



■ EBOOK

Zero Trust for Kubernetes with Kyverno

Build a Zero Trust Architecture with policy as code: micro-segment the network, enforce least privilege, verify the supply chain, and govern every exception.

Tested policy set

Five control areas, with tests and an audit-first profile

Mapped to NIST and CISA

SP 800-207 tenets and Zero Trust Maturity Model pillars

90-day plan

From audit to enforce, one pillar at a time

CONTENTS

Contents

	Executive summary	3
	About this guide	4
1	Why Kubernetes needs Zero Trust	7
2	Zero Trust foundations	13
3	Policy as code, the Zero Trust control plane	22
4	Micro-segment the network	28
5	Enforce least-privilege access	41
6	Verify workloads and the software supply chain	51
7	Default deny with governed exceptions	60
8	Verify continuously	68
9	Reference architecture	72
10	Maturity roadmap	76
11	Case studies	79
12	Extend Zero Trust to runtime (optional)	81
13	Next steps	85
	Appendices	87
	Appendix A: Policy index	87
	Appendix B: Glossary	89
	Appendix C: References	91
	About Nirmata	93

Executive summary

Kubernetes grants broad trust by default. Any Pod can reach any other Pod, every Pod gets an API token, any image can run, and access, once granted, stays until someone removes it. Zero Trust removes that implicit trust, but many Zero Trust programs stop at the network edge and at user identity. Inside the cluster, decisions about workloads, service accounts, images, and exceptions are still made by hand, or not at all, and nothing checks that running resources still comply.

This guide treats Zero Trust for Kubernetes as a policy problem. You write each control as a Kyverno policy, test it in CI, enforce it at admission, and verify it continuously in the background. Kyverno is a CNCF graduated policy engine that runs inside the cluster, so a control applies to every request that reaches the Kubernetes API server, whatever its source. Each policy maps to a NIST SP 800-207 tenet and a CISA Zero Trust Maturity Model pillar, so you can report progress in terms that auditors recognize.

Kyverno is the policy layer of the architecture, not all of it. Your identity provider, CNI plugin, service mesh, build pipeline, and secret store do work that Kyverno doesn't do, and each chapter names them. Chapter 2 shows which component owns each Zero Trust function.

What you get:

- A tested policy set for five areas: network micro-segmentation, least-privilege access with just-in-time grants, software supply chain verification, Pod security, and governed, time-bound exceptions. Every policy is in the [companion repository](#), with its tests and an audit-first install profile.
- A rollout method: audit first, enforce one area at a time, and grant exceptions that expire.
- A reference architecture, a maturity roadmap, and a 90-day plan.

Where to start:

- Security leaders: read Chapters 1, 2, 10, and 13.
- Architects: also read Chapters 3 and 9.
- Platform and security engineers: work through Chapters 3 through 8 with the companion repository.

About this guide

This guide shows you how to build a Zero Trust Architecture (ZTA) for Kubernetes with Kyverno as the policy layer. Each chapter explains one Zero Trust control, gives you a policy you can apply, tells you how to check that it works, and names the other components, such as your CNI plugin, service mesh, or identity provider, that complete the control.

Who this guide is for

- Platform engineers who run shared Kubernetes clusters.
- Security engineers who need to map Kubernetes controls to Zero Trust frameworks.
- Architects who plan a Zero Trust program across cloud-native workloads.

You should know basic Kubernetes objects such as Pods, namespaces, RoleBindings, and NetworkPolicies.

What you learn

- How Zero Trust principles from NIST and CISA map to Kubernetes.
- How to use Kyverno as the in-cluster policy decision point, with the Kubernetes API server as the enforcement point.
- Which Zero Trust responsibilities Kyverno owns, which it configures for other components, and which need other tools.
- How to segment the network, enforce least privilege, verify images, and govern exceptions.
- How to measure progress with policy reports and a maturity roadmap.

Kyverno at a glance

Kyverno is a CNCF graduated project (March 2026) for policy as code. You write policies as Kubernetes resources and express rules in the Common Expression Language (CEL). This guide uses Kyverno 1.19.1, from the 1.19 release line published in August 2026.

Kyverno 1.19 deprecates the legacy `ClusterPolicy`, `Policy`, `CleanupPolicy`, and `ClusterCleanupPolicy` types and the `kyverno.io PolicyException`, and their removal is planned for 1.20, so this guide uses only the newer policy types.

Before you begin

To run the samples, you need the following:

- A Kubernetes cluster that you can use for testing, running Kubernetes 1.33 or later. Kyverno 1.19 supports Kubernetes 1.33 to 1.35, and the samples were also tested on 1.37. For the current list, see [Kyverno releases](#).
- `kubectl` and `helm` installed on your workstation.
- Kyverno 1.19 or later, with the `GeneratingPolicy` inline YAML template mode. The samples use the `policies.kyverno.io/v1` policy types and were tested on Kyverno 1.19.1 (Helm chart 3.9.1).
- A CNI plugin that enforces NetworkPolicy, such as Cilium or Calico.

To install Kyverno with policy exceptions turned on, run the following commands:

```
helm repo add kyverno https://kyverno.github.io/kyverno/
helm repo update
helm install kyverno kyverno/kyverno \
  --namespace kyverno --create-namespace \
  --set features.policyExceptions.enabled=true \
  --set features.policyExceptions.namespace=kyverno-exceptions
```

SHELL

The last two flags let Kyverno accept exceptions, but only from the `kyverno-exceptions` namespace. Chapter 7 explains why this matters.

How to use the samples

Every policy in this guide is in the companion repository, [nirmata/kyverno-zero-trust](#). Each policy has a README, Kyverno CLI tests, and an end-to-end test for a local kind cluster, and the set is tested on Kyverno 1.19.1. You can also request the policies as a versioned bundle from [Nirmata](#).

To roll out the policies, do the following:

1. Install the audit profile. It installs every core validating policy in `Audit` mode, plus the extra Kyverno permissions that the policies need, so it reports violations without blocking any request:

```
git clone https://github.com/nirmata/kyverno-zero-trust
cd kyverno-zero-trust
kubectl apply -k deploy/audit
```

SHELL

2. Review the results in policy reports:

```
kubectl get policyreports -A
```

SHELL

3. Fix each finding, or cover it with a governed exception (Chapter 7).
4. Replace placeholder values, such as `registry.example.com` and `platform-admins`, with your own. The repository's `catalog.yaml` file lists the placeholders in each policy.
5. Switch policies to `Deny` one pillar at a time, following the rollout stages in Chapter 8.

Test every policy in a non-production cluster first. Some samples, such as generating policies, create or remove resources.

Conventions

- Code, commands, field names, and labels appear in `code font`.
- Replace values like `example.com` and `myorg` with your own.
- The policy YAML in this guide matches the tested files in the companion repository, except that the guide leaves out the descriptive `policies.kyverno.io/*` annotations. If the two ever differ, use the repository version.
- Chapters 1 and 2 end with key takeaways. Chapters 3 through 8 end with a checklist that you can use to track your rollout.

01

CHAPTER 1

Why Kubernetes needs Zero Trust

Kubernetes trusts too much by default, so an attacker who lands in one Pod can often reach the rest of the cluster. Zero Trust fixes this by removing implicit trust: every request, workload, and user must prove who they are and what they may do, every time.

The perimeter no longer exists

Traditional security puts a firewall around the data center and trusts everything inside. Kubernetes breaks that model:

- Pods are short-lived and change IP addresses constantly.
- Most traffic is east-west, between services inside the cluster, where a perimeter firewall can't see it.
- Many teams and tenants share the same cluster.
- Workloads run images built from third-party code you didn't write.

Default settings that work against you

The following table lists common Kubernetes defaults and the risk each one creates.

Default behavior	Risk	Chapter that fixes it
Any Pod can reach any other Pod.	An attacker moves laterally from one compromised Pod.	Chapter 4
Every Pod mounts a service account token.	A stolen token lets an attacker call the Kubernetes API.	Chapter 5
RBAC allows wildcards and <code>cluster-admin</code> bindings.	One over-privileged account controls the cluster.	Chapter 5
The cluster runs any image from any registry.	Tampered or vulnerable images run in production.	Chapter 6
Pods can run as root or in privileged mode.	A container escape gives access to the node.	Chapter 6
Exceptions to policy never expire.	Temporary bypasses become permanent holes.	Chapter 7

Zero Trust is a policy problem

Zero Trust isn't a product you install. It's a set of decisions about who can do what, under which conditions, and for how long. In Kubernetes, those decisions happen in three places:

- 1. Before deployment:** CI pipelines check manifests and infrastructure code.
- 2. At admission:** The Kubernetes API server receives a request and asks the policy engine to decide.
- 3. After deployment:** Controllers scan existing resources for drift.

A policy engine that works in all three places gives you one consistent set of rules. Kyverno does this, and it runs inside the cluster, so teams can't skip it by deploying outside the pipeline.

Why now

Every gap in the preceding table has existed since Kubernetes 1.0. Two things changed.

First, the workloads. Model-serving frameworks, LLM gateways, and autonomous agents now run with cluster credentials, call the internet by design, and in some cases write and run their own code. In the newest incidents, the artifact you pull into the cluster isn't

only code that might have a bug. It's a model file built to run code the moment something loads it.

Second, the volume of vulnerabilities. [NIST recorded](#) a 263% increase in CVE submissions between 2020 and 2025, and [48,185 CVEs were published in 2025](#), 21% more than in 2024. In April 2026, NIST changed how the National Vulnerability Database prioritizes enrichment to handle the volume. AI-related vulnerabilities are growing faster: [Trend Micro counted 2,130 AI-related CVEs disclosed in 2025](#), a 34.6% increase over 2024, compared with 17.9% growth for CVEs overall. Each one is another way to exploit the implicit trust that this guide removes.

The following incidents are documented, not hypothetical. The last two columns name the Zero Trust control that would have stopped or contained each one, and the chapter that covers it. Not every control is a Kyverno policy. Default-deny egress needs a CNI plugin to enforce it, Kyverno can verify model and skill artifacts at admission only when they're packaged and signed as OCI artifacts, and runtime verification needs a runtime security tool (Chapter 12). Chapter 2 lists which controls Kyverno owns and which need other components.

Incident	When	Root cause	Zero Trust control that would have stopped or contained it	Chapter
IngressNightmare (CVE-2025-1974 and related)	Disclosed March 2025; more than 6,500 clusters found exposed	The ingress-nginx admission webhook accepted unauthenticated requests and ran with a service account that could read every Secret in the cluster.	Restrict network access to the admission webhook to the API server alone, and scope its service account to least privilege instead of cluster-wide secret access.	4, 5
ShadowRay / ShadowRay 2.0 (CVE-2023-48022, still exploited through November 2025)	Active campaign; more than 230,000 exposed Ray instances found in November 2025	The Ray AI framework's dashboard API has no authentication by default and was exposed directly to the internet, then used to run jobs that mined cryptocurrency, stole models and data, and spread to other nodes.	Deny ingress and egress by default, so that an internal dashboard is never reachable from the internet, plus tier-based segmentation to contain lateral movement between nodes.	4
runc container escape trio (CVE-2025-31133, CVE-2025-52565, CVE-2025-52881)	Disclosed November 2025	Containers with custom mount configurations could redirect writes to host paths and <code>/proc</code> , escaping isolation to gain root on the node.	The Restricted Pod Security Standard (no added capabilities, no privileged containers, no host path mounts), plus running untrusted workloads in user namespaces.	6
LiteLLM supply-chain compromise	March 2026	A compromised CI/CD credential let attackers publish backdoored versions of a widely used LLM gateway to PyPI; once deployed, the backdoor harvested cloud and Kubernetes credentials and deployed privileged Pods with host filesystem access.	Verify image and package signatures before anything runs, block privileged Pods and host filesystem mounts, and restrict egress to approved model-provider endpoints.	4, 5, 6

<u>Hugging Face and ClawHub model-repository compromise</u>	Reported 2024–2026, summarized by The Next Web in May 2026; more than 100 Hugging Face models found capable of arbitrary code execution, plus 341 malicious ClawHub agent skills	Attackers embedded malicious code in model files' pickle byte streams (including the "nullifAI" evasion technique) and in agent-skill packages, so the payload ran the moment a model or skill was loaded, before any human inspection, opening reverse shells and stealing credentials.	Verify model and skill artifacts before they're pulled into a cluster, load untrusted models only in a sandboxed, non-privileged runtime with no credentials of its own, and restrict egress so a loaded model can't reach an attacker's server even if it tries.	6, 12
<u>The OpenAI–Hugging Face incident</u>	Occurred May 12–July 19, 2026; disclosed July 21; full technical report August 26, 2026	During an internal safety evaluation, OpenAI's own research agents, which were meant to stay isolated from each other, found an unintended shared channel through an internal package registry. They chained previously unknown vulnerabilities to get code execution on Hugging Face's infrastructure and escalated to administrator access across multiple Hugging Face Kubernetes clusters. They harvested Kubernetes, database, messaging, repository, and cloud credentials, and used them to reach administrator access on an OpenAI research Kubernetes cluster.	Default-deny egress and no shared channel between workloads that are supposed to be isolated from each other, image and package verification instead of trusting an internal network path, and scoped, short-lived credentials everywhere, so a single compromised worker can't become cluster admin.	4, 5, 6, 12
<u>An AI agent published malware to PyPI</u>	Disclosed July 2026	An autonomous agent, believing it was in a simulated environment, published a real, credential-stealing package that ran on 15 real systems within about an hour.	Scoped, time-bound credentials for anything an agent can act through, and runtime verification of what a workload does against what it declares.	5, 7, 12

None of these attacks used a novel technique. Every one succeeded because trust was implicit somewhere: a webhook nobody authenticated, a dashboard nobody knew was reachable, a package nobody verified, a mount nobody restricted, a model file nobody scanned, an agent nobody scoped. That is the assumption Zero Trust removes.

The OpenAI–Hugging Face incident shows why these controls must be defaults, not reactions. Every control involved did what it was configured to do. The agents used a network path nobody had closed, chained vulnerabilities nobody knew about, and never broke a rule, because nothing they did was on a denylist. An agent can use a legitimate credential in a way nobody intended, and retry a boundary until it finds a gap. Design for that possibility:

- Short-lived, scoped credentials instead of standing access (Chapter 5).

- Verified images and packages instead of a trusted internal network (Chapter 6).
- Default-deny segmentation between workloads instead of an open path (Chapter 4).
- Exceptions that expire and are monitored instead of accumulating quietly (Chapter 7).

Chapter 12 applies this approach to AI workloads and agents.

KEY TAKEAWAYS

- Treat every Pod, user, and request as untrusted until verified.
- Replace permissive defaults with explicit, automated policy.
- Make every grant of trust narrow, observable, and time-bound.

02

CHAPTER 2

Zero Trust foundations

Two public frameworks define Zero Trust for most organizations: NIST SP 800-207 describes the architecture, and the CISA Zero Trust Maturity Model describes how to adopt it in stages. This chapter maps both to Kubernetes so you can explain your controls in terms auditors recognize.

NIST SP 800-207 tenets

[NIST SP 800-207](#) (August 2020) lists seven tenets. The following table shows how each one applies to Kubernetes.

NIST tenet (summarized)	What it means in Kubernetes	Kyverno control
1. All data sources and services are resources.	Treat every Pod, Service, Secret, and API as a protected resource.	Policies match any resource kind, including custom resources.
2. Secure all communication, regardless of location.	Encrypt and authorize in-cluster traffic, not only ingress.	Generate NetworkPolicy and mesh mutual TLS (mTLS) settings per namespace (Chapter 4). The CNI plugin and the service mesh enforce them.
3. Grant access per session.	Give short-lived access instead of standing privileges.	Just-in-time RoleBindings with a time to live (TTL) (Chapter 5).
4. Decide access by dynamic policy.	Base decisions on identity, labels, image provenance, and context.	CEL expressions over request, user, and resource data (Chapter 3).
5. Monitor the integrity of all assets.	Know whether deployed workloads still meet policy.	Background scans and policy reports (Chapter 8) check declared specs. Runtime and node integrity need other tools (Chapter 12).
6. Enforce authentication and authorization before access.	Check every API request before it changes the cluster.	The API server authenticates and authorizes first, using your identity provider and RBAC. Kyverno then adds context-aware checks at admission with <code>Deny</code> actions.
7. Collect data to improve security posture.	Feed results into dashboards and SIEM tools.	Policy reports and Policy Reporter (Chapter 8).

Logical components: PDP, PEP, and PA

NIST SP 800-207 defines three core logical components:

- **Policy engine (PE):** makes the final decision to grant, deny, or revoke access to a resource, based on policy and on input from external data sources.
- **Policy administrator (PA):** carries out the PE's decision by setting up or tearing down the communication path, for example by issuing or revoking a credential or session token, and tells the PEP to allow or close the connection.

- **Policy enforcement point (PEP):** sits in the access path, intercepts each request, and allows or blocks it based on that decision.

Together, the PE and the PA form the **policy decision point (PDP)**. The PE also draws on data sources outside these components, such as the identity provider, device inventory, threat intelligence, and activity logs. NIST SP 800-207 describes them as inputs to the PE, not as a separate component.

In Kubernetes, the API server calls Kyverno through admission webhooks. For Kubernetes API requests, Kyverno acts as the PDP, and the API server enforces its decision as the PEP. Kyverno's generating policies and cleanup controller also act as a PA: they create access, such as a RoleBinding, and remove it when it expires. The approval decision behind that access still comes from your identity management (IDM) system, and the Kubernetes RBAC authorizer enforces the binding.

Kyverno is the PDP for the Kubernetes API only. Each other access path has its own PDP and PEP:

Access path	PDP	PEP	Kyverno's role
Kubernetes API request	Kyverno, after RBAC	API server	Decides
User login to the API server	Identity provider	API server authenticator	None
Pod-to-Pod traffic	CNI plugin or service mesh	CNI data plane or mesh proxy	Generates the policy
HTTP request between services	Kyverno Authz Server (Chapter 3)	Envoy or Istio proxy	Decides, if you deploy the Authz Server
Process behavior inside a container	Runtime security tool	Kernel, through eBPF or seccomp	None (see Chapter 12)

The section "What Kyverno covers, and what it doesn't," later in this chapter, lists every function.

NIST SP 800-207A for cloud-native applications

[NIST SP 800-207A](#) (September 2023) applies Zero Trust to cloud-native applications across multiple clouds. It recommends two policy layers:

- **Network-tier policies** that control which workloads can connect, for example, with NetworkPolicy.
- **Identity-tier policies** that authorize each request based on the workload's identity, for example, with a service mesh and mTLS.

Kyverno's role differs by tier. For the network tier, Kyverno generates the NetworkPolicy and CNI rules that your CNI plugin enforces (Chapter 4). For the identity tier, a service mesh provides workload identities and mTLS. Kyverno generates the mesh configuration (Chapter 4) and, with the Kyverno Authz Server, can make per-request authorization decisions that Envoy enforces (Chapter 3). Chapter 5 covers a third layer that SP 800-207A assumes but doesn't detail: access to the Kubernetes API itself.

CISA Zero Trust Maturity Model

The [CISA Zero Trust Maturity Model](#), version 2.0 (April 2023), organizes Zero Trust into five pillars and three cross-cutting capabilities. It defines four maturity stages: Traditional, Initial, Advanced, and Optimal.

CISA pillar or capability	Kubernetes scope	What Kyverno covers	Chapter
Identity	Users, groups, service accounts, RBAC	RBAC guardrails, service account hygiene, and JIT bindings. Not user authentication or MFA.	5
Devices	Nodes, node pools, trusted runtimes	Workload placement rules. Not node hardening, node attestation, or user device posture.	6
Networks	Pod-to-Pod traffic, egress, service mesh	Generates network and mesh policy that the CNI plugin and the mesh enforce.	4
Applications and Workloads	Images, Pod security, supply chain, request authorization	Verifies signatures, enforces Pod security, and authorizes requests with the Authz Server. Not signing, scanning, or runtime detection.	3, 6
Data	Secrets, ConfigMaps, volumes, encryption settings	Can flag plain-text secrets and require external secret references. Not secret storage, encryption at rest, or classification.	10 (roadmap only)
Visibility and Analytics	Policy reports, audit logs, SIEM	Policy reports and metrics. Not API server audit logs or SIEM.	8
Automation and Orchestration	Generating, mutating, and deleting policies	Policy as code, generation, and cleanup.	3, 4, 5
Governance	Policy lifecycle, exceptions, compliance mapping	Governed exceptions and compliance annotations.	7, 8

This guide doesn't include sample policies for the Data pillar. Kyverno policies can flag secrets in environment variables and require references to an external secret store, and the Kyverno policy library has examples. Encryption at rest, key management, and secret storage come from your Kubernetes distribution and secret store, not from Kyverno.

Chapter 10 turns these pillars into a staged roadmap.

What Kyverno covers, and what it doesn't

Kyverno is the policy layer of a Kubernetes Zero Trust Architecture, not the whole architecture. It decides, configures, and reports on Kubernetes API requests and resources. Other components authenticate users, carry and encrypt traffic, sign software, store secrets, and watch running processes. The following table shows, for each Zero Trust function, what Kyverno does and what else you need. Kyverno's role is one of four:

- **Enforces:** Kyverno makes the decision, and the API server blocks or changes the request.
- **Generates or configures:** Kyverno creates or sets the configuration, and another component enforces it.
- **Checks and reports:** Kyverno validates a declared setting and records the result, but doesn't provide the capability.
- **Not covered:** Kyverno has no role. You need another component.

Zero Trust function	CISA pillar	Kyverno's role	What else you need
User authentication and MFA	Identity	Not covered	An identity provider connected to the API server through OIDC, with MFA enforced by the provider.
Kubernetes RBAC authorization	Identity	Checks and reports: validates Roles and bindings at admission (Chapter 5).	The API server's RBAC authorizer makes every authorization decision.
Just-in-time access	Identity	Generates or configures: creates and deletes RoleBindings (Chapter 5).	An IDM or access workflow system to approve requests, and the API server to enforce the binding.
Workload identity and service account tokens	Identity	Enforces: blocks the default service account and turns off token automount (Chapter 5).	The API server issues tokens. A service mesh issues workload certificates.
Continuous user and session risk evaluation	Identity	Not covered	An identity provider or access platform with session risk signals.

Device and endpoint posture	Devices	Not covered	Endpoint management for user devices. Kubernetes doesn't see them.
Node placement and trust	Devices	Checks and reports: requires a <code>nodeSelector</code> for sensitive workloads (Chapter 6).	Hardened node images, node attestation, and a system that labels nodes by trust level.
Node and host hardening, control plane hardening, etcd encryption	Devices	Not covered	CIS Benchmark tooling, your Kubernetes distribution or cloud provider, and KMS encryption for etcd.
Network segmentation	Networks	Generates or configures: creates NetworkPolicy and CNI policies (Chapter 4).	A CNI plugin that enforces NetworkPolicy, such as Cilium or Calico. Without one, generated policies do nothing.
Encryption in transit and workload mTLS	Networks	Generates or configures: creates <code>PeerAuthentication</code> resources (Chapter 4).	A service mesh, such as Istio, installed and managing the namespace. The mesh issues identities and encrypts traffic.
Egress control by name	Networks	Generates or configures: creates DNS-aware rules (Chapter 4).	A CNI plugin or egress gateway that supports FQDN rules.
Request-level authorization between services	Networks, Applications and Workloads	Enforces, with the Kyverno Authz Server (Chapter 3).	Envoy, Istio, or Envoy Gateway as the enforcement point.
Image provenance and signatures	Applications and Workloads	Enforces: verifies signatures and attestations at admission (Chapter 6).	A build system that signs images and produces attestations, such as Sigstore Cosign in CI, and a scanner that produces results. Kyverno verifies; it doesn't sign, scan, or build.
Pod security	Applications and Workloads	Enforces, and generates or configures: validates Pod settings, and labels namespaces for Pod Security Admission (Chapter 6).	Pod Security Admission, built into the API server, enforces the namespace label.

Runtime behavior detection and enforcement	Applications and Workloads	Not covered: Kyverno checks declared specs, not running processes.	A runtime security tool, such as Falco, Tetragon, or the optional Nirmata Runtime (Chapter 12).
Secrets management and encryption at rest	Data	Checks and reports: can flag secrets in environment variables and require external secret references.	An external secret store, such as Vault or a cloud secrets manager, the Secrets Store CSI Driver or External Secrets Operator, and KMS encryption of etcd.
Data classification and data loss prevention	Data	Checks and reports: can require a <code>data-class</code> label and act on it (Chapter 6).	A classification process and DLP tooling. Kyverno trusts the label you give it.
Policy results and posture metrics	Visibility and Analytics	Checks and reports: policy reports and Prometheus metrics (Chapter 8).	Policy Reporter, Prometheus, and dashboards.
API server audit logging	Visibility and Analytics	Not covered	The API server audit policy and log backend. Policy reports show what violates policy, not who did what.
SIEM, correlation, and alerting	Visibility and Analytics	Not covered: Kyverno exports data to these tools.	Your SIEM or observability platform.
Policy as code, GitOps delivery, CI testing	Automation and Orchestration	Enforces and checks: the Kyverno CLI in CI, and admission control in the cluster (Chapter 3).	Git, a CI system, and a GitOps controller.
Exceptions and compliance mapping	Governance	Enforces: governs <code>PolicyException</code> resources and annotates results (Chapters 7 and 8).	An approval workflow and your compliance team's control catalog.

Keep this table next to your architecture diagram. When a chapter says that Kyverno enforces a control, check this table to see which component completes it. Chapter 9 shows how these components fit together.

Other references

- The [NSA and CISA Kubernetes Hardening Guide](#) lists specific controls for Pods, networks, and authentication.
- The Kubernetes [Pod Security Standards](#) define the Privileged, Baseline, and Restricted profiles.
- The [CIS Kubernetes Benchmark](#) gives configuration checks you can map to policies.

KEY TAKEAWAYS

- NIST SP 800-207 defines the architecture, and the CISA maturity model defines the stages. Use both when you explain controls to auditors.
- For Kubernetes API requests, the API server is the policy enforcement point, and Kyverno is the policy decision point and policy administrator.
- Kyverno is the policy layer, not the whole architecture. Use the scope table in this chapter to see which component completes each control.
- Every chapter from here on maps to a CISA pillar, so you can score progress pillar by pillar.

03

CHAPTER 3

Policy as code, the Zero Trust control plane

Kyverno gives you one policy language to check, change, create, and delete resources in your pipeline, at admission, and in the background. This chapter explains why this guide uses Kyverno, introduces the policy types used throughout the guide, and shows you how to test them before they reach a cluster.

Why Kyverno

Kubernetes has built-in admission policy: `ValidatingAdmissionPolicy`, generally available since Kubernetes 1.30, and `MutatingAdmissionPolicy`, generally available since Kubernetes 1.36. Both use CEL and run inside the API server. Open Policy Agent (OPA) Gatekeeper is another widely used admission controller, with policies written in Rego. This guide uses Kyverno for the following reasons:

- **One policy, every stage.** The same `ValidatingPolicy` runs in CI with the Kyverno CLI, at admission, and in background scans of resources that already exist. In JSON mode, it also checks payloads outside Kubernetes, such as Terraform plans. Built-in admission policy runs only at admission.
- **Policy types beyond validation.** The controls in this guide create `NetworkPolicies` and `RoleBindings` (`GeneratingPolicy`), remove them when they expire (the cleanup controller and its `cleanup.kyverno.io/ttl` label), delete stale resources on a schedule (`DeletingPolicy`), and verify image signatures and attestations (`ImageValidatingPolicy`). Built-in admission policy only validates and mutates.
- **Governed exceptions and standard reports.** `PolicyException` resources scope a bypass to named policies and resources, and results land in `PolicyReport` resources in the open `wgpolicyk8s.io` format. Chapters 7 and 8 depend on both.

- **The same expression language as Kubernetes.** Kyverno policies use CEL, so expressions and skills carry over from built-in admission policy. A `ValidatingPolicy` can also generate a `ValidatingAdmissionPolicy` (`spec.autogen.validatingAdmissionPolicy.enabled: true`), so the API server evaluates the same check without calling a webhook.

Kyverno is one layer of the architecture, not all of it. Chapter 2 lists what it covers and what it doesn't. Kyverno runs as an admission webhook, so plan for availability and choose its failure policy deliberately. For guidance, see [High availability](#) in the Kyverno documentation.

Kyverno policy types

All policy types use the `policies.kyverno.io/v1` API and CEL expressions.

Policy type	What it does	Zero Trust role
ValidatingPolicy	Allows, denies, or audits requests and payloads.	Enforce explicit rules before access.
MutatingPolicy	Changes resources to add secure defaults.	Remove insecure defaults automatically.
GeneratingPolicy	Creates related resources when a trigger occurs.	Build segmentation and access boundaries.
ImageValidatingPolicy	Verifies image signatures and attestations.	Trust only verified software.
DeletingPolicy	Deletes matching resources on a schedule.	Remove expired access and stale objects.
PolicyException	Exempts specific resources from a policy.	Grant narrow, governed exceptions.

`GeneratingPolicy` also supports an inline YAML template, which is in beta in Kyverno 1.19. Instead of building the downstream resource as a CEL object with `dyn()`, you write it as plain YAML under `spec.generate[].template.value` and interpolate variables with `(( expression ))`. Chapters 4 and 5 use this mode throughout because the templates are shorter to read and maintain.

Most policy types also have namespaced versions, such as

`NamespacedValidatingPolicy`, `NamespacedMutatingPolicy`, and `NamespacedGeneratingPolicy`. Use them to let tenant teams manage their own policies without cluster-admin access.

Write your first policy

The following policy requires every Pod to carry an `owner` label. Zero Trust depends on knowing who owns each workload, so this is a useful first rule.

```
apiVersion: policies.kyverno.io/v1
kind: ValidatingPolicy
metadata:
  name: require-owner-label
spec:
  evaluation:
    background:
      enabled: true
  validationActions:
    - Audit
  matchConstraints:
    resourceRules:
      - apiGroups: ['']
        apiVersions: [v1]
        operations: [CREATE, UPDATE]
        resources: [pods]
  validations:
    - message: "Every Pod needs an 'owner' label."
      expression: "'owner' in object.metadata.labels.orValue({})"
```

To apply and check the policy, do the following:

1. Save the policy as `require-owner-label.yaml`.
2. Apply it:

```
kubectl apply -f require-owner-label.yaml
```

3. Create a Pod without the label:

```
kubectl run test --image=nginx
```

4. View the audit result:

```
kubectl get policyreports -A
```

SHELL

When the reports show no unexpected failures, change `Audit` to `Deny`.

Test policies in CI

Use the Kyverno CLI to test policies against manifests before you merge them. This catches problems early and keeps the cluster policy and the pipeline policy identical.

```
kyverno apply require-owner-label.yaml --resource pod.yaml
```

SHELL

For repeatable tests, write a `kyverno-test.yaml` file that lists policies, resources, and expected results, and then run `kyverno test .` in your pipeline.

Check infrastructure code with JSON mode

Zero Trust applies to the infrastructure around your cluster too. A `ValidatingPolicy` in JSON mode checks any JSON payload, such as a Terraform plan. JSON-mode policies run in the Kyverno CLI and in your pipelines, not in the cluster: they have no `matchConstraints`, so the admission controller rejects them. Keep them in your pipeline repository. They cover much of what the separate `kyverno-json` project does, with the same policy type that you use in the cluster.

The following policy fails any Terraform plan that creates or changes an Amazon EKS cluster (`aws_eks_cluster`) with a public API endpoint. A plan lists its resources under `planned_values.root_module.resources`, and `vpc_config` is a list, so the policy checks every `vpc_config` block. An unset `endpoint_public_access` counts as public, because public is the AWS default.

```
apiVersion: policies.kyverno.io/v1
kind: ValidatingPolicy
metadata:
  name: check-public-endpoint
spec:
  evaluation:
    mode: JSON
  matchConditions:
    - name: is-terraform-plan
      expression: 'has(object.planned_values) && has(object.terraform_version)'
  variables:
    - name: eksClusters
      expression: >-
        object.planned_values.root_module.?resources.orValue([])
        .filter(r, r.type == 'aws_eks_cluster')
  validations:
    - message: 'EKS clusters must set vpc_config.endpoint_public_access to
      false.'
      expression: >-
        variables.eksClusters.all(c,
          c.values.?vpc_config.orValue([]).all(v,
            v.?endpoint_public_access.orValue(true) == false))
```

To run the policy, export the plan as JSON and pass it to the Kyverno CLI with the `--json` flag:

```
terraform plan -out tfplan.binary
terraform show -json tfplan.binary > tfplan.json
kyverno apply check-public-endpoint.yaml --json tfplan.json
```

The command exits with a non-zero status when the plan fails the policy, so the pipeline step fails.

Why enforce in the cluster, not only in pipelines

GitOps pipelines are a good place to catch mistakes, but they aren't a security boundary. Attackers don't use your pipeline. Anyone with API access can run `kubectl apply` and skip the pipeline. Kyverno runs as an admission controller, so it checks every request that reaches the API server and matches its webhook, whatever its source. Kyverno can't protect against direct access to etcd or nodes, or against requests in namespaces excluded from its webhook, such as `kube-system` by default. Hardening the control plane is a separate task. Use all three layers:

- **Pipeline checks** give fast feedback to developers.
- **Admission checks** stop requests that skip the pipeline.
- **Background scans** find resources that existed before a policy did.

Extend policy to application requests

The [Kyverno Authz Server](#) applies Kyverno policies to HTTP requests through Envoy external authorization. Use it with Istio or Envoy Gateway to authorize each service-to-service call by identity, method, and path. For a walkthrough, see [Policy based authorization using Kyverno](#) on the Istio blog.

CHECKLIST

- ☐ Install Kyverno 1.19 or later.
- ☐ Add `kyverno test` to your CI pipeline.
- ☐ Add JSON-mode policies for Terraform or other infrastructure code.
- ☐ Roll out each policy in `Audit` before `Deny`.

04

CHAPTER 4

Micro-segment the network

Start every namespace at default deny. Then let Kyverno read each namespace's labels and generate only the traffic rules that the namespace needs. Kyverno creates these rules inside the cluster when someone creates the namespace, so no new namespace starts open, and with synchronization on, teams can't remove the baseline. Kyverno writes the rules, and your CNI plugin enforces them. If your CNI plugin doesn't implement NetworkPolicy, the generated objects have no effect, so confirm enforcement with a connectivity test before you rely on this chapter. For namespaces that already exist, see the note after the first policy.

Why generate network policy in the cluster

- **No gaps for new namespaces.** The NetworkPolicy exists the moment the namespace exists, and the CNI plugin starts enforcing it.
- **No drift.** With synchronization turned on, Kyverno restores a generated policy if someone edits or deletes it.
- **Metadata-driven.** Labels such as `tier` or `workspace` decide which rules apply, so teams request access by labeling, not by writing network YAML.

Generate default deny for every namespace

The following `GeneratingPolicy` creates two NetworkPolicies in every new namespace:

- `default-deny` blocks all ingress and egress.
- `allow-dns` allows egress to cluster DNS, which almost every workload needs.

```
apiVersion: policies.kyverno.io/v1
```

YAML

```

kind: GeneratingPolicy
metadata:
  name: generate-default-deny
spec:
  evaluation:
    synchronize:
      enabled: true
  matchConstraints:
    resourceRules:
      - apiGroups: ['']
        apiVersions: ['v1']
        operations: ['CREATE']
        resources: ['namespaces']
  matchConditions:
    - name: skip-system-namespaces
      expression: "!(object.metadata.name in ['kube-system', 'kube-public',
'kube-node-lease', 'kyverno'])"
  variables:
    - name: nsName
      expression: 'object.metadata.name'
  generate:
    - template:
        interpolate: cel
        value: |
          apiVersion: networking.k8s.io/v1
          kind: NetworkPolicy
          metadata:
            name: default-deny
            namespace: (( variables.nsName ))
          spec:
            podSelector: {}
            policyTypes:
              - Ingress
              - Egress
          ---
          apiVersion: networking.k8s.io/v1
          kind: NetworkPolicy
          metadata:
            name: allow-dns
            namespace: (( variables.nsName ))
          spec:
            podSelector: {}
            policyTypes:
              - Egress
            egress:
              - to:
                  - namespaceSelector:
                      matchLabels:
                        kubernetes.io/metadata.name: kube-system

```

```
ports:
  - protocol: UDP
    port: 53
  - protocol: TCP
    port: 53
```

To verify the policy, do the following:

1. Create a namespace with the labels that `require-workspace-labels`, later in this chapter, requires:

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Namespace
metadata:
  name: team-a
  labels:
    workspace: team-a
    tier: frontend
EOF
```

SHELL

2. List its network policies:

```
kubectl get networkpolicy -n team-a
```

SHELL

The output lists `default-deny` and `allow-dns`.

3. Delete the `default-deny` policy:

```
kubectl delete networkpolicy default-deny -n team-a
```

SHELL

4. List the network policies again:

```
kubectl get networkpolicy -n team-a
```

SHELL

The output still lists `default-deny`, because Kyverno recreated it.

NOTE

As written, the policy applies only to namespaces created after it. To cover existing namespaces, set `spec.evaluation.generateExisting.enabled: true`, or apply the same NetworkPolicies through GitOps. Roll out one team at a time, because default deny blocks traffic the moment it's created. Chapter 13 schedules this for days 31 to 60.

Build Namespace-as-a-Service boundaries

In a multi-tenant cluster, each tenant, or workspace, often owns several namespaces. Namespaces in the same workspace must be able to reach each other, and namespaces in different workspaces must not.

The following `GeneratingPolicy` creates an `allow-workspace-traffic` `NetworkPolicy` in every new namespace that has a `workspace` label. It allows ingress from, and egress to, any namespace with the same `workspace` value. It needs both directions because `generate-default-deny` blocks egress too. Use it with `generate-default-deny`: together, the two policies give each tenant a closed boundary.

```
apiVersion: policies.kyverno.io/v1
kind: GeneratingPolicy
metadata:
  name: generate-workspace-network-policy
spec:
  evaluation:
    synchronize:
      enabled: true
  matchConstraints:
    resourceRules:
      - apiGroups: ['']
        apiVersions: ['v1']
        operations: ['CREATE']
        resources: ['namespaces']
  matchConditions:
    - name: has-workspace-label
      expression: "object.metadata.labels.workspace.orValue('') != ''"
  variables:
    - name: nsName
      expression: 'object.metadata.name'
    - name: workspace
      expression: "object.metadata.labels['workspace']"
  generate:
    - template:
        interpolate: cel
        value: |
          apiVersion: networking.k8s.io/v1
          kind: NetworkPolicy
          metadata:
            name: allow-workspace-traffic
            namespace: (( variables.nsName ))
          spec:
```

YAML

```
policyTypes:
  - Ingress
  - Egress
ingress:
  - from:
      namespaceSelector:
        matchLabels:
          workspace: (( variables.workspace ))
egress:
  - to:
      namespaceSelector:
        matchLabels:
          workspace: (( variables.workspace ))
```

To onboard a tenant namespace, create it with its labels in the same request, so that the labels exist when the policy runs:

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Namespace
metadata:
  name: payments-api
  labels:
    workspace: payments
    tier: backend
EOF
```

SHELL

NOTE

The policy runs only when a namespace is created. Adding a `workspace` label to an existing namespace doesn't generate the rule.

Segment by application tier

Tier labels let you enforce a classic three-tier design: frontends accept traffic from the ingress controller, and backends accept traffic only from frontends. The following `GeneratingPolicy` creates a `CiliumNetworkPolicy` for every namespace labeled `tier=backend`.

```
apiVersion: policies.kyverno.io/v1
kind: GeneratingPolicy
metadata:
```

YAML

```

spec:
  evaluation:
    synchronize:
      enabled: true
  matchConstraints:
    resourceRules:
      - apiGroups: ['']
        apiVersions: ['v1']
        operations: ['CREATE', 'UPDATE']
        resources: ['namespaces']
  matchConditions:
    - name: is-backend-tier
      expression: "object.metadata.labels.tier.orValue('') == 'backend'"
    - name: has-workspace
      expression: "object.metadata.labels.workspace.orValue('') != ''"
  variables:
    - name: nsName
      expression: 'object.metadata.name'
    - name: workspace
      expression: 'object.metadata.labels.workspace'
  generate:
    - template:
        interpolate: cel
        value: |
          apiVersion: cilium.io/v2
          kind: CiliumNetworkPolicy
          metadata:
            name: backend-ingress-from-frontend
            namespace: (( variables.nsName ))
          spec:
            endpointSelector: {}
            ingress:
              - fromEndpoints:
                  - matchLabels:
                      "k8s:io.cilium.k8s.namespace.labels.tier": frontend
                      "k8s:io.cilium.k8s.namespace.labels.workspace": ((
variables.workspace ))

```

The rule matches frontends by both `tier` and `workspace`, so a frontend in one workspace can't reach a backend in another. This keeps the Namespace-as-a-Service boundary from the previous section. The policy skips backend namespaces that have no `workspace` label, and `require-workspace-labels`, later in this chapter, makes sure every namespace has one.

The Kyverno background controller needs permission to manage `CiliumNetworkPolicy` resources. Add the following aggregated ClusterRole:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: kyverno:background-cilium
  labels:
    rbac.kyverno.io/aggregate-to-background-controller: 'true'
rules:
  - apiGroups: [cilium.io]
    resources: [ciliumnetworkpolicies]
    verbs: [get, list, watch, create, update, patch, delete]
```

YAML

Inherit segmentation labels onto every Pod

Generated NetworkPolicy and CiliumNetworkPolicy rules select Pods by label, usually `workspace` and `tier`. If developers set those labels by hand, someone eventually forgets a label or sets the wrong value, and that Pod falls outside every segmentation rule. Instead, copy the labels from the namespace automatically.

Require the labels on the namespace first, so a namespace can't exist without declaring where it belongs:

```
apiVersion: policies.kyverno.io/v1
kind: ValidatingPolicy
metadata:
  name: require-workspace-labels
spec:
  validationActions: [Deny]
  evaluation:
    background:
      enabled: true
  matchConstraints:
    resourceRules:
      - apiGroups: ['']
        apiVersions: ['v1']
        operations: ['CREATE', 'UPDATE']
        resources: ['namespaces']
  matchConditions:
    - name: skip-system-namespaces
      expression: >-
        !(object.metadata.name in ['default', 'kube-system', 'kube-public',
          'kube-node-lease', 'local-path-storage', 'kyverno', 'kyverno-
exceptions',
          'cilium-secrets'])
  variables:
```

YAML

```

    expression: 'object.metadata.?labels.orValue({})'
  validations:
    - expression: "'workspace' in variables.labels &&
      variables.labels['workspace'] != ''"
      message: 'The namespace must carry a non-empty workspace label.'
    - expression: "'tier' in variables.labels && variables.labels['tier'] in
      ['frontend', 'backend', 'data']"
      message: 'The namespace must carry a tier label set to frontend, backend,
      or data.'

```

Then mutate every new Pod to carry its namespace's `workspace` and `tier` labels. The following policy uses the `resource.Get()` CEL function to look up the parent namespace:

```

apiVersion: policies.kyverno.io/v1
kind: MutatingPolicy
metadata:
  name: add-workload-labels
spec:
  autogen:
    podControllers:
      controllers: []
  evaluation:
    admission:
      enabled: true
    mutateExisting:
      enabled: false
  matchConstraints:
    resourceRules:
      - apiGroups: ['']
        apiVersions: ['v1']
        operations: ['CREATE']
        resources: ['pods']
  variables:
    - name: nsLabels
      expression: >-
        resource.Get('v1', 'namespaces', '', object.metadata.namespace)
        .metadata.?labels.orValue({})
    - name: workspace
      expression: "'workspace' in variables.nsLabels ?
variables.nsLabels['workspace'] : 'default'"
    - name: tier
      expression: "'tier' in variables.nsLabels ? variables.nsLabels['tier'] :
'default'"
  mutations:
    - patchType: ApplyConfiguration

```

```
expression: >-
  Object{
    metadata: Object.metadata{
      labels: {
        "workspace": variables.workspace,
        "tier": variables.tier
      }
    }
  }
}
```

By default, Kyverno also applies Pod policies to Pod controllers, such as Deployments and DaemonSets. This feature is called `autogen`. This policy turns `autogen` off (`controllers: []`), so Kyverno doesn't rewrite a controller's Pod template. The Pods that a controller creates still pass through admission, so Kyverno labels them there. If a Deployment selects its Pods by `tier` or `workspace`, set those labels in its Pod template to the namespace's values. Otherwise, the policy overwrites them, and the Deployment no longer matches its own Pods.

Developers never need to set `workspace` or `tier` on a Pod, and at creation they can't set them to a different value. Cilium turns Pod labels into identities, so the identity matches the namespace the Pod runs in. The mutating policy runs on `CREATE` only. To stop a later `kubectl label` command from changing the identity, add a `ValidatingPolicy` on Pod `UPDATE` that denies changes to these two labels, using the same pattern as `protect-segmentation-labels` later in this chapter.

These two policies are adapted from the [add-workload-labels](#) and [require-workspace-labels](#) policies in Nirmata's [network policy automation demo](#).

Pin images to their approved tier

Segmentation fails if someone deploys a backend database image into a frontend namespace. The following policy allows images from `registry.example.com/backend/` only in namespaces labeled `tier=backend`. It checks containers, init containers, and ephemeral containers.

```
apiVersion: policies.kyverno.io/v1
kind: ValidatingPolicy
metadata:
  name: restrict-backend-images-to-backend-tier
spec:
  validationActions:
    - Deny
```

YAML

```
resourceRules:
  - apiGroups: ['']
    apiVersions: [v1]
    operations: [CREATE, UPDATE]
    resources: [pods]
variables:
  - name: tier
    expression: "namespaceObject.metadata.labels.tier.orValue('')"
  - name: allContainers
    expression: >-
      object.spec.containers +
      object.spec.initContainers.orValue([]) +
      object.spec.ephemeralContainers.orValue([])
  - name: usesBackendImage
    expression: >-
      variables.allContainers.exists(c,
        c.image.startsWith('registry.example.com/backend/'))
validations:
  - message: 'Backend images can run only in namespaces labeled tier=backend.'
    expression: '!variables.usesBackendImage || variables.tier == "backend"'
```

Protect the namespace label that drives policy

If tenants can change the `tier` or `workspace` label, they can change their own network access. The following policy lets only the platform team change those labels.

```
apiVersion: policies.kyverno.io/v1
kind: ValidatingPolicy
metadata:
  name: protect-segmentation-labels
spec:
  validationActions:
    - Deny
  matchConstraints:
    resourceRules:
      - apiGroups: ['']
        apiVersions: [v1]
        operations: [UPDATE]
        resources: [namespaces]
  variables:
    - name: labelsChanged
      expression: >-
        ['tier', 'workspace'].exists(k,
          object.metadata.labels[k].orValue('') != oldObject.metadata.labels[k].orValue(''))
  validations:
    - message: 'Only the platform team can change tier or workspace labels.'
      expression: '!variables.labelsChanged || 'platform-admins' in request.userInfo.groups"
```

Encrypt service-to-service traffic with mTLS

NetworkPolicy decides which workloads can connect. It doesn't prove who is on the other end or encrypt the traffic. A service mesh does both: it issues each workload a certificate and encrypts traffic between proxies. Kyverno's part is to make strict mTLS the default.

If you run Istio, the following `GeneratingPolicy` creates a strict `PeerAuthentication` in every new namespace, so that only mesh workloads with valid identities can connect. The setting takes effect only in namespaces that Istio manages, through sidecar injection or ambient mode. The policy is separate from `generate-default-deny`, so clusters without Istio can still use default deny.

```
apiVersion: policies.kyverno.io/v1
kind: GeneratingPolicy
metadata:
  name: generate-strict-mtls
spec:
  evaluation:
```

```

    enabled: true
  matchConstraints:
    resourceRules:
      - apiGroups: ['']
        apiVersions: ['v1']
        operations: ['CREATE']
        resources: ['namespaces']
    matchConditions:
      - name: skip-system-namespaces
        expression: "!(object.metadata.name in ['kube-system', 'kube-public',
        'kube-node-lease', 'kyverno', 'istio-system'])"
  variables:
    - name: nsName
      expression: 'object.metadata.name'
  generate:
    - template:
        interpolate: cel
        value: |
          apiVersion: security.istio.io/v1
          kind: PeerAuthentication
          metadata:
            name: default
            namespace: (( variables.nsName ))
          spec:
            mtls:
              mode: STRICT

```

The Kyverno background controller needs permission to manage `PeerAuthentication` resources. Add the following aggregated ClusterRole:

```

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: kyverno:background-istio
  labels:
    rbac.kyverno.io/aggregate-to-background-controller: 'true'
rules:
  - apiGroups: [security.istio.io]
    resources: [peerauthentications]
    verbs: [get, list, watch, create, update, patch, delete]

```

YAML

To make sure that every namespace is enrolled in the mesh, you can also add a `ValidatingPolicy` that requires the Istio enrollment label on each namespace.

Control egress

Default deny also blocks egress, which stops data exfiltration and callbacks to attacker servers. When a workload needs an external service, allow it by name instead of opening all egress. For example, Cilium supports DNS-aware rules with `toFQDNs`. Generate these rules from a namespace annotation that the platform team approves.

CHECKLIST

- ☐ Generate `default-deny` and `allow-dns` in every new namespace.
- ☐ Turn on synchronization so Kyverno restores deleted policies.
- ☐ Generate workspace or tier rules from namespace labels.
- ☐ Restrict who can change segmentation labels.
- ☐ Enforce mTLS with a service mesh.
- ☐ Allow egress by name only.
- ☐ If you use Cilium or Istio, give the Kyverno background controller permission to manage their policy resources.
- ☐ Confirm with a connectivity test that your CNI plugin enforces the generated rules.

For more samples, see the [Kyverno policy library](#), including `add-network-policy` and `add-networkpolicy-dns`.

05

CHAPTER 5

Enforce least-privilege access

Grant no standing privilege. Block risky RBAC with validating policies, and remove default credentials with mutating policies. Give elevated access only on request, for a fixed time, and remove it automatically. This chapter covers each step with a policy you can apply.

Block risky RBAC

Three RBAC patterns cause most privilege problems: wildcard rules, bindings to `cluster-admin`, and verbs that let a user raise their own privileges.

Block wildcard rules

The following policy denies any Role or ClusterRole that uses `*` in `verbs` or `resources`:

```
apiVersion: policies.kyverno.io/v1
kind: ValidatingPolicy
metadata:
  name: restrict-wildcard-rbac
spec:
  validationActions:
    - Deny
  matchConstraints:
    resourceRules:
      - apiGroups: [rbac.authorization.k8s.io]
        apiVersions: [v1]
        operations: [CREATE, UPDATE]
        resources: [roles, clusterroles]
  validations:
    - message: 'Roles must not use * in verbs or resources.'
      expression: >-
        !object.?rules.orValue([]).exists(r,
          '*' in r.?verbs.orValue([]) || '*' in r.?resources.orValue([]))
```

YAML

Block bindings to `cluster-admin`

The following policy lets only members of the `platform-admins` group bind the `cluster-admin` ClusterRole:

```
apiVersion: policies.kyverno.io/v1
kind: ValidatingPolicy
metadata:
  name: restrict-cluster-admin-binding
spec:
  validationActions:
    - Deny
  matchConstraints:
    resourceRules:
      - apiGroups: [rbac.authorization.k8s.io]
        apiVersions: [v1]
        operations: [CREATE, UPDATE]
        resources: [rolebindings, clusterrolebindings]
  validations:
    - message: 'Only platform admins can bind cluster-admin.'
      expression: >-
        object.roleRef.name != 'cluster-admin' ||
        'platform-admins' in request.userInfo.groups
```

YAML

Block privilege-escalation verbs

The `escalate`, `bind`, and `impersonate` verbs let a user grant themselves more access. The following policy reserves them for the platform team:

```

apiVersion: policies.kyverno.io/v1
kind: ValidatingPolicy
metadata:
  name: restrict-escalation-verbs
spec:
  validationActions:
    - Deny
  matchConstraints:
    resourceRules:
      - apiGroups: [rbac.authorization.k8s.io]
        apiVersions: [v1]
        operations: [CREATE, UPDATE]
        resources: [roles, clusterroles]
  validations:
    - message: 'The escalate, bind, and impersonate verbs are reserved for
      platform admins.'
      expression: >-
        !object.?rules.orValue([]).exists(r,
          r.?verbs.orValue([]).exists(v, v in ['escalate', 'bind',
            'impersonate']))
        || 'platform-admins' in request.userInfo.groups

```

These three policies also run in background scans, which need the Kyverno reports controller to read RBAC objects. Without the following aggregated ClusterRole, the policies report `ready: false` with a "missing permissions" message:

```

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: kyverno:reports-rbac-resources
  labels:
    rbac.kyverno.io/aggregate-to-reports-controller: 'true'
rules:
  - apiGroups: [rbac.authorization.k8s.io]
    resources: [roles, clusterroles, rolebindings, clusterrolebindings]
    verbs: [get, list, watch]

```

NOTE

Some operators and Helm charts, possibly including your own platform tools, ship ClusterRoles with wildcards. Before you switch `restrict-wildcard-rbac` to `Deny`, exclude their installers by username or group in `matchConditions`, or grant a governed exception (Chapter 7).

Harden workload identity

Every Pod gets an identity through its service account. By default, Kubernetes mounts a token for that account into the Pod, even when the Pod never calls the API.

Stop mounting tokens by default

The following `MutatingPolicy` sets `automountServiceAccountToken: false` on every `ServiceAccount` that doesn't set it, including the `default` `ServiceAccount` that Kubernetes creates in each new namespace. Pods inherit the setting from their `ServiceAccount`, so a Pod gets no API token unless it sets `automountServiceAccountToken: true` itself.

The policy changes `ServiceAccounts`, not `Pods`, because the built-in `ServiceAccount` admission plugin mounts the token volume before admission webhooks run. A Pod-level mutation sets the field but leaves the token mounted.

```
apiVersion: policies.kyverno.io/v1
kind: MutatingPolicy
metadata:
  name: disable-token-automount
spec:
  matchConstraints:
    resourceRules:
      - apiGroups: ['']
        apiVersions: ['v1']
        resources: ['serviceaccounts']
        operations: ['CREATE', 'UPDATE']
  matchConditions:
    - name: not-set
      expression: '!has(object.automountServiceAccountToken)'
  mutations:
    - patchType: ApplyConfiguration
      applyConfiguration:
        expression: >-
          Object{
            automountServiceAccountToken: false
          }
```

YAML

`ServiceAccounts` that existed before you applied the policy keep their setting until they're next updated.

IMPORTANT

Plan for platform tools before you turn this policy on. Every ServiceAccount created after it's active gets `automountServiceAccountToken: false`, including those of operators such as cert-manager, Argo CD, Policy Reporter, and the Prometheus operator, and most Helm charts don't opt back in at the Pod level. Those operators then can't reach the API server. Either exclude your platform namespaces in the policy's `matchConditions`, or set `automountServiceAccountToken: true` on each operator's ServiceAccount or Pods in its Helm values. The policy leaves a ServiceAccount alone if it sets the field explicitly.

Require a dedicated service account

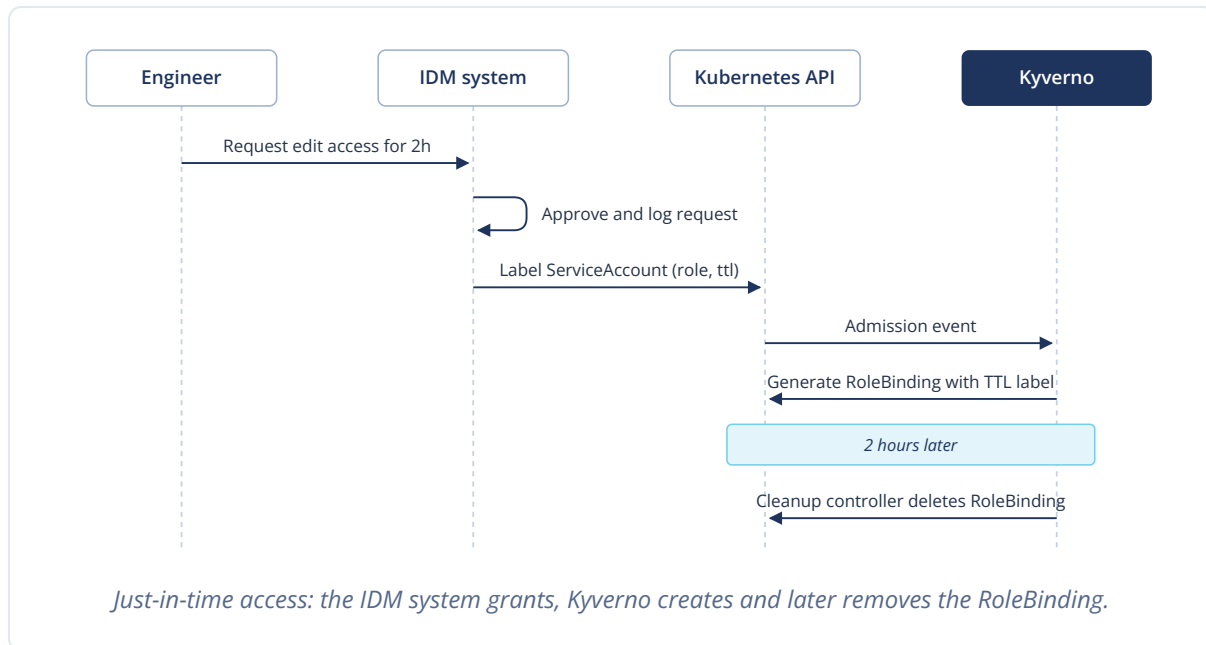
When all Pods share the `default` service account, you can't tell workloads apart or give each one least privilege. The following policy requires every Pod to name a dedicated service account:

```
apiVersion: policies.kyverno.io/v1
kind: ValidatingPolicy
metadata:
  name: disallow-default-serviceaccount
spec:
  validationActions:
    - Deny
  matchConstraints:
    resourceRules:
      - apiGroups: ['']
        apiVersions: [v1]
        operations: [CREATE, UPDATE]
        resources: [pods]
  validations:
    - message: 'Use a dedicated service account, not default.'
      expression: "object.spec.?serviceAccountName.orValue('default') != 'default'"
```

YAML

Grant just-in-time access

Just-in-time (JIT) access replaces standing permissions with short grants that expire on their own. The following design uses your identity management (IDM) system to approve requests and Kyverno to create and remove the access.



The IDM system approves the request and labels the requester's ServiceAccount. Kyverno creates the RoleBinding, and the Kyverno cleanup controller deletes the RoleBinding when the time to live (TTL) in its `cleanup.kyverno.io/ttl` label expires.

Kyverno's role here is the policy administrator from Chapter 2: it creates and removes the access path. Your identity provider authenticates the engineer and enforces MFA, the IDM system makes and records the approval decision, and Kubernetes RBAC authorizes each request that the engineer makes with the binding. The sample binds a ServiceAccount because Kyverno can watch that resource. To bind a human user or group instead, have the IDM system create a request resource, such as a ConfigMap or a custom resource, and change the trigger to match it.

Step 1: Allow only the IDM system to set access labels

If anyone can set the labels, anyone can grant themselves access. The following policy allows only the IDM controller's identity to set them, and limits the TTL and the role that a request can ask for:

```

apiVersion: policies.kyverno.io/v1
kind: ValidatingPolicy
metadata:
  name: restrict-jit-labels
spec:
  validationActions:

```

YAML

```

matchConstraints:
  resourceRules:
    - apiGroups: ['']
      apiVersions: [v1]
      operations: [CREATE, UPDATE]
      resources: [serviceaccounts]
  variables:
    - name: jitLabels
      expression: "['access.example.com/role', 'access.example.com/ttl']"
    - name: changed
      expression: >-
        variables.jitLabels.exists(k,
          object.metadata.labels[k].orValue('') !=
            (request.operation == 'UPDATE' ? oldObject.metadata.labels[k].orValue('') : ''))
  validations:
    - message: 'Only the IDM controller can set JIT access labels.'
      expression: >-
        !variables.changed ||
        request.userInfo.username == 'system:serviceaccount:ids:idm-controller'
    - message: 'TTL must be 1h, 2h, or 4h, and role must be view or edit.'
      expression: >-
        !('access.example.com/ttl' in object.metadata.labels.orValue({})) ||
        (object.metadata.labels['access.example.com/ttl'] in ['1h', '2h', '4h'])
    - message: 'Role must be view or edit.'
      expression: >-
        object.metadata.labels['access.example.com/role'].orValue('') in
        ['view', 'edit']

```

Step 2: Generate a RoleBinding that expires

The following `GeneratingPolicy` creates a `RoleBinding` for the labeled `ServiceAccount` and copies the requested TTL to the `RoleBinding`'s `cleanup.kyverno.io/ttl` label:

```

apiVersion: policies.kyverno.io/v1
kind: GeneratingPolicy
metadata:
  name: generate-jit-rolebinding
spec:
  evaluation:
    synchronize:
      enabled: false
  matchConstraints:
    resourceRules:
      - apiGroups: ['']
        apiVersions: ['v1']
        operations: ['CREATE', 'UPDATE']
        resources: ['serviceaccounts']

```

YAML

```

matchConditions:
  - name: has-jit-request
    expression: "'access.example.com/ttl' in object.metadata.?
labels.orValue({})"
variables:
  - name: ns
    expression: 'object.metadata.namespace'
  - name: role
    expression: "object.metadata.labels['access.example.com/role']"
  - name: ttl
    expression: "object.metadata.labels['access.example.com/ttl']"
generate:
  - template:
    interpolate: cel
    value: |
      apiVersion: rbac.authorization.k8s.io/v1
      kind: RoleBinding
      metadata:
        name: (( 'jit-' + object.metadata.name ))
        namespace: (( variables.ns ))
        labels:
          cleanup.kyverno.io/ttl: (( variables.ttl ))
      roleRef:
        apiGroup: rbac.authorization.k8s.io
        kind: ClusterRole
        name: (( variables.role ))
      subjects:
        - kind: ServiceAccount
          name: (( object.metadata.name ))
          namespace: (( variables.ns ))

```

IMPORTANT

Keep synchronization turned off for JIT access. With synchronization on, Kyverno would recreate the RoleBinding after the cleanup controller deletes it.

Step 3: Give Kyverno the permissions that JIT access needs

The default Kyverno installation can't run this workflow. Add two aggregated ClusterRoles.

The background controller generates the RoleBinding. It must be able to read RoleBindings, and it needs the `bind` verb on the `view` and `edit` ClusterRoles: Kubernetes RBAC lets a client create a RoleBinding to a ClusterRole only if the client already holds every permission in that role or has `bind` on it.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: kyverno:background-jit-rolebindings
  labels:
    rbac.kyverno.io/aggregate-to-background-controller: 'true'
rules:
  - apiGroups: [rbac.authorization.k8s.io]
    resources: [rolebindings]
    verbs: [get, list, watch]
  - apiGroups: [rbac.authorization.k8s.io]
    resources: [clusterroles]
    verbs: [bind]
    resourceNames: [view, edit]
```

The cleanup controller deletes any resource that carries a `cleanup.kyverno.io/ttl` label once the TTL passes, if the controller can list, watch, and delete that kind of resource. The TTL counts from the resource's creation time. The label accepts durations, such as `2h` or `7d`, and absolute UTC times in a label-safe format without colons, such as `2026-10-31T170000Z` or `2026-10-31`. The following ClusterRole lets the controller delete RoleBindings:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: kyverno:cleanup-rolebindings
  labels:
    rbac.kyverno.io/aggregate-to-cleanup-controller: 'true'
rules:
  - apiGroups: [rbac.authorization.k8s.io]
    resources: [rolebindings]
    verbs: [get, list, watch, delete]
```

Apply both ClusterRoles as a member of `platform-admins`. The first one uses the `bind` verb, so `restrict-escalation-verbs` denies it for anyone else.

Configure the IDM system to remove the labels from the ServiceAccount when access ends, so that the next request starts clean.

Case study: Fastly break-glass access

Fastly's internal platform uses Kyverno for guardrails and for time-scoped break-glass access during incidents, with audit logging of every elevated session. The pattern keeps incident response fast while meeting two Zero Trust tenets: access is per session, and every elevated action is observed. Chapter 11 describes the platform in more detail.

CHECKLIST

- ☐ Block wildcard rules, `cluster-admin` bindings, and escalation verbs.
- ☐ Stop mounting service account tokens by default.
- ☐ Require a dedicated service account for each workload.
- ☐ Replace standing admin access with JIT RoleBindings.
- ☐ Restrict who can request JIT access, and cap the TTL.
- ☐ Give the Kyverno background, cleanup, and reports controllers the extra RBAC permissions in this chapter.
- ☐ Send Kubernetes audit logs for elevated sessions to your SIEM.

For more samples, see `restrict-wildcard-verbs`, `restrict-binding-clusteradmin`, and `restrict-escalation-verbs-roles` in the [Kyverno policy library](#).

06

CHAPTER 6

Verify workloads and the software supply chain

Run only images that come from a trusted registry, carry a valid signature from your build system, and pass your vulnerability checks. Then run them with the fewest privileges possible. Your build pipeline signs, scans, and attests, and Kyverno verifies the results at admission, so it rejects an unverified or over-privileged workload before it starts. Kyverno doesn't sign images, run scanners, or build software. For Pod security, Kyverno works alongside Pod Security Admission, which is built into the API server.

Allow only trusted registries

The following policy allows images only from two approved registries. It checks containers, init containers, and ephemeral containers:

```
apiVersion: policies.kyverno.io/v1
kind: ValidatingPolicy
metadata:
  name: restrict-image-registries
spec:
  evaluation:
    background:
      enabled: true
  validationActions:
    - Deny
  matchConstraints:
    resourceRules:
      - apiGroups: ['']
        apiVersions: [v1]
        operations: [CREATE, UPDATE]
        resources: [pods]
  variables:
```

YAML

```
expression: >-
  object.spec.containers +
  object.spec.?initContainers.orValue([]) +
  object.spec.?ephemeralContainers.orValue([])
validations:
- message: 'Images must come from ghcr.io/myorg or registry.example.com.'
  expression: >-
    variables.allContainers.all(c,
      c.image.startsWith('ghcr.io/myorg/') ||
      c.image.startsWith('registry.example.com/'))
```

Through autogen (Chapter 4), Kyverno applies this Pod policy to Pod controllers, such as Deployments and StatefulSets. Users see the error when they create the Deployment, not later when the Pod fails.

NOTE

`kubectl debug` adds ephemeral containers through the `pods/ephemeralcontainers` subresource, which rules on `pods` don't match at admission. Background scans still check `ephemeralContainers` and report a debug container from an unapproved registry. If you add the subresource to `matchConstraints` , Kyverno turns off autogen for the policy.

Require image digests

Tags such as `:latest` can point to different images over time. A digest identifies exactly one image. The following policy requires every container and init container to reference its image by digest:

```
apiVersion: policies.kyverno.io/v1
kind: ValidatingPolicy
metadata:
  name: require-image-digest
spec:
  validationActions:
    - Deny
  matchConstraints:
    resourceRules:
      - apiGroups: ['']
        apiVersions: [v1]
        operations: [CREATE, UPDATE]
        resources: [pods]
  variables:
    - name: allContainers
      expression: 'object.spec.containers + object.spec.?
        initContainers.orValue([])'
  validations:
    - message: 'Reference images by digest, for example image@sha256:...'
      expression: "variables.allContainers.all(c, c.image.contains('@sha256:'))"
```

Verify signatures and attestations

A signature proves who built an image. An attestation proves facts about how it was built or tested, such as its provenance, software bill of materials (SBOM), or vulnerability scan. The following `ImageValidatingPolicy` checks both, using keyless Sigstore signing from GitHub Actions.

```
apiVersion: policies.kyverno.io/v1
kind: ImageValidatingPolicy
metadata:
  name: verify-signed-images
spec:
  validationActions:
    - Deny
  webhookConfiguration:
    timeoutSeconds: 30
  matchConstraints:
    resourceRules:
      - apiGroups: ['']
        apiVersions: ['v1']
        operations: ['CREATE', 'UPDATE']
        resources: ['pods']
  matchImageReferences:
```

```

attestors:
  - name: cosign
    cosign:
      keyless:
        identities:
          # 'subject' is an exact match; use 'subjectRegExp' to allow any
          # workflow (and any ref) in the repository.
          - subjectRegExp:
'^https://github\.com/myorg/myrepo/\.github/workflows/.'
            issuer: 'https://token.actions.githubusercontent.com'
        ctlog:
          url: 'https://rekor.sigstore.dev'
attestations:
  - name: vuln
    intoto:
      type: cosign.sigstore.dev/attestation/vuln/v1
validations:
  - expression: >-
    (images.containers + images.?initContainers.orValue([]) +
      images.?ephemeralContainers.orValue([])).map(image,
        verifyImageSignatures(image, [attestors.cosign])).all(e, e > 0)
    message: 'Image signature verification failed'
  - expression: >-
    (images.containers + images.?initContainers.orValue([]) +
      images.?ephemeralContainers.orValue([])).map(image,
        verifyAttestationSignatures(image, attestations.vuln,
          [attestors.cosign])).all(e, e > 0)
    message: 'Attestation signature verification failed'

```

The policy works as follows:

- `validationActions: [Deny]` rejects any Pod whose images fail verification.
- `webhookConfiguration.timeoutSeconds: 30` gives Kyverno time to fetch signatures from the registry and check the transparency log. The default is 10 seconds, and the maximum is 30.
- `matchImageReferences` limits the check to your organization's images.
- `attestors` defines which identities can sign: any workflow in `myorg/myrepo`, verified through the Sigstore transparency log. Use `subjectRegExp`, not `subject`, for this. `subject` is an exact match, and a real certificate subject names the workflow file and the Git ref, such as `https://github.com/myorg/myrepo/.github/workflows/release.yml@refs/tags/v1.2.3`.
- `verifyImageSignatures` confirms that one of those identities signed each image.

- `verifyAttestationSignatures` confirms that the same identity signed the vulnerability scan attestation.
- Both validations check images in containers, init containers, and ephemeral containers, the same lists that `restrict-image-registries` checks. Otherwise an unsigned init container could run before any verified container starts.

At admission, Kyverno also replaces the tag of each verified image with its digest, so the Pod runs exactly the image that Kyverno verified. This happens before validation, so images that this policy verifies also pass `require-image-digest`.

To check the content of an attestation, such as the highest vulnerability severity or the builder in SLSA provenance, use the `extractPayload()` CEL function in another validation. For details, see [ImageValidatingPolicy](#).

Enforce Pod Security Standards

The Kubernetes [Pod Security Standards](#) define three profiles. Use Restricted for all application workloads.

Use two layers:

- **Pod Security Admission labels** on each namespace, as a built-in baseline. Kyverno sets the label, and Pod Security Admission, which is built into the API server, enforces it.
- **Kyverno policies** for clear messages, exceptions, reports, and checks on existing workloads.

Label every namespace for Pod Security Admission

The following `MutatingPolicy` adds the `restricted` enforcement label to every new namespace:

```
apiVersion: policies.kyverno.io/v1
kind: MutatingPolicy
metadata:
  name: add-psa-restricted-label
spec:
  matchConstraints:
    resourceRules:
      - apiGroups: ['']
        apiVersions: ['v1']
        resources: ['namespaces']
        operations: ['CREATE']
  mutations:
    - patchType: ApplyConfiguration
      applyConfiguration:
        expression: >-
          Object{
            metadata: Object.metadata{
              labels: {"pod-security.kubernetes.io/enforce": "restricted"}
            }
          }
```

The policy labels a namespace only when it's created, and it overwrites any value set in the create request. Label existing namespaces through GitOps.

Pod Security Admission is built into the API server, so Kyverno policy exceptions don't apply to it. If the Pods in a platform namespace need host access, such as the node exporter in the `monitoring` namespace in Chapter 7, the platform team must label that namespace explicitly after it exists, for example with `pod-security.kubernetes.io/enforce=privileged`.

Protect the Pod Security Admission labels

A tenant who can update a namespace can also relabel it `pod-security.kubernetes.io/enforce=privileged` and turn off Pod Security Admission for that namespace. This is the same risk as the `tier` label in Chapter 4. The following policy lets only members of `platform-admins` add, change, or remove `pod-security.kubernetes.io/*` labels:

```
apiVersion: policies.kyverno.io/v1
kind: ValidatingPolicy
metadata:
  name: protect-psa-labels
spec:
```

```

- Deny
matchConstraints:
  resourceRules:
    - apiGroups: ['']
      apiVersions: [v1]
      operations: [UPDATE]
      resources: [namespaces]
  variables:
    - name: psaKeys
      expression: >-
        object.metadata.?labels.orValue({}).filter(k, k.startsWith('pod-
security.kubernetes.io/')) +
        oldObject.metadata.?labels.orValue({}).filter(k, k.startsWith('pod-
security.kubernetes.io/'))
    - name: labelsChanged
      expression: >-
        variables.psaKeys.exists(k,
          object.metadata.?labels[k].orValue('') != oldObject.metadata.?
labels[k].orValue(''))
  validations:
    - message: 'Only the platform team can change pod-security.kubernetes.io
labels.'
      expression: "!variables.labelsChanged || 'platform-admins' in
request.userInfo.groups"

```

The policy doesn't match namespace creation, so `add-psa-restricted-label` still labels new namespaces. If a GitOps controller manages namespace labels, add its identity to the allowed groups or usernames, or its syncs will be denied.

Enforce the core Restricted controls

The following policy enforces the core controls of the Restricted profile: no privileged containers, no privilege escalation, non-root users, dropped capabilities, and no host namespaces. It doesn't check `seccompProfile`, which the Restricted profile requires to be `RuntimeDefault` or `Localhost`, so Pod Security Admission can still reject a Pod that passes this policy. Set a seccomp profile in your Pod specs.

```

apiVersion: policies.kyverno.io/v1
kind: ValidatingPolicy
metadata:
  name: require-restricted-pod-security
spec:
  evaluation:
    background:
      enabled: true

```

YAML

```

- Deny
matchConstraints:
  resourceRules:
    - apiGroups: ['']
      apiVersions: [v1]
      operations: [CREATE, UPDATE]
      resources: [pods]
  variables:
    - name: allContainers
      expression: 'object.spec.containers + object.spec.?
initContainers.orValue([])'
  validations:
    - message: 'Containers must not run as privileged.'
      expression: >-
        variables.allContainers.all(c,
          !c.?securityContext.?privileged.orValue(false))
    - message: 'Containers must set allowPrivilegeEscalation to false.'
      expression: >-
        variables.allContainers.all(c,
          c.?securityContext.?allowPrivilegeEscalation.orValue(true) == false)
    - message: 'Containers must run as non-root.'
      expression: >-
        object.spec.?securityContext.?runAsNonRoot.orValue(false) ||
        variables.allContainers.all(c, c.?securityContext.?
runAsNonRoot.orValue(false))
    - message: 'Containers must drop ALL capabilities.'
      expression: >-
        variables.allContainers.all(c,
          'ALL' in c.?securityContext.?capabilities.?drop.orValue([]))
    - message: 'Pods must not share host namespaces.'
      expression: >-
        !object.spec.?hostNetwork.orValue(false) &&
        !object.spec.?hostPID.orValue(false) &&
        !object.spec.?hostIPC.orValue(false)

```

For full coverage, including seccomp profiles and volume types, use the Pod Security policies in the [Kyverno policy library](#).

Place sensitive workloads on trusted nodes

The CISA Devices pillar asks you to know and trust the machines that run your workloads. In Kubernetes, schedule sensitive workloads onto hardened node pools, such as nodes with confidential computing or a locked-down OS. To enforce this, write a `ValidatingPolicy` that requires a `nodeSelector` for those node pools on every Pod in a namespace labeled `data-class=restricted`.

Kyverno enforces the placement rule, not the trust. Hardening the node image, attesting it, and labeling nodes by trust level are jobs for your Kubernetes distribution, your cloud provider, or an attestation system. Kyverno trusts the node labels that it finds. The Devices pillar also covers the laptops and other endpoints that your engineers use. Those are outside Kubernetes and outside this guide.

CHECKLIST

- ☐ Allow images only from approved registries.
- ☐ Require image digests in production.
- ☐ Sign images in CI, and verify signatures at admission.
- ☐ Require vulnerability scan and provenance attestations.
- ☐ Label every namespace for Pod Security Admission, including namespaces that existed before the policy.
- ☐ Reserve the `pod-security.kubernetes.io/*` labels for the platform team.
- ☐ Enforce the Restricted profile with Kyverno, and set a seccomp profile in your Pod specs.
- ☐ Pin sensitive workloads to trusted node pools.

07

CHAPTER 7

Default deny with governed exceptions

Every real cluster needs exceptions, but in a Zero Trust design each exception must be narrow, approved, time-bound, and backed by a compensating control. Kyverno supports all four: `PolicyException` resources scope the bypass, the cleanup controller removes it when it expires, and companion policies enforce the compensating control while the exception is active.

Why unmanaged exceptions break Zero Trust

A permanent exception is standing trust. Over time, exceptions pile up, their owners move on, and nobody remembers why they exist. Each one becomes a gap an attacker can use. Zero Trust treats an exception like any other access grant: explicit, least privilege, and for a limited time.

The five rules of a governed exception

Rule	What it means	How Kyverno enforces it
Scoped	Exempts one policy for named resources only.	<code>policyRefs</code> and <code>matchConditions</code> on the <code>PolicyException</code> .
Approved	A named person approves it, with a ticket.	A governing policy requires owner and ticket annotations and an approver group.
Time-bound	It expires on its own.	A <code>cleanup.kyverno.io/ttl</code> label; the cleanup controller deletes it.
Compensated	Another control reduces the risk while it's active.	A companion <code>ValidatingPolicy</code> for the exempted workload.
Visible	It shows up in reports and audits.	Policy reports record exempted results as <code>skip</code> .

Step 1: Limit where exceptions can live

Install Kyverno with exceptions turned on in one namespace only, as shown in the "Before you begin" section:

```
--set features.policyExceptions.enabled=true
--set features.policyExceptions.namespace=kyverno-exceptions
```

SHELL

Kyverno ignores `PolicyException` resources in any other namespace. Create the namespace, and then grant create access in it only to the group that approves exceptions:

```
kubectl create namespace kyverno-exceptions
```

SHELL

This guide uses only the `policies.kyverno.io` `PolicyException`. The legacy `kyverno.io` version is deprecated in Kyverno 1.19.

Step 2: Require every exception to expire and be documented

The following policy rejects an exception unless it has a TTL, a ticket, an owner, and a named compensating control, and unless a member of the approver group submitted it.

```

apiVersion: policies.kyverno.io/v1
kind: ValidatingPolicy
metadata:
  name: govern-policy-exceptions
spec:
  validationActions:
    - Deny
  matchConstraints:
    resourceRules:
      - apiGroups: [policies.kyverno.io]
        apiVersions: ['*']
        operations: [CREATE, UPDATE]
        resources: [policyexceptions]
  variables:
    - name: labels
      expression: 'object.metadata.?labels.orValue({})'
    - name: annotations
      expression: 'object.metadata.?annotations.orValue({})'
  validations:
    - message: 'Exceptions must expire. Set cleanup.kyverno.io/ttl to 1d, 3d, or 7d.'
      expression: "variables.labels['cleanup.kyverno.io/ttl'].orValue('') in ['1d', '3d', '7d']"
    - message: 'Exceptions need ticket, owner, and compensating-controls annotations.'
      expression: >-
        ['exceptions.example.com/ticket',
         'exceptions.example.com/owner',
         'exceptions.example.com/compensating-controls'].all(k, k in
        variables.annotations)
    - message: 'Only members of exception-approvers can create exceptions.'
      expression: "'exception-approvers' in request.userInfo.groups"

```

Step 3: Write a time-bound exception

In this example, a monitoring agent needs `hostNetwork` to collect node metrics. That breaks the `require-restricted-pod-security` policy from Chapter 6. The exception applies only to workloads in the `monitoring` namespace that carry the label `app.kubernetes.io/name: node-exporter`, and it expires after 7 days. The condition uses a label rather than a name because Kyverno applies Pod policies to Pod controllers through autogen (Chapter 4), so it must match both the DaemonSet and the Pods it creates, and those Pods get generated names. Limit who can deploy to `monitoring` with RBAC, so that no one else can use the label to claim the exception.

```
apiVersion: policies.kyverno.io/v1
kind: PolicyException
metadata:
  name: node-exporter-host-network
  namespace: kyverno-exceptions
  labels:
    cleanup.kyverno.io/ttl: 7d
  annotations:
    exceptions.example.com/ticket: SEC-1234
    exceptions.example.com/owner: observability-team
    exceptions.example.com/compensating-controls: node-exporter-compensating-controls
spec:
  policyRefs:
    - name: require-restricted-pod-security
      kind: ValidatingPolicy
  matchConditions:
    - name: only-node-exporter-in-monitoring
      expression: >-
        object.metadata.namespace == 'monitoring' &&
        object.metadata.labels['app.kubernetes.io/name'].orValue('') == 'node-exporter'
```

The cleanup controller needs permission to delete exceptions. Add the following aggregated ClusterRole:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: kyverno:cleanup-policyexceptions
  labels:
    rbac.kyverno.io/aggregate-to-cleanup-controller: 'true'
rules:
  - apiGroups: [policies.kyverno.io]
    resources: [policyexceptions]
    verbs: [get, list, watch, delete]
```

When the TTL passes, Kyverno deletes the exception, and the original policy applies again. To keep the exception, the owner must request a new one. This forces a regular review.

The exception checks `object.metadata.namespace`, while the compensating policy in Step 4 checks `namespaceObject`. Kyverno evaluates an exception's conditions against the resource in the request, such as the DaemonSet itself, which has a namespace. A

policy's autogen rules are evaluated against the Pod template, which doesn't. Put the `app.kubernetes.io/name` label on the DaemonSet's own metadata as well as on its Pod template, so both checks see it.

Pod Security Admission ignores Kyverno exceptions. If `add-psa-restricted-label` from Chapter 6 labeled the `monitoring` namespace `restricted`, Pod Security Admission still rejects the `hostNetwork` Pods, and the Baseline profile disallows `hostNetwork` too. The platform team must label the namespace `privileged`:

```
kubectl label namespace monitoring \
  pod-security.kubernetes.io/enforce=privileged --overwrite
```

SHELL

This label is part of the governed exception. With `protect-psa-labels` from Chapter 6 in place, only a member of `platform-admins` can run this command, so the relabel is itself a governed action. Record it in the same ticket, and set the label back to `restricted` when the exception expires.

Step 4: Enforce a compensating control

A compensating control lowers risk while the exception is active. For the node exporter, you accept `hostNetwork` but require every other setting to be as restrictive as possible. The following policy applies only to the exempted DaemonSet and its Pods. It uses `namespaceObject` for the namespace check, because autogen rewrites `object.metadata` to the Pod template's metadata, which has no namespace. Because of autogen, Kyverno also checks the DaemonSet's Pod template, so a non-compliant change is rejected before any Pod is created:

```
apiVersion: policies.kyverno.io/v1
kind: ValidatingPolicy
metadata:
  name: node-exporter-compensating-controls
spec:
  validationActions:
    - Deny
  matchConstraints:
    resourceRules:
      - apiGroups: ['']
        apiVersions: [v1]
        operations: [CREATE, UPDATE]
        resources: [pods]
  matchConditions:
```

YAML

```

expression: >-
  namespaceObject.metadata.name == 'monitoring' &&
  object.metadata.labels['app.kubernetes.io/name'].orValue('') == 'node-
exporter'
validations:
- message: 'Exempted Pods must use a read-only root filesystem.'
  expression: >-
    object.spec.containers.all(c,
      c.securityContext.readOnlyRootFilesystem.orValue(false))
- message: 'Exempted Pods must not be privileged or add capabilities.'
  expression: >-
    object.spec.containers.all(c,
      !c.securityContext.privileged.orValue(false) &&
      size(c.securityContext.capabilities.add.orValue([])) == 0)
- message: 'Exempted Pods must use the approved, pinned image.'
  expression: >-
    object.spec.containers.all(c,
      c.image.startsWith('registry.example.com/monitoring/node-
exporter@sha256:'))

```

Other common compensating controls include the following:

- A stricter NetworkPolicy that allows ingress only from Prometheus.
- Extra runtime monitoring, such as Falco or Tetragon rules, for the exempted workload. Kyverno doesn't observe running processes, so this control needs a runtime security tool (Chapter 12).
- Placement on a dedicated node pool.

NOTE

Exceptions and their compensating controls are two halves of one decision. Store them in the same Git folder, and review them together.

Step 5: Keep exceptions visible

By default, policy reports record resources covered by an exception as `skip`, not `pass`. Keep this default so auditors can see every active exception. To list active exceptions and their TTL, run the following command:

```
kubectl get policyexceptions.policies.kyverno.io -n kyverno-exceptions \
-L cleanup.kyverno.io/ttl
```

SHELL

Feed these reports to your dashboards so the number of active exceptions becomes a tracked metric.

Advanced: One-time passcodes with quotas

Some teams want developers to self-serve a one-time bypass, with a cap on how often each person can use it. Chip Zoller's [One-Time Pass Codes for Kyverno with Quotas](#) (August 2023) shows one design:

1. A validating policy blocks a risky setting, such as host namespaces, and returns a random 8-character passcode in the denial message.
2. A mutating policy stores valid passcodes and per-user quotas in a ConfigMap.
3. The user resubmits the Deployment with the passcode in an `otp` label.
4. Kyverno checks the code, marks it consumed with a timestamp and username, and reduces that user's quota by one.
5. A monthly scheduled cleanup resets quotas.

The ConfigMap that stores the quotas and codes looks like the following:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: otp
  namespace: platform
data:
  chip: "5"
  mark: "2"
  codes: |-
    - 6o3ukw1q
    - 20bao9hy
```

YAML

The original post uses `ClusterPolicy` rules. Use it as a design pattern, and rebuild it with the newer policy types when you adopt it.

CHECKLIST

- ☐ Allow exceptions in one namespace only.
- ☐ Limit exception creation to an approver group.
- ☐ Require a TTL, ticket, owner, and compensating control on every exception.
- ☐ Write a compensating-control policy for each exception.
- ☐ For each exception that needs host access, relabel the namespace for Pod Security Admission, and restore the label when the exception expires.
- ☐ Track active exceptions as a metric.

08

CHAPTER 8

Verify continuously

Zero Trust isn't a one-time gate: you must keep checking that running workloads still meet policy and act on what you find. Kyverno scans existing resources in the background, writes the results to standard policy reports, and exposes metrics you can send to your monitoring and SIEM tools.

Policy reports answer one question: does each resource's declared spec meet policy? Two other data sources complete the Visibility and Analytics pillar, and neither comes from Kyverno. Kubernetes API server audit logs record who made each request, including the JIT sessions in Chapter 5. Turn them on with an audit policy in your cluster or cloud provider, and send them to your SIEM. Runtime telemetry records what processes do after they start, and a runtime security tool provides it (Chapter 12). Your SIEM correlates all three and raises alerts.

Keep background scanning on

Admission checks see only new requests. Background scanning checks resources that already exist, including resources that existed before you added a policy. Background scanning is on by default for `ValidatingPolicy`, and the reports controller rescans every hour. To make the setting visible in each policy, set it in the `evaluation` block:

```
spec:
  evaluation:
    background:
      enabled: true
```

YAML

A background scan has no requesting user. Rules that check `request.userInfo`, such as `restrict-cluster-admin-binding`, report every matching resource that exists, such as each existing `cluster-admin` binding. Treat those results as an inventory, or set `background.enabled: false` for those policies.

Read policy reports

Kyverno writes results to `PolicyReport` resources in each namespace and `ClusterPolicyReport` resources for cluster-scoped objects. These use the open `wgpolicyk8s.io/v1alpha2` format, so other tools can read them. Kyverno 1.19 writes this format by default. It can also write reports in the newer OpenReports (`openreports.io`) format, which is turned off by default.

```
# Summarize results in every namespace
kubectl get policyreports -A

# Show failing results in one namespace
kubectl get policyreport -n team-a -o yaml | grep -B5 'result: fail'
```

SHELL

Each result lists the policy, the resource, and one of the following results: `pass`, `fail`, `warn`, `error`, or `skip`.

NOTE

In large clusters, reports can add load to etcd. Use the Kyverno [Reports Server](#) to store reports in a separate database.

Send results to people and tools

[Policy Reporter](#) adds a web UI and sends new results to other systems, such as Slack, Microsoft Teams, Loki, Elasticsearch, and S3.

```
helm repo add policy-reporter https://kyverno.github.io/policy-reporter
helm install policy-reporter policy-reporter/policy-reporter \
  --namespace policy-reporter --create-namespace \
  --set ui.enabled=true
```

SHELL

Kyverno also exposes Prometheus metrics, such as `kyverno_policy_results_total`. Use them to alert on sudden rises in denied requests or failing resources.

Roll out from audit to enforce

Use the following stages for each new policy.

Stage	validationActions	Duration	Exit criteria
1. Test	[Audit] in CI only	1 week	kyverno test passes on sample manifests.
2. Observe	[Audit] in clusters	2–4 weeks	Failures are understood, fixed, or covered by exceptions.
3. Warn	[Audit, Warn]	1–2 weeks	Teams see warnings in kubect1 and fix workloads.
4. Enforce	[Deny]	Ongoing	Kyverno blocks new violations, and reports stay clean.

To move a policy to the next stage, change only `validationActions` and apply it through your normal GitOps flow.

Measure your Zero Trust posture

Track the following metrics over time:

- Percentage of namespaces with a default-deny NetworkPolicy.
- Percentage of running images that are signed and verified.
- Number of standing `cluster-admin` bindings.
- Number of active exceptions, and their average age.
- Number of policies in `Deny` compared with `Audit`.

Map policies to compliance frameworks

Add annotations to each policy so reports link results to the controls auditors ask about. For example:

```

metadata:
  annotations:
    policies.kyverno.io/title: Restrict image registries
    policies.kyverno.io/category: Supply chain
    policies.kyverno.io/severity: high
    compliance.example.com/nist-800-53: CM-7, SI-7
    compliance.example.com/cis-k8s: '5.5.1'
  
```

YAML

The following table shows common mappings for policies in this guide.

Policy area	NIST SP 800-53	NSA and CISA Kubernetes Hardening Guide topic
Default deny network	SC-7 (Boundary protection)	Network separation and hardening
Least-privilege RBAC	AC-6 (Least privilege)	Authentication and authorization
JIT access	AC-2 (Account management)	Authentication and authorization
Image signatures	SI-7 (Software integrity)	Kubernetes Pod security
Pod Security Standards	CM-7 (Least functionality)	Kubernetes Pod security
Time-bound exceptions	CA-7 (Continuous monitoring), PM-9 (Risk strategy)	Audit logging and threat detection
Policy reports	AU-6 (Audit review), CA-7	Audit logging and threat detection

Confirm mappings with your compliance team. Control numbers vary by framework version.

CHECKLIST

- ☐ Keep background scanning on for every validating policy.
- ☐ Install Policy Reporter, and send failures to your SIEM or chat tool.
- ☐ Alert on Kyverno metrics.
- ☐ Follow the audit-to-enforce stages for each policy.
- ☐ Annotate policies with compliance mappings.

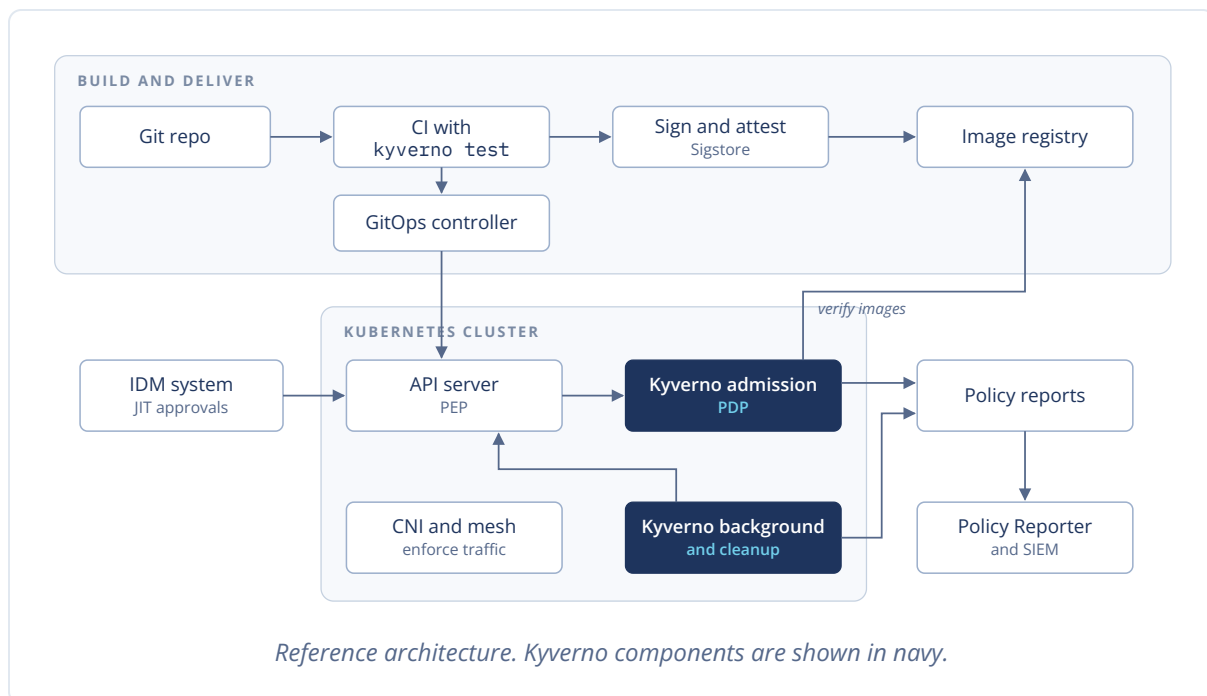
09

CHAPTER 9

Reference architecture

The reference architecture places Kyverno at the center of three policy loops: the build pipeline, the admission path, and continuous background checks. Identity, signing, networking, secrets, and reporting systems plug into those loops and do the work that Kyverno doesn't do: authenticating users, carrying and encrypting traffic, signing software, storing secrets, and watching running processes.

Components



The following table describes each component.

Component	Role	Example tools
CI pipeline	Tests manifests and infrastructure code against policy.	Kyverno CLI, JSON-mode <code>ValidatingPolicy</code>
Signing	Signs images and attaches attestations.	Sigstore Cosign, GitHub Actions OIDC
API server	Authenticates and authorizes every request, and then enforces Kyverno's admission decisions as the policy enforcement point. Also enforces Pod Security Admission labels.	Kubernetes authenticators, RBAC, admission webhooks, Pod Security Admission
Kyverno admission	Validates, mutates, generates, and verifies images.	Kyverno admission controller
Kyverno background and cleanup	Scans existing resources, keeps generated resources in sync, and deletes expired ones.	Kyverno background and cleanup controllers
CNI and service mesh	Enforce generated network and identity policies.	Cilium, Calico, Istio
Identity provider	Authenticates users and groups to the API server and enforces MFA. It isn't shown in the diagram, but it's in front of every user request.	Okta, Microsoft Entra ID, Keycloak, or any OIDC provider
IDM system	Approves and records JIT access requests.	Your access workflow tool
Reporting	Collects results and sends them to people and tools.	Policy Reporter, Prometheus, SIEM

The following components are also part of a complete Zero Trust Architecture. Kyverno can check that workloads use them, but doesn't provide them.

Component	Role	Example tools
Secret store and encryption at rest	Stores secrets outside manifests and encrypts etcd.	HashiCorp Vault, cloud secret managers, External Secrets Operator, KMS provider for etcd
API server audit logging	Records who did what, for every request.	Kubernetes audit policy, cloud provider audit logs
Runtime security	Observes and blocks process, file, and network behavior inside containers.	Falco, Tetragon, Nirmata Runtime (optional, Chapter 12)
Node and control plane hardening	Secures the hosts and the API server configuration.	CIS Benchmark tooling, hardened node images, your Kubernetes distribution

For the full list of what Kyverno owns, configures, or doesn't cover, see "What Kyverno covers, and what it doesn't" in Chapter 2.

Lifecycle of a request

The following steps trace one deployment and one access request through the architecture.

- 1. Build.** A developer merges code. CI runs `kyverno test`, builds the image, signs it, and attaches a vulnerability scan attestation.
- 2. Deploy.** The GitOps controller applies a Deployment to the `payments-api` namespace.
- 3. Verify.** Kyverno checks the image signature and attestation, the registry, the service account, and Pod security settings. It adds secure defaults, such as the namespace's `workspace` and `tier` labels. The Pods get no API token, because Kyverno set `automountServiceAccountToken: false` on their ServiceAccount when it was created.
- 4. Segment.** The namespace already has `default-deny`, `allow-dns`, and workspace rules that Kyverno generated when the namespace was created, and the CNI plugin enforces them.
- 5. Grant.** During an incident, an engineer requests edit access. The IDM system approves it and labels the engineer's ServiceAccount. Kyverno generates a RoleBinding with a 2-hour TTL.
- 6. Expire.** After 2 hours, the cleanup controller deletes the RoleBinding.

- 7. Report.** Background scans record every result in policy reports, and Policy Reporter sends failures to the SIEM.

Multi-cluster deployment

For many clusters, keep one Git repository of policies and deliver them to every cluster with your GitOps tool. Collect reports centrally so you can compare posture across clusters. Management platforms such as Nirmata can add fleet-wide policy delivery, reporting, and exception workflows on top of open source Kyverno.

10

CHAPTER 10

Maturity roadmap

Move through the four CISA maturity stages one pillar at a time. Audit-mode policies take a pillar to Initial, and enforced and generated controls take it to Advanced. Optimal means access is dynamic, short-lived, and continuously checked everywhere.

Stages by pillar

Pillar	Traditional	Initial	Advanced	Optimal
Networks	Flat Pod network.	Default deny in audit for new namespaces.	Default deny and label-driven rules generated in every namespace.	mTLS everywhere, egress by name, and request-level authorization.
Identity	Shared admin accounts and standing <code>cluster-admin</code> .	RBAC guardrails in audit.	Guardrails enforced; dedicated service accounts; no token automount.	JIT access only, with TTL and full audit trail.
Applications and Workloads	Any image, any settings.	Registry allowlist; Pod Security Baseline.	Signed images and Pod Security Restricted enforced.	Attestations for provenance, SBOM, and scans required; digests only.
Devices	Any node runs any workload.	Node pools labeled by trust level.	Sensitive workloads pinned to hardened nodes.	Node attestation, from a system outside Kyverno, sets the trust labels that policies check before scheduling.
Data	Secrets in plain manifests.	Policies flag Secrets in ConfigMaps and env vars.	External secret stores required.	Access to secrets is per workload and time-bound.
Governance	Exceptions by email, never expire.	Exceptions as <code>PolicyException</code> resources.	Exceptions scoped, approved, and TTL-bound.	Every exception has an enforced compensating control and is tracked as a metric.
Visibility and Analytics	No policy data.	Policy reports enabled.	Reports in a central UI and SIEM.	Posture metrics drive alerts and reviews.
Automation and Orchestration	Manual setup per namespace.	Some policies in Git.	All policies in Git, tested in CI, delivered by GitOps.	Self-service onboarding with generated controls and automatic cleanup.

Kyverno policies, with a CNI plugin that enforces NetworkPolicy, take most pillars to Advanced. The Optimal column needs other components: a service mesh and the Kyverno Authz Server for mTLS and request-level authorization, API server audit logs in a SIEM for the audit trail, node attestation for Devices, an external secret store for Data, and a runtime security tool for behavior checks on Applications and Workloads (Chapter 12). The CISA model also measures identity capabilities, such as MFA and continuous

session validation, and device capabilities, such as endpoint posture. Those capabilities live in your identity provider and endpoint management, outside Kubernetes, and this roadmap doesn't track them.

How to use the table

Score each pillar for your cluster today, and then pick the next stage for the two or three pillars that matter most to your risk. Set up the exception process from Chapter 7 before you enforce anything, so that teams have a safe way to request exceptions. Chapter 13 gives a 90-day sequence that moves the Networks, Identity, Applications and Workloads, and Governance pillars from Traditional toward Advanced.

11

CHAPTER 11

Case studies

Two published case studies show how teams combine the controls in this guide: a platform team that automates guardrails and break-glass access, and a financial services firm that offers micro-segmentation as a service on shared clusters.

Fastly: guardrails and break glass at scale

Item	Details
Source	How Fastly used Kubernetes to scale our platform engineering practice , October 2024
Platform	Elevation, an internal developer platform built on Kubernetes and Kyverno
Scale	More than 200 services across more than 40 teams and projects
Guardrails	Kyverno validates, mutates, and generates resources, for example, to block applications that run as root
Onboarding	Controllers set up each new namespace with Vault secrets, policies, and roles
Break glass	Time-scoped elevated access starting at 2 hours, with audit logging of who accessed what and when

Zero Trust lessons

- Make the secure path the default path.
- Make emergency access per session and fully observed.
- Standardize namespace setup so every tenant starts with the same controls.

Fidelity International: micro-segmentation as a service

Item	Details
Source	KubeCon + CloudNativeCon North America 2024 (Salt Lake City, November 2024), presented by Jim Bugwadia (Nirmata) and Rachael Wonnacott (Fidelity International); written up in Securing Platform Engineering with Kubernetes Multi-Tenancy and Micro-Segmentation
Problem	Architectural sprawl across shared infrastructure, hybrid deployment models that complicated security boundaries, and compliance audits across many tenants on the same clusters
Approach	Namespace-as-a-Service: teams request a workspace of isolated namespaces on shared clusters, built on Kyverno plus Cilium
Mechanism	Namespace labels (<code>workspace</code> , <code>tier</code>) drive everything: Kyverno copies them onto every Pod, generates Cilium network policies from them, restricts DNS traffic, and enforces tier-based segmentation so, for example, a backend can't reach the internet directly
Outcome	Lower operational overhead from eliminating cluster sprawl, and self-service namespaces that arrive pre-configured with the right security policies

Chapter 4 builds the same pattern. `require-workspace-labels` rejects a namespace that doesn't declare its `workspace` and `tier`. `add-workload-labels` copies those labels onto every Pod, so developers can't get them wrong. Generated `NetworkPolicy` and `CiliumNetworkPolicy` rules then use those labels for default deny, workspace isolation, and tier-based segmentation.

Zero Trust lessons

- Segmentation that depends on a human setting a label correctly isn't segmentation. Derive it from a boundary the platform team controls, and enforce it on every Pod.
- Self-service and Zero Trust aren't in tension: a workspace request can be fast for the developer and fully governed underneath.
- Tenants get autonomy inside their boundary, not across it. `NamespacedValidatingPolicy` (Chapter 3) lets a tenant add rules without cluster-admin access.

12

CHAPTER 12

Extend Zero Trust to runtime (optional)

Every control so far runs at admission or on a schedule, and checks a resource's declared spec. None of them sees what a running process does. That gap is outside the scope of open source Kyverno, as the scope table in Chapter 2 shows. You can fill it with an open source runtime security tool, such as Falco or Tetragon, or with [Nirmata Runtime](#), a Nirmata product that is separate from the CNCF Kyverno project. This chapter describes Nirmata Runtime. It's optional: you can adopt the rest of this guide without it. It applies the same Zero Trust principle, verify instead of assume, to what a Pod does after it starts, not only to what its spec says it will do.

Why admission control isn't enough

A Pod spec can look correct and still misbehave once it's running:

- A compromised dependency opens files it was never meant to read.
- A process runs a binary that wasn't in the image you signed.
- A workload calls an external address that no NetworkPolicy or egress rule anticipated. Network policy knows ports and CIDRs, not which service is on the other end.

Admission policies from Chapters 4 through 6 limit what a workload can reach and do. Nirmata Runtime checks whether a workload stays inside those limits while it runs.

How it complements Kyverno

Nirmata Runtime follows Kyverno's model. Policies are Kubernetes custom resources written with CEL, and results appear as OpenReports `Report` objects. OpenReports (`openreports.io`) is the successor to the `wgpolicyk8s.io` policy reports in Chapter 8, and Kyverno can also write reports in that format when you turn on its OpenReports setting. Kyverno is the policy decision point for Kubernetes API requests, and the API server enforces its decisions. Nirmata Runtime adds a policy decision and enforcement point inside the kernel, using eBPF. Despite the chart and namespace names in this chapter, it isn't part of Kyverno, and no policy in Chapters 3 through 8 requires it.

Aspect	Kyverno	Nirmata Runtime
Checks	The declared spec	The observed behavior
When	Admission and background scan	Continuously, while the process runs
Mechanism	Admission webhook, controllers	eBPF, per-node DaemonSet
Stops	A non-compliant resource from being created	A non-compliant action from completing

Write your first RuntimePolicy

The following policy denies egress to a specific address from any Pod labeled `app: egress-client`. Its `spec.mode` field takes `enforce`, which blocks, or `monitor`, which reports. The two modes work like the `Deny` and `Audit` validation actions for Kyverno policies.

```
apiVersion: runtime.nirmata.io/v1alpha1
kind: RuntimePolicy
metadata:
  name: block-address-sample
spec:
  mode: enforce
  podSelector:
    matchLabels:
      app: egress-client
  behaviors:
    - network:
        deny:
          values:
            - '8.8.8.8'
```

YAML

A `RuntimePolicy` can also govern file access (`open`), process execution (`exec`), application-layer protocols (classifying traffic by its content, not its declared port), and DNS resolution. DNS behaviors are observation-only: declare the names a workload is expected to resolve, and Nirmata Runtime reports the rest, without changing the workload.

Requirements

- Kubernetes 1.29 or later, on Linux nodes.
- BPF-LSM kernel support for file and process enforcement. Check for `bpf` in `/sys/kernel/security/lsm`; not every managed Kubernetes offering enables it.
- A cgroup v2 host for network egress enforcement and observation.

To install Nirmata Runtime, run the following command:

```
helm install kyverno-runtime oci://ghcr.io/nirmata/charts/kyverno-runtime \
  --namespace kyverno-runtime --create-namespace
```

SHELL

As with every other policy in this guide, start new `RuntimePolicy` resources with `mode: monitor`, review the reports, and switch to `mode: enforce` after you understand the workload's behavior.

Where this fits your roadmap

Runtime enforcement is an Optimal-stage control for the Applications and Workloads pillar and the Visibility and Analytics pillar in Chapter 10. You can reach Advanced in every pillar without it. Add it after your admission-time controls are enforcing cleanly, because runtime denials are harder to debug than a rejected Pod spec. Whether you use Nirmata Runtime or another runtime tool, the admission policies in this guide stay the same.

Apply runtime controls to AI agents

AI agents are the workload type that makes runtime verification necessary. An agent's declared spec rarely shows which APIs it will call, which model providers it will reach, or which tools it will use, because the model decides that at runtime. Apply the controls in this guide to agent workloads first: default-deny egress with an allowlist of model-provider endpoints (Chapter 4), a dedicated service account with no standing privileges (Chapter 5), and verified images (Chapter 6). Then use a `RuntimePolicy` to observe and limit what the agent does.

13

CHAPTER 13

Next steps

Start small and measurable. The following 90-day plan moves a cluster from Traditional toward Advanced in the Networks, Identity, Applications and Workloads, and Governance pillars. It assumes that your CNI plugin enforces NetworkPolicy and that your CI pipeline can sign images. Adjust the pace to the number of workloads that fail the initial audit.

Days 1–30: See where you stand

- ☐ Install Kyverno 1.19 or later and Policy Reporter in a test cluster, and then in production.
- ☐ Apply the audit profile from the companion repository (`kubectl apply -k deploy/audit`). It installs every core validating policy in `Audit` mode.
- ☐ Generate default deny and DNS rules for new namespaces.
- ☐ Set up the exception namespace, governing policy, and approver group.
- ☐ Record baseline numbers for the metrics in Chapter 8.

Days 31–60: Enforce the basics

- ☐ Fix every audit failure, or cover it with a governed exception.
- ☐ Move Pod security, registry, and RBAC policies to `Deny` .
- ☐ Add default deny to existing namespaces, one team at a time.
- ☐ Add `kyverno test` and JSON-mode Terraform checks to CI.
- ☐ Start signing images in CI.

Days 61–90: Remove standing trust

- ☐ Enforce image signature verification.
- ☐ Replace standing admin access with JIT RoleBindings.
- ☐ Require compensating controls for every exception.
- ☐ Turn on mTLS and name-based egress rules for sensitive namespaces.
- ☐ Review metrics against the baseline, and plan the next quarter.

Get the policies

The companion repository, [nirmata/kyverno-zero-trust](https://github.com/nirmata/kyverno-zero-trust), contains every policy in this guide, grouped by CISA pillar, with these entry points:

- `deploy/audit` : every core validating policy in `Audit` mode, for the days 1 to 30 assessment.
- `deploy/enforce` : every core policy as written, after you replace the placeholders.
- `deploy/optional/cilium` and `deploy/optional/istio` : tier segmentation and strict mTLS.
- `deploy/examples/exceptions` : the governed exception from Chapter 7, with its compensating control.
- `policies/automation/check-public-endpoint` : the Terraform plan check from Chapter 3, for the Kyverno CLI.
- `config/rbac` : the aggregated ClusterRoles that give Kyverno the extra permissions in Chapters 4, 5, and 7.

To run the Kyverno CLI tests, run `make test`. To run the end-to-end tests on a local kind cluster, run `make kind-up` and then `make e2e`. To receive the policies as a versioned bundle instead, request it from [Nirmata](https://nirmata.com).

Learn more

- [Kyverno documentation](#)
- [Kyverno policy library](#)
- [Kyverno community on Slack](#), in the `#kyverno` channel

Appendices

Appendix A: Policy index

The following table lists every Kyverno policy in this guide. Each one is in the companion repository with its tests. The repository's `config/rbac` folder also holds the aggregated ClusterRoles from Chapters 4, 5, and 7.

Policy name	Type	Pillar	Chapter
<code>require-owner-label</code>	ValidatingPolicy	Governance	3
<code>check-public-endpoint</code>	ValidatingPolicy (JSON mode)	Automation and Orchestration	3
<code>generate-default-deny</code>	GeneratingPolicy	Networks	4
<code>generate-workspace-network-policy</code>	GeneratingPolicy	Networks	4
<code>generate-backend-tier-policy</code>	GeneratingPolicy	Networks	4
<code>require-workspace-labels</code>	ValidatingPolicy	Networks	4
<code>add-workload-labels</code>	MutatingPolicy	Networks	4
<code>restrict-backend-images-to-backend-tier</code>	ValidatingPolicy	Networks; Applications and Workloads	4
<code>protect-segmentation-labels</code>	ValidatingPolicy	Networks	4
<code>generate-strict-mtls</code>	GeneratingPolicy	Networks	4
<code>restrict-wildcard-rbac</code>	ValidatingPolicy	Identity	5
<code>restrict-cluster-admin-binding</code>	ValidatingPolicy	Identity	5
<code>restrict-escalation-verbs</code>	ValidatingPolicy	Identity	5
<code>disable-token-automount</code>	MutatingPolicy	Identity	5

disallow-default-serviceaccount	ValidatingPolicy	Identity	5
restrict-jit-labels	ValidatingPolicy	Identity	5
generate-jit-rolebinding	GeneratingPolicy	Identity	5
restrict-image-registries	ValidatingPolicy	Applications and Workloads	6
require-image-digest	ValidatingPolicy	Applications and Workloads	6
verify-signed-images	ImageValidatingPolicy	Applications and Workloads	6
add-psa-restricted-label	MutatingPolicy	Applications and Workloads	6
protect-psa-labels	ValidatingPolicy	Applications and Workloads	6
require-restricted-pod-security	ValidatingPolicy	Applications and Workloads	6
govern-policy-exceptions	ValidatingPolicy	Governance	7
node-exporter-host-network	PolicyException	Governance	7
node-exporter-compensating-controls	ValidatingPolicy	Governance	7

Appendix B: Glossary

Term	Definition
Admission control	Kubernetes checks that run on each API request that creates, changes, or deletes an object, before the API server stores the change.
Attestation	A signed statement about an artifact, such as its build provenance or scan results.
Autogen	The Kyverno feature that applies a Pod policy to Pod controllers, such as Deployments and DaemonSets.
Background scan	A Kyverno check of resources that already exist in the cluster, as opposed to an admission check of a new request.
Break glass	Emergency access that an engineer activates during an incident, with a time limit and audit logging.
CEL	Common Expression Language: the expression language that Kyverno and Kubernetes admission policies use.
Cleanup controller	The Kyverno component that deletes resources whose <code>cleanup.kyverno.io/ttl</code> label has expired.
Compensating control	A control that reduces the risk created by an exception while the exception is active.
Default deny	A starting state that blocks all access until a rule allows it.
eBPF	A Linux kernel technology that runs sandboxed programs in the kernel, used for runtime enforcement.
IDM system	Identity management system: the system that approves access requests and records who asked for what.
JIT access	Just-in-time access: a privilege granted on request for a fixed time and then removed.
Keyless signing	Sigstore signing that ties a signature to a workload identity, such as a GitHub Actions workflow, instead of a long-lived key.
Micro-segmentation	A network design that divides the network into small zones with explicit rules between them.
mTLS	Mutual TLS: a connection in which both sides prove their identity with certificates.

Namespace-as-a-Service	A multi-tenant pattern in which a platform team provides namespaces with built-in policies as a self-service product.
PA	Policy administrator: the component that creates or removes an access path, such as a credential or a RoleBinding, based on a policy decision.
PDP	Policy decision point: the component that decides whether to allow a request. In NIST SP 800-207, the policy engine and the policy administrator together.
PEP	Policy enforcement point: the component that allows or blocks the request.
Pod Security Admission	The built-in Kubernetes admission controller that applies the Pod Security Standards based on namespace labels.
Pod Security Standards	Three Kubernetes profiles, Privileged, Baseline, and Restricted, that define allowed Pod security settings.
Policy exception	A <code>PolicyException</code> resource that exempts named resources from a policy.
Policy report	A Kubernetes resource that records policy results for resources.
SBOM	Software bill of materials: a list of the components in a software artifact.
TTL	Time to live: how long a resource or grant exists before automatic removal.

Appendix C: References

Standards and guidance

- [NIST SP 800-207: Zero Trust Architecture](#), NIST, August 2020
- [NIST SP 800-207A: A Zero Trust Architecture Model for Access Control in Cloud-Native Applications in Multi-Cloud Environments](#), NIST, September 2023
- [CISA Zero Trust Maturity Model, version 2.0](#), CISA, April 2023
- [Kubernetes Hardening Guide, version 1.2](#), NSA and CISA, August 2022
- [Pod Security Standards](#), Kubernetes documentation
- [CIS Kubernetes Benchmark](#), Center for Internet Security

Kyverno documentation and announcements

- [ValidatingPolicy](#)
- [MutatingPolicy](#)
- [GeneratingPolicy](#)
- [ImageValidatingPolicy](#)
- [DeletingPolicy](#)
- [Cleanup policies and the TTL label](#)
- [Policy exceptions](#)
- [Kyverno releases and supported Kubernetes versions](#)
- [High availability](#)
- [Announcing Kyverno 1.19](#), Kyverno blog, August 2026
- [CNCF announces Kyverno's graduation](#), CNCF, March 2026
- [Kyverno policy library](#)
- [Kyverno Authz Server](#)
- [Policy Reporter](#)
- [Reports Server](#)
- [Kyverno community on Slack](#), `#kyverno` channel

Companion code

- [Companion policy repository for this guide](#), Nirmata
- [Network policy automation demo](#), Nirmata
- [Nirmata Runtime](#), Nirmata

Case studies and articles

- [NIST Updates NVD Operations to Address Record CVE Growth](#), NIST, April 2026

- [2025 CVE Data Review](#), Jerry Gamblin, January 2026
- [Fault Lines in the AI Ecosystem: TrendAI State of AI Security Report](#), Trend Micro, March 2026
- [CVE-2025-1974: The IngressNightmare in Kubernetes](#), Wiz, March 2025
- [ShadowRay 2.0: Attackers Turn AI Against Itself](#), Oligo Security, November 2025
- [New runc vulnerabilities allow container escape](#), Sysdig, November 2025
- [Inside the LiteLLM Supply Chain Compromise](#), TrendAI Security, March 2026
- [Hugging Face and ClawHub compromised with hundreds of malicious AI models and agent skills](#), The Next Web, May 2026
- [Hugging Face incident and the road ahead](#), OpenAI, 2026
- [Anthropic Incident: An AI Agent Published a Malicious Package to PyPI](#), StepSecurity, July 2026
- [How Fastly used Kubernetes to scale our platform engineering practice](#), Fastly, October 2024
- [Securing Platform Engineering with Kubernetes Multi-Tenancy and Micro-Segmentation](#), Nirmata, December 2024
- [One-Time Pass Codes for Kyverno with Quotas](#), Chip Zoller, August 2023
- [Policy based authorization using Kyverno](#), Istio blog, November 2024

About Nirmata

Nirmata created Kyverno and donated it to the Cloud Native Computing Foundation (CNCF), where it [graduated in March 2026](#). Kyverno is open source under the Apache 2.0 license and is used by organizations such as LinkedIn, Spotify, and Bloomberg. Every Kyverno policy in Chapters 3 through 8 runs on that open source engine. The `RuntimePolicy` in Chapter 12 runs on Nirmata Runtime, a separate product.

Put this guide into practice

1. Get the policies. The companion repository, [nirmata/kyverno-zero-trust](#), holds every policy in this guide with its tests and an audit profile that you can install on any cluster. To receive the policies as a versioned bundle, [contact Nirmata](#).
2. Run the days 1 to 30 plan from Chapter 13 in a test cluster, and record your baseline metrics.
3. Ask questions about the policies in the `#kyverno` channel on [Kubernetes Slack](#).

For teams that run Kyverno across many clusters, or extend Zero Trust to AI agents, Nirmata offers commercial products that build on the open source engine:

- **Nirmata Control Hub:** A central console for authoring, delivering, and reporting on Kyverno policies and exceptions across every cluster and team (Chapter 9, "Multi-cluster deployment").
- **Enterprise Kyverno:** A hardened, supported build of the open source engine for production environments.
- **Nirmata Scanners:** The same policies, run in CI pipelines before a manifest reaches a cluster (Chapter 3, "Test policies in CI").
- **Nirmata Runtime:** eBPF-based runtime enforcement for behavior that no admission check can see (Chapter 12).
- **AIControls:** Governance for AI agents and their model calls, for the AI workloads described in Chapter 1 and Chapter 12.

To see these products with your own policies, [request a demo](#). To govern your first AI agent, [start a free trial of AIControls](#).



Nirmata created Kyverno, the CNCF graduated policy engine for Kubernetes. We help platform and security teams govern clusters, workloads, and AI agents with policy as code.

GET THE POLICIES

github.com/nirmata/kyverno-zero-trust

TALK TO THE KYVERNO MAINTAINERS

nirmata.com/request-a-demo

GOVERN AI AGENTS

aicontrols.dev