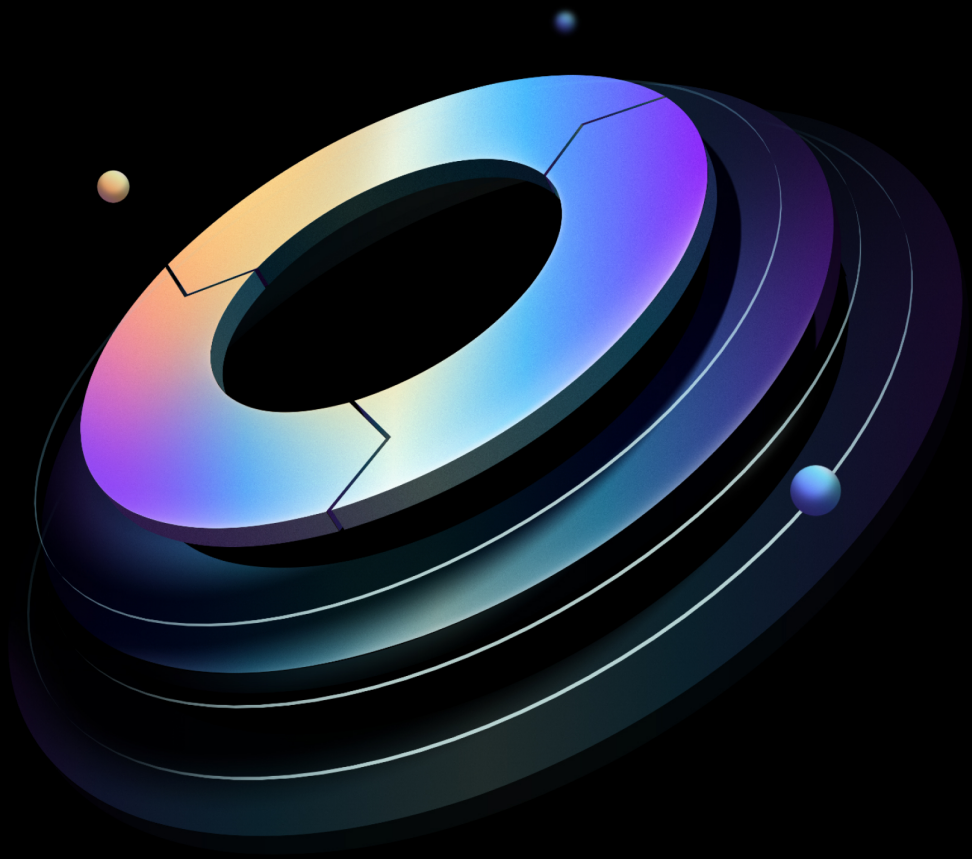
HashiCorp

Vault Enterprise on OpenShift: Solution Design Guide

Built from commit 7fca46d

HashiCorp | Validated Designs

15 June 2026



HashiCorp

Contents

0.1 Introduction

> **Note**
>
> This solution design guide for Vault Enterprise on Red Hat OpenShift is *emerging guidance*. We continually update this guide as we identify common best practices and configurations to help customers get the most from the product.

The purpose of this document is to provide implementation guidance for teams deploying Vault Enterprise on Red Hat OpenShift Container Platform (OCP). It describes platform-aligned patterns, operational considerations, and design decisions for running Vault as a service on OpenShift.

Vault supports multiple deployment models in Kubernetes environments, with a spectrum of client-centric agent and operator solutions intended to streamline secrets management for containerized workloads. In this context, teams may operate the Vault service external to a Kubernetes platform, or deploy it within a cluster, potentially alongside consuming application workloads. This guide focuses specifically on deployments where Vault server runs inside OpenShift as a highly available service, providing secrets management for containerized or traditional applications outside of the immediate cluster.

The content emphasizes architectural choices and day-2 operational practices that are specific to OpenShift, including security controls, storage behavior, networking and routing, lifecycle management, and platform integrations. This content aims to reduce deployment ambiguity and promote consistent, supportable configurations across environments.

Note that while this guide focuses on OpenShift specifically, much of this guidance is also applicable to Kubernetes generally.

Client implementation specifics, application integrations and consumption patterns are not covered in this guide. For detailed information on how to access Vault-managed secrets from your OpenShift-based applications, we recommend reading the following Vault Secrets Operator docs as a starting point:

- Vault Secrets Operator
- Run the Vault Secrets Operator on OpenShift
- Vault Secrets Operator bundle on the Red Hat Ecosystem Catalog
- hashicorp/vault-secrets-operator on GitHub

0.2 Platform concepts

This section introduces the core platform concepts that customers need to operate, secure, and support workloads for Vault on Red Hat OpenShift. It frames these concepts from an ownership and operational perspective first, then layers in how they apply when deploying HashiCorp Vault on OpenShift.

0.2.1 Projects

OpenShift organizes workloads around *projects*, which act as the primary unit of ownership, isolation, and policy enforcement. Platform teams use projects to define clear boundaries between teams, environments, and operational responsibilities.

OpenShift builds on Kubernetes, but it extends the Kubernetes namespace model with additional defaults and controls. A project maps one-to-one with a Kubernetes namespace, while adding opinionated constructs such as role bindings, quotas, and security constraints. These extensions allow teams to encode governance and guardrails centrally rather than relying on each application team to configure them correctly. These features go hand-in-hand with the operational models of HashiCorp Vault.

Platform teams typically provision and manage projects, while application teams consume them. This separation supports consistent access control, predictable resource allocation, and clearer escalation paths when issues arise.

OpenShift runs on major public cloud providers (AWS, Azure, GCP, IBM Cloud) or in on-premises self-managed environments. Regardless of deployment model, the project abstraction remains consistent, which allows administrators to apply the same operational patterns.

Figure 1: OpenShift core concepts

Figure 1. OpenShift core concepts (Projects, Policy roles, policy bindings, users)

Review Red Hat's OpenShift Container Platform Projects to better understand the core platform concepts.

0.3 Deployment options

This section describes the recommended deployment options and example tooling patterns for running Vault Enterprise on OpenShift. We provide these examples as reference architectures. Your organization may select a different toolchain or integration pattern based on existing platform standards, security controls, and operational constraints.

For security considerations on OpenShift, refer to the official documentation:

- Red Hat - Using RBAC to define and apply permissions
- Red Hat - Managing security context constraints
- Red Hat - Reference of security context constraints commands

0.3.1 Package & install manager - Helm

OpenShift supports Helm as a package and install manager that standardizes and simplifies the deployment of containerized applications on Kubernetes. Helm charts define repeatable, auditable deployments which can optionally integrate with GitOps workflows.

We recommend using the official Vault Helm chart to install and configure Vault on OpenShift, with a GitOps-based workflow as the preferred operating model. The chart supports multiple deployment patterns based on the provided values. Refer to the Initial configuration section of the Vault: Operating Guide for Adoption for general cluster configuration details.

- Reference Helm values repository for this HVD: a set of reference configurations that map this guide's recommendations to concrete deployment scenarios you can adapt to your environment.
- Docs: Run Vault on OpenShift

0.3.2 GitOps - Argo CD

OpenShift GitOps uses Argo CD to manage the Vault deployment lifecycle. It integrates with your version control system (VCS), treating Helm chart configuration as the source of truth. We recommend standardizing Vault deployment and management through Argo CD to ensure consistency,

auditability, and controlled operations across environments. Deliver changes through Git, with Argo CD continuously reconciling cluster state to the desired configuration. Limit manual modifications outside of this flow to exceptional cases.

On OpenShift, install the Argo CD Operator from OperatorHub. For multi-cluster deployments, scope access using Argo CD Projects to enforce clear boundaries between environments. When enabling auto-sync across multiple Vault clusters, ensure ApplicationSets and subchart references are uniquely named and isolated to avoid conflicts. Refer to the Argo CD Projects documentation for additional details.

0.3.3 Helm values precedence

When Argo CD renders the Vault Helm charts, value precedence follows this order (highest to lowest):

```
parameters > valuesObject > values > valueFiles > chart defaults
```

The lowest-precedence layer, chart defaults, is the set of values bundled in the chart's `values.yaml`. Platform teams must understand the configuration precedence order to avoid misconfiguration. Subcharts must not define opaque defaults through helper templates. Because application teams often lack visibility into these templates, hidden defaults make troubleshooting difficult and increase operational risk.

Keep configuration values explicit and document any overrides. This enables scalable Vault deployments across OpenShift projects while minimizing environment-specific customizations.

Refer to the official Red Hat documentation for additional details:

- Red Hat - Understanding OpenShift GitOps
- Red Hat - GitOps CLI Argo CD Reference
- GitHub: Red Hat Developer - OpenShift GitOps Usage Guide

0.4 High availability topologies

Vault Enterprise achieves high availability (HA) through clustering with integrated storage based on the Raft consensus algorithm. On OpenShift, the platform manages Pod placement and lifecycle dynamically, so topology decisions that affect Raft quorum require explicit scheduling and storage

constraints. Pod rescheduling, persistent storage, and topology-aware scheduling all affect how Vault maintains quorum and recovers from failures.

The following sections cover OpenShift-specific topology decisions for HA cluster design: Pod distribution across failure domains, storage topology, redundancy zones, and Service topology.

For foundational concepts about Vault clustering, quorum, redundancy zones, and Autopilot, refer to Vault Enterprise Architecture.

0.5 StatefulSet architecture

Vault requires a StatefulSet rather than a Deployment because the StatefulSet controller provides capabilities essential for running a distributed consensus system on Kubernetes. Each Pod receives a predictable DNS name based on its ordinal index and maintains its own PersistentVolumeClaim that persists across Pod rescheduling.

0.5.1 Pod identity and storage

The StatefulSet controller assigns each Pod a stable network identity based on its ordinal index. For a StatefulSet named `vault`, the Pods receive DNS names such as `vault-0.vault-internal`, `vault-1.vault-internal`, and `vault-2.vault-internal`. Vault uses these stable identities for Raft peer communication and cluster formation.

Each Pod maintains its own PersistentVolumeClaim (PVC) that persists across Pod rescheduling. When Kubernetes recreates the Pod, the new Pod mounts the same PVC, preserving the Raft data directory. This behavior ensures that a restarted Pod rejoins the cluster with its existing state rather than as a new member.

The Vault Helm chart creates PVCs through `volumeClaimTemplates` in the StatefulSet specification. Configure the storage requirements in the Helm values file under `server.dataStorage`. For Raft storage configuration details, refer to the Storage topology section.

0.5.2 Anti-affinity and Pod distribution

Pod anti-affinity rules prevent the scheduler from co-locating multiple Vault Pods on the same node, which would create a single point of failure.

The Vault Helm chart defaults to `requiredDuringSchedulingIgnoredDuringExecution` with `topologyKey: kubernetes.io/hostname`. If you override this setting with `preferredDuringSchedulingIgnor`, the scheduler may co-locate Pods under resource pressure.

This default anti-affinity keeps Vault Pods on separate nodes. It does not enforce distribution across higher-level failure domains such as availability zones or rack groups. Topology labels provide fault isolation only when each label value maps to infrastructure that can fail independently. Apply the labels consistently across all Vault-eligible nodes. For example, different values for `topology.kubernetes.io/zone` do not improve isolation if the nodes still share the same power feed, network path, or storage failure point. Configure explicit `topologySpreadConstraints` with DoNotSchedule, as described in the Pod topology spread constraints section, so the scheduler enforces the intended spread when placing new Pods.

For specific anti-affinity configuration values, refer to the affinity configuration in the Vault Helm chart documentation.

0.5.3 Pod disruption budgets

A Pod Disruption Budget (PDB) limits how many Pods a voluntary disruption can remove simultaneously. Voluntary disruptions include node drains during cluster upgrades, maintenance operations, and manual Pod deletions. Pod Disruption Budgets (PDBs) do not affect involuntary disruptions such as hardware failures.

For a Vault cluster to maintain quorum, a majority of voting Pods must remain available. Configure a PDB that ensures the cluster retains quorum during voluntary disruptions. When you enable HA mode (`server.ha.enabled: true`), the Vault Helm chart creates a PDB by default (`server.ha.disruptionBudget.enabled` defaults to `true`). The chart expresses the PDB using `maxUnavailable` and lets you override it with `server.ha.disruptionBudget.maxUnavailable`.

For a 5-pod cluster, the Helm chart calculates `maxUnavailable` to 2, which allows node drains to evict up to two Pods simultaneously while maintaining quorum with the remaining three voting members. This default preserves quorum but leaves no margin for error. If a third Pod fails while two are already unavailable, the cluster loses quorum. Setting `maxUnavailable` to 1 provides additional headroom by keeping four voters available during each eviction, at the cost of slower drain operations.

For a 6-pod redundancy zones cluster, the Helm chart defaults `maxUnavailable` to 1 when `server.ha.raft.redundancyZones.enabled` is **`true`**. The PDB cannot distinguish between voting and non-voting Pods, and the standard formula `floor((n-1)/2)` evaluates to 2 for 6 Pods. Allowing 2 simultaneous evictions risks quorum loss when both evicted Pods are voting members. If you override `maxUnavailable` through `server.ha.disruptionBudget.maxUnavailable`, ensure the value accounts for the reduced number of voting members in a redundancy zones architecture.

When the Kubernetes control plane initiates a node drain, it sends eviction requests that the API server validates against PDBs. The API server rejects evictions that would violate the PDB constraint, so the drain process evicts Pods sequentially as PDB conditions allow. This behavior prevents drain operations from simultaneously evicting enough Vault Pods to break quorum.

0.6 Redundancy zones

Redundancy zones provide automatic failover within failure domains by grouping one voting and one or more non-voting Vault nodes per zone. When a voter fails, Autopilot can promote a stable non-voting server to restore the intended voter distribution across zones. Autopilot first prefers a stable server from the same zone. If no same-zone candidate is available, Autopilot can temporarily place two voters in one surviving zone. After the failed zone recovers and a stable server is available there again, Autopilot demotes the temporary extra voter and returns to one voter per zone. We recommend this architecture for production Vault deployments on OpenShift. In this section, zone refers to a failure domain. On managed platforms, a zone usually maps to a cloud availability zone. On self-managed OpenShift, a zone can map to another independent failure domain, such as a rack group or power domain, that you expose through node labels.

For more information on redundancy zones and Autopilot behavior, refer to Redundancy Zones.

0.6.1 Recommended: 6-pod architecture with redundancy zones

Deploy 6 Vault Pods across 3 zones, with 2 Pods per zone. On self-managed OpenShift, those zones can represent independent rack groups, power domains, or other low-latency failure boundaries. In each zone, one Pod serves as a voting member and one as a non-voting member. This architecture provides the following benefits:

- **Intra-zone failover**: If a voting Pod fails, Autopilot first attempts to promote the non-voting Pod in the same zone to voting status, restoring the intended one-voter-per-zone layout. If no

same-zone candidate is available or meets stability criteria, Autopilot promotes a stable non-voter from another zone, temporarily placing two voters in that zone. The cluster can tolerate up to 4 Pod failures, but only in a sequential scenario where each promotion completes before the next failure. For a side-by-side summary of the 6-pod redundancy-zones architecture and the 5-pod fallback architecture, refer to the Architecture comparison table.

- **Cross-zone resilience**: With one voting member in each zone, the cluster tolerates the loss of any single zone because the remaining 2 of 3 voters preserve quorum. If Autopilot completes a promotion after the first zone loss, the cluster can tolerate a second sequential zone loss only when that second loss removes the remaining zone with a single voter.
- **Reduced recovery time**: Non-voting Pods replicate Raft state from the leader. Because a non-voter already holds a current copy of the data, promotion changes Raft membership instead of requiring a full data resynchronization.
- **Smaller quorum window**: With 3 voters, the Raft leader needs only one additional acknowledgment to commit writes, so it does not wait for the slowest voter. If one zone experiences higher latency or storage contention, the voter in the third zone can satisfy quorum independently.

This architecture requires the PodTopologyLabels admission controller, which enables Pods to obtain node topology labels through the Downward API. Kubernetes 1.35 and later versions enable this feature by default. OpenShift 4.22 and newer includes the corresponding Kubernetes version. You also need a Vault Helm chart version that supports redundancy zones automation on Kubernetes (v0.33.0 or later). Verify feature and chart availability in your platform and Helm chart release notes.

In Vault Helm chart versions that support redundancy zones, set `server.ha.raft.redundancyZones.enabled` to `true` and add `autopilot_redundancy_zone = "VAULT_REDUNDANCY_ZONE"` entry to `server.ha.raft.config`. When nodes have the `topology.kubernetes.io/zone` label, the chart exposes that value to each Pod through the `VAULT_REDUNDANCY_ZONE` environment variable. At startup, the chart substitutes that value into the Raft storage configuration. If a Pod does not receive the zone label, Vault startup fails, so verify Pod labels before you rely on zone assignment. Autopilot then uses the zone value for voter placement and promotion decisions.

0.6.2 On-premises and custom failure domains

On self-managed OpenShift where you control node labeling and node lifecycle, the topology inputs for redundancy zones and scheduling constraints are not limited to cloud availability zones.

You can use any failure domain as a zone if it is a real fault boundary and your node provisioning and replacement process consistently applies the correct zone label to every node in that domain. Common examples in on-premises data centers include separate data halls, rack rows with independent power and network paths, and separate buildings within the same campus. Do not treat domains as independent when they still share the same power feed, network path, or storage failure point. Evaluate failure dimensions such as power distribution, network uplink, and rack independence to identify independent domains. Map those domains to node labels and use them as topology inputs for scheduling and redundancy zone configuration.

Decision framework

Use the following framework to select the appropriate topology based on your failure domain characteristics.

3 or more independent failure domains. When the deployment has at least 3 independent failure domains, deploy the 6-pod reference architecture with redundancy zones. On managed platforms, these are typically cloud availability zones. On self-managed OpenShift, use the failure domains identified in the On-premises and custom failure domains section. We recommend at least 3 zones because Autopilot targets one voter per zone. In a 2-zone layout, the cluster converges to 2 voters, so loss of either zone loses quorum. Vault can run with 2 configured zones, but we do not recommend that layout for production.

In the 3-zone reference topology, loss of one zone removes 1 voter and 1 non-voter. The remaining 2 voters preserve quorum. Autopilot then promotes a stable non-voter in a surviving zone to restore the 3-voter count. That temporary placement can leave one surviving zone with 2 voters and the other with 1. One-voter-per-zone resumes after the failed zone recovers and Autopilot demotes the extra voter.

Exactly 2 failure domains. A single Vault cluster across exactly 2 failure domains does not meet zone-failure HA goals. Without redundancy zones, a 3-2 voter split loses quorum when the 3-voter domain fails. With redundancy zones, the cluster converges to 2 voters regardless of Pod placement, so the loss of either zone loses quorum.

We recommend splitting the environment into 3 independent failure domains where possible. When 3 domains are not achievable, deploy separate Vault clusters in each failure domain and connect them with DR replication or performance replication to provide site-level resilience. For multi-cluster replication architecture, refer to the Multi-region architectures section.

If you must run a single cluster across exactly 2 failure domains as a constrained fallback, deploy 5 voting Pods without redundancy zones and place them 3-2. Accept that loss of the 3-voter domain causes an outage. Configure `topologySpreadConstraints` to help maintain the intended 3-2 shape using a zone key. The Helm chart's default node-level anti-affinity helps distribute Pods across different nodes within each zone. For fallback architecture details, refer to the Fallback: 5-pod architecture without redundancy zones section.

No independent higher-level failure domains. When all nodes share the same power, cooling, network uplink, and storage infrastructure with no failure domain above the individual host, deploy 5 voting Pods without redundancy zones. When `topology.kubernetes.io/zone` does not represent a real failure domain, zone-based scheduling and redundancy zones provide no fault isolation above the node level. Node-level anti-affinity (`topologyKey: kubernetes.io/hostname`) remains the minimum distribution constraint. DR replication is the minimum recommendation for site-level resilience because the entire site shares a single affected scope. Performance replication addresses read locality and horizontal scale but is not a substitute for disaster recovery. For architecture details, refer to the Fallback: 5-pod architecture without redundancy zones section. For DR replication architecture, refer to the Multi-region architectures section.

Label mapping

Map failure domains to node labels consistently so the scheduler and Vault evaluate the same topology. Incorrect or inconsistent labels can make placement appear balanced without creating real fault isolation.

Map your chosen failure domain to `topology.kubernetes.io/zone` for redundancy zones. The Helm chart's redundancy-zones feature reads zone assignment from this label. Use the same key in `topologySpreadConstraints` to keep Vault and the scheduler aligned. On self-managed OpenShift where you control node labeling end to end and your storage and network topology align with the label, map custom failure domains to `topology.kubernetes.io/zone` before deploying Vault. On managed platforms that already expose `topology.kubernetes.io/zone`, such as Red Hat OpenShift Service on AWS (ROSA), Azure Red Hat OpenShift (ARO), and OpenShift Dedicated, use the platform-provided label values directly.

0.6.3 Pod topology spread constraints

Pod topology spread constraints provide fine-grained control over how the scheduler distributes Pods across failure domains. Unlike anti-affinity rules that prevent co-location, topology spread constraints control how the scheduler distributes Pods across specified topology keys. The appropriate constraint configuration depends on your topology.

By default, the Vault Helm chart does not set `topologySpreadConstraints`. When you do not define `server.topologySpreadConstraints`, the scheduler may still apply Kubernetes cluster-level default topology spread constraints. These defaults use `ScheduleAnyway` semantics, so they remain soft preferences and do not prevent uneven placement under resource pressure. For production deployments, configure `server.topologySpreadConstraints` with DoNotSchedule so the scheduler enforces the configured Pod distribution across failure domains when placing new Pods.

3 failure domains (reference topology). Set `topology.kubernetes.io/zone` as the topology key with `whenUnsatisfiable` set to DoNotSchedule. If placing a new Pod would increase zone imbalance beyond `maxSkew`, the scheduler leaves that Pod Pending instead of placing it in the wrong zone. The scheduler does not move existing Pods to rebalance the cluster. This maintains the intended 2-2-2 shape for the 6-pod redundancy zones architecture and the 2-2-1 shape for a 5-pod cluster during scheduling.

2 failure domains (constrained fallback). Use `topology.kubernetes.io/zone` as the topology key to help maintain the intended 3-2 placement for the 5-pod architecture. Set `maxSkew` to 1 and `whenUnsatisfiable` to DoNotSchedule. The scheduler distributes Pods across the two zones but cannot achieve even placement with an odd replica count, so expect one zone to hold one more Pod than the other.

No independent failure domains. When `topology.kubernetes.io/zone` does not map to a real failure domain, or all nodes carry the same zone value, topology spread constraints on that key do not provide fault isolation above the node level. In this environment, node-level anti-affinity (`topologyKey: kubernetes.io/hostname`) remains the minimum distribution constraint. Configure topology spread constraints on `topology.kubernetes.io/zone` only if you later expose real failure domains through node labels.

0.6.4 Fallback: 5-pod architecture without redundancy zones

This fallback topology preserves quorum across Pod failures but does not provide automatic voter replacement. Deploy 5 voting Pods without redundancy zones when any of the following conditions apply:

1. The OpenShift version does not yet support the PodTopologyLabels admission controller (versions before 4.22), preventing the Helm chart from automating redundancy zone assignment.
2. The deployment environment does not have 3 or more independent failure domains, as described in the On-premises and custom failure domains section.

All 5 Pods are voting members, so the cluster tolerates 2 simultaneous Pod failures. A 5-voter cluster maintains quorum after losing a failure domain only when that domain contains no more than two voters.

When you have 3 independent failure domains but cannot use redundancy zones, place the 5 voters in a 2-2-1 distribution across those domains.

A 3-2 split across 2 failure domains does not tolerate loss of the 3-voter domain. If your environment has only 2 failure domains, deploy separate Vault clusters in each domain and connect them with DR replication or performance replication instead of running a single cluster that cannot survive zone loss. If a single cluster across 2 domains is unavoidable, place voters 3-2 and accept the zone-failure risk.

In environments with only 1 independent failure domain, rely on node-level anti-affinity across nodes.

When the platform version is the only blocker, treat this architecture as interim and plan OpenShift and Helm chart upgrades so you can enable redundancy zones. When your environment cannot provide 3 independent failure domains, the 5-pod architecture is the appropriate long-term topology.

For the 5-pod architecture without redundancy zones, refer to the Vault Raft Reference Architecture.

0.6.5 Migration from 5-pod to 6-pod architecture

Migrate to the 6-pod redundancy-zones architecture only when your environment has 3 independent failure domains and your OpenShift release supports the pod topology labels required for

redundancy zones. If your environment cannot provide 3 independent failure domains, the 5-pod architecture remains the appropriate long-term topology and this migration does not apply.

A Helm upgrade updates the StatefulSet template, but existing Vault Pods do not adopt redundancy zone settings until you recreate them. Because the Vault server StatefulSet uses the OnDelete update strategy, complete the OpenShift and Helm changes first, then recreate the existing Pods one at a time before you scale the cluster to 6 replicas.

The general migration approach includes the following steps:

1. Upgrade OpenShift to version 4.22 or later so the platform supports the pod topology labels required for redundancy zones
2. Update the Vault Helm chart to the latest supported version that includes redundancy-zones support (v0.33.0 or later)
3. Configure redundancy zones and topology spread constraints in the Vault Helm chart
4. Recreate the existing Vault Pods one at a time so they start with the updated redundancy zone configuration
5. Scale the StatefulSet from 5 to 6 replicas
6. Run `vault operator raft autopilot state` to verify that Autopilot correctly identifies zones and designates one voter and one non-voter per zone

Coordinate this migration during a maintenance window. Apply the OpenShift and Helm changes first, recreate the existing Pods one at a time, and then scale to 6 replicas. Validate zone assignment and Autopilot state as the recreated Pods rejoin and again after the scale event to confirm that the cluster converges on the intended one-voter, one-non-voter pattern in each zone.

0.6.6 Architecture comparison

The following table compares the two architectures to help you select the appropriate topology.

Aspect	6-pod with redundancy zones	5-pod without redundancy zones
OpenShift version	4.22 or later (Kubernetes 1.35)	No specific minimum
Helm chart version	v0.33.0 or later	Latest supported Vault Helm chart version

Aspect	6-pod with redundancy zones	5-pod without redundancy zones
Failure domain requirement	At least 3 independent failure domains mapped to 3 zone values. A 2-zone redundancy-zones layout converges to 2 voters, so loss of either zone loses quorum	Constrained fallback when the platform cannot support redundancy zones or the environment cannot provide 3 independent failure domains
Voting members	3	5
Non-voting members	3 (1 per zone)	0
Automatic voter replacement	Yes. Autopilot can promote a stable non-voting server to restore the intended voter distribution across zones. Autopilot demotes temporary extra voters after the failed zone recovers and a stable server is available there again	No, the cluster operates with fewer voters until the Pod recovers
Pod failure tolerance	Up to 4 Pods in a sequential failure scenario, but only if each failure occurs one at a time and Autopilot completes each promotion before the next failure	2 Pods

Aspect	6-pod with redundancy zones	5-pod without redundancy zones
Single zone failure	Tolerated (2 of 3 voters remain)	Tolerated only when no single failure domain contains more than 2 voters. A 2-2-1 distribution across 3 failure domains tolerates loss of any single domain. A 3-2 split across 2 domains does not tolerate loss of the 3-voter domain. A single cluster across 2 failure domains does not meet zone-failure HA goals. We recommend 3 failure domains or separate clusters with DR/performance replication
Sequential second zone failure	Conditional: the cluster tolerates a second zone loss only if it affects the 1-voter zone after the first successful promotion. After the first zone loss and promotion, one surviving zone temporarily holds 2 voters and the other holds 1	Not tolerated

For values files that implement both topologies in this comparison, refer to the reference Helm values for this HVD. The `-rz` profiles deploy the recommended 6-pod redundancy-zones architecture with the matching `topologySpreadConstraints` and `autopilot_redundancy_zone` settings, and `values-awskms-no-rz.yaml` deploys the 5-pod fallback for environments that cannot run redundancy zones.

0.7 Multi-region architectures

Multi-region Vault deployments provide geographic redundancy and can improve latency for globally distributed applications. Vault Enterprise supports two replication modes for multi-region architectures: disaster recovery (DR) replication and performance replication (PR).

DR replication creates a standby cluster in a secondary region that can assume operations if the primary cluster becomes unavailable. The secondary cluster replicates data from the primary, but it does not serve client requests during normal operation. For DR replication architecture and failover procedures, refer to Vault Enterprise Architecture.

Performance replication replicates Vault data to secondary clusters that can serve client requests independently, though tokens and leases remain local to each cluster. This approach reduces latency for applications in remote regions and distributes load across multiple clusters. Performance replication secondaries forward write operations and certain consistency-sensitive requests to the primary cluster. For performance replication design and scaling considerations, refer to Performance Replication.

On OpenShift, multi-region architectures require network connectivity between clusters for replication traffic. Configure OpenShift networking to allow the primary and secondary Vault clusters to communicate over ports 8200/tcp (API) and 8201/tcp (cluster). For OpenShift-specific network configuration and replication traffic considerations, refer to the Networking, routes, and TLS section in this guide.

0.8 Service topology

In HA mode, the Vault Helm chart creates Services you can use to target the active node, standby nodes, or all server Pods. The `<release>-active` and `<release>-standby` Services select Pods by the `vault-active` label, which Vault maintains when the server configuration includes the `service_registration "kubernetes"{}` stanza.

The following table describes the Service endpoints that the Vault Helm chart creates in HA mode.

Service	Purpose
`<release>`	Selects all Vault server Pods for this release. Exposes ports `8200/tcp` (API) and `8201/tcp` (cluster). Route client API traffic to this Service.
`<release>-active`	Selects the active Vault Pod (`vault-active: "true"`). Exposes ports `8200/tcp` and `8201/tcp`. Use this Service when traffic must reach the active node, such as replication traffic on port `8201/tcp`.
`<release>-standby`	Selects standby Vault Pods (`vault-active: "false"`). Use this Service only when you have a specific requirement to target standby nodes.
`<release>-internal`	Headless Service (`clusterIP: None`) providing a stable DNS name per Pod, such as `vault-0.vault-internal`, for Raft peer communication. Do not use this Service for client traffic.

Route client API traffic through `<release>` on port `8200/tcp` so requests can reach all Vault Pods. With Vault Enterprise, standby nodes operate as performance standbys that handle the majority of common client requests locally, so distributing traffic across all Pods reduces load on the active node. Performance standbys automatically forward write operations and other requests that modify shared state to the active node. For replication traffic, use `<release>-active` on port `8201/tcp` because replication must connect to the active node.

A headless Service without a ClusterIP returns all Pod IPs in DNS responses. If you use this approach for client traffic, create a dedicated headless Service for that purpose. Do not use `<release>-internal`, which Vault uses for Raft peer communication.

Not all OpenShift deployments include cloud load balancer integration. On-premises and self-managed OpenShift clusters typically use MetalLB, HAProxy, or external load balancers. When configuring Services for external access, select the appropriate Service type for your platform.

0.9 OpenShift platform considerations

OpenShift provides additional configuration options that affect Vault Pod scheduling and resource allocation. This section covers two design concerns for a Vault Enterprise deployment on OpenShift: node placement, which determines where Vault Pods run, and resource management, which determines how the cluster allocates CPU and memory to Vault.

0.9.1 Node placement strategies

Node placement controls which worker nodes the scheduler runs Vault Pods on. The choice affects fault isolation, resource availability during bursts, and the cluster's ability to reschedule Pods during node drains and unplanned node loss. This section covers the node selection mechanisms available in the Vault Helm chart and the per-zone node count required by the reference topology.

For the reference 6-pod, 3-zone topology, provision at least 3 Vault-eligible worker nodes per zone. Two nodes per zone satisfy `kubernetes.io/hostname` Pod anti-affinity for the 2 Vault Pods in each zone. The third node gives the scheduler in-zone scheduling headroom during node drains, machine configuration rollouts, OpenShift upgrades, and unplanned node loss. The reference topology therefore requires 9 worker nodes; weigh this headroom against the recovery delay you accept without it.

Without the third node, anti-affinity blocks rescheduling inside the zone and `topologySpreadConstraints` with DoNotSchedule blocks rescheduling to another zone, so the evicted Pod remains Pending until the drained node returns.

This headroom helps only when the Pod's PersistentVolumeClaim can reattach to another node in the same zone, as with zone-local block storage. A local PersistentVolume bound to a single node cannot follow, so the Pod stays Pending until its original node returns, regardless of spare capacity.

Control placement through `server.nodeSelector`, `server.affinity`, and `server.tolerations` in the Helm values. The redundancy-zones feature reads zone assignment from `topology.kubernetes.io/zone`, so align node labels with that key as described in the Label mapping section.

When you override `server.affinity`, include the chart's default `podAntiAffinity` block from `values.yaml` alongside your node affinity rules so the override does not weaken the `kubernetes.io/hostname` anti-affinity guarantee. To reserve dedicated nodes for Vault, taint those nodes and

pair `server.tolerations` with a `nodeAffinity` rule or `server.nodeSelector` so Vault Pods schedule only there and other workloads cannot.

Verify that project-wide node selectors and default tolerations in the Vault namespace do not conflict with the placement rules configured through the Vault Helm chart.

We recommend deploying Vault on a dedicated worker node pool for production workloads. Dedicated nodes prevent application workloads from competing with Vault, at the cost of higher infrastructure spend and a separate node pool to maintain. If your organization cannot allocate dedicated nodes, shared nodes are a supported fallback; the Resource management section covers the required resource profile for both models.

The Cluster Autoscaler and the Descheduler can move Vault Pods outside planned maintenance windows, triggering an avoidable leader election and a brief write interruption. Prevent the Cluster Autoscaler from scaling down nodes that host Vault Pods, especially when those Pods use zone-local or node-local storage that cannot move to another node. Ensure any Descheduler policy respects the Vault PodDisruptionBudget rather than evicting Vault Pods to rebalance the cluster.

Before draining the node that hosts the active Vault Pod, transfer leadership with `vault operator step-down` and confirm a new active node, so the write interruption happens at a controlled moment rather than mid-operation. Increase `server.terminationGracePeriodSeconds` from its default of 10 seconds so the `kubelet` does not send SIGKILL while Vault is still draining in-flight requests during shutdown. Involuntary disruptions such as unplanned node loss still cause an election, so this practice reduces rather than removes write interruption.

0.9.2 Resource management

Resource requests, limits, Quality of Service (QoS) class, and Pod priority influence Vault scheduling and eviction behavior on OpenShift. This section recommends resource profiles for dedicated and shared node deployments and identifies platform controls that must preserve those profiles.

The Vault Helm chart does not set resource requests or limits by default. Without explicit resources or namespace defaults, Vault Pods receive BestEffort QoS, which provides the least eviction protection under node memory pressure. Refer to the Kubernetes documentation on Pod selection for kubelet eviction.

Set equal memory requests and limits on every container in the Pod, including init containers, extra containers, and admission-injected sidecars. Any container with mismatched or omitted requests

and limits downgrades the entire Pod's QoS class, so disable automatic sidecar injection in the Vault namespace unless the injected containers carry matching CPU and memory requests and limits.

Size the memory request and limit from the observed peak working set and vault-benchmark results, with enough headroom to avoid Vault restarts during peak load. Size Vault against node allocatable capacity so it accounts for platform reservations and DaemonSet usage.

Set `server.resources.requests.cpu` to reserve enough CPU for Vault's normal load, covering at least steady-state usage plus operational headroom. CPU limits, by contrast, can throttle Vault during bursts even when the node has idle CPU, which increases latency for CPU-intensive operations. Refer to Hardware sizing for Vault servers for initial sizing.

The deployment model determines whether to cap CPU. Equal memory requests and limits, together with a dedicated PriorityClass, already give Vault strong eviction protection in both models, so the remaining decision is whether the cost of a CPU limit is justified. On dedicated nodes, omit the CPU limit so Vault can burst into idle CPU, which produces Burstable QoS. This is the preferred profile: no application workloads compete for the node's memory, so a stronger guarantee adds little, while uncapped CPU avoids throttling the active node.

On shared nodes, set CPU requests equal to CPU limits on every container, including init containers and injected sidecars, to reach Guaranteed QoS, and size both from benchmarked peak demand. A co-tenant memory spike can exhaust the node faster than the `kubelet` evicts, and Guaranteed QoS gives Vault the strongest out-of-memory protection in that case. The cost is CPU throttling once demand exceeds the limit, which on a voting Pod can delay Raft heartbeats and trigger an unnecessary leader election. Size the limit from peak rather than steady-state demand, monitor throttling against latency objectives, and move Vault to dedicated nodes rather than raising the limit without bounds.

OpenShift admission controls can silently rewrite the resource configuration the Helm chart applies and defeat the intended QoS class. LimitRange defaults, ResourceQuota constraints, the Cluster Resource Override Operator, and VerticalPodAutoscaler all mutate Pod resources, so design the deployment to exclude Vault from these controls.

Set `server.priorityClassName` to a dedicated PriorityClass that ranks Vault above application workloads and below the platform's system-critical Pods, such as the networking and storage agents that run as `system-node-critical`. Set `preemptionPolicy` to Never on this class so Vault waits in the scheduling queue for capacity instead of evicting application workloads, because OpenShift honors PodDisruptionBudgets only at a best-effort level during preemption. Priority re-

duces eviction risk, but spare per-zone capacity remains the primary recovery mechanism for node drains and node loss.

The following table compares resource management considerations for dedicated and shared node deployment models.

Consideration	Dedicated nodes (recommended)	Shared nodes (fallback)
Resulting QoS class	Burstable (memory request=limit; CPU limit omitted).	Guaranteed (CPU and memory request=limit).
Memory request and limit	Set equal for every container.	Set equal for every container.
CPU limit	Omit, so Vault can use idle CPU on the dedicated node during bursts.	Set equal to the CPU request, sized from benchmarked peak. Throttling can delay Raft heartbeats and force an election; escalate to dedicated nodes rather than raise the limit.
PriorityClass	Strongly recommended. Create a dedicated class with `preemptionPolicy` set to Never, ranked above application workloads. Confirm your networking and storage agents use `system-node-critical`, or another class above Vault, so Vault never outranks the components it depends on.	Strongly recommended; co-tenant pressure makes Vault's eviction-ranking protection matter more. Use the same dedicated non-preempting class and run the same agent-priority check.

Consideration	Dedicated nodes (recommended)	Shared nodes (fallback)
Cost and operational overhead	Higher. A separate node pool to provision and maintain.	Lower. Reuses existing capacity, but co-tenancy adds the QoS and throttling risks in the other rows.

Evicting the active Vault Pod triggers a Raft leader election and briefly prevents the cluster from processing writes. Resource configuration and PriorityClass reduce eviction risk but do not eliminate it under severe node pressure.

0.10 Performance standbys

Performance standbys handle most requests locally and forward write operations and other requests that modify shared persistent state to the active node.

Performance standby behavior is separate from voting and non-voting membership. Voting and non-voting membership affects quorum and leader election, while performance standby behavior affects how Vault handles client requests on standby nodes.

For foundational performance standby concepts and behavior, refer to Vault Enterprise Architecture.

This guide assumes your Vault Enterprise license includes the performance standbys feature. If your license does not include this feature, standby nodes operate as standard standbys that forward all requests, including reads, to the active node.

0.10.1 Deploying performance standbys

Deploy additional standby capacity by increasing the StatefulSet replica count. The Vault Helm chart configures replicas through `server.ha.replicas`. Autopilot manages Raft membership and can add voting and non-voting members based on your cluster configuration.

Performance standbys increase the total number of Raft peers, which affects cluster communication overhead. Each additional Raft peer receives log entries from the leader, so monitor replication lag

and leader resource utilization when scaling. For read-heavy workloads such as Transit operations and key-value secret reads, add non-voting Pods to increase throughput without affecting quorum. With redundancy zones enabled, Autopilot manages non-voting membership automatically.

0.11 Scaling and Autopilot behavior

The StatefulSet controller manages Vault Pod scaling, while Vault's Autopilot feature manages Raft cluster membership. These systems interact during scale-up and scale-down operations.

0.11.1 Scaling up

When you increase the StatefulSet replica count, the StatefulSet controller creates new Pods with sequential ordinal indices. Each new Pod starts Vault, which attempts to join the existing Raft cluster. Autopilot evaluates the new node and determines whether it becomes a voting or non-voting member based on the cluster configuration and zone topology.

0.11.2 Scaling down

When you scale down a StatefulSet, Kubernetes deletes Pods in descending ordinal order. For example, scaling from 6 to 5 replicas deletes `vault-5`. The StatefulSet controller manages Pod lifecycle but has no awareness of Raft cluster membership. When Kubernetes deletes a Pod, its Raft peer entry remains in the cluster configuration as a dead server. By default, Kubernetes retains the PVC for the deleted Pod, which protects the Pod's Raft data.

Scaling the StatefulSet back up recreates the Pod with the same ordinal index and reattaches the retained PVC. Because the PVC still contains valid Raft data, the Pod resumes its previous cluster identity without requiring data synchronization from the leader.

Before permanently reducing the replica count, verify cluster health using `vault operator raft list-peers` and confirm that the cluster can maintain quorum with fewer voting members. Run `vault operator raft remove-peer` for the departing peer before you reduce the replica count. After the scale-down completes, delete the orphaned PVC for the removed Pod. If you already removed the Raft peer but then scale back up, the returning Pod cannot rejoin with its existing data. You must delete the PVC and allow the Pod to join as a new member.

0.11.3 Dead server cleanup

Autopilot removes dead servers automatically when you enable `cleanup_dead_servers`. In the reference 6-pod, 3-zone topology, set `min_quorum` to 3 so automatic cleanup does not remove a required voter while the cluster is restoring zone balance. If a failed voter is the last remaining server in its zone, Autopilot does not remove it automatically.

Set `dead_server_last_contact_threshold` to the smallest value that still exceeds the longest Pod recovery time in your environment; 15m is a reasonable starting point when Pod recovery completes within minutes. This threshold determines when Autopilot removes a peer that has stopped sending heartbeats. If Autopilot removes a non-voter before its Pod recovers, the returning Pod cannot rejoin the cluster with its existing data. You must delete the Pod's PersistentVolumeClaim and allow it to join as a new member.

For Autopilot configuration and behavior, refer to the Autopilot documentation.

0.12 Vault Autopilot upgrades

We recommend Autopilot Upgrades combined with the StatefulSet OnDelete update strategy for Vault Enterprise version upgrades on OpenShift. Autopilot Upgrades treats the version transition as a controlled Raft membership change, promoting target-version candidates and demoting old-version voters in an order that preserves quorum and transfers leadership without an unplanned election. A manual upgrade that deletes standbys first and the leader last still forces a leader election when Kubernetes deletes the leader Pod, which causes a write outage during the election timeout window, and the manual flow has no automated check that the cluster can safely tolerate the next deletion. The recommended procedure differs between the 6-pod redundancy zones topology and the 5-pod fallback topology because each starts from a different voter and non-voter composition.

This section covers the recommended upgrade strategy, the procedure for each topology, and automation fundamentals. Dead server cleanup, described in the Dead server cleanup section, is not relevant during an upgrade because Pods rejoin with their existing PersistentVolumeClaims and the same Raft IDs.

0.12.1 Upgrade strategy and the scale-out anti-pattern

We recommend in-place Pod replacement under the OnDelete update strategy, which is the Vault Helm chart default. Update the Helm release with the target Vault image, and then delete Pods in the order specified by the topology-specific procedures in the following sections to trigger replacement on the target image. Each Pod retains its PersistentVolumeClaim and rejoins the cluster with the same Raft ID. Because the StatefulSet template change does not restart Pods, you control the deletion order.

Do not scale the StatefulSet out to add target-version Pods and then scale it back in. StatefulSet scale-down removes the highest-ordinal Pods first, and the Vault Helm chart sets `podManagementPolicy` to Parallel, so scale-down can happen concurrently. Scaling from 6 to 9 to add `vault-6`, `vault-7`, and `vault-8`, then scaling back to 6, removes the freshly upgraded Pods and leaves the original old-version Pods in place. The result is the opposite of the intended upgrade.

Direct `oc delete pod` bypasses the eviction API and therefore the chart's PodDisruptionBudget, described in the Pod disruption budgets section. The StatefulSet controller still recreates the deleted Pod from the current template, which now references the target Vault image.

When you enable redundancy zones, the chart's default PodDisruptionBudget permits one eviction at a time because `maxUnavailable` is 1. In the 5-pod fallback topology, the default `maxUnavailable` follows the formula `floor((n-1)/2)`, which permits two evictions for five replicas. Because these procedures delete one Pod at a time, automation must enforce pacing instead of relying on the PodDisruptionBudget.

0.12.2 Recommended upgrade procedure for the 6-pod redundancy zones topology

In the 6-pod redundancy zones topology, you upgrade the non-voter in each zone first, so target-version candidates are already in place before Autopilot changes any voter membership. The cluster maintains quorum because Autopilot keeps at least one voter in each zone and demotes an old-version voter only after a target-version voter can replace it. You do not need to change voter membership manually.

The procedure has two operator actions separated by an Autopilot-driven voter swap: replace the existing non-voters with target-version Pods, wait for Autopilot to swap voters, then replace the remaining old-version Pods.

1. Apply the Helm upgrade so the StatefulSet template uses the target Vault image. The On-Delete update strategy means no Pods restart yet.
2. Delete the non-voter Pods one at a time. Wait for each rescheduled Pod to return healthy on the target image before deleting the next. Deleting one at a time provides a per-Pod recovery checkpoint, preserves performance-standby read capacity during the upgrade, and avoids interaction with concurrent operations such as node drains. After this step, each redundancy zone holds one old-version voter and one target-version non-voter.
3. Wait for Autopilot to complete the voter swap. You do not act during this phase. Autopilot promotes the 3 target-version non-voters to voters, demotes 2 of the 3 old-version voters (keeping the leader for now), transfers leadership to a target-version voter, and then demotes the old leader.
4. Confirm the 3 remaining old-version Pods are now non-voters by reading `vault operator raft autopilot state`.
5. Delete the old-version non-voter Pods one at a time so they restart on the target image. Wait for each to return healthy on the target image before deleting the next.
6. Confirm Autopilot returns to `idle` and all Pods report the target version.

Throughout the procedure the cluster maintains quorum because non-voter deletions do not affect the voter count, and Autopilot keeps at least one voter per zone during promotion and demotion.

0.12.3 Upgrade procedure for the 5-pod fallback topology

The 5-pod cluster has no non-voters that can return on the target version before Autopilot changes voter membership. Each returning Pod rejoins as a voter using its existing PersistentVolumeClaim. Autopilot demotes the first returning voters back to non-voters until 3 target-version servers exist in the cluster, at which point Autopilot transitions to `promoting` and stops demoting target-version servers. Autopilot then promotes the target-version non-voters, demotes the old-version voters, and transfers leadership without manual voter changes. Delete Pods strictly one at a time and upgrade the leader last.

The procedure has two operator actions separated by an Autopilot-driven voter swap: replace 3 of the 4 standby Pods so Autopilot accumulates 3 target-version servers and starts the swap, wait for Autopilot to finish the swap, then replace the 2 remaining old-version Pods (the original leader and the one untouched standby).

1. Apply the Helm upgrade so the StatefulSet template uses the target Vault image. The On-

Delete update strategy means no Pods restart yet.

2. Delete 3 of the 4 standby Pods one at a time, leaving the leader and one standby for later. Wait for each Pod to return healthy on the target image before deleting the next. The first and second returning Pods rejoin as voters and Autopilot demotes them back to non-voters because the cluster is in `await-new-voters`. The third returning Pod also rejoins as a voter, but its arrival gives Autopilot 3 target-version servers against 2 old-version voters, which transitions the upgrade to `promoting`. That third Pod stays a voter, alongside the 2 stable target-version non-voters and the 2 remaining old-version voters.
3. Wait for Autopilot to complete the voter swap. You do not act during this phase. Autopilot promotes the 2 target-version non-voters to voters, demotes the remaining old-version voter that is not the leader, transfers leadership to a target-version voter, and then demotes the old leader.
4. Confirm the 2 remaining old-version Pods are now non-voters by reading `vault operator raft autopilot state`.
5. Delete the 2 old-version non-voter Pods one at a time so they restart on the target image. Wait for each to return healthy on the target image before deleting the next.
6. Confirm Autopilot returns to `idle` and all Pods report the target version.

A healthy 5-voter cluster tolerates 2 unavailable voters, but Autopilot's demotions during this procedure temporarily reduce the configured voter count to as few as 3, where the cluster tolerates only 1 unavailable voter. Delete only one Pod at a time, and wait until Autopilot reports the previous one as healthy on the target version before deleting the next.

0.12.4 Automating the upgrade procedure

The upgrade procedure has deterministic steps and repetitive verification, which makes it a good automation candidate. Two reasonable patterns exist, and both share the same fundamentals.

The first pattern is a pipeline-driven, manually triggered upgrade. A continuous-delivery pipeline runs the upgrade as a job. The existing GitOps flow applies the target Vault image to the Helm release. Because the StatefulSet uses OnDelete, the specification change does not restart Pods. The upgrade pipeline then performs the Pod deletions and Autopilot state checks. Per-cluster environment approvals provide the safety gate so that each cluster requires a separate approval before the pipeline acts on it.

For multi-cluster deployments with replication, upgrade downstream clusters before their upstream

sources, working one tier at a time so the primary upgrades last. This order prevents an upstream cluster from running a newer Vault version than its downstream replicas. For the full replication-aware upgrade ordering, refer to the Vault replicated deployment upgrade guide.

The second pattern is a dedicated OpenShift operator. A purpose-built operator watches a custom resource that describes the desired Vault image and orchestrates the same steps. The operator pattern offers tighter integration with OpenShift but requires more engineering investment to build and maintain.

Regardless of pattern, the automation must implement the following safety fundamentals:

- **Run a pre-flight health gate.** Before any Pod deletion, confirm `healthy` is true, `optimistic_failure_tolerance` is at least 1, the previously deleted Pod has returned with `version` equal to the target, and active replication is steady. Refuse to proceed if the pre-flight check fails.
- **Define an explicit starting order.** For a single cluster, follow the topology-specific procedure in the preceding sections. For a multi-cluster deployment with replication, follow the same downstream-before-upstream order described earlier in this section so the primary upgrades last.
- **Wait for stabilization between steps.** After each Pod replacement, wait until the replacement server appears in `/sys/storage/raft/autopilot/state` with `healthy` true, `version` equal to the target version, and `upgrade_info.status` either remaining in the expected transition state for the current step or advancing toward `idle`. Autopilot reports the upgrade through a sequence of transition statuses such as `await-new-voters`, `promoting`, `demoting`, `leader-transfer`, `await-new-servers`, and `await-server-removal` before returning to `idle`. Treat forward movement through these statuses as healthy progress, not failure.
- **Fail closed on safety violations.** If `healthy` becomes false, `optimistic_failure_tolerance` drops to 0, or the upgrade status regresses, stop and surface the problem. Do not delete the next Pod.

We recommend identifying Pods through Vault's Kubernetes service registration labels to simplify the automation. When the Vault configuration includes the `service_registration "kubernetes"{}` stanza (the chart's default), Vault applies the `vault-active`, `vault-sealed`, `vault-initialized`, `vault-perf-standby`, and `vault-version` labels to its own Pods. Reading these labels through the Kubernetes API retrieves Pod status without requiring Vault authentication. Use `vault-active=true` to find the leader and skip Pods where `vault-sealed=true` or `vault-initialized=false`. The labels do not distinguish voters from non-voters, so read voter status

from `vault operator raft autopilot state` when you need that detail. For label semantics, refer to the Vault Kubernetes service registration documentation.

For the full Autopilot state response schema, refer to the Autopilot state API reference. For Autopilot Upgrades concepts and prerequisites, refer to the Autopilot documentation.

0.13 Storage topology

Vault integrated storage uses PersistentVolumeClaims for Raft data persistence. The storage configuration affects cluster reliability, performance, and recovery capabilities. For integrated storage internals and Raft configuration parameters, refer to the Vault integrated storage documentation.

Each Vault Pod requires its own PVC with ReadWriteOnce (RWO) access mode. Integrated storage requires exclusive access to the data directory. Do not use ReadWriteMany (RWX) access modes, which allow multiple Pods to mount the same volume simultaneously.

Select a StorageClass that provides low latency and high IOPS. Raft consensus involves frequent disk writes for log entries and snapshots. Storage performance directly affects Raft election timeouts and cluster stability. For storage sizing and IOPS requirements, refer to Hardware sizing for Vault servers.

Storage requirements vary based on your workload mix. A write-heavy workload, such as high-volume KV secret updates or authentication at scale, produces significantly more storage I/O than a read-heavy workload because each write commits a Raft log entry to disk across a majority of voters. Slow disk on any single voter delays the entire Raft commit pipeline, and when storage cannot keep pace with the commit rate, followers miss heartbeat windows, triggering unnecessary leader elections and write unavailability. We recommend p99 write latency below 10 ms on `vault.raft-storage.put` and p99 read latency below 5 ms on `vault.raft-storage.get` for the storage backing Vault data PVCs. Use vault-benchmark to simulate your expected workload against a non-production cluster and validate that your StorageClass, CPU, and memory configuration meet performance requirements before deploying to production.

The following table outlines storage considerations across OpenShift variants.

Platform	Storage options	Considerations
Self-managed OpenShift (on-premises or cloud-hosted)	Local storage, storage area network (SAN), NFS	Configure zone topology if using distributed storage. Local storage provides lowest latency but complicates Pod rescheduling. We do not recommend NFS for Vault integrated storage because it does not provide the filesystem consistency guarantees integrated storage requires.
Red Hat OpenShift Service on AWS (ROSA), a Red Hat and AWS co-managed service	Amazon EBS (gp3, io1, io2)	The default `gp3` StorageClass is suitable. Consider `io1` or `io2` for high-throughput clusters.
Azure Red Hat OpenShift (ARO), a Red Hat and Azure co-managed service	Azure Managed Disks (Premium solid-state drive (SSD), Ultra Disk)	Premium SSD is suitable for most deployments. Ultra Disk provides lower latency for high-performance requirements.
OpenShift Dedicated (GCP or AWS), a Red Hat managed service	Cloud provider storage (varies by hosting platform)	Use the default StorageClass or request a high-performance class from your platform provider.

For foundational integrated storage architecture and design considerations, refer to Vault Enterprise Architecture.

0.14 Networking, routes, and TLS

This section covers OpenShift-specific networking considerations for Vault Enterprise, including external access patterns, TLS termination strategies, route configuration, network policy design, and cross-cluster replication connectivity.

Vault requires network connectivity for two traffic types: client API traffic on port 8200 and Raft cluster traffic on port 8201. Clients outside the cluster reach the API through an external access path, such as an OpenShift Route, a LoadBalancer Service, or an NGINX Ingress Controller. Each path has different requirements for TLS, load balancing, and access control.

0.15 Internal cluster networking

0.15.1 Raft peer communication

Vault Pods use port 8201/tcp for Raft consensus traffic (log replication, heartbeats, leader election). Within a cluster, port 8200/tcp (API) handles the one-time `retry_join` bootstrap handshake when a node first joins the cluster, but steady-state Raft communication occurs exclusively on port 8201 beyond this point. The headless Service `<release>-internal` provides a DNS record for each Pod, such as vault-0.vault-internal.vault.svc. Each Vault node advertises its cluster address using this name, and Raft peers communicate over these addresses on port 8201.

The `retry_join` stanza in the Raft storage configuration enables Vault Pods to discover peers and form the cluster automatically. Use `auto_join` with the Kubernetes provider (go-discover) for dynamic peer discovery instead of hardcoding individual Pod addresses. The following configuration uses the Kubernetes provider:

```
storage "raft" {
  path = "/vault/data"

  retry_join {
    auto_join             = "provider=k8s namespace=<namespace> label_selector=\"app.
        kubernetes.io/name=<chart-name>, component=server\""
    auto_join_scheme      = "https"
    leader_tls_servername = "vault.apps.example.com"
    leader_ca_cert_file   = "/vault/userconfig/vault-tls/ca.crt"
  }
}
```

Replace <namespace> with the target namespace and <chart-name> with the value of the app.kubernetes.io/name label on the Vault Pods. This label defaults to vault (the chart name) unless overridden through nameOverride in the Helm values.

Always set leader_tls_servername when using auto_join, and set it to a Subject Alternative Name (SAN) on the listener certificate, such as the external hostname. Auto-join discovers Pods by IP, and Pod IPs never appear as certificate SANs, so without this parameter TLS validation during Raft bootstrap fails. leader_tls_servername overrides the hostname used for that validation so the joining node checks against a SAN that actually exists on the certificate.

When embedding this in the Helm server.ha.raft.config value, use Helm template expressions so Helm resolves the values at render time:

```
auto_join = "provider=k8s namespace={{.Release.Namespace}} label_selector=\"app.
    kubernetes.io/name={{ template "vault.name" . }}, component=server\""
```

The quotes inside {{ }} above are Go template syntax processed by Helm before Helm parses the result as HashiCorp Configuration Language (HCL). After rendering, verify that the label_selector value remains valid HCL with escaped inner quotes, for example auto_join = "provider=k8s namespace=vault label_selector=\"app.kubernetes.io/name=vault, component=server\"". Keep a space after the comma in the rendered label_selector so the value parses correctly.

The auto_join approach uses the Kubernetes API to discover Vault Pods by label selector, eliminating the need to maintain one retry_join block per Pod. This works regardless of the replica count and adapts automatically when scaling the StatefulSet. When using the Vault Helm chart in HA mode, the Helm chart creates the required RBAC automatically: the chart's server-discovery-role.yaml and server-discovery-rolebinding.yaml templates grant the Vault ServiceAccount the get, watch, list, update, and patch verbs on Pods in the namespace. server.serviceAccount.serviceDiscovery.enabled controls this behavior, and it defaults to **true**. If you pre-provision the service account externally or disable server.serviceAccount.serviceDiscovery.enabled, create this Role and RoleBinding manually.

0.15.2 DNS requirements

Even though auto_join discovers peers by Pod IP through the Kubernetes API, each node advertises its cluster address as its stable DNS name in the <release>-internal headless Service,

and steady-state Raft communication on port 8201 targets those names. Broken DNS therefore prevents cluster formation, so verify resolution for all Vault Pods before initializing the cluster:

```
oc exec vault-0 -n vault -- nslookup vault-0.vault-internal.vault.svc
oc exec vault-0 -n vault -- nslookup vault-1.vault-internal.vault.svc
oc exec vault-0 -n vault -- nslookup vault-2.vault-internal.vault.svc
```

If DNS resolution fails, verify that the headless Service exists and its selector matches the Vault Pods. DNS resolution depends on the `openshift-dns` namespace being accessible from the Vault namespace through network policies.

0.16 External access patterns

OpenShift provides multiple mechanisms for exposing Vault to clients outside the cluster. The appropriate pattern depends on your platform variant, security requirements, and existing infrastructure.

0.16.1 OpenShift routes

OpenShift Routes are the platform-native mechanism for exposing HTTP and HTTPS services to external clients. The OpenShift router, based on HAProxy, runs as a set of Pods in the `openshift-ingress` namespace and watches for Route objects across all namespaces.

The Vault Helm chart creates a Route when you set `server.route.enabled` to **`true`**. Configure the Route hostname through `server.route.host`.

For Vault deployments, use **passthrough** TLS termination as the default. Passthrough preserves end-to-end, unbroken TLS between the client and the Vault listener, consistent with HashiCorp's standard guidance for Vault deployments. The router forwards raw TLS traffic to Vault without termination or inspection, and the client validates Vault's listener certificate directly.

Because the client sees the Vault listener certificate, that certificate **must include a SAN matching the external hostname** used in the Route. If the SAN does not match the hostname the client connects to, TLS validation fails. The certificate must also include the internal service DNS name so that in-cluster clients (Vault CLI, Vault Agent, or local application workloads) can validate the certificate when connecting through the Kubernetes service.

Example certificate SANs for a Vault deployment in namespace vault with hostname vault.apps.example.com:

```
Subject:    CN=vault.apps.example.com

X509v3 Subject Alternative Name:
    DNS:vault.apps.example.com ←          external clients via Route / LB
    DNS:vault.vault.svc ←                 in-cluster clients via ClusterIP Service
    DNS:vault.vault.svc.cluster.local ←   full internal DNS
    IP Address:127.0.0.1 ←                in-pod Vault CLI (VAULT_ADDR=https://
        127.0.0.1:8200)
```

The external SAN (vault.apps.example.com) enables passthrough route matching. The internal SANs (vault.vault.svc) enable in-cluster access without `VAULT_SKIP_VERIFY`. The 127.0.0.1 IP SAN is optional. Include it if the Helm chart sets `VAULT_ADDR` to `https://127.0.0.1:8200` (the default when you enable TLS).

```
server:
  route:
    enabled: true
    activeService: false
    host: vault.apps.example.com
    tls:
      termination: passthrough
```

Setting `server.route.activeService` to **`false`** routes traffic to the <release> Service, which targets all Vault Pods. With Vault Enterprise, performance standbys serve most read operations locally, so distributing traffic across all Pods improves read operation throughput.

Setting `server.route.activeService` to **`true`** sends Route traffic for port 8200 to the active Vault Pod through <release>-active. It does not expose port 8201 for replication traffic. For cluster replication communication on port 8201, provision a separate exposure mechanism, either a LoadBalancer Service (recommended) or a raw TCP passthrough Route as described in the cross-cluster networking for replication section.

Re-encrypt termination is a valid alternative when organizational policy requires the router to terminate TLS for all services, or when the router needs L7 HTTP visibility (for example, header-based routing or rate limiting). Re-encrypt terminates the client TLS session at the router and establishes a new TLS connection to the Vault backend. However, re-encrypt breaks the end-to-end TLS chain, which has two implications:

- **TLS certificates auth method**: Vault's cert auth method requires the client to present its certificate during the TLS handshake. With re-encrypt, the router terminates the client TLS session, so Vault never sees the client certificate. Use passthrough if your deployment

relies on TLS certificate authentication. Starting with Vault 1.17, the listener supports `x_forwarded_for_client_cert_header` to read the client certificate from an HTTP header forwarded by the proxy, which enables cert auth behind L7 proxies, provided the proxy forwards the certificate (for example, X-Forwarded-Client-Cert).

- **Router as trust boundary**: With re-encrypt, the router briefly holds the decrypted request in memory before re-encrypting it to the backend. The router becomes a trust boundary that you must secure accordingly.

The following table compares TLS termination strategies for Vault on OpenShift.

Strategy	How it works	Vault compatibility	When to use
Passthrough	Router forwards raw TLS traffic to Vault without termination. Client validates Vault's listener certificate directly. Passthrough preserves end-to-end TLS.	**Recommended (default)**. Consistent with HashiCorp's standard TLS guidance. Supports all Vault authentication methods, including TLS certificate authentication natively.	Default for production deployments. Required when end-to-end, unbroken TLS is a compliance requirement.
Re-encrypt	Router terminates client TLS, re-encrypts to backend using Vault's listener certificate. Router validates backend certificate against `destinationCACertificate`.	Supported. Breaks end-to-end TLS. TLS cert auth requires Vault 1.17+ with `x_forwarded_for_client_cert_header` and a proxy that forwards the client certificate.	Deployments where the external certificate authority differs from the internal certificate authority and a single certificate cannot satisfy both trust domains.

Strategy	How it works	Vault compatibility	When to use
Edge	Router terminates TLS and forwards plaintext HTTP to the backend.	Not recommended. The Vault listener must always serve TLS. Disabling TLS on the listener (`tls_disable = true`) removes encryption between the router and Vault Pod.	Not applicable for production Vault deployments.

When using passthrough termination, external clients must trust Vault's listener certificate. Use a certificate issued by a corporate PKI or public CA with a SAN that matches the external hostname. The `leader_tls_servername` setting allows you to also use this certificate for Raft peer validation. Refer to the Certificate trust and TLS section for additional details.

Route certificate configuration (re-encrypt only)

On OpenShift versions before 4.19, or when you do not use `.spec.tls.externalCertificate`, re-encrypt Routes require TLS certificates and private keys **inline** in the Route specification. On these versions, unlike Kubernetes Ingress resources that reference kubernetes.io/tls Secrets natively, Route resources cannot source certificates from Kubernetes Secrets. Passthrough routes do not use these fields, and in this case the Vault listener certificate handles client trust directly.

This API design creates a significant security concern for Vault deployments: when you embed certificates in Helm values, the private key appears in **clear text** in the `values.yaml` file. If you commit this file to a Git repository, the private key becomes visible to anyone with read access.

```
server:
  route:
    tls:
      termination: reencrypt
      # WARNING: These values contain sensitive key material in clear text.
      # Do NOT commit this file to Git with actual certificate content.
      certificate: |
        -----BEGIN CERTIFICATE-----
        <external certificate>
        -----END CERTIFICATE-----
      key: |
        -----BEGIN RSA PRIVATE KEY-----
        <private key - SENSITIVE>
        -----END RSA PRIVATE KEY-----
      caCertificate: |
        -----BEGIN CERTIFICATE-----
        <external CA certificate>
        -----END CERTIFICATE-----
      destinationCACertificate: |
        -----BEGIN CERTIFICATE-----
        <internal CA certificate for backend validation>
        -----END CERTIFICATE-----
```

Avoid storing private keys in version control. For OpenShift versions before 4.19, inject certificate content at deploy time with `--set-file`, for example `--set-file server.route.tls.key=./certs/tls.key`. This approach requires manual CLI execution and does not integrate directly with GitOps workflows. On OpenShift 4.19 and later, the `externalCertificate` field removes this limitation, as the following paragraph describes. For GitOps secret-handling patterns on earlier versions, refer to the GitOps fundamentals section.

OpenShift 4.16 introduced the RouteExternalCertificate feature gate to allow routes to reference a kubernetes.io/tls secret instead of embedding certificate content inline. As of OpenShift 4.19, this feature is generally available. The Vault Helm chart passes the entire `server.route.tls` object through to `spec.tls`, so you can supply `externalCertificate` through `server.route.tls.externalCertificate` in the Helm values. This eliminates the need for `--set-file` injection or GitOps workarounds when using re-encrypt routes on OpenShift 4.19+.

Route backend validation

When using re-encrypt termination, the `destinationCACertificate` field contains the CA certificate that the router uses to validate the Vault Pod's listener certificate. This field must contain the

CA certificate that signed the certificate Vault presents on port 8200.

Provide the CA certificate that signed the Vault listener certificate as the `destinationCACertificate`. For details on certificate strategies, refer to the Certificate trust and TLS section.

0.16.2 Gateway API (OpenShift 4.19+)

Gateway API is the Kubernetes-standard successor to Ingress. OpenShift 4.19 introduced Gateway API as generally available, supporting Gateway, HTTPRoute, and GRPCRoute resources through the Ingress Operator. Gateway API resources reference TLS certificates from secrets natively and offer stronger traffic management controls than Ingress.

The Vault Helm chart does not generate Gateway API resources. If your organization adopts Gateway API, create HTTPRoute and Gateway resources as supplementary manifests managed alongside the Helm release (for example through Kustomize overlays) and do not configure Ingress or Route resources as part of the Vault deployment.

0.16.3 External load balancers

Not all OpenShift deployments include cloud load balancer integration. On-premises and self-managed OpenShift clusters typically use F5 BIG-IP, MetalLB, HAProxy, or other external load balancers in front of the OpenShift router infrastructure.

When an external load balancer sits in front of the OpenShift router, configure it for **SSL passthrough** to the router Pods. The external load balancer forwards encrypted traffic to the OpenShift router without terminating TLS. The router then handles traffic according to the route's TLS termination strategy (passthrough or re-encrypt).

With **passthrough** (recommended), TLS flows end-to-end from the client to the Vault listener. The load balancer and router both forward the encrypted stream without termination while the client validates the Vault listener certificate directly.

With **re-encrypt**, TLS terminates in stages:

1. **Client to external load balancer**: SSL passthrough forwards encrypted traffic to the router.
2. **Router terminates and re-encrypts**: Router terminates the client TLS session and opens a new one to the Vault backend, validating the listener certificate against `destinationCACertificate`.

Configure the external load balancer backend pool to target the OpenShift router Pod IPs or node ports. The load balancer distributes traffic across available router Pods, which then route requests based on the Server Name Indication (SNI) hostname to the appropriate route.

0.16.4 F5 NGINX Ingress Controller

> **Note**
>
> This section covers the F5 NGINX Ingress Controller (VirtualServer CustomResourceDefinitions (CRDs), apiGroup k8s.nginx.org), not the Kubernetes community Ingress NGINX project, which retired in March 2026. The F5 controller is open source (Apache 2.0) and actively supported.

Some OpenShift deployments use the F5 NGINX Ingress Controller alongside, or instead of, the default HAProxy-based router. NGINX Ingress Controller uses VirtualServer custom resources instead of OpenShift Routes.

Criteria	OpenShift Route (HAProxy)	NGINX VirtualServer
Default on OpenShift	Yes. OpenShift deploys the HAProxy-based router out of the box.	No. Requires separate NGINX Ingress Controller installation.
Frontend TLS certificate	Inline in Route spec, or through `externalCertificate` referencing a kubernetes.io/tls Secret (OpenShift 4.19+). On older versions, requires `--set-file` injection or GitOps workarounds.	References kubernetes.io/tls Secrets natively. Private keys never appear in manifests or Helm values.

Criteria	OpenShift Route (HAProxy)	NGINX VirtualServer
Secret management	With `externalCertificate` (4.19+), certificates stay in Secrets protected by RBAC. On older versions, private keys appear in clear text in the Route object or values.yaml.	Certificates stored in Kubernetes Secrets, protected by RBAC. Compatible with cert-manager, Sealed Secrets, and External Secrets Operator without additional workarounds.
Backend TLS validation	`destinationCACertificate` field in Route spec (inline CA).	EgressMTLS Policy resource referencing a CA Secret.
Advanced L7 features	Basic path-based routing, host-based routing, rate limiting annotations.	WAF, advanced rate limiting, JSON Web Token (JWT) validation, header manipulation, circuit breaking.
When to use	Standard OpenShift deployments where the built-in router satisfies requirements. Most common for Vault deployments.	Deployments that require advanced traffic management, or where the organization standardizes on NGINX across Kubernetes platforms.

Use OpenShift routes when the default HAProxy-based router meets your requirements. Use NGINX VirtualServer when the cluster runs NGINX Ingress Controller and your organization requires advanced traffic management capabilities.

VirtualServer configuration

With re-encrypt termination, the ingress layer (OpenShift Route or NGINX VirtualServer) must trust the CA that signed Vault's listener certificate so it can validate the backend Pod. The ingress layer terminates the client TLS session and opens a new connection to the Vault backend, so it needs the listener CA to verify the backend Pod's identity. Passthrough termination forwards the raw TLS stream without inspection, so no backend trust configuration applies. Configure backend trust for re-encrypt as follows:

- **OpenShift Route (re-encrypt):** The `destinationCACertificate` field contains the listener CA certificate inline in the Route spec.
- **NGINX VirtualServer (re-encrypt):** An EgressMTLS Policy references the listener CA certificate from a Kubernetes Secret.

Without backend CA validation configured, the ingress layer connects to Vault over TLS but does not verify the identity of the backend Pod.

```
apiVersion: k8s.nginx.org/v1
kind: VirtualServer
metadata:
  name: vault
  namespace: vault
spec:
  host: vault.example.com
  tls:
    secret: vault-external-tls
  policies:
  - name: vault-egress-mtls
  upstreams:
  - name: vault
    service: vault
    port: 8200
    tls:
      enable: true
  routes:
  - path: /
    action:
      pass: vault
---
apiVersion: k8s.nginx.org/v1
kind: Policy
metadata:
  name: vault-egress-mtls
  namespace: vault
spec:
  egressMTLS:
    trustedCertSecret: service-ca-nginx
    verifyServer: true
    verifyDepth: 2
    serverName: vault.vault.svc
```

Configure `tls.secret` to reference the external TLS certificate, `upstreams[].tls.enable: true` to enable backend TLS, `trustedCertSecret` to validate the backend certificate against the listener

CA, and `verifyServer: true` to enforce validation. The `serverName` field overrides the hostname for backend TLS validation, matching the SANs on the listener certificate. For all available options, see the VirtualServer and VirtualServerRoute resource specification.

The trustedCertSecret references an OpenShift Container Platform (OCP) secret containing the CA that signed the listener certificate. If the CA is only available in a ConfigMap (for example, when using service-ca), synchronize the CA data into a secret using Kustomize secretGenerator, an init container, or a CronJob. For detailed GitOps automation patterns, refer to the GitOps fundamentals section.

VirtualServer's native secret references provide a significant advantage for GitOps workflows: cert-manager creates and renews the external certificate secret automatically, and the VirtualServer references it by name. No certificate content appears in Git, and you need no additional secrets management tooling for the external certificate.

0.16.5 Client IP preservation

Client IP preservation and TLS termination are independent concerns. TLS termination determines where the encrypted session ends, as covered in the preceding sections. Client IP preservation determines whether Vault sees the real client address or the proxy's address. However, the TLS termination choice constrains which client IP mechanism is available.

Without client IP preservation, Vault audit logs record the router or load balancer Pod IP as `remote_address`, and IP-based policy constraints (`token_bound_cidrs`, `bound_cidrs`) cannot function correctly.

Proxy layer	Mechanism	Vault listener parameters
L7 (proxy terminates TLS)	X-Forwarded-For HTTP header, the proxy injects the client IP into the request header before forwarding to Vault.	`x_forwarded_for_authorized_addrs` set to the proxy Pod Classless Inter-Domain Routing (CIDR).

Proxy layer	Mechanism	Vault listener parameters
L4 (direct LB to Vault, bypassing router)	PROXY protocol, the LB prepends a protocol header containing the client IP to the TCP stream. Does **not** work with passthrough routes (the OpenShift router does not send PROXY protocol to backends).	`proxy_protocol_behavior` and `proxy_protocol_authorized_addrs` set to the proxy Pod CIDR.

L7: X-Forwarded-For

When the proxy terminates TLS (re-encrypt route, NGINX VirtualServer), it has HTTP visibility and injects the X-Forwarded-For header. Configure Vault to trust this header from the specific proxy IPs:

```
listener "tcp" {
  address       = "[::]:8200"
  tls_cert_file = "/vault/userconfig/vault-tls/tls.crt"
  tls_key_file  = "/vault/userconfig/vault-tls/tls.key"

  x_forwarded_for_authorized_addrs     = ["<proxy-ip-1>/32", "<proxy-ip-2>/32"]
  x_forwarded_for_hop_skips            = 0
  x_forwarded_for_reject_not_present   = true
  x_forwarded_for_reject_not_authorized = true
}
```

Set `x_forwarded_for_authorized_addrs` to the IPs of the router or ingress controller Pods that send the X-Forwarded-For header. Scope this as narrowly as possible — trusting a broad CIDR (for example, the entire pod network) allows any Pod in that range to spoof client IPs. When Vault receives a request from an authorized address, it reads the client IP from the header and records it as `remote_address` in the audit log.

Impact on internal Pod traffic: Internal clients (application workloads, Vault Agent, Vault Secrets Operator (VSO)) typically connect directly to the Vault service without passing through the ingress layer. These direct connections do not carry an X-Forwarded-For header and do not originate from an authorized proxy address, so the X-Forwarded-For (XFF) configuration does not affect them. In this

case, Vault uses the connection source IP as `remote_address`. However, if a service mesh or HTTP client library in the cluster automatically injects X-Forwarded-For headers on direct connections, `x_forwarded_for_reject_not_authorized = true` rejects those requests because the source is not in the authorized list. If this occurs, either suppress the header on the client side, add the relevant source CIDR ranges to the authorized list, or set `x_forwarded_for_reject_not_authorized` to `false` (which allows the request but ignores the untrusted header).

L4: PROXY protocol

When using a direct L4 path to Vault (for example, a LoadBalancer service that bypasses the OpenShift router), you cannot inject HTTP headers. To preserve the client IP at L4, use the PROXY protocol (v1 or v2), which Vault supports natively.

> **Warning**
>
> The OpenShift HAProxy router does not forward PROXY protocol headers to backend Pods for passthrough routes, so the backend cannot observe the original client IP. PROXY protocol support on the Ingress Controller applies only to the external load balancer-to-router connection, not router-to-backend. To use PROXY protocol with Vault, configure the L4 load balancer to connect directly to Vault, bypassing the router.

```
listener "tcp" {
  address       = "[::]:8200"
  tls_cert_file = "/vault/userconfig/vault-tls/tls.crt"
  tls_key_file  = "/vault/userconfig/vault-tls/tls.key"

  proxy_protocol_behavior        = "allow_authorized"
  proxy_protocol_authorized_addrs = ["<proxy-ip-1>/32", "<proxy-ip-2>/32"]
}
```

`proxy_protocol_behavior` value	Behavior
`allow_authorized`	Use the client IP from PROXY protocol if the source is in the authorized list. Use the connection source IP otherwise.
`deny_unauthorized`	Reject connections without a valid PROXY protocol header from an authorized source.

`proxy_protocol_behavior` value	Behavior
`use_always`	Always use the client IP from PROXY protocol, regardless of source.

PROXY protocol requires end-to-end support across the entire proxy chain. Configure every component in the path (external LB, OpenShift router, Vault listener) consistently. Not all L4 products support PROXY protocol. Enabling PROXY protocol on Vault without a corresponding upstream configuration causes connection failures.

For a complete reference, see the Vault TCP listener documentation.

0.17 Network policies

OpenShift supports Kubernetes NetworkPolicy resources for controlling traffic to and from Vault Pods. In environments where the default network policy is deny-all, you must explicitly allow the traffic categories that Vault requires.

0.17.1 Required network policies

The following table describes the network policies required for a Vault deployment.

Policy	Direction	Purpose	Source/Destination
Intra-namespace	Ingress + Egress	Raft peer communication between Vault Pods on ports 8200/tcp and 8201/tcp	Same namespace
DNS	Egress	DNS resolution for Service names and Raft peer cluster addresses	`openshift-dns` namespace, ports 5353/tcp and 5353/udp

Policy	Direction	Purpose	Source/Destination
Ingress controller	Ingress	External API traffic from the router or ingress controller to Vault Pods on port 8200	OpenShift router namespace (`openshift-ingress`) or NGINX namespace
Kubernetes API	Egress	Service registration (vault-active label updates), `auto_join` Pod discovery, and Kubernetes auth method	Kubernetes API server, port 6443
Monitoring	Ingress	Prometheus metrics scraping	`openshift-monitoring` namespace

0.17.2 Additional egress policies

Configure additional egress policies for external services that Vault connects to. The following are common examples. Your deployment may require additional policies depending on the auth methods, secrets engines, and audit devices in use:

- **LDAP/LDAPS**: Port 636 (LDAPS) or 389 (LDAP) to directory servers for authentication if used.
- **External KMS**: Ports required by your key management service for auto-unseal (for example, AWS KMS, Azure Key Vault, GCP Cloud KMS, or a Public Key Cryptography Standards (PKCS)#11 hardware security module (HSM)).
- **Replication targets**: Ports 8200/tcp and 8201/tcp to secondary Vault clusters for DR or performance replication.
- **Audit log destinations**: Ports required by Syslog or socket audit devices, if used.

When network policies are in place, timeouts and connection failures are the first symptoms of a missing policy. Verify network policies whenever you encounter connectivity issues between Vault and external services.

For ingress from the OpenShift router, create a NetworkPolicy that allows traffic from the `openshift-ingress` namespace (label `policy-group.network.openshift.io/ingress: ""`) to Vault Pods. Adapt the same pattern for NGINX Ingress Controller by targeting its namespace instead.

0.18 Cross-cluster networking for replication

Multi-cluster Vault deployments using disaster recovery or performance replication require network connectivity between clusters on ports 8200/tcp (API) and 8201/tcp (cluster). Replication traffic must reach the active Vault node in each cluster, so use the `<release>-active` Service for replication endpoints.

0.18.1 Connectivity requirements

Traffic	Port	Direction		Service target
Replication setup	8200	Secondary	Primary	`<release>-active` on the primary cluster. The secondary initiates the connection to the primary's API endpoint to establish the replication relationship.
Replication stream	8201	Secondary	Primary	`<release>-active` on the primary cluster. The secondary maintains a persistent connection to the primary's cluster port for receiving replicated data.

Within the cluster, the <release> and <release>-active Services both expose port 8201/tcp, and intra-cluster Raft communication needs no additional configuration. For **cross-cluster replication**, port 8201 has no external exposure by default: the Route created by `server.route.enabled` maps only to port 8200/tcp, and those Services default to ClusterIP, so neither reaches port 8201 from outside the cluster. You must provision a separate external exposure mechanism for port 8201 as a supplementary manifest, for example a LoadBalancer Service or TCP passthrough route managed through Kustomize overlays.

Vault's cluster port (8201) uses gRPC over HTTP/2. The mechanism used to expose the <release>-active service must support HTTP/2 passthrough without protocol downgrade or inspection.

Use a LoadBalancer Service or raw TCP passthrough for multi-cluster replication connectivity. A LoadBalancer Service associated with the <release>-active Service passes traffic directly to the active Pod without protocol inspection or termination. Configure the load balancer in L4/TCP mode. Any provider that integrates with the Kubernetes LoadBalancer Service type is sufficient, such as a cloud native load balancer, F5 BIG-IP, or MetalLB.

Pattern	HTTP/2 support	Considerations
LoadBalancer Service (recommended)	Full. L4 passthrough, no protocol inspection.	Requires cloud load balancer integration or MetalLB. Exposes ports 8200/tcp and 8201/tcp directly.
OpenShift Route with passthrough	Requires verification. Configure the HAProxy-based router to support HTTP/2 passthrough on the cluster port.	Route operates on port 443, requiring port mapping from 443 to 8201. Verify HTTP/2 passthrough behavior with your router configuration.

Vault uses its own internal CA for cluster replication traffic on port 8201. Vault establishes certificate trust between primary and secondary clusters automatically during the replication setup process.

Ensure that network policies on both clusters allow egress to the remote cluster and ingress from the remote cluster on ports 8200/tcp and 8201/tcp.

0.18.2 Network latency requirements

Network latency between availability zones must be less than 8 ms for Raft peers within a single cluster (Raft reference architecture). For replication between clusters, latency requirements are less strict because replication operates asynchronously, but sustained or excessive latency increases replication lag, which can lead to unpredictable application failure modes.

0.19 Certificate trust and TLS

Vault on OpenShift relies on TLS for both client API access and Raft peer communication, so certificate design determines how clients and cluster members establish trust. This section recommends a single-certificate architecture, then covers backend certificate options for re-encrypt, volume configuration, provisioning and rotation, and a usage matrix that maps each connection path to its certificate. The guidance applies whether you expose Vault through OpenShift Routes or an NGINX-based ingress.

0.19.1 Recommended: single certificate from a trusted CA

The simplest and recommended TLS architecture for Vault on OpenShift uses a **single certificate** issued by a trusted CA (corporate PKI or public CA), with a SAN matching the external hostname. Configure the Vault listener with this certificate and use passthrough termination (or an L4 load balancer) for external access. You need no additional certificate infrastructure.

With `leader_tls_servername` set to the same external hostname in the `retry_join` configuration (see Raft peer communication), this single certificate covers both external client access and the Raft bootstrap handshake.

```
listener "tcp" {
  address         = "[::]:8200"
  cluster_address = "[::]:8201"
  tls_cert_file   = "/vault/userconfig/vault-tls/tls.crt"
  tls_key_file    = "/vault/userconfig/vault-tls/tls.key"
  tls_min_version = "tls12"
}
```

Each profile in the reference Helm values for this HVD implements this single-certificate pattern end to end, pairing a passthrough Route with the listener certificate paths above and a `leader_tls_servername` set to a hostname SAN on that certificate.

0.19.2 Related guidance

The **OpenShift routes** and **External load balancers** sections in the Networking, routes, and TLS section cover passthrough and re-encrypt behavior, certificate requirements, and external load balancers in front of the OpenShift router. The **Raft peer communication** section in Internal cluster networking covers port 8200 vs 8201 (API and `retry_join` bootstrap versus steady-state Raft and internal cluster TLS).

0.19.3 Backend certificate options for re-encrypt

When using re-encrypt, the Vault listener needs a backend certificate that the router can validate. The certificate source is independent of the recommendation above, use whatever fits your environment:

- cert-manager Operator for Red Hat OpenShift — the preferred option. Supports configurable renewal windows, external and internal CAs, and integrates with the Kubernetes secret lifecycle.
- **OpenShift service-ca operator** — a built-in option for minimal or greenfield setups where cert-manager is not yet available. The Vault Helm chart automates service-ca integration through `server.serviceCA.enabled` v0.32.0+. Service-ca has operational limitations: Vault does not auto-reload certificates (renewal requires a Pod restart), and the renewal window is neither configurable nor documented.

Regardless of the tool used, the certificate requirements are the same: SANs matching the Service DNS name (for example, `vault.<namespace>.svc`), and a CA certificate available for `destinationCACertificate` (Route) or `trustedCertSecret` (NGINX VirtualServer). Do not use service-ca for passthrough or L4 deployments. Its certificates use an internal CA that external clients do not trust.

0.19.4 Volume configuration

Vault Pods require the TLS certificate, private key, and CA certificate for the listener. Store the certificate and key in a kubernetes.io/tls secret (tls.crt, tls.key). If cert-manager includes the CA certificate in the same Secret (cert-manager populates ca.crt automatically), a single volume is sufficient.

```
server:
  extraEnvironmentVars:
    VAULT_CACERT: /vault/userconfig/vault-tls/ca.crt

  volumes:
    - name: vault-tls
      secret:
        secretName: vault-tls
        defaultMode: 0400

  volumeMounts:
    - name: vault-tls
      mountPath: /vault/userconfig/vault-tls
      readOnly: true
```

`VAULT_CACERT` points to the CA certificate that signed the listener certificate, so the in-pod Vault CLI trusts the listener. `leader_ca_cert_file` in the Raft configuration also points to this CA. If the CA certificate comes from a separate source (for example, a ConfigMap), mount it as an additional volume and adjust the paths accordingly.

0.19.5 Certificate provisioning with cert-manager

cert-manager automates the issuance and renewal of certificates from your PKI solution. With passthrough, this is the only certificate needed, as it serves both external clients and Raft peers. With re-encrypt, cert-manager can also manage the route's frontend certificate.

The integration requires two resources: an **Issuer** (or ClusterIssuer) that defines how cert-manager connects to the PKI, and a **Certificate** that describes the certificate to request.

Issuer vs ClusterIssuer: An Issuer is namespace-scoped and can only issue certificates within its own namespace. A ClusterIssuer is cluster-scoped and can issue certificates in any namespace. Use a namespace-scoped Issuer when you need to isolate PKI credentials to the Vault namespace.

The following example shows an Automatic Certificate Management Environment (ACME)-based Issuer that connects to a corporate PKI server using External Account Binding (EAB) for authentication and a DNS01 webhook solver for domain validation:

```yaml
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: corporate-pki-issuer
  namespace: vault
  annotations:
    argocd.argoproj.io/sync-wave: "-6"
spec:
  acme:
    server: https://pki.example.com/acme/v2/directory
    caBundle: <base64-encoded-corporate-root-ca>
    externalAccountBinding:
      keyID: <eab-key-id>
      keySecretRef:
        key: hmac
        name: pki-eab-secret
    privateKeySecretRef:
      name: cert-manager-solver-account
    solvers:
    - dns01:
        webhook:
          groupName: acme.example.com
          solverName: cert-manager-dns-solver
```

The Certificate resource describes the certificate to request. cert-manager watches for Certificate resources, requests the certificate from the configured Issuer, and stores the result in a Kubernetes Secret.

```yaml
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: vault-tls
  namespace: vault
  annotations:
    argocd.argoproj.io/sync-wave: "-5"
spec:
  commonName: vault.apps.example.com
  dnsNames:
  - vault.apps.example.com
  isCA: false
  issuerRef:
    kind: Issuer
    name: corporate-pki-issuer
  secretName: vault-tls
  duration: 2160h
  renewBefore: 360h
  subject:
    countries:
    - <country-code>
    localities:
    - <city>
    organizationalUnits:
    - <organizational-unit>
    organizations:
    - <organization>
    provinces:
    - <state-or-province>
  usages:
  - server auth
  - digital signature
  - key encipherment
  privateKey:
    algorithm: RSA
    size: 4096
```

Using the certificate

With **passthrough** (recommended), mount the cert-manager Secret (vault-tls) directly into the vault Pod as the listener certificate. You need no certificate extraction, as the Vault listener serves the certificate to both external clients and Raft peers.

With **re-encrypt** and **OpenShift Routes**, on OpenShift 4.19+ use `server.route.tls.externalCertificate` in the Helm values to reference the cert-manager Secret by name,

and you need no certificate extraction or `--set-file` injection. The Helm chart passes the entire `server.route.tls` object through to `spec.tls`, so the `externalCertificate` field is available directly. On older versions, extract the certificate from the secret and inject it into the Helm values at deploy time using `--set-file` flags (for example, `--set-file server.route.tls.certificate`, `--set-file server.route.tls.key`). For **NGINX VirtualServer**, reference the Secret by name in the VirtualServer tls.secret field. You need no extraction.

For detailed cert-manager usage instructions covering a variety of supported issuer types, see the official documentation.

Manual certificate provisioning

When cert-manager is not available, obtain certificates from your PKI team and inject them into the Helm values using the same `--set-file` pattern. Track certificate expiry dates and plan renewals manually.

0.19.6 Certificate rotation

Regardless of certificate source (corporate PKI, cert-manager, service-ca, or manual provisioning), Vault does not automatically reload certificates from disk. Vault continues serving the old certificate from memory until you reload the application. Therefore, perform a **rolling restart** of the Vault Pods so each Pod loads the new certificate. Monitor the certificate notAfter date and schedule restarts before expiry as you would for any other application that relies on a valid certificate.

If using passthrough, the Route resource needs no specific action after a rotation because the Pod presents the listener certificate directly. With re-encrypt on OpenShift 4.19 and later, the Route references the certificate Secret through `externalCertificate` and also needs no action after rotation. With inline re-encrypt certificates on earlier versions, re-run the Helm upgrade with `--set-file` flags or resync the Argo CD Application.

Source-specific considerations

Certificate source	Renewal mechanism	Operational window
cert-manager	Automatic, based on renewBefore (for example, 15 days before expiry). cert-manager updates the Secret in place.	Predictable and configurable: days or weeks to schedule a Pod restart.
Manual (corporate PKI)	Manual; coordinate with your PKI team. Replace the secret contents and reschedule the Vault Pods before expiry.	Depends on your PKI team's process. Track notAfter proactively.
Service-ca	Automatic, but the renewal threshold is neither configurable nor documented.	Undocumented and not configurable. Plan a controlled Pod restart before expiry rather than automating around the renewal timing.

0.19.7 Certificate usage matrix

The following table summarizes which certificate each connection path uses.

Connection path	Certificate presented	Validated by	Validation method
External client Route (passthrough)	Vault listener certificate	Client browser or API consumer	CA trust store
External client Route (re-encrypt)	Route frontend certificate	Client browser or API consumer	CA trust store
Router Vault Pod (re-encrypt)	Vault listener certificate	OpenShift router	`destinationCACertificate` in Route spec
External client VirtualServer	VirtualServer tls.secret certificate	Client browser or API consumer	CA trust store

Connection path	Certificate presented	Validated by	Validation method
VirtualServer → Vault Pod	Vault listener certificate	NGINX	EgressMTLS Policy with trustedCertSecret
Raft bootstrap → Vault Pod (port 8200)	Vault listener certificate	Raft client (joining Vault Pod)	`leader_ca_cert_file` + `leader_tls_servername`
Steady-state Raft (port 8201)	Vault internally-generated cluster TLS	Vault	Automatic, managed by Vault internally
Vault CLI (in-pod) → Vault	Vault listener certificate	Vault CLI	`VAULT_CACERT` environment variable

0.19.8 Deployment sequence

The following sequence orders certificate provisioning, deployment, and verification so that TLS material exists before Vault Pods start.

1. **Provision the TLS certificate**: Request a certificate from your CA (through cert-manager or manually) with SANs matching the external hostname and internal service DNS name. Store the certificate, private key, and CA certificate in a kubernetes.io/tls secret in the Vault namespace.
2. **Deploy Vault with Helm**: Run `helm install` with the listener configured to use the certificate secret. Configure the route (passthrough or re-encrypt) with the appropriate hostname.
3. **Initialize and unseal**: Once Pods are running, initialize the Vault cluster and unseal each. Raft peers join automatically through `retry_join` with `leader_tls_servername` set to a SAN on the listener certificate.
4. **Verify**: Confirm Raft cluster formation, route accessibility, and certificate chain validation from external clients.

0.20 Vault seal/unseal and external key management

We recommend enabling Vault auto-unseal for deployments on OpenShift. Auto-unseal allows Vault Pods to automatically enter service after restart or rescheduling without requiring manual intervention, simplifying operational workflow and improving cluster recovery characteristics.

You can implement auto-unseal using either a cloud Key Management Service (KMS) or a supported Hardware Security Module (HSM) that exposes a Public Key Cryptography Standards (PKCS)#11 interface.

The choice between KMS and HSM is typically driven by organizational security and compliance requirements. In OpenShift environments running on public cloud infrastructure, a cloud KMS is often the most straightforward option due to native integration and simplified operational management.

When using either KMS or HSM, Vault must authenticate to the device to perform seal and unseal operations. Multiple credential delivery mechanisms are possible, but we recommend avoiding static credentials and instead using Pod or workload identity wherever the platform supports it. The Authentication to external systems section covers this design consideration.

For each seal strategy wired into a complete server configuration, refer to the reference Helm values for this HVD: `values-awskms-rz.yaml` for cloud KMS auto-unseal through workload identity, and `values-hsm-rz.yaml` for PKCS#11 HSM auto-unseal using the init-container library delivery pattern.

0.20.1 HSM client library delivery to Vault Pods

HSM-backed auto-unseal on OpenShift requires the vendor's PKCS#11 shared library inside the Vault container before Vault starts. The delivery pattern you choose affects update cadence, image ownership boundaries, and storage dependencies across environments. This section compares three common patterns.

For seal stanza configuration, refer to the Vault Enterprise detailed design. For PKCS#11 seal reference documentation, refer to PKCS#11 seal.

Each pattern places the library in the container through a different mechanism, which determines who owns updates and how a changed library reaches running Pods. The three patterns differ mainly in how tightly the library couples to the Vault image:

- **Custom Vault image**: Embeds the PKCS#11 library directly in the Vault image you run.
- **Init container plus `emptyDir`**: Copies the library from a separate artifact image into a shared `emptyDir` that the Vault container mounts.
- **Pre-provisioned shared volume**: Stores the library on ReadWriteMany (RWX) or ReadOnlyMany (ROX) shared storage that every Vault Pod mounts read-only.

The following table compares these patterns across update cadence, reproducibility, rollback, failure mode, supply-chain integrity, and runtime dependencies so you can select the one that fits your environment.

Dimension	Custom Vault image	Init container plus `emptyDir`	Pre-provisioned shared volume
Library update cadence and coupling to the Vault image	Coupled: a library change requires a new image build and tag.	Decoupled: the library ships as a separate digest-pinned artifact image on its own cadence.	Decoupled: volume updates change the library without a Vault image change.
Reproducibility and GitOps	The image digest fully describes the running library.	Reproducible when the artifact image is digest-pinned; the Vault digest plus the artifact digest describe the runtime.	The image digest does not describe the library; identical digests can run different library versions depending on volume contents.

Dimension	Custom Vault image	Init container plus `emptyDir`	Pre-provisioned shared volume
Rollback model	Revert the Vault image reference in your Helm values, then replace Pods.	Revert the artifact image reference in your Helm values, then replace Pods.	Restore the previous library version on the volume, then replace Pods. In-place overwrites live outside the manifest and GitOps does not track them; pinning an immutable version path keeps the rollback in Git.
Failure mode and affected scope	A faulty library reaches Pods only when you replace Pods in a controlled order.	A faulty library reaches a Pod only when you recreate it, the same as the custom image; a bad artifact surfaces as an init-container failure that blocks that single Pod's startup, so you catch it during a controlled rollout.	An in-place overwrite or backend outage affects every Pod at once. Publishing immutable, versioned libraries and pinning the version path in the manifest narrows the scope to recreated Pods, but never matches the pull-time integrity of a signed image.

Dimension	Custom Vault image	Init container plus `emptyDir`	Pre-provisioned shared volume
External runtime dependency	None at runtime; the registry is only a pull-time dependency, as with any image.	None at steady-state runtime, only at Pod startup. The init container pulls the artifact image and copies the library into a local `emptyDir` at the same point the Vault image is pulled, so it adds a second pull-time registry dependency but no ongoing one; once the Pod is running the library sits on node-local storage and the registry can be unreachable without affecting it. The pull recurs on each Pod recreation, exactly as for the Vault image.	Startup and runtime dependency; the ReadWriteMany or ReadOnlyMany backend must stay reachable from every zone, or Pods cannot start. If the backend fails while Pods are running, those Pods can hang waiting on the volume, and some vendor clients read from it throughout operation, not just at startup.

Dimension	Custom Vault image	Init container plus `emptyDir`	Pre-provisioned shared volume
Supply-chain integrity and verification	Packaged as an Open Container Initiative (OCI) image, the library falls under whatever supply-chain controls you already apply to Vault images, such as signing, scanning, and admission-time verification.	A second OCI image under the same controls; apply the signing, scanning, and verification you use for the Vault image.	The volume sits outside the image supply-chain path, so those controls do not apply. Any principal with write access to the share can replace the library, which calls for compensating controls: restricted write role-based access control (RBAC) on the publishing path, a read-only mount, and checksum verification.

Dimension	Custom Vault image	Init container plus `emptyDir`	Pre-provisioned shared volume
Best fit when	Your team already owns the Vault image build pipeline and accepts an image rebuild on every library change and every Vault upgrade.	A separate team owns and releases the HSM client library on its own cadence and you want it decoupled from the Vault image: you run the stock `vault-enterprise:<version>-ent.hsm-ubi` image, so a Vault upgrade needs no library rebuild; you pin the artifact image by digest for GitOps reproducibility; the artifact inherits the same OCI supply-chain controls (signing, scanning, admission-time verification) as the Vault image; you can roll the library forward or back by changing one digest without touching the Vault image; and you avoid owning a custom Vault image build pipeline.	You must keep custom image builds out of the Vault upgrade path, or a ReadWriteMany or ReadOnlyMany delivery standard already exists in your environment.

We recommend the init container with a digest-pinned artifact image as the default delivery pattern. Choose the custom image or the shared volume only where your environment matches the conditions in the table above. For the shared volume, give each library version its own immutable path instead of overwriting the library in place.

The Vault Helm chart supports the init container and shared-volume patterns through `server.extraInitContainers`, `server.volumes`, and `server.volumeMounts` without a chart fork. Mount the delivered library read-only into the Vault container. For the shared-volume pattern, use a backend that supports multi-node access across every zone where Vault schedules Pods: ReadWriteMany for in-place updates, or ReadOnlyMany for an immutable, pre-published volume. ReadWriteOnce cannot span nodes. Seed and update the volume with a separate publishing step, most commonly a Kubernetes Job that mounts the volume, writing each library version to its own immutable path and verifying its checksum.

The chart defaults to the OnDelete update strategy, so StatefulSet template changes, such as image references, take effect only when you replace Pods in a controlled order. The update strategy does not control in-place shared-volume changes, so the changed library reaches any Pod that restarts.

The following guidance applies to all three patterns:

- Use the glibc-based Red Hat Universal Base Image (UBI) `hashicorp/vault-enterprise:<version>-ent.hsm-ubi` image, which matches the runtime that most vendor PKCS#11 libraries target. Confirm with your HSM vendor that their library supports this image.
- Store client certificates, private keys, partition credentials, HSM personal identification numbers (PINs), and other authentication material in Kubernetes Secret objects or an approved external secret workflow. Inject `VAULT_HSM_PIN` through `server.extraSecretEnvironmentVars`, sourced from a Kubernetes Secret, so credentials never render into Helm values, the StatefulSet manifest, a ConfigMap, an artifact image, or a shared volume. Use ConfigMaps only for non-sensitive client configuration such as `Chrystoki.conf` for Thales Luna.
- HSM reachability is a Vault startup dependency regardless of delivery pattern: an unreachable HSM prevents Pods from unsealing. Provide a reachable HSM, typically as a high-availability group, for each primary and disaster recovery cluster.

0.21 Authentication to external systems

Depending on the features enabled, Vault may frequently need to authenticate to external systems to deliver core functionality. Examples include cloud APIs used by various secrets engines, external identity providers used by authentication methods, telemetry or storage integrations, and key management systems used for auto-unseal.

In OpenShift environments, external integrations introduce an important design consideration: how Vault obtains and manages the credentials required to access those dependencies. Use the platform's native workload or Pod identity mechanism instead of distributing static credentials to Vault Pods or Vault plugin configurations. Major cloud platforms support various forms of Pod identity, such as GCP cluster workload identity federation and AWS IAM Roles for Service Accounts, all of which are compatible with Vault.

0.22 Cluster initialization and bootstrap

Vault initialization on OpenShift uses the standard `vault operator init` (or `sys/init`) workflow. Operators commonly run the command from within a Vault Pod with `oc exec` when the service is not yet externally reachable, but the procedure itself is identical to other Vault deployments.

```
$ oc exec -n vault -it vault-0 -- /bin/sh
$ vault operator init
```

After initialization, Vault returns a root token and recovery (or unseal) keys to the operator. Handle this data as you would in any other Vault initialization scenario. See the official documentation for further details.

0.23 GitOps fundamentals

This section covers design considerations for deploying and managing Vault Enterprise on OpenShift through GitOps and Helm-based workflows. The first part describes platform-agnostic patterns that apply regardless of tool choice. The second addresses Argo CD design decisions specific to Vault on OpenShift.

For an introduction to Helm and Argo CD as deployment tooling, refer to the Deployment options section in this guide.

0.23.1 Vault GitOps design patterns

These patterns apply to any declarative delivery tool, such as Argo CD, Flux, or a CI/CD pipeline that renders and applies Helm templates.

Vault is not a typical GitOps workload. As a stateful, security-critical service, it introduces constraints that standard application deployments do not face:

- **Ordering dependencies**: TLS certificates, CA bundles, and PKI resources must exist before the Vault Helm release deploys. A standard sync submits all resources at once. Vault Pods crash-loop if the certificate secret does not yet exist.
- **Sensitive values**: Helm values may reference TLS private keys, license keys, and seal credentials. These must never appear in Git in plaintext.
- **StatefulSet behavior**: Sync and pruning operations interact with StatefulSet ordering guarantees. A misconfigured prune policy can delete a PersistentVolumeClaim (PVC) mid-upgrade, breaking the Raft quorum.
- **Operator-managed resources**: cert-manager and the service-ca operator populate secrets and ConfigMaps after the GitOps tool creates the initial empty resources. If the tool treats this as drift, it either blocks sync or reverts the operator's work.

These constraints lead to one core design principle: **split the deployment into two phases**: prerequisites first (Issuers, Certificates, CA ConfigMaps, NetworkPolicies), then the Vault Helm release. Keep sensitive values out of Git using tooling such as Sealed Secrets, External Secrets Operator, Argo CD Vault Plugin, or Kustomize post-rendering.

Organize the Git repository with per-environment overlay directories. This structure works identically whether consumed by a GitOps controller, a CI/CD pipeline, or a manual Helm upgrade invocation:

```
vault-config/├──
 overlays/│├──
    vault-nonprd/│├──
       kustomization.yaml│├──
       values.yaml│└──
       assets/            # Issuer, Certificate, NetworkPolicy│├──
    vault-prd/│├──
       kustomization.yaml│├──
       values.yaml│└──
       assets/│└──
    vault-dr/│├──
        kustomization.yaml│├──
        values.yaml│└──
        assets/
```

Teams that do not use a GitOps tool can apply the same two-phase model directly with Helm: either sequential `helm upgrade --install` calls or an umbrella chart with subchart dependencies. Refer to the Vault Helm chart reference.

0.23.2 OpenShift GitOps design decisions

Red Hat OpenShift GitOps provides an enterprise-supported Argo CD distribution and serves as Red Hat's recommended GitOps solution for OpenShift. The following decisions target Argo CD but translate to other tools. For official documentation, refer to the "Introduction to GitOps with OpenShift" and "Argo CD Declarative Setup" guides.

Application structure

Use the multi-source Application pattern: one source references the Vault Helm chart from an OCI registry, the other references your Git repository for environment-specific values and supplementary Kustomize manifests. This cleanly separates the upstream chart lifecycle from organizational configuration and lets Argo CD deploy cert-manager Issuers, Certificates, and networking resources alongside the Helm release as a single unit.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: vault
  namespace: openshift-gitops
spec:
  project: vault-project
  sources:
    - repoURL: https://github.com/org/vault-config.git
      targetRevision: main
      ref: values
      path: overlays/vault-prd  # Kustomize overlay: Issuer, Certificate, NP

    - repoURL: oci://registry.example.com/helm/hashicorp/vault
      targetRevision: 0.33.0
      chart: vault
      helm:
        valueFiles:
          - '$values/overlays/vault-prd/values.yaml'

  destination:
    server: https://kubernetes.default.svc
    namespace: vault

  ignoreDifferences:
    - group: ""
      kind: Secret
      name: vault-tls
      jsonPointers:
        - /data

  syncPolicy:
    syncOptions:
      - CreateNamespace=false
```

The first source clones the Git repository and deploys the Kustomize overlay (cert-manager resources, networking manifests). The second pulls the Vault Helm chart and renders it with the environment-specific values. The `ignoreDifferences` block prevents drift detection on the cert-manager-managed vault-tls Secret.

Dependency ordering

The two-phase prerequisite model maps to Argo CD in two ways. Choose based on operational preference:

- **Single Application with sync waves**: annotate the Issuer at wave -6, the Certificate at wave -5, and the Vault Helm release at the default wave 0. All resources share a single sync status, which is simpler to manage but offers less granular control.
- **Separate Applications under App-of-Apps**: a vault-prereqs Application and a vault-app Application, ordered by sync-wave annotations on the Application resources themselves, or gated by a PreSync resource hook. This gives each phase an independent sync lifecycle at the cost of an extra Application to manage.

Explicit ordering through mechanisms like sync-waves is often essential to ensure that prerequisite resources deploy before the Vault Pods start.

Handling operator-managed drift

The cert-manager and service-ca operator write into Secrets and ConfigMaps after Argo CD creates them. Argo CD may interpret this as drift. Therefore, it may be necessary to configure ignoreDifferences on the /data field of Secret/vault-tls (cert-manager or service-ca) and ConfigMap/service-ca (if using service-ca). Without this, self-heal may revert operator-managed content, break certificate rotation, or leave the Argo CD Application permanently out of sync.

Sync policy

Enable `automated.selfHeal` with caution, as it reverts manual changes during Vault maintenance operations such as seal migration, manual unseal, or Raft peer removal. Disable `automated.prune` or use `PrunePropagationPolicy` to prevent accidental deletion of PersistentVolumeClaims (PVCs) or secrets during manifest reorganization. Set `CreateNamespace=**false**` so the platform team provisions the Vault namespace with proper RBAC, quotas, and SecurityContextConstraints (SCCs) before the application deploys. Use `RespectIgnoreDifferences=**true**` to prevent automated sync from overwriting operator-managed fields. See Argo CD: Automated Sync Policy for additional information.

Project scoping

Contain Vault Applications within a dedicated Argo CD Project to limit which repositories, namespaces, and clusters they can target. Without this boundary, a misconfigured ApplicationSet can sync resources into the wrong namespace or deploy to an unintended cluster, both of which are scenarios that can cause outages.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: AppProject
metadata:
  name: vault-project
  namespace: openshift-gitops
spec:
  sourceRepos:
    - 'https://github.com/org/vault-config.git'
    - 'oci://registry.example.com/helm/hashicorp/*'
  destinations:
    - namespace: vault-*
      server: https://kubernetes.default.svc
  clusterResourceWhitelist:
    - group: ''
      kind: Namespace
```

Helm values precedence

Argo CD applies Helm values in a fixed precedence order. When using subcharts, verify that subchart defaults do not silently override your top-level values. This is a common source of debugging time, especially for teams without access to the underlying templates. For the full precedence order, refer to the Helm values precedence subsection in the Deployment options section. For a deeper exploration, see the Argo CD Helm documentation.

Multi-cluster deployments

For a multi-region Vault with DR or performance replication, use Argo CD ApplicationSets with a list generator to produce one Application per target cluster. Each cluster references its own overlay directory for environment-specific values (replication role, hostnames, certificates) while sharing the same Helm chart version and base configuration.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: vault-clusters
  namespace: openshift-gitops
spec:
  generators:
    - list:
        elements:
          - cluster: cluster-primary
            url: https://api.primary.example.com:6443
            overlay: vault-prd-primary
          - cluster: cluster-dr
            url: https://api.dr.example.com:6443
            overlay: vault-prd-dr
  template:
    metadata:
      name: 'vault-{{cluster}}'
    spec:
      project: vault-project
      sources:
        - repoURL: https://github.com/org/vault-config.git
          targetRevision: main
          ref: values
          path: 'overlays/{{overlay}}'
        - repoURL: oci://registry.example.com/helm/hashicorp/vault
          targetRevision: 0.33.0
          chart: vault
          helm:
            valueFiles:
              - '$values/overlays/{{overlay}}/values.yaml'
      destination:
        server: '{{url}}'
        namespace: vault
```

The primary and DR secondary clusters use different Vault configurations (listener addresses and replication stanza) through their respective overlay directories, but share the same chart version and project scoping. See Multi-cluster Management with GitOps, ApplicationSet documentation, and Cluster sharding across Argo CD Application Controller replicas for more information. For network connectivity between clusters, refer to the cross-cluster networking for replication subsection in the Networking, routes, and TLS section.

Common anti-patterns

Review your Argo CD configuration against the following failure patterns before promoting to production.

- **Committing TLS private keys to Git**: anyone with repository read access can impersonate the Vault endpoint.
- **`selfHeal` without `ignoreDifferences`**: Argo CD reverts operator-managed secrets, breaking certificate rotation.
- **prune without safeguards**: can delete PVCs or Secrets during manifest changes.
- **Unscoped Argo CD Project**: a misconfigured ApplicationSet can target production clusters unintentionally.
- **Missing sync-wave annotations**: Vault deploys before certificates exist; TLS fails on first sync.
- **`VAULT_SKIP_VERIFY` as a workaround**: masks TLS misconfigurations instead of diagnosing the certificate chain.
- **Unseal or recovery keys in Git**: full cluster compromise. Use KMS auto-unseal.

Exposing and routing Vault telemetry, logs, and health signals on OpenShift requires coordinating Vault-side configuration with platform-level collection and forwarding infrastructure. This section covers the configuration requirements for exposing, collecting, forwarding, and alerting on Vault telemetry and logs on OpenShift. The Vault-side configuration applies to any Prometheus-compatible metrics stack and log aggregation platform, including OpenShift Cluster Monitoring, the LGTM stack (Loki, Grafana, Tempo, and Mimir), Elastic, Splunk, and Datadog. The following subsections cover Vault-side configuration first, then platform-specific integration points such as ServiceMonitor label matching and log forwarding rules. Each profile in the reference Helm values for this HVD implements the telemetry, logging, audit, and health probe settings from this section as part of a complete server configuration you can adapt.

0.24 Telemetry

Telemetry metrics provide measurable signals about Vault cluster health, request latency, and resource consumption. Use these metrics to detect anomalies, trigger alerts, and inform capacity planning decisions. This section covers the Prometheus metrics endpoint configuration, metrics collection design for OpenShift, and alerting rule strategy.

Vault exposes Prometheus-format metrics at the `/v1/sys/metrics` endpoint. For telemetry stanza parameters and recommended defaults, refer to Vault: Solution Design Guide - Detailed Design: telemetry stanza.

0.24.1 Enable the Prometheus metrics endpoint

Vault requires authentication to access `/v1/sys/metrics` by default. Set `unauthenticated_metrics_access` to `true` in the listener-level telemetry block so your monitoring stack can scrape health and performance data from each Vault Pod without a token. With the default authenticated-only access, the following limitations affect metrics collection:

- **Standby visibility loss**: Regular standbys cannot validate tokens locally, so they respond with a 307 redirect to the active node instead of serving local metrics. Prometheus receives only the active node's metrics, losing visibility into the health, resource usage, and performance of each standby Pod.
- **Sealed Pod gap**: Sealed Pods reject all requests before Vault processes authentication, making metrics inaccessible until unseal completes.
- **DR secondary visibility gaps**: DR secondaries cannot run performance standbys, so every non-active Pod is a regular standby that redirects metrics requests to the active node rather than serving local data. With authenticated access using an orphan batch token, only the unsealed active node returns metrics. Unauthenticated access is the only configuration that returns local metrics from every Pod in a DR secondary cluster.

Performance standby Pods can validate tokens locally and serve authenticated metrics requests, but unauthenticated access provides consistent per-pod metrics collection regardless of Vault Pod role, including sealed and standby Pods.

Configure a top-level `telemetry` block outside the listener stanza, and a `telemetry` block inside the `listener "tcp"` stanza. Place both blocks in the `server.ha.raft.config` Helm chart value as HashiCorp Configuration Language (HCL).

```
listener "tcp" {
  telemetry {
    unauthenticated_metrics_access = true
  }
}

telemetry {
  prometheus_retention_time = "12h"
  disable_hostname          = true
}
```

In the top-level telemetry block, set `prometheus_retention_time` to 12h, lower than the Vault default of 24h. The retention window must exceed the scrape interval so metric series do not expire

between scrapes, and 12h also tolerates temporary scraping interruptions, such as Prometheus Pod restarts, without losing metric continuity.

Set `disable_hostname` to **`true`** to prevent Vault from embedding the Pod hostname in metric names. Without `disable_hostname`, metric names include the Pod hostname and break time series continuity when OpenShift replaces Pods.

0.24.2 Metrics collection on OpenShift

After you enable the metrics endpoint, configure your monitoring stack to discover and scrape each Vault Pod. The configuration varies depending on whether you use OpenShift built-in monitoring, a standalone Prometheus Operator, or a direct metrics collection agent.

Set `serverTelemetry.serviceMonitor.enabled` to **`true`** in the Helm chart values to create a ServiceMonitor custom resource. The Helm chart applies a default label of `release: prometheus` to the ServiceMonitor. If your Prometheus instance uses a different selector, override the label through `serverTelemetry.serviceMonitor.selectors` to match your Prometheus configuration.

The Helm chart sets `serverTelemetry.serviceMonitor.matchLabels` to an empty map by default. In HA mode, when `matchLabels` is empty, the ServiceMonitor template applies `vault-active: "true"` as a fallback, which scrapes only the active Pod. Override `matchLabels` to `vault-internal: "true"` to scrape all Pods.

Scraping only the active Pod misses standby metrics, including read request latency from performance standbys and per-pod Raft health.

Performance standby Pods emit metrics independently, and in read-heavy deployments, their metrics provide most of the request latency data. Key metrics that are unavailable when you scrape only the active Pod include:

- Write request forwarding latency
- Per-pod Raft storage health
- Heartbeat connectivity to the active Pod
- Read request latency from performance standbys

Keep the default scrape interval of `30s`. This interval provides sufficient granularity for alerting and dashboards. Shorter intervals increase metric storage and CPU overhead without meaningfully improving anomaly detection for Vault workloads.

Vault disaster recovery (DR) and performance replication (PR) secondary clusters also serve the `/v1/sys/metrics` endpoint when you enable `unauthenticated_metrics_access`. Monitor replication secondaries to verify that each Vault Pod remains unsealed, track replication health, and confirm promotion readiness before failover. For replication health monitoring recommendations, refer to Monitor enterprise replication.

OpenShift built-in monitoring. The Cluster Monitoring Operator (CMO) manages OpenShift built-in monitoring and includes the Prometheus Operator, so you do not need to install a separate operator. OpenShift's default Prometheus does not scrape Pods in user namespaces. Enable user workload monitoring by setting `enableUserWorkload: true` in the `cluster-monitoring-config` ConfigMap in the `openshift-monitoring` namespace. This deploys a dedicated Prometheus instance that discovers ServiceMonitors in user namespaces, including the Vault ServiceMonitor.

The Cluster Observability Operator (COO) provides optional monitoring stack customization but is not required for Vault metrics collection or alerting.

Standalone Prometheus Operator. Standalone installations use the same ServiceMonitor and `matchLabels` configuration described in this section. Install the Prometheus Operator and its custom resource definitions before deploying the Vault Helm chart. The Helm chart renders the ServiceMonitor manifest when you set `serverTelemetry.serviceMonitor.enabled` to **`true`**. If you have not installed the ServiceMonitor custom resource definition, the Helm chart installation fails.

Direct metrics collection without Prometheus Operator. Organizations using metrics collection agents that do not rely on ServiceMonitor custom resources, such as Grafana Alloy with Kubernetes service discovery, can scrape the Vault metrics endpoint directly. Configure your metrics agent to target `/v1/sys/metrics?format=prometheus` on each Vault Pod using the `<release>-internal` headless Service for Pod discovery. The Vault-side configuration for unauthenticated access, hostname handling, and retention applies regardless of scraping method.

0.24.3 Alerting

Alert rules convert telemetry data into actionable notifications for your operations team. Define thresholds that detect degraded Vault health or performance before degradation affects users.

Set `serverTelemetry.prometheusRules.enabled` to **`true`** and define rules in `serverTelemetry.prometheusRules.rules` to create a PrometheusRule custom resource. Setting `global.serverTelemetry.prometheusOperator` to **`true`** achieves the same result. Set `serverTelemetry`

`.prometheusRules.selectors` to match the `ruleSelector` on your Prometheus custom resource if it differs from the default `release: prometheus`.

After you enable user workload monitoring, user-defined PrometheusRule alerts route through the default platform Alertmanager. Alerts from user-defined projects appear in the OpenShift Console under **Observe** > **Alerting** alongside platform alerts. With a standalone Prometheus Operator, alerts route through the configured Alertmanager instance. Organizations not using the Prometheus Operator define equivalent alert rules in their alerting platform, such as Grafana Alerting or Mimir Ruler, using the same Vault metrics.

For critical alert categories, recommended thresholds, and detailed metric guidance for Vault Enterprise, refer to Vault: Operating Guide for Adoption - Monitoring and observability: telemetry.

OpenShift provides the following pod-level metrics in addition to Vault's own telemetry:

- **Restart counts**: Indicate crash loops or failed health checks
- **Memory usage**: Signals risk of out-of-memory (OOM) termination, which causes a Pod restart and service interruption until the replacement Pod completes initialization
- **CPU throttling**: Degrades request latency without triggering pod restarts
- **Disk usage**: Increases steadily on write-heavy clusters because BoltDB backing files (`vault.db` and `raft.db`) grow over time and do not shrink automatically

0.25 Logging

Vault produces two types of logs: **operational logs** and **audit logs**. Operational logs record server activity, errors, and diagnostic information. Audit logs record API requests and responses, including failed authentication attempts, for security analysis. Configure your log forwarding pipeline to collect and route both log types from the Vault namespace.

0.25.1 Operational logs

Operational logs are the primary data source for troubleshooting Vault server issues and understanding cluster behavior. Configure log format and collection to ensure these logs reach your aggregation platform in a structured, parseable format.

Set `server.logFormat` to `"json"` in the Helm chart values. JSON format enables field-level filtering in log aggregation pipelines.

Vault defaults to `info` log level, which captures errors, warnings, and lifecycle events. This level provides sufficient detail for production monitoring and initial troubleshooting.

Do not run `debug` or `trace` log levels in production. These levels significantly increase log volume and degrade Vault performance. Use them only during active troubleshooting and revert to `info` immediately after.

By default, the Vault container writes operational logs to standard error (stderr).

When you install the Red Hat OpenShift Logging Operator, it deploys log collection and forwarding infrastructure and automatically captures container output. Configure a ClusterLogForwarder custom resource to define log outputs and routing rules. The operator supports forwarding to destinations such as Splunk, Loki, Kafka, and Syslog.

You can substitute any Kubernetes-compatible log collector, such as Fluent Bit, Grafana Alloy, or Vector, for the OpenShift Logging Operator.

For additional guidance on operational log content, refer to Vault: Operating Guide for Adoption - Monitoring and observability: operational logs.

0.25.2 Audit logs

Vault halts all client operations when every configured audit device fails. Configure at least two audit devices with different output paths for redundancy, and choose paths that integrate with container log collection.

Configure the `file` audit device with `file_path` set to `stdout` to write audit log entries to the container's standard output. Your log collector captures audit entries alongside operational logs from the same container output stream. Writing to stdout does not require a PersistentVolumeClaim (PVC) for the audit device.

We recommend a `file` audit device writing to stdout as the primary device. Add a `socket` audit device sending to an external TCP collector as the secondary device. If no external TCP collector is available, use a second `file` audit device writing to PVC-backed storage. The Helm chart creates a PVC and mounts it at `/vault/audit` when you set `server.auditStorage.enabled` to **`true`**. Different output paths prevent a single pipeline failure from disabling both audit devices. For audit device types and configuration parameters, refer to the Vault audit devices documentation.

0.25.3 Audit log forwarding and SIEM integration

Operational logs and audit logs serve different purposes and must reach different downstream systems. Separate them in your log forwarding pipeline so that operational logs feed monitoring workflows and audit logs feed security and compliance analysis.

Because the container log stream merges stdout and stderr, identify audit entries by the `type` field (`request` or `response`) and operational entries by the `@level` field. Route operational logs to your monitoring system and audit logs to your security information and event management (SIEM) platform.

The ClusterLogForwarder supports this separation through `parse` and `drop` filters that match on `.structured.type` field values, with separate pipelines routing audit and operational logs to different outputs. Standalone log collectors, such as Fluent Bit, Grafana Alloy, or Vector provide equivalent field-based filtering and pipeline routing.

For the full list of observable audit log patterns, refer to Vault: Operating Guide for Adoption - Monitoring and observability: usage patterns. For the list of privileged endpoints that require SIEM alerts, refer to Vault: Operating Guide for Adoption - Monitoring and observability: privileged endpoints.

0.26 Health endpoint monitoring

The health API provides a lightweight way to verify the operational state of each Vault Pod independently of telemetry metrics and log analysis. Use the health endpoint for Kubernetes readiness and liveness probes and for external uptime monitoring.

On OpenShift, the Vault Helm chart uses the health endpoint for readiness and liveness probes. The `/v1/sys/health` API returns HTTP status codes that reflect the state of the Vault Pod you query. For status code definitions and alerting guidance, refer to Vault: Operating Guide for Adoption - Monitoring and observability: health API and the Vault /sys/health API documentation.

Probe design depends on which Vault states each probe must treat as healthy. Readiness signals Pod health to consumers such as external load balancers, Kubernetes Operators, and rollout tooling. Liveness controls when the `kubelet` restarts the container. A liveness query string that treats sealed or uninitialized states as unhealthy can restart Pods that operate as designed. A readiness query string that treats standby, performance standby, or disaster recovery (DR) secondary states as unhealthy can mark healthy Pods unavailable.

0.26.1 Readiness probe

The readiness probe drives the Pod Ready condition and provides a role-aware health signal that Kubernetes Operators, rollout tooling, and external probers can align with.

We recommend the following readiness path:

```
/v1/sys/health?standbyok=true&perfstandbyok=true&drsecondarycode=200
```

Once the cluster initializes and all Pods unseal, this query string returns HTTP 200 for every healthy Pod in the cluster, so external probers see a uniform ready signal across the cluster. Sealed and uninitialized Pods continue to return their default non-2xx status (503 sealed, 501 uninitialized), which is the intended behavior for readiness.

By default, the chart enables `server.readinessProbe.enabled` but leaves `server.readinessProbe.path` unset, which configures an `exec` probe that runs `vault status`. Setting `path` switches the chart to an `httpGet` probe against that path. Set `server.readinessProbe.path` to the recommended readiness path, and configure external load balancers separately to probe the same path so both checks use the same `/v1/sys/health` semantics.

The chart-default values for `failureThreshold`, `initialDelaySeconds`, `periodSeconds`, and `timeoutSeconds` are a reasonable starting point for the readiness probe.

0.26.2 Liveness probe

The liveness probe determines when the `kubelet` stops and restarts the Vault container. A sealed Vault Pod is not unhealthy: unsealing requires an external action, and a restart does not recover a sealed cluster. The recommended liveness query string treats sealed and uninitialized states as healthy in addition to the states the readiness path accepts.

We recommend the following liveness path:

```
/v1/sys/health?standbyok=true&perfstandbyok=true&drsecondarycode=200&sealedcode=204&
    uninitcode=204
```

The added query parameters cover Vault states that must not trigger a restart:

- Setting `sealedcode=204` makes a sealed Pod report healthy to the `kubelet` so that auto-unseal failures or operator-driven seals do not cause restart loops.

- Setting `uninitcode=204` makes an uninitialized Pod report healthy during first bootstrap.

By default, the chart sets `server.livenessProbe.enabled` to **`false`** and `server.livenessProbe.path` to `/v1/sys/health?standbyok=true`. Enable the liveness probe and set `server.livenessProbe.path` to the recommended liveness path. The chart default returns a non-2xx status for sealed Pods, so the `kubelet` treats sealed Pods as failed and restarts the container.

Keep the chart default `server.livenessProbe.initialDelaySeconds` of 60. Do not lower `initialDelaySeconds` without measuring startup time in your environment. Auto-unseal dependencies and Raft join can extend startup beyond the default.

0.27 Common anti-patterns

The following patterns create observability gaps that are difficult to detect until an incident exposes them. Review your configuration against this list before promoting to production.

Anti-pattern	Risk
Enabling only one audit device	Vault halts all client operations when the audit device fails.
Scraping only the active Vault Pod	Loses per-pod visibility into standby metrics. Refer to the Metrics collection on OpenShift section.
Omitting `disable_hostname` in the top-level telemetry block	Metric names include the Pod hostname, which causes series discontinuity when OpenShift replaces Pods and breaks dashboards and alert rules that reference those series.

0.28 Failure scenarios and recovery

Understanding failure scenarios helps you design resilient Vault clusters and plan recovery procedures. The following scenarios describe common failures and their impact on OpenShift deployments.

0.28.1 Pod failure

A Pod failure occurs when a Vault process crashes or the container fails health checks. The `kubelet` restarts the Vault container according to the Pod restart policy. If Kubernetes recreates the Pod, the StatefulSet controller preserves the Pod identity and the new Pod mounts the same PersistentVolumeClaim (PVC). The restarted Vault instance rejoins the Raft cluster using its persisted state. Probe timing affects how quickly the cluster detects and recovers from Pod failures.

0.28.2 Node failure

A node failure causes all Pods on that node to become unavailable. After a configurable timeout, the node controller marks the node NotReady and applies `node.kubernetes.io/not-ready` and `node.kubernetes.io/unreachable` taints with the `NoExecute` effect. Pods tolerate these taints for a default of 300 seconds before eviction.

The StatefulSet controller creates replacement Pods and the Kubernetes scheduler assigns them to available nodes. If the Pod's PVC uses zone-local storage, the replacement Pod must schedule in the same zone. If the PVC uses a local persistent volume bound to a specific node, the replacement Pod must return to that node.

The cluster's quorum tolerance determines whether it can sustain the disruption. In the reference 6-pod redundancy zones topology, the cluster has 3 voters. When one node fails and its Vault Pod becomes unavailable, if that Pod is a voter, the remaining 2 of 3 voters maintain quorum. If the failed Pod is a non-voter, the failure does not affect quorum. In the 5-pod topology, the cluster preserves quorum as long as at least 3 voting Pods remain available. With the recommended node-level anti-affinity, a single node failure normally removes 1 Vault Pod and leaves 4 of 5 voters available.

Node failure recovery time depends on the node controller detection period, taint toleration duration, Pod replacement scheduling, PVC reattachment, and Vault startup. These stages vary by environment, CSI driver, and node controller configuration. If the Pod's PVC uses zone-local storage, or a local persistent volume bound to the failed node, the replacement Pod can remain `Pending` after permanent loss of that zone or node. Recovery then depends on the storage backend and CSI driver and can require manual intervention.

For permanent loss of a non-voting Pod, dead server cleanup removes the failed peer from the Raft configuration after the configured threshold, as described in the Dead server cleanup sec-

tion. For permanent loss of a voting Pod, Autopilot can also remove the failed voter automatically, but it may wait until the cluster satisfies safety conditions, such as preserving `min_quorum` and avoiding removal of the last remaining server in a redundancy zone. Use manual `vault operator raft remove-peer` only when Autopilot cannot remove the failed voter automatically. If Autopilot removes a non-voting peer before the Pod recovers, the returning Pod cannot rejoin with its existing PersistentVolumeClaim. For details, refer to the Dead server cleanup section.

0.28.3 Failure domain loss

Failure domain loss affects all nodes and storage within a failure domain. A 6-pod redundancy-zones cluster maintains quorum after any single zone loss when you distribute Pods with one voter per zone. A 5-pod cluster spread across 3 failure domains, such as 2-2-1, maintains quorum when a failure domain goes offline. A 5-pod cluster split 3-2 across 2 failure domains loses quorum if the 3-pod domain fails.

When Vault PersistentVolumes use zonal block storage or local persistent volumes bound to individual nodes, replacement Pods for the failed domain can remain `Pending` because the volumes bind to the lost zone or node and cannot move automatically to surviving zones. Some storage backends, such as regional or zone-redundant disks, can recover across zones, but that behavior depends on the storage platform and CSI driver rather than Vault. Pods resume with their existing data only when the failed domain returns or the storage backend supports cross-zone recovery.

Remaining Pods serve requests while quorum holds. In a redundancy-zones cluster, Autopilot promotes a stable non-voter in a surviving zone to restore the 3-voter count, which temporarily places two voters in one surviving zone. After that promotion, one surviving zone holds 2 voters and the other holds 1. The cluster tolerates a second sequential zone loss only if the second loss hits the 1-voter zone. The surviving zone that receives the temporary extra voter depends on which stable non-voter Autopilot promotes. Entire-site loss requires DR replication, refer to the Multi-region architectures section.

0.28.4 Split-brain prevention

Split-brain prevention relies on Raft consensus semantics. A Vault node cannot become the leader without receiving votes from a majority of the cluster. If network partitions divide the cluster, only the partition containing a majority can elect a leader and serve requests. Nodes in minority partitions

cannot elect a leader and cannot process requests that require an active node until the network restores connectivity.

Health check timing affects failure detection and recovery speed. Aggressive probe intervals detect failures quickly but can cause unnecessary restarts during transient issues. Conservative intervals reduce false positives but delay failure detection. Probe timing affects how quickly Kubernetes routes traffic away from unhealthy Pods and restarts failed processes. For detailed probe configuration guidance, refer to the Health endpoint monitoring section.

0.29 Common anti-patterns

The following table describes deployment patterns that can cause reliability issues or reduce cluster resilience.

Anti-pattern	Risk
Not configuring explicit `topologySpreadConstraints`	Without explicit constraints, the scheduler relies on soft defaults that do not prevent uneven placement under resource pressure. Configure `server.topologySpreadConstraints` with DoNotSchedule as described in the Pod topology spread constraints section.
Even replica counts without redundancy zones (2, 4)	Even numbers do not improve fault tolerance over odd numbers. A 4-node cluster tolerates only 1 failure, the same as a 3-node cluster, but with higher resource cost.
Routing replication traffic through <release> instead of <release>-active	Replication traffic on port 8201/tcp must reach the active Vault node. The <release> service distributes traffic to all Pods, so replication requests may reach standby nodes that cannot process them. Use the <release>-active service for replication traffic.

Anti-pattern	Risk
Not customizing health probes	By default, the Vault Helm chart enables a readiness probe that runs `vault status` and disables the liveness probe. Customize probe paths to account for standby, performance standby, DR secondary, sealed, and uninitialized states. Without tailored probes, the `kubelet` may restart Pods unnecessarily during expected state transitions, or fail to detect an unresponsive Vault process that continues to count toward quorum.
Undersized resource requests	Pods with low resource requests are eviction candidates during node memory pressure. Leader eviction causes cluster disruption and election delays.
Spanning exactly 2 availability zones	Quorum loss risk. If you split voting members evenly, a zone-to-zone partition prevents either side from reaching quorum. If you place a majority in one zone, losing that zone causes an outage. Use at least 3 zones for production deployments.
Ignoring network latency requirements	Raft consensus requires responses within election timeouts. Network latency exceeding 8 ms can cause election instability and leader oscillation.

0.30 Conclusion

Treat the deployment of Vault Enterprise on OpenShift as a platform engineering discipline, not a routine application installation. A production-ready deployment depends on informed, deliberate choices across topology, storage, networking, TLS, resource management, lifecycle operations, and observability. Manage Vault through Helm and GitOps, use explicit scheduling controls, align Raft

topology with real failure domains, preserve end-to-end TLS wherever possible, with careful attention to monitoring, logging, audit forwarding, and health probes to reflect Vault's actual operating states.

OpenShift supplies foundational primitives for ownership, isolation, scheduling, routing, and lifecycle management, while sound Vault system configuration addresses cluster quorum, cryptographic trust, auditability, and predictable recovery. A successful deployment reconciles both sets of concerns into a repeatable operating model: OpenShift becomes a durable platform for delivering Vault Enterprise as a resilient and supportable secrets management service.

1 Changelog

1.1 2026-05

- Emerging guidance first release

2 Credits

Thank you to everyone who contributed to this validated design guidance. This cross-functional team, representing many departments within HashiCorp, authored, edited, reviewed, and iterated this content to provide prescription and recommendations for using HashiCorp software in enterprise scenarios.

- Chris Zembower
- Francisco Lee
- Huseyin Unal
- Mohamed Jouini