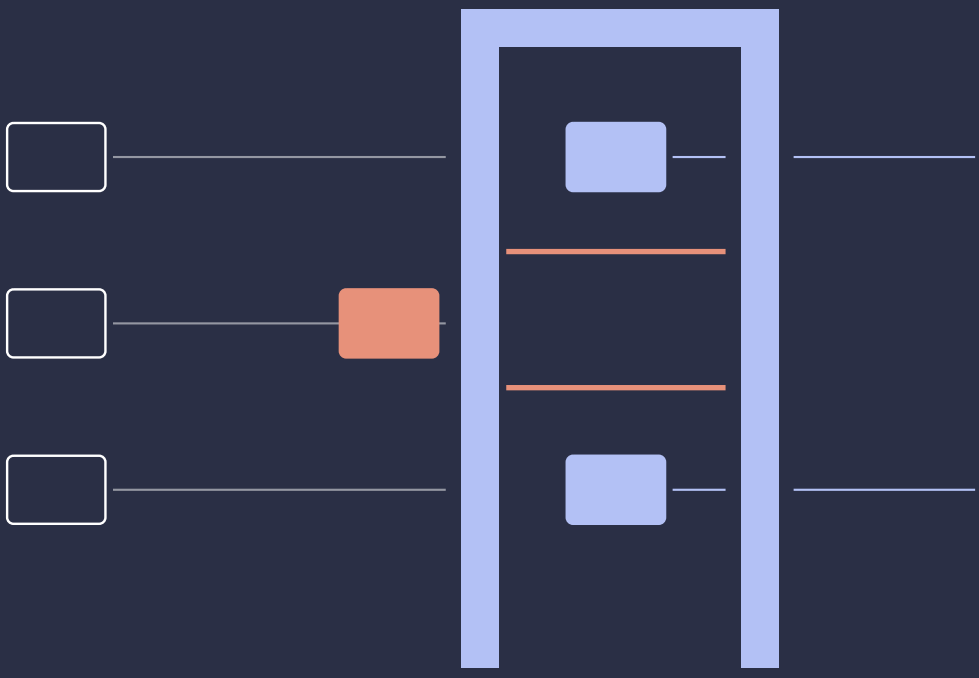
nyrvex.com

Kyverno

Hands-On

Policy as Code for Kubernetes:
validate, mutate, generate, repair



Interview-ready with minimal text

Kyverno

Hands-On

Policy as Code for Kubernetes: validate, mutate, generate, repair

Interview-ready with minimal text

Thomas Zachmann

Kyverno Hands-On. Policy as Code for Kubernetes: validate, mutate, generate, repair.
As of November 4, 2026. © 2026 Thomas Zachmann. License: CC BY-ND 4.0.

Written for Kyverno 1.19 (Chart 3.9) with an eye on 1.20, which was available as a release candidate at the time of writing. The commands, policies, and sample output in this book have not been run on a cluster or with the Kyverno CLI. Output is illustrative, and details may vary by version. Try everything in a lab first, never on a production cluster.

Typefaces: Charter, Fira Sans, JetBrains Mono.

Contents

How to Read This Book 6

Getting Started **9**
1 What Is Kyverno? 10
2 Admission Control and Webhooks 13
3 The Architecture 15
4 The Lab 18

Thinking in Policies **21**
5 The Policy Family 22
6 The First ValidatingPolicy 24
7 Match: What Gets Checked? 27
8 CEL in One Hour 29
9 Audit, Warn, Deny 32
10 Fail or Ignore 34

The Tools **37**
11 MutatingPolicy 38
12 GeneratingPolicy 41
13 ImageValidatingPolicy 44
14 DeletingPolicy 48
15 Pod Security Standards 50
16 PolicyException 52
17 Policy Reports 54

The Classic World **57**
18 Reading a ClusterPolicy 58
19 Validate: pattern and deny 60
20 Variables and JMESPath 62
21 Preconditions and foreach 64
22 Mutate, Generate, verifyImages 66
23 Old Cleanups and Exceptions 68
24 Migrating to CEL 70

Tests and Operations **73**
25 Kyverno CLI 74
26 Tests 76
27 CI and GitOps 79
28 High Availability 82
29 Performance 85
30 Monitoring 87
31 Upgrades 89

Troubleshooting **91**
32 The Method 92
33 The Cluster Is Locked Out 94
34 The Policy Doesn't Take Effect 97
35 Generate and Background Are Stuck 99
36 Image Verification Fails 101

The Interview **103**
37 Kyverno, Gatekeeper, VAP 104
38 The Questions 106
39 Exercises 110
40 Cheat Sheet 113

Index 116

How to Read This Book

You have an interview coming up and you need to know Kyverno. Not someday. Soon. This book is built for exactly that: little text, lots of small examples, and for every topic the question you'll actually be asked.

The Rules

- **Type everything yourself.** Don't copy anything. Your fingers learn too.
- **One command at a time.** There are no scripts. You type, you watch what happens.
- **Lab only.** One bad policy can lock you out of a cluster. A kind cluster is all you need, see chapter 4.
- **Understand the why first.** Interviews rarely test YAML fields. They test whether you can explain what happens when Kyverno goes down.

> **REMEMBER**
>
> The commands, policies, and output in this book have not been run. The output is shortened sample output. Your text will differ.

Two Worlds

Kyverno is in the middle of a rebuild, and that's an interview topic in itself:

- **New:** CEL policies such as `ValidatingPolicy`. This is the future and the core of this book.
- **Old:** `ClusterPolicy` with JMESPath. Deprecated since 1.19, but still present in almost every existing cluster.

Parts II and III show the new world. Part IV shows the old one, so you can read and migrate it.

The Boxes

> **PICTURE THIS**
>
> *A comparison from everyday life. Remember it and you can work out the topic from there.*

```
kubectl get vpol
```

Memory aid: A command to type, with a memory aid underneath.

> **PITFALL**
>
> The mistake almost everyone makes. And that interviewers love to ask about.

> **IN THE INTERVIEW**
>
> **A typical question?**
>
> An answer in two or three sentences. It can be that short.

Your Study Plan

You have ...	Then read ...
1 evening	Chapters 1, 2, 3, 5, 6, 9, and 10, then chapters 38 (The Questions) and 40 (cheat sheet).
2 days	Day 1: Parts I and II. Day 2: chapters 11 to 13, 16, 18, 33, and 38.
1 week	Day 1: Parts I and II. Then one part per day. Day 6: the exercises from chapter 39. Day 7: review the questions.

PART I

Getting Started

What Kyverno is, where it sits in the API server, and how to build a lab.

1

What Is Kyverno?

Rules for your cluster, written as Kubernetes resources.

Kyverno is a policy engine for Kubernetes. The name comes from Greek and means “to govern”. The core is a single idea:

A rule is a resource like any other.

You write policies in YAML, create them with `kubectl apply`, and manage them with Git.

The Problem

RBAC says **who** may create something. RBAC doesn’t say **what it must look like**. A developer with the right to create Pods can therefore also create this Pod:

```
spec:
  containers:
    - name: app
      image: whatever:latest
      securityContext:
        privileged: true
```

Kyverno closes exactly this gap.

> **PICTURE THIS**
>
> *RBAC is the bouncer who checks your ID. Kyverno is the dress code: you may come in, but not dressed like that.*

What Kyverno Can Do

Capability	Example
Validate	Reject Pods with `:latest`.
Mutate	Add missing labels or a `securityContext`.
Generate	Create a NetworkPolicy for every Namespace.
Verify images	Allow only signed images.
Delete	Remove finished Pods after one hour.
Report	Show violations by existing resources as a report.

Why Kyverno Is So Popular

- **Kubernetes-native.** Policies are CRDs. `kubectl`, Helm, and GitOps work as usual.
- **More than validation.** Mutating, generating, and cleaning up are built in.
- **CNCF.** Kyverno is a Cloud Native Computing Foundation project.
- **One language.** The new policies use CEL, the same language as the API server itself.

> **PITFALL**
>
> Kyverno only checks what goes through the API server. It replaces neither RBAC nor NetworkPolicies nor runtime security. What happens inside a container after it starts, it can't see.

IN THE INTERVIEW

What is Kyverno?

A policy engine for Kubernetes. It runs as an admission webhook and can validate, mutate, and generate resources, verify image signatures, and clean up resources. Policies are normal Kubernetes resources.

IN THE INTERVIEW

Why isn't RBAC enough?

RBAC decides on verbs and resource kinds, that is, who may do what. It can't check the content of an object. Kyverno checks exactly that content, for example `privileged: true`.

KEY TAKEAWAYS

- Policies are resources.
- RBAC: who. Kyverno: how.
- Validate, mutate, generate, verify images, delete, report.

2

Admission Control and Webhooks

The last checkpoint before something lands in etcd.

Every change goes through the API server. There's a fixed chain there:

```
kubectl apply
  -> Authentication        Who are you?
  -> Authorization         Are you allowed to? (RBAC)
  -> Mutating Admission    Change the object
  -> Schema validation     Is the object valid?
  -> Validating Admission  Check the object
  -> etcd
```

Remember the order: **mutate first, then validate**. That way, validation checks the final result, not the first draft.

Webhooks

The API server can outsource admission decisions. With a `ValidatingWebhookConfiguration` or `MutatingWebhookConfiguration` you tell it: "Send me these requests." It then sends an `AdmissionReview` over HTTPS to the service and waits for the answer.

```
API server  --AdmissionReview-->  Kyverno
API server  <--allowed: true/false--  Kyverno
```

A mutating webhook also responds with a JSON Patch.

> **PICTURE THIS**
>
> *The API server is passport control. For certain travelers, the officer calls an outside agency and waits for the answer. If the agency can't be reached, the officer has to decide: wave everyone through or stop everyone.*

The Most Important Fields

Field	Meaning
`rules`	Which resources and operations are sent?
`namespaceSelector`	Only Namespaces with certain labels.
`failurePolicy`	`Fail` or `Ignore`, if the webhook doesn't respond.
`timeoutSeconds`	How long the API server waits.

The last sentence of that picture is the most important question in this book. We get to it in chapter 10.

Built-In Alternative

Since Kubernetes 1.30, `ValidatingAdmissionPolicy` (VAP) has been a stable feature. The API server evaluates CEL expressions itself, without a webhook. More on this in chapter 37.

> **PITFALL**
>
> Webhooks sit in the path of **every** matching request. A slow webhook makes the whole API server slow. A dead webhook can block it. So webhooks are part of your control plane, not just another application.

IN THE INTERVIEW

What happens between `kubectl apply` and etcd?

Authentication, authorization, mutating admission, schema validation, validating admission, then storage in etcd. Kyverno hooks into the two admission phases as a webhook.

IN THE INTERVIEW

Why does mutation run before validation?

Validation should see the final object. If it ran earlier, a later mutation could make an already checked object invalid again.

KEY TAKEAWAYS

- AuthN, AuthZ, Mutating, Schema, Validating, etcd.
- The API server asks Kyverno via `AdmissionReview`.
- A webhook is part of the control plane.

3

The Architecture

Four controllers, each with exactly one job.

Controller	Job	Scales?
Admission	Answers webhooks: validates, mutates, verifies images.	Yes, all replicas work
Background	Generates and mutates existing resources.	No, one leader
Reports	Creates policy reports.	No, one leader
Cleanup	Runs deletion policies.	Yes

> **PICTURE THIS**
>
> *The admission controller is the clerk at the counter: it has to answer right away. The background controller is the back office: it works through orders. The reports controller keeps the files. The cleanup controller is the garbage truck with a fixed schedule.*

The Path of a Request

```
API server --> Webhook --> Engine --> Response
                  ^
                  | Policy Cache
                  |
           Policies (CRDs)
```

- The **webhook** accepts the `AdmissionReview`.
- The **engine** applies the matching policies.
- The **webhook controller** adjusts the webhook configuration to the installed policies. That way, Kyverno only receives the resources a policy actually needs.
- The **Cert Renewer** renews the certificates for the webhooks.

UpdateRequests

Generating takes time and must not hold up a request. So the admission controller creates an order, an `UpdateRequest`. The background controller then works through it.

```
Namespace created  -> Admission -> UpdateRequest
                                        |
                                Background Controller
                                        |
                                   NetworkPolicy
```

Kyverno's Webhooks

```
kubectl get validatingwebhookconfigurations,mutatingwebhookconfigurations | grep kyverno
```

Memory aid: Show all the phone lines from the API server to Kyverno.

The two most important are `kyverno-resource-validating-webhook-cfg` and `kyverno-resource-mutating-webhook-cfg`. They route normal resources to Kyverno. The others protect Kyverno's own objects, such as policies and exceptions.

> **PITFALL**
>
> A backend outage hits the controllers differently. If the background controller goes down, no generated resources appear, but validation keeps running. If the admission controller goes down, the entire API server can be affected. Plan according to this ranking.

IN THE INTERVIEW

What controllers does Kyverno have?

Admission for webhooks, background for generate and mutate-existing, reports for policy reports, cleanup for deletion. The admission controller is mandatory and the most critical. It can scale horizontally; background and reports work with a leader.

IN THE INTERVIEW

How does the API server know what to send to Kyverno?

Through the webhook configurations. Kyverno maintains them automatically based on the installed policies, so that only matching resources arrive.

KEY TAKEAWAYS

- Admission, Background, Reports, Cleanup.
- Generating runs asynchronously via `UpdateRequest`.
- Kyverno maintains its own webhooks.

4

The Lab

One kind cluster, one Helm chart, and you're good to go.

What You Need

Docker, `kind`, `kubectl`, and Helm. Later, the Kyverno CLI as well (chapter 25).

```
kind create cluster --name kyverno-lab
```

Memory aid: **kind** stands for **K**ubernetes **in** **D**ocker: a cluster as a container.

```
kubectl get nodes
```

Memory aid: Is the cluster alive?

Installation

```
helm repo add kyverno https://kyverno.github.io/kyverno/
```

Memory aid: Add a new compartment to the Helm shelf.

```
helm repo update
```

Memory aid: Update the catalog.

```
helm install kyverno kyverno/kyverno -n kyverno --create-namespace --version 3.9.2
```

Memory aid: Name, chart, Namespace, pinned version. Chart 3.9.2 corresponds to Kyverno 1.19.2.

```
kubectl -n kyverno get pods
```

Memory aid: Wait until all four controllers are `Running`.

```
NAME                                             READY   STATUS
kyverno-admission-controller-6d9f...             1/1     Running
kyverno-background-controller-7c4b...            1/1     Running
```

```
kyverno-cleanup-controller-5f8d...              1/1     Running
kyverno-reports-controller-84bc...              1/1     Running
```

> **REMEMBER**
>
> This installation has one replica each and is meant for the lab only. In production you need at least three admission replicas, see chapter 28.

What Got Installed

```
kubectl api-resources | grep -i kyverno
```

Memory aid: Shows all new resource kinds and their short names.

A Namespace to Play In

```
kubectl create namespace demo
```

Memory aid: Our test objects live here.

```
kubectl label namespace demo team=lab
```

Memory aid: We'll use this label later to target Namespaces.

IN THE INTERVIEW

How do you install Kyverno?

With the official Helm chart, in its own Namespace, and with a pinned version. In production I set three admission replicas and two each for the other controllers.

KEY TAKEAWAYS

- `kind create cluster`, `helm install`.
- Always pin the chart version.
- Four Pods must be `Running`.

PART II

Thinking in Policies

Policy types, CEL, Match, Audit, and Fail. The foundation of every interview.

5

The Policy Family

One type for each job.

A `ClusterPolicy` used to do everything. Today, every job has its own type. They all belong to the API group `policies.kyverno.io` and use **CEL** (Common Expression Language).

Type	Short name	Job
`ValidatingPolicy`	`vpol`	Check resources.
`MutatingPolicy`	`mpol`	Change resources.
`GeneratingPolicy`	`gpol`	Create resources.
`ImageValidatingPolicy`	`ivpol`	Check images and signatures.
`DeletingPolicy`	`dpol`	Delete resources on a schedule.

Each type applies cluster-wide. Each one also has a variant for a single Namespace: `NamespacedValidatingPolicy` (`nvpol`) and so on.

> **PICTURE THIS**
>
> *It used to be a Swiss Army knife. Now it's a toolbox. Each tool does less, but does its one thing properly.*

Old vs. New

Old: `kyverno.io`	**New:** `policies.kyverno.io`
`ClusterPolicy`, `Policy` with `validate`	`ValidatingPolicy`
`ClusterPolicy`, `Policy` with `mutate`	`MutatingPolicy`
`ClusterPolicy`, `Policy` with `generate`	`GeneratingPolicy`
`ClusterPolicy`, `Policy` with `verifyImages`	`ImageValidatingPolicy`
`CleanupPolicy`, `ClusterCleanupPolicy`	`DeletingPolicy`
`PolicyException` in `kyverno.io`	`PolicyException` in `policies.kyverno.io`
JMESPath: `{{ request.object }}`	CEL: `object`

The Timeline

Version	Date	What happened
1.15	2025	CEL policies appear.
1.17	Feb. 2026	`ValidatingPolicy` and others become stable (`v1`). `ClusterPolicy` and `CleanupPolicy` are marked for deprecation.
1.19	Aug. 2026	Feature parity. Old types officially deprecated, last version with full support.
1.20	Available as RC	Old policies are no longer enforced.

As of this book: 1.19.2 is the current stable version, 1.20.0-rc.1 was released on November 2, 2026.

> **PITFALL**
>
> The upgrade to 1.20 is a cliff. An old `ClusterPolicy` stays readable, but is no longer enforced. If you upgrade blindly, you silently lose your security rules. Details are in chapters 24 and 31.

IN THE INTERVIEW

`ClusterPolicy` or `ValidatingPolicy`?

For anything new, `ValidatingPolicy` and its siblings. `ClusterPolicy` has been deprecated since 1.19, and from 1.20 on it's no longer enforced. I migrate existing policies beforehand.

IN THE INTERVIEW

Why did Kyverno switch to CEL?

CEL is the Kubernetes standard, also used by `ValidatingAdmissionPolicy`. That way, Kyverno shares language and structure with the API server, and policies can even be generated as native VAPs.

KEY TAKEAWAYS

- `vpol`, `mpol`, `gpol`, `ivpol`, `dpol`.
- One type per job, optionally `Namespaced`.
- `ClusterPolicy` is on its way out.

6

The First ValidatingPolicy

Pods without the `app` label no longer get through.

require-app-label.yaml

```yaml
apiVersion: policies.kyverno.io/v1
kind: ValidatingPolicy
metadata:
  name: require-app-label
spec:
  validationActions: [Deny]
  matchConstraints:
    resourceRules:
      - apiGroups: [""]
        apiVersions: [v1]
        operations: [CREATE, UPDATE]
        resources: [pods]
  validations:
    - expression: "'app' in object.metadata.?labels.orValue({})"
      message: "The label app is missing."
```

Three blocks carry everything:

- `validationActions`: What happens on a violation? `Deny` rejects.
- `matchConstraints`: Which resources does the policy apply to?
- `validations`: Which condition must be true?

> **REMEMBER**
>
> The expression describes what **must hold**. If it's `true`, all is well. With the old `deny` rule it was the other way around. Nearly everyone mixes this up when migrating.

Try It Out

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

Memory aid: A policy is a resource. So create it the normal way.

```
kubectl get vpol
```

Memory aid: Short name. Wait until the policy is ready.

```
kubectl -n demo run test --image=nginx:1.29
```

Memory aid: `kubectl run` sets the label `run=test` on its own, but no `app`.

```
Error from server: admission webhook
"vpol.validate.kyverno.svc-fail" denied the request:
The label app is missing.
```

```
kubectl -n demo run test --image=nginx:1.29 --labels=app=web
```

Memory aid: With the label, the Pod gets through.

The Deployment Case

The policy matches `pods`. What happens with a Deployment with no label in its template? The Deployment itself is accepted. Only the ReplicaSet controller fails when creating the Pods. The user just sees a Deployment that never becomes ready.

Autogen fixes this. Kyverno derives rules for the Pod controllers:

```
spec:
  autogen:
    podControllers:
      controllers: [deployments, statefulsets, jobs, cronjobs]
```

Now the `kubectl apply` of the Deployment itself fails, with a clear message.

> **PITFALL**
>
> A policy on Pods without Autogen reports the error in the wrong place. The developer sees a green `apply`, and the Pods never show up. The cause is hiding in the ReplicaSet's events.

IN THE INTERVIEW

What is Autogen?

Kyverno automatically derives rules for Deployments, StatefulSets, Jobs, and CronJobs from a Pod rule. That way, a violation fails right when the controller is created, not later at the Pod.

KEY TAKEAWAYS

- `validationActions`, `matchConstraints`, `validations`.
- The expression describes what must hold.
- Autogen puts errors in the right place.

7

Match: What Gets Checked?

The most expensive mistake is a policy that hits too much.

The filter has two stages. Both come from the world of `ValidatingAdmissionPolicy`.

- `matchConstraints`: coarse and fast. Resource kind, operation, Namespace.
- `matchConditions`: fine-grained, with CEL.

match-example.yaml (excerpt)

```yaml
spec:
  matchConstraints:
    resourceRules:
      - apiGroups: [apps]
        apiVersions: [v1]
        operations: [CREATE, UPDATE]
        resources: [deployments]
    namespaceSelector:
      matchLabels:
        team: lab
    excludeResourceRules:
      - apiGroups: [apps]
        apiVersions: [v1]
        operations: ["*"]
        resources: [deployments/scale]
  matchConditions:
    - name: production-only
      expression: >-
        object.metadata.?labels.?stage.orValue('') == 'prod'
```

Field	Effect
`resourceRules`	Group, version, operation, resource.
`excludeResourceRules`	The counterpart: these rules drop out.
`namespaceSelector`	By labels of the Namespace.
`objectSelector`	By labels of the object itself.
`matchConditions`	Any CEL condition. All must be true.

> **PICTURE THIS**
>
> *First the coarse sieve, then the fine one. Whatever the coarse sieve filters out costs no compute time.*

Don't Forget `apiGroups`

- Pods and Namespaces live in the core group `""`.
- Deployments live in `apps`, NetworkPolicies in `networking.k8s.io`.
- The resource name is plural and lowercase: `pods`, not `Pod`.

> **PITFALL**
>
> With `operations: [CREATE]`, `UPDATE` goes unchecked. Anyone who first creates a harmless Pod and then changes a field gets around the rule. Check `CREATE` and `UPDATE` together, unless you have a good reason.

> **PITFALL**
>
> `namespaceSelector` looks at **labels**. The name is only reachable through the automatic label `kubernetes.io/metadata.name`. Anyone who may set labels on Namespaces can thereby label themselves out of a policy. Protect these labels with RBAC or a policy of their own.

> **IN THE INTERVIEW**
>
> **How do you exclude `kube-system`?**
>
> Globally in the webhook, via `config.webhooks` in the Helm chart. That's the default. For individual policies, via `namespaceSelector`. The global exclusion saves calls, but then no policy at all applies there.

> **IN THE INTERVIEW**
>
> **`matchConstraints` or `matchConditions`?**
>
> `matchConstraints` for the coarse filter, because they control what reaches Kyverno at all. `matchConditions` for everything only CEL can express.

KEY TAKEAWAYS

- First `matchConstraints`, then `matchConditions`.
- Check `CREATE` and `UPDATE` together.
- Labels on Namespaces are security boundaries.

8

CEL in One Hour

Five patterns cover almost every policy.

CEL is a small expression language. There are no loops and no assignments, but there are list functions. An expression evaluates to a value.

What You Have Available

Variable	Contents
`object`	The new object.
`oldObject`	The old object on `UPDATE`.
`request`	The request: user, Namespace, operation.
`variables`	Your own variables.
`namespaceObject`	The resource's Namespace.

Pattern 1: Safe Access

If a field is missing, you get a runtime error. `?` protects against that:

```
object.spec.?securityContext.?runAsNonRoot.orValue(false)
```

Read it like this: "If the path exists, take the value, otherwise `false`."

Pattern 2: Checking Lists

```
object.spec.containers.all(c, c.image.contains(':'))
object.spec.containers.exists(c, c.name == 'app')
object.spec.containers.filter(c, !has(c.resources)).size()
object.spec.containers.map(c, c.image)
```

Pattern 3: All Containers

Don't forget the init containers:

```
object.spec.containers + object.spec.?initContainers.orValue([])
```

Pattern 4: Variables

Variables avoid repetition and make for better messages:

image-hygiene.yaml (excerpt)

```
spec:
  variables:
    - name: images
      expression: >-
        (object.spec.containers
          + object.spec.?initContainers.orValue([]))
        .map(c, c.image)
    - name: latest
      expression: >-
        variables.images.filter(i,
          !image(i).containsDigest()
            && image(i).tag() == 'latest')
  validations:
    - expression: "size(variables.latest) == 0"
      messageExpression: >-
        'No :latest: ' + variables.latest.join(', ')
```

Pattern 5: Kyverno Libraries

Kyverno extends CEL with its own functions:

Function	Purpose
`image(i)`	Breaks down an image reference: `registry()`, `tag()`, `containsDigest()`.
`resource.Get(...)`	Reads a resource from the API server.
`generator.Apply(...)`	Creates resources in a `GeneratingPolicy`.

```
resource.Get("v1", "configmaps",
    "kyverno", "allowed-registries")
  .data.registries.split(",")
```

> **PITFALL**
>
> `has(object.spec.foo.bar)` only protects the last field. If `foo` is already missing, it still blows up. Use `object.spec.?foo.?bar` and set a default at the end with `orValue`.

> **PITFALL**
>
> A runtime error in CEL is a policy error, not a violation. It triggers `failurePolicy`. With `Fail`, the request is rejected, even if the object is perfectly fine.

IN THE INTERVIEW

How do you check all containers of a Pod?

I combine `containers` and `initContainers` with `orValue([])` and use `all` or `filter`. Depending on the rule, ephemeral containers belong in there too. The check shouldn't skip any container.

IN THE INTERVIEW

Why is CEL suited to admission?

CEL isn't Turing-complete; there are no endless loops. Expressions are fast, have bounded cost, and can be type-checked in advance. That fits a path that has no time to spare.

KEY TAKEAWAYS

- `.?field.orValue(...)` instead of crashes.
- `all`, `exists`, `filter`, `map`.
- Always include `initContainers`.
- Runtime errors trigger `failurePolicy`.

9

Audit, Warn, Deny

Observe first, then block.

Action	Effect on violation
`Deny`	The request is rejected.
`Audit`	The request goes through. The violation lands in the report.
`Warn`	The request goes through. The client sees a warning.

`Audit` and `Warn` can be combined: `validationActions: [Audit, Warn]`.

> **PICTURE THIS**
>
> *`Deny` is the barrier, `Warn` the roadside sign, `Audit` the speed camera: you get through, but it's on record.*

The Safe Rollout

1. Roll out the policy with `[Audit]`.
2. Read the reports: who's in violation?
3. Inform teams, fix resources, sort out exceptions.
4. Switch to `[Audit, Warn]` so that everyone sees it in `kubectl`.
5. Switch to `[Deny]` only at zero violations.

Admission and Background

Every policy can run at two points in time:

- **Admission:** on a request.
- **Background:** as a regular scan of existing resources. The default interval is one hour.

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

Both are on by default. A new violation from today shows up at admission, an old Pod from last year in the background scan.

PITFALL

`Deny` also blocks `UPDATE`s of existing, non-compliant objects. A team may then not even be able to scale or patch an old Deployment, because the object doesn't meet the new rule. That's why `Audit` comes first.

PITFALL

The background scan has no user. There's no real request there. Rules that use `request.userInfo` must set `evaluation.background.enabled: false`.

IN THE INTERVIEW

How do you introduce a new policy without disrupting teams?

First `Audit`, then read reports, inform teams, and fix things. After that `Warn`, so everyone sees it, and `Deny` only at zero violations. All of it goes through Git, with tests, in stages per environment.

IN THE INTERVIEW

How do you find violations in existing resources?

With the background scan. It evaluates existing resources regularly and writes the result to policy reports.

KEY TAKEAWAYS

- `Deny`, `Audit`, `Warn`.
- Rollout: Audit, Warn, Deny.
- Admission for new things, Background for what's already there.

10

Fail or Ignore

What happens when Kyverno doesn't answer?

The most important question about any webhook.

`failurePolicy`	**Kyverno goes down**	**Risk**
`Fail`	Requests are rejected.	Availability: deployments grind to a halt.
`Ignore`	Requests go through unchecked.	Security: rules don't apply.

The default for the new policies is `Fail`.

> **PICTURE THIS**
>
> *A power outage at the security airlock. `Fail` means the door stays shut and nobody gets in. `Ignore` means the door stays open and everybody gets in. The bank picks one, the hospital the other.*

Decide per Policy

The choice depends on the purpose of the rule:

soft-labels.yaml (excerpt)

```yaml
spec:
  failurePolicy: Ignore
  webhookConfiguration:
    timeoutSeconds: 3
  validationActions: [Audit]
```

- **Security rule** such as "no privileged Pods": `Fail`.
- **Convenience or hygiene rule** such as a team label: `Ignore`.

The Timeout

`webhookConfiguration.timeoutSeconds` is between 1 and 30 seconds; the default is 10. When it expires, the `failurePolicy` kicks in. Long checks such as image signatures need more time, hygiene rules less.

PITFALL

`Fail` plus a single replica is a trap. One node restart is enough, and nobody can create Pods anymore, not even the new Kyverno Pod, if its Namespace weren't excluded. That's why the Kyverno Namespace is excluded by default.

REMEMBER

There's a third answer: also generate critical rules as a `ValidatingAdmissionPolicy`. It runs in the API server and needs no webhook. That's covered in chapter 28.

IN THE INTERVIEW

`failurePolicy`: `Fail` or `Ignore`?

It depends on the rule. Security rules get `Fail`, so nobody slips through during an outage. Hygiene rules get `Ignore`, so they can't bring the platform down. `Fail` requires HA and tested exclusions.

IN THE INTERVIEW

What happens if a CEL expression throws a runtime error?

That counts as a policy error, and the `failurePolicy` decides. With `Fail`, the request is rejected. With `Ignore`, the rule is skipped. That's why I write expressions defensively with `orValue`.

KEY TAKEAWAYS

- `Fail`: secure, but a risk to availability.
- `Ignore`: available, but a risk to security.
- Decide per rule; the default is `Fail`.

PART III

The Tools

Mutating, generating, images, deleting, Pod Security, exceptions, and reports.

11

MutatingPolicy

Fill in what's missing instead of rejecting.

Mutation makes life easier for teams: instead of reporting “field missing”, Kyverno sets a sensible default.

pod-defaults.yaml

```yaml
apiVersion: policies.kyverno.io/v1
kind: MutatingPolicy
metadata:
  name: pod-defaults
spec:
  matchConstraints:
    resourceRules:
      - apiGroups: [""]
        apiVersions: [v1]
        operations: [CREATE]
        resources: [pods]
  mutations:
    - patchType: ApplyConfiguration
      applyConfiguration:
        expression: >-
          Object{
            metadata: Object.metadata{
              labels: {"security/defaults": "set"}
            },
            spec: Object.spec{
              securityContext: Object.spec.securityContext{
                runAsNonRoot: object.spec.?securityContext
                  .?runAsNonRoot.orValue(true)
              }
            }
          }
```

The `orValue(true)` keeps what the user set and applies a default otherwise.

Two Patch Types

patchType	When
ApplyConfiguration	Setting fields. Reads like the target object.
JSONPatch	Precise operations, such as replacing a list element.

```
- patchType: JSONPatch
  jsonPatch:
    expression: >-
      [JSONPatch{
        op: "add",
        path: "/metadata/labels/checked",
        value: "yes"
      }]
```

> **PICTURE THIS**
>
> `ApplyConfiguration` *is a form where you fill in only the fields you want.* `JSONPatch` *is a work order: go to line 3 and replace the word.*

Mutating Existing Resources

Admission only sees new requests. With `mutateExisting`, Kyverno also changes resources that already exist, by way of the background controller.

```
spec:
  evaluation:
    mutateExisting:
      enabled: true
  targetMatchConstraints:
    resourceRules:
      - apiGroups: [apps]
        apiVersions: [v1]
        operations: ["*"]
        resources: [deployments]
```

`matchConstraints` describes the trigger, `targetMatchConstraints` the target.

Webhook Order

Multiple mutations can affect one another. `reinvocationPolicy` controls whether a policy runs again after later changes.

> **PITFALL**
>
> Mutation is invisible. The object in Git and the object in the cluster then differ. GitOps tools report that as drift, and in the worst case Argo CD and Kyverno fight each other. The Kyverno docs recommend Server-Side Diff for this. Alternatively, configure `ignoreDifferences` for those fields, or set the fields directly in Git.

PITFALL

Never mutate something the user set deliberately. Overwriting `image` or `resources` is a trap for troubleshooting. Defaults yes, secret changes no.

IN THE INTERVIEW

When do you mutate, and when do you validate?

Mutate when there's a clear default, such as `runAsNonRoot` or a label. Validate when the team has to decide for itself, such as with resource limits. I often combine both: mutation sets the default, validation enforces it.

KEY TAKEAWAYS

- `ApplyConfiguration` for fields, `JSONPatch` for precision.
- `mutateExisting` plus `targetMatchConstraints`.
- Watch out for GitOps drift.

12

GeneratingPolicy

An event triggers new resources.

The classic example: every new Namespace gets a default-deny NetworkPolicy.

tenant-netpol.yaml

```yaml
apiVersion: policies.kyverno.io/v1
kind: GeneratingPolicy
metadata:
  name: tenant-netpol
spec:
  evaluation:
    synchronize:
      enabled: true
  matchConstraints:
    resourceRules:
      - apiGroups: [""]
        apiVersions: [v1]
        operations: [CREATE, UPDATE]
        resources: [namespaces]
    objectSelector:
      matchExpressions:
        - {key: tenant, operator: Exists}
  variables:
    - name: policies
      expression: >-
        [{
          "apiVersion": dyn("networking.k8s.io/v1"),
          "kind": dyn("NetworkPolicy"),
          "metadata": dyn({"name": "default-deny"}),
          "spec": dyn({
            "podSelector": dyn({}),
            "policyTypes": dyn(["Ingress", "Egress"])
          })
        }]
  generate:
    - expression: >-
        generator.Apply(object.metadata.name,
          variables.policies)
```

`generator.Apply(namespace, list)` creates the objects in the target Namespace. `dyn(...)` tells CEL that the map values may have different types.

Cloning

Instead of typing data into the YAML, you copy a template:

clone-monitoring.yaml (excerpt)

```
spec:
  variables:
    - name: source
      expression: >-
        resource.Get("networking.k8s.io/v1", "networkpolicies",
          "netpol-templates", "allow-monitoring")
  generate:
    - expression: >-
        generator.Apply(object.metadata.name,
          [variables.source])
```

> **PICTURE THIS**
>
> *A retail chain. When a new branch opens, the basic equipment arrives automatically: fire extinguisher, house rules, cash register. When headquarters changes the house rules, they get swapped everywhere.*

The Switches

`evaluation.…`	Effect
`synchronize`	If someone deletes or changes the target, Kyverno restores it.
`generateExisting`	Also applies to triggers that already exist.
`orphanDownstreamOnPolicyDelete`	The target stays when the policy is deleted.

RBAC

The background controller creates the objects. For anything beyond the default permissions it needs additional permissions. Otherwise, the `UpdateRequest` stays stuck in an error state.

> **PITFALL**
>
> Without `generateExisting`, the policy only applies to future Namespaces. Existing Namespaces stay unprotected, and nobody notices. After rollout, check that the objects really exist everywhere.

PITFALL

With `synchronize`, the target effectively belongs to Kyverno. Manual changes get reverted. A team that tweaks its NetworkPolicy "just for a moment" is in for a surprise. Document this, or deliberately choose `synchronize: false` for such objects.

IN THE INTERVIEW

What do you use Generate for?

For tenant baseline equipment: default-deny NetworkPolicy, ResourceQuota, LimitRange, roles. That way, every new Namespace gets the rules automatically, and `synchronize` keeps them up to date.

IN THE INTERVIEW

Why does Generate run asynchronously?

Generate can take a long time and affect several objects. That must not hold up an admission request. So the admission controller creates an `UpdateRequest`, and the background controller works through it.

KEY TAKEAWAYS

- `generator.Apply` creates, `resource.Get` reads the template.
- `synchronize` keeps targets up to date.
- `generateExisting` for what already exists.
- The background controller needs matching RBAC.

13

ImageValidatingPolicy

Only signed images get into the cluster.

Supply chain security means checking **who** built an image, not just **what it's called**. A tag is only a name. A signature proves origin.

The Building Blocks

- **Signature:** The publisher signs the image digest.
- **Cosign:** Sigstore tool for signing, with a key or keyless via OIDC.
- **Notary:** An alternative using X.509 certificates.
- **Attestation:** A signed statement about the image, such as an SBOM or SLSA provenance.

A Policy with Cosign

verify-shop.yaml

```yaml
apiVersion: policies.kyverno.io/v1
kind: ImageValidatingPolicy
metadata:
  name: verify-shop-images
spec:
  validationActions: [Deny]
  webhookConfiguration:
    timeoutSeconds: 15
  matchConstraints:
    resourceRules:
      - apiGroups: [""]
        apiVersions: [v1]
        operations: [CREATE, UPDATE]
        resources: [pods]
  matchImageReferences:
    - glob: "registry.example.com/shop/*"
  attestors:
    - name: team
      cosign:
        key:
          data: |
            -----BEGIN PUBLIC KEY-----
            ...
```

```
            -----END PUBLIC KEY-----
  validations:
    - message: "Image is not signed by the team."
      expression: >-
        images.containers.map(i,
          verifyImageSignatures(i, [attestors.team]))
        .all(n, n > 0)
```

`verifyImageSignatures` returns the number of valid signatures. `matchImageReferences` limits the check to matching images.

Keyless

Without a key of your own, you check who signed, such as a specific GitHub workflow:

```
attestors:
  - name: ci
    cosign:
      keyless:
        identities:
          - subject: https://github.com/company/shop/...
            issuer: https://token.actions...
      ctlog:
        url: https://rekor.sigstore.dev
```

Verifying Attestations

```
attestations:
  - name: sbom
    intoto:
      type: https://spdx.dev/Document
validations:
  - message: "No signed SBOM."
    expression: >-
      images.containers.map(i, verifyAttestationSignatures(i,
        attestations.sbom, [attestors.team])).all(n, n > 0)
```

With `extractPayload(i, attestations.sbom)` you read the content of the attestation and check it in CEL.

> **PICTURE THIS**
>
> *A tag is a sticker anyone can apply. A signature is the company seal, an attestation the notarized certificate of origin.*

Tag or Digest?

Tags are mutable. If you check `:1.2` and a different image with the same tag runs afterward, the check is worthless. So enforce digests or let Kyverno resolve the reference.

> **PITFALL**
>
> With `Fail`, Deployments depend on the registry. If the registry goes down or is slow, the check fails. Raise `timeoutSeconds` and keep pull credentials up to date.

> **PITFALL**
>
> Check all container kinds. `images.containers` alone leaves out `initContainers` and `ephemeralContainers`, because both are separate keys in `images`. Collect them in a variable. For ephemeral containers, the policy must also match the subresource `pods/ephemeralcontainers`. An attacker doesn't have to start their code in the main container.

```
variables:
  - name: allImages
    expression: >-
      images.?containers.orValue([])
      + images.?initContainers.orValue([])
      + images.?ephemeralContainers.orValue([])
```

IN THE INTERVIEW

How do you enforce signed images?

With an `ImageValidatingPolicy`. It matches images by glob and verifies the signature with an attestor, meaning a Cosign key, keyless identity, or Notary certificate. I can additionally verify and evaluate attestations such as an SBOM.

IN THE INTERVIEW

Why isn't a list of allowed registries enough?

The list only says where an image comes from, not who built it. Anyone who can push to the registry gets through. A signature proves it came from your CI.

KEY TAKEAWAYS

- Signature instead of name.
- `attestors`, `attestations`, `validations`.
- `verifyImageSignatures` and `verifyAttestationSignatures`.
- High timeout, and digests instead of tags.

14

DeletingPolicy

Cleaning up on schedule.

A `DeletingPolicy` deletes resources on a cron schedule.

delete-finished-pods.yaml

```yaml
apiVersion: policies.kyverno.io/v1
kind: DeletingPolicy
metadata:
  name: delete-finished-pods
spec:
  schedule: "*/15 * * * *"
  matchConstraints:
    resourceRules:
      - apiGroups: [""]
        apiVersions: [v1]
        resources: [pods]
    namespaceSelector:
      matchExpressions:
        - {key: tenant, operator: Exists}
  conditions:
    - name: finished
      expression: >-
        object.status.?phase.orValue('')
          in ['Succeeded', 'Failed']
    - name: older-than-1h
      expression: >-
        time.now()
          - timestamp(object.metadata.creationTimestamp)
          > duration('1h')
```

You can also choose `deletionPropagationPolicy`: `Foreground`, `Background`, or `Orphan`.

The TTL Shortcut

For individual objects, a label is enough:

```
kubectl -n demo label pod test cleanup.kyverno.io/ttl=2h
```

Memory aid: An expiration date on the package. After two hours, Kyverno cleans up.

RBAC

The cleanup controller needs permission to delete the resources. You grant it through an additional ClusterRole. Otherwise, nothing happens.

> **PITFALL**
>
> One wrong condition deletes things en masse. Test the condition first as a `ValidatingPolicy` in `Audit`, or with `kubectl get` and `jq`. A `DeletingPolicy` has no recycle bin.

> **IN THE INTERVIEW**
>
> **What do you use deletion policies for?**
>
> For finished Pods and Jobs, expired exceptions, or test environments with an expiration date. They spare you hand-written CronJobs and come with clear conditions in CEL.

KEY TAKEAWAYS

- `schedule`, `matchConstraints`, `conditions`.
- Label `cleanup.kyverno.io/ttl` for individual objects.
- Dry-run the condition first.

15

Pod Security Standards

Kubernetes already wrote the rules.

The **Pod Security Standards** (PSS) define three levels:

Level	Contents
`privileged`	No restrictions. For system services.
`baseline`	Prevents known privilege escalations.
`restricted`	Strict hardening based on proven rules.

Built in for this is **Pod Security Admission** (PSA), controlled via labels on the Namespace:

```
kubectl label namespace demo pod-security.kubernetes.io/enforce=baseline
```

Memory aid: The label is the rule. Level `baseline` for this Namespace.

PSA or Kyverno?

	PSA	Kyverno
Exceptions	Coarse: Namespace, user, RuntimeClass.	Fine-grained: per workload.
Message	Fixed.	Your own text.
Dependency	None, in the API server.	Webhook.
Rule scope	Only the three levels.	Anything.

The two are often combined: PSA as the base, Kyverno for the fine points.

PSS with CEL

The new policies don't know the old `podSecurity` shorthand. You write each check as an expression:

pss-restricted.yaml (excerpt)

```
spec:
  validationActions: [Audit]
```

```
  variables:
    - name: containers
      expression: >-
        object.spec.containers
          + object.spec.?initContainers.orValue([])
  validations:
    - expression: >-
        variables.containers.all(c, c.?securityContext
          .?allowPrivilegeEscalation.orValue(true) == false)
      message: "allowPrivilegeEscalation: false is missing."
    - expression: >-
        variables.containers.all(c, 'ALL' in c.?securityContext
          .?capabilities.?drop.orValue([]))
      message: "capabilities.drop must contain ALL."
```

You'll find ready-made rules in the policy library at `kyverno.io/policies`. You don't have to reinvent them.

IN THE INTERVIEW

PSA or Kyverno for Pod Security?

PSA is built in, needs no webhook, and is robust, but it's coarse. Kyverno offers fine-grained exceptions, custom messages, and reports. I often use PSA as the baseline and Kyverno for the details.

KEY TAKEAWAYS

- `privileged`, `baseline`, `restricted`.
- PSA is controlled by a Namespace label.
- CEL policies check each rule individually.

16

PolicyException

Exceptions are normal. Uncontrolled exceptions are not.

Exceptions used to go into the policy itself. That makes the policy unreadable, and every team needs write access to it. So a `PolicyException` is a resource of its own.

legacy-batch.yaml

```yaml
apiVersion: policies.kyverno.io/v1
kind: PolicyException
metadata:
  name: legacy-batch-latest
  namespace: policy-exceptions
spec:
  policyRefs:
    - name: image-hygiene
      kind: ValidatingPolicy
  matchConditions:
    - name: legacy-batch-only
      expression: >-
        object.metadata.namespace == 'shop' &&
        object.metadata.name.startsWith('legacy-batch')
  expiresAt: "2026-12-31T23:59:59Z"
  properties:
    ticket: SEC-4711
    owner: team-shop
```

- `policyRefs`: Which policy does the exception apply to?
- `matchConditions`: Which objects?
- `expiresAt`: When does it end?

> **REMEMBER**
>
> Exceptions are off by default. You turn them on in the Helm chart: `features.policyExceptions.enabled=true` and `features.policyExceptions.namespace=policy-exceptions`.

PICTURE THIS

A doctor's note: with a name, a reason, and an expiration date. It doesn't get written into the law. It sits in a separate file, and someone guards that file.

Governing Exceptions

- A dedicated Namespace, and only the security team may write there.
- Exceptions go through pull requests with review.
- Every exception has `expiresAt`, a ticket, and an owner. Enforce that with a policy of its own.
- A `DeletingPolicy` clears away expired exceptions.

PITFALL

Anyone who can freely create `PolicyException`s can bypass every policy. The Namespace for exceptions is a security boundary. Limit write access with RBAC, and never set the value `"*"` for all Namespaces unless you know exactly why.

IN THE INTERVIEW

How do you handle exceptions?

As a separate `PolicyException` resource with an expiration date, ticket, and owner, in a protected Namespace and with review. That keeps the policy clean, and the exceptions stay auditable.

KEY TAKEAWAYS

- `PolicyException` is off by default.
- `policyRefs`, `matchConditions`, `expiresAt`.
- The exception Namespace is a security boundary.

17

Policy Reports

What's violating what right now?

Policy reports are resources. They follow a standard from the Kubernetes Policy Working Group (`wgpolicyk8s.io`). That stays the same in Kyverno 1.20.

```
kubectl get polr -A
```

Memory aid: **polr** as in **pol**icy **r**eport. One report per resource, with counters.

```
kubectl get cpolr
```

Memory aid: The same for cluster-wide resources.

```
NAMESPACE  NAME     KIND  NAME  PASS  FAIL  WARN  ERROR  SKIP
demo       0b2d...  Pod   test  2     1     0     0      0
```

```
kubectl -n demo get polr 0b2d730e -o yaml
```

Memory aid: Shows the individual results. Fill in the name from the list.

```
results:
  - policy: image-hygiene
    result: fail
    message: "No :latest: nginx:latest"
    source: KyvernoValidatingPolicy
    properties:
      process: background scan
```

Results

Result	Meaning
`pass`	Rule satisfied.
`fail`	Violation.
`warn`	Warning.
`error`	Error during evaluation.
`skip`	Skipped, for example because of an exception.

Visualizing

With many reports, `kubectl` gets tedious. **Policy Reporter** shows them in a UI and sends notifications to Slack, Teams, or Prometheus.

> **REMEMBER**
>
> Reports need storage in etcd. With many resources, that can strain capacity. Keep an eye on their number. Optionally, reports can be stored elsewhere via OpenReports (`openreports.enabled`, off by default).

IN THE INTERVIEW

How do you see who's violating policies?

Through policy reports, which Kyverno writes at admission and during the background scan. `kubectl get polr -A` gives the overview. For daily work I use Policy Reporter and Prometheus.

KEY TAKEAWAYS

- `polr` in the Namespace, `cpolr` cluster-wide.
- `pass`, `fail`, `warn`, `error`, `skip`.
- Reports take up etcd space. Keep an eye on them.

PART IV

The Classic World

ClusterPolicy, JMESPath, and migration. You'll keep running into them for a long time.

18

Reading a ClusterPolicy

The old form that's still sitting in almost every cluster.

> **REMEMBER**
>
> This part is for reading, operating, and migrating. Write new policies as a CEL type. `ClusterPolicy` and `Policy` have been deprecated since 1.19.

A `ClusterPolicy` applies cluster-wide, a `Policy` to one Namespace. Both contain a list of **rules**. Each rule does exactly one thing: `validate`, `mutate`, `generate`, or `verifyImages`.

require-app-label-old.yaml

```yaml
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: require-app-label
spec:
  background: true
  rules:
    - name: check-app
      match:
        any:
          - resources:
              kinds: [Pod]
      exclude:
        any:
          - resources:
              namespaces: [kube-system]
      validate:
        failureAction: Enforce
        message: "The label app is missing."
        pattern:
          metadata:
            labels:
              app: "?*"
```

`match` and `exclude`

- `any`: One of the conditions is enough (OR).
- `all`: All must match (AND).

- Filters exist for `kinds`, `names`, `namespaces`, `selector`, `operations`, `namespaceSelector`, and for users via `subjects`, `roles`, `clusterRoles`.

```
exclude:
  any:
    - clusterRoles: [cluster-admin]
```

`failureAction`

Old	New
`Enforce`	`validationActions: [Deny]`
`Audit`	`validationActions: [Audit]`
`emitWarning: true`	`validationActions: [Warn]`

Older policies still set the field at the policy level as `spec.validationFailureAction`.

> **PITFALL**
>
> The names are deceptive: `failurePolicy` governs **Kyverno going down**. `failureAction` governs a **violation of the rule**. Those are two separate switches, and it's a popular interview question.

> **IN THE INTERVIEW**
>
> **`failureAction` or `failurePolicy`?**
>
> `failureAction` says what happens on a violation: `Enforce` or `Audit`. `failurePolicy` says what happens when the webhook doesn't respond: `Fail` or `Ignore`. In the new policies, the former is called `validationActions`.

KEY TAKEAWAYS

- One policy, many rules, each rule one kind.
- `match` and `exclude` with `any` and `all`.
- `failureAction` is not `failurePolicy`.

19

Validate: pattern and deny

Two ways to express a rule.

`pattern`: This Is How It Must Look

A `pattern` is a piece of YAML that must match the object.

```
validate:
  pattern:
    spec:
      containers:
        - image: "!*:latest"
          resources:
            limits:
              memory: "?*"
```

Character	Meaning
`*`	Any number of characters.
`?`	Exactly one character. `?*` means: not empty.
`!`	Not.
`|`	Or.
`>`, `<`	Comparison, such as `>=2`.

A list in a pattern applies to **all** elements of the list.

Anchors

Parentheses around the field name change the logic:

Anchor	Meaning
`(field)`	Condition: check the rest only if this matches.
`=(field)`	If the field exists, it must match.
`^(field)`	At least one list element must match.
`X(field)`	The field must not exist.
`+(field)`	Mutation: set only if missing.

```
pattern:
  spec:
    =(securityContext):
      =(runAsUser): ">0"
```

> **PICTURE THIS**
>
> *A pattern is a stencil. If it lies on the object and fits, all is well. Anchors are holes in the stencil with conditions.*

`deny`: This Is Forbidden

`deny` flips the logic. If the conditions are true, the request is rejected.

```
validate:
  message: "Deletion for admins only."
  deny:
    conditions:
      all:
        - key: "{{ request.operation }}"
          operator: Equals
          value: DELETE
```

Important operators are `Equals`, `NotEquals`, `AnyIn`, `AllIn`, `AnyNotIn`, `AllNotIn`, `GreaterThan`, and `LessThan`.

> **IN THE INTERVIEW**
>
> **`pattern` or `deny`?**
>
> `pattern` describes what an object should look like and is good for field checks. `deny` describes forbidden conditions and can check variables and operations such as `DELETE`. In CEL, both exist only as expressions, with `deny` using inverted logic.

KEY TAKEAWAYS

- `pattern`: stencil. `deny`: forbidden situation.
- `?*` means not empty.
- Anchors control conditions.

20

Variables and JMESPath

Double curly braces everywhere.

In the old world, everything sits in `{{ }}`. Inside is a **JMESPath** expression, a query language for JSON.

Variable	Contents
`{{ request.object }}`	The new object.
`{{ request.oldObject }}`	The old object.
`{{ request.operation }}`	`CREATE`, `UPDATE`, `DELETE`.
`{{ request.namespace }}`	The Namespace.
`{{ request.userInfo.username }}`	The requesting user.
`{{ serviceAccountName }}`	The ServiceAccount.
`{{ images.containers }}`	Parsed image references.
`{{ element }}`	The current element in `foreach`.

Reading JMESPath

```
request.object.spec.containers[].name
request.object.spec.containers[?securityContext.privileged]
request.object.metadata.labels.team || 'unknown'
request.object.spec.containers[] | length(@)
request.object.metadata.name | to_upper(@)
```

- `[]` flattens a list.
- `[?...]` filters.
- `||` sets a default.
- `|` passes the result along, and `@` is the current element.

`context`: Data from Outside

```yaml
context:
  - name: allowed
    configMap:
      name: allowed-registries
```

```
      namespace: kyverno
  - name: count
    apiCall:
      urlPath: /api/v1/namespaces/{{ request.namespace }}/pods
      jmesPath: items | length(@)
```

There's also `variable` for your own values, `imageRegistry` for image data, and `globalReference` for Kyverno's global cache.

Old	New in CEL
`{{ request.object }}`	`object`
`context` with `apiCall`	`variables` with `resource.Get`
`{{ x || 'y' }}`	`.?x.orValue('y')`
`[?...]`	`.filter(...)`

> **PITFALL**
>
> If a value in `{{ }}` is missing, the substitution fails, and the rule returns an error. Set a default with `||`. Kyverno also evaluates variables inside text. If you literally need `{{`, you have to protect it with a backslash, for example in Helm templates.

IN THE INTERVIEW

How do you read a ConfigMap in a policy?

In the `ClusterPolicy` via `context` with `configMap`. In the new world via `variables` with `resource.Get`. That way, lists such as allowed registries stay editable outside the policy.

KEY TAKEAWAYS

- `{{ request.object... }}` is old, `object` is new.
- `||` for defaults, `|` for passing along.
- `context` fetches ConfigMaps and API data.

21

Preconditions and foreach

When a rule applies, and how it walks through lists.

Preconditions

`match` selects by kind and name. `preconditions` use variables to decide whether the rule runs at all. If they're false, the rule is skipped (`skip`), not rejected.

```
preconditions:
  all:
    - key: "{{ request.operation }}"
      operator: AnyIn
      value: [CREATE, UPDATE]
    - key: "{{ request.object.metadata.labels.stage || '' }}"
      operator: Equals
      value: prod
```

`any` means OR, `all` means AND. In CEL this becomes `spec.matchConditions`.

`foreach`

`foreach` walks through a list. The element is called `element`, the position `elementIndex`.

```
validate:
  message: "Image {{ element.image }} is not allowed."
  foreach:
    - list: request.object.spec.containers
      deny:
        conditions:
          any:
            - key: "{{ element.image }}"
              operator: Equals
              value: "*:latest"
```

`foreach` also exists in `mutate` and `generate`:

```
mutate:
  foreach:
    - list: request.object.spec.containers
      patchStrategicMerge:
```

```
        spec:
          containers:
            - name: "{{ element.name }}"
              image: "registry.io/{{ element.image }}"
```

> **PICTURE THIS**
>
> *`preconditions` is the question "Is this even my job?" `foreach` is the inspector who opens every box on the pallet individually.*

In CEL, `foreach` goes away. You use `all()`, `exists()`, `filter()`, and `map()`.

> **PITFALL**
>
> Mutations with `foreach` over lists are tricky. An index in the JSON Patch can shift because of earlier changes. That's why, for deletions, the list runs backward (`order: Descending`). With nested loops, test this especially thoroughly.

IN THE INTERVIEW

What's the difference between `match` and `preconditions`?

`match` filters which resources are even candidates and controls the webhooks. `preconditions` then evaluates variables and skips the rule on `false`. In CEL they're called `matchConstraints` and `matchConditions`.

KEY TAKEAWAYS

- `preconditions` false: `skip`, not rejection.
- `foreach` with `element`.
- In CEL: `matchConditions` and list functions.

22

Mutate, Generate, verifyImages

The other three rule kinds in brief.

`mutate`

```yaml
mutate:
  patchStrategicMerge:
    metadata:
      labels:
        +(team): unknown
    spec:
      containers:
        - (name): "*"
          imagePullPolicy: IfNotPresent
```

- `+(team)`: set only if missing.
- `(name): "*"`: for every container.

For precision, there's `patchesJson6902`. For existing resources, `mutateExistingOnPolicyUpdate` and `targets`. In the new policies, `patchType: ApplyConfiguration` and `JSONPatch` take their place.

`generate`

```yaml
match:
  any:
    - resources:
        kinds: [Namespace]
generate:
  apiVersion: networking.k8s.io/v1
  kind: NetworkPolicy
  name: default-deny
  namespace: "{{ request.object.metadata.name }}"
  synchronize: true
  data:
    spec:
      podSelector: {}
      policyTypes: [Ingress, Egress]
```

Instead of `data`, you can also use `clone` with the `namespace` and `name` of the template.

`verifyImages`

```
verifyImages:
  - imageReferences:
      - "registry.example.com/shop/*"
    mutateDigest: true
    verifyDigest: true
    required: true
    attestors:
      - entries:
          - keys:
              publicKeys: |-
                -----BEGIN PUBLIC KEY-----
                ...
                -----END PUBLIC KEY-----
```

`mutateDigest` replaces the tag with the digest. That way, exactly the verified image runs.

Autogen in the Old World

Autogen is on by default for `ClusterPolicy`. You can see the generated rules in the status as `autogen-...`, and you turn them off with an annotation:

```
metadata:
  annotations:
    pod-policies.kyverno.io/autogen-controllers: none
```

IN THE INTERVIEW

Why does `verifyImages` insert the digest?

So that the verified image is also the one that runs. A tag can change between verification and pull. With a digest, that gap doesn't exist.

KEY TAKEAWAYS

- `+(field)` only if missing.
- `generate` with `data` or `clone`.
- `mutateDigest` closes the tag gap.

23

Old Cleanups and Exceptions

CleanupPolicy and PolicyException from `kyverno.io`.

`ClusterCleanupPolicy`

```yaml
apiVersion: kyverno.io/v2
kind: ClusterCleanupPolicy
metadata:
  name: old-pods
spec:
  schedule: "0 * * * *"
  match:
    any:
      - resources:
          kinds: [Pod]
  conditions:
    any:
      - key: "{{ target.status.phase }}"
        operator: Equals
        value: Succeeded
```

Here the object itself is called `target`. Internally, the cleanup controller creates CronJobs for this. The new form is the `DeletingPolicy` from chapter 14.

Old `PolicyException`

```yaml
apiVersion: kyverno.io/v2
kind: PolicyException
metadata:
  name: batch-exception
  namespace: policy-exceptions
spec:
  exceptions:
    - policyName: require-app-label
      ruleNames: [check-app, autogen-check-app]
  match:
    any:
      - resources:
          kinds: [Pod]
          namespaces: [shop]
```

Note the `autogen-...`: the exception must also name the generated rules.

Old	New
`kyverno.io PolicyException`	`policies.kyverno.io PolicyException`
`exceptions[].policyName`	`policyRefs[].name`, `kind`
`match`	`matchConditions` with CEL
Time via cleanup	`expiresAt`

KEY TAKEAWAYS

- `CleanupPolicy` becomes `DeletingPolicy`.
- With old exceptions, don't forget the `autogen-` rules.
- New exceptions name the policy and the type.

24

Migrating to CEL

How to switch 200 ClusterPolicies over safely.

Why Now?

From 1.20 on, an old policy is no longer enforced. It stays readable, but it no longer protects anything.

The Translation

`ClusterPolicy`	**CEL policy**
`rules[].match`	`matchConstraints`, `matchConditions`
`rules[].exclude`	`matchConstraints.excludeResourceRules`
`rules[].context`	`variables`
`preconditions`	`matchConditions`
`validate.pattern`	`validations` with `expression`
`validate.deny`	`validations`, invert the logic
`validate.foreach`	`all()`, `exists()`, `exists_one()`
`validate.podSecurity`	Not available. Each rule individually.
`validate.manifests`	Not available.
`allowExistingViolations`	Not available. Use exceptions.
`failureActionOverrides`	Not available. Use exceptions.
`mutate.patchStrategicMerge`	`patchType: ApplyConfiguration` or `JSONPatch`
`generate.data`, `clone`	`generate` with CEL
`verifyImages`	`ImageValidatingPolicy`
`background`	`evaluation.background.enabled`

One Rule, One Policy

A `ClusterPolicy` with five rules becomes several policies. `applyRules` goes away. You split the rules up and use `matchConditions`.

The Plan

1. **Inventory:** `kubectl get cpol,pol -A` and count the policies in Git.
2. **Clean up:** What's superfluous? What does PSA already cover?

3. **Translate:** with the Migration Guide, one policy at a time.
4. **Test:** The existing CLI tests keep running. Check the new policy against the same examples.
5. **Run in parallel:** New policy in `Audit` alongside the old one. Compare reports.
6. **Switch over:** New policy to `Deny`, remove the old one.
7. **Upgrade:** Only when no old policy is active anymore.

> **PICTURE THIS**
>
> *Moving from one bridge to another. You close the old one only once traffic is provably running over the new one.*

> **PITFALL**
>
> With `deny`, the logic flips. What used to mean "forbidden if true" now means "allowed if true". A forgotten `!` turns the rule into its opposite. So write negative tests first.

> **PITFALL**
>
> Autogen behaves differently. With `ClusterPolicy` it was on automatically; with the new types you turn on `autogen.podControllers` deliberately. Forget that, and Deployments get through again, and the errors only show up at the Pods.

IN THE INTERVIEW

How do you migrate from `ClusterPolicy` to CEL?

Inventory, cleanup, translate with the Migration Guide, reuse the tests. Then run new policies in `Audit` alongside the old one and compare the reports. Only then switch over and upgrade to 1.20. Until then I deliberately stay on 1.19.

KEY TAKEAWAYS

- One rule becomes one policy.
- Invert `deny`, turn on Autogen.
- Run in parallel in `Audit`, then switch over.

PART V

Tests and Operations

CLI, CI, GitOps, high availability, performance, and upgrades.

25

Kyverno CLI

Check policies without needing a cluster.

```
brew install kyverno
```

Memory aid: Or as a plugin: `kubectl krew install kyverno`.

```
kyverno version
```

Memory aid: Does the CLI match the cluster version? It should.

Applying a Policy

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

Memory aid: Read it as a sentence: “Apply this policy to this resource.” Everything happens locally.

```
Applying 1 policy rule(s) to 1 resource(s)...
pass: 0, fail: 1, warn: 0, error: 0, skip: 0
```

```
kyverno apply require-app-label.yaml --cluster
```

Memory aid: Against everything in the current cluster. Read-only, a cheap preview of the rollout.

```
kyverno apply policy.yaml --resource pod.yaml --exception
exception.yaml
```

Memory aid: Also tests the exception. Expected result: `skip`.

The Trial Run in the Cluster

`kubectl` helps too, once policies are in the cluster. A server dry run goes through the admission chain but saves nothing:

```
kubectl apply -f pod.yaml --dry-run=server
```

Memory aid: `server`, not `client`. Kyverno checks, but nothing lands in etcd.

> **REMEMBER**
>
> For experiments without installing anything, there's the Kyverno Playground in the browser (`playground.kyverno.io`). It also supports the new policy types.

IN THE INTERVIEW

How do you test a policy before it reaches the cluster?

Locally with `kyverno apply`, and `kyverno test` in CI. Then `--cluster` against existing resources as a preview and `--dry-run=server` for individual objects. In the cluster, it starts as `Audit`.

KEY TAKEAWAYS

- `kyverno apply` works without a cluster.
- `--cluster` shows what would fail among existing resources.
- `--dry-run=server` goes through the webhooks.

26

Tests

A policy without a test is a guess.

Unit Tests with `kyverno test`

Put a file named `kyverno-test.yaml` next to the policy.

tests/image-hygiene/kyverno-test.yaml

```yaml
apiVersion: cli.kyverno.io/v1alpha1
kind: Test
metadata:
  name: image-hygiene
policies:
  - ../../policies/image-hygiene.yaml
resources:
  - resources.yaml
exceptions:
  - ../../exceptions/legacy-batch.yaml
variables: values.yaml
results:
  - isValidatingPolicy: true
    policy: image-hygiene
    kind: Pod
    resources: [good]
    result: pass
  - isValidatingPolicy: true
    policy: image-hygiene
    kind: Pod
    resources: [latest, no-tag]
    result: fail
  - isValidatingPolicy: true
    policy: image-hygiene
    kind: Pod
    resources: [legacy-batch-1]
    result: skip
```

`values.yaml` supplies data that's missing locally, such as the labels of a Namespace:

tests/image-hygiene/values.yaml

```yaml
apiVersion: cli.kyverno.io/v1alpha1
kind: Value
```

```
metadata:
  name: values
namespaceSelector:
  - name: shop
    labels:
      team: lab
```

```
kyverno test tests/
```

Memory aid: Finds all test files in the folder and runs them.

> **PICTURE THIS**
>
> *Every rule needs two test cases: one that must pass and one that must fail. A door has only been tested once the right key opens it and the wrong one doesn't.*

E2E with Chainsaw

Chainsaw is the Kyverno project's test tool. It runs against a real cluster and checks how the webhooks behave.

e2e/image-hygiene/chainsaw-test.yaml (excerpt)

```
apiVersion: chainsaw.kyverno.io/v1alpha1
kind: Test
metadata:
  name: image-hygiene
spec:
  steps:
    - name: good-pod-allowed
      try:
        - apply:
            file: pod-good.yaml
    - name: latest-is-rejected
      try:
        - apply:
            file: pod-latest.yaml
            expect:
              - check:
                  ($error != null): true
```

```
chainsaw test e2e/
```

Memory aid: Plays the steps against the current cluster.

	`kyverno test`	**Chainsaw**
Needs	No cluster.	A real cluster.
Checks	The policy's logic.	The whole interaction.
Speed	Seconds.	Minutes.

PITFALL

Test `skip` cases too. A policy that never matches produces no errors and looks healthy. So check that the rule actually hits something.

IN THE INTERVIEW

How do you test policies?

Unit tests with `kyverno test`: positive, negative, and exception cases, in every pull request. Plus E2E with Chainsaw in a kind cluster, because only that really tests webhooks, Autogen, and RBAC.

KEY TAKEAWAYS

- One `pass` and one `fail` per rule.
- `kyverno test` is fast, Chainsaw is real.
- Test `skip` too.

27

CI and GitOps

Policies live in Git like any other code.

The Pipeline

```
Pull Request
  -> Lint and schema
  -> kyverno test
  -> Chainsaw in kind
  -> Review
Merge
  -> Argo CD or Flux syncs
```

.github/workflows/policies.yaml (excerpt)

```yaml
jobs:
  unit:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v7
      - uses: kyverno/action-install-cli@v0.2.0
        with:
          release: v1.19.2
      - run: kyverno test tests/
```

Checking Workloads in CI

`kyverno apply` can also run in the application repo against rendered manifests:

```
helm template shop ./chart > manifest.yaml
kyverno apply policies/ --resource manifest.yaml
```

That way, developers see the error in the pull request, not at deployment. This is **Shift Left**.

GitOps Pitfalls

Problem	Solution
Argo CD reports drift after mutation.	Server-Side Diff with `IncludeMutationWebhook=true`, otherwise `ignoreDifferences`.
Drift on aggregated ClusterRoles.	Exempt these roles with `ignoreDifferences`.
Argo CD and the Kyverno chart use the same label.	Choose a different tracking method in Argo CD.
Post-delete hook gets in the way.	Set `config.preserve=false`.
Generated objects appear as foreign.	Clarify ownership: only one party manages an object.
Kyverno CRDs are too large.	Use `ServerSideApply=true`, avoid `Replace=true`.
Reports clutter the UI.	Hide reports in Argo CD.
Policy applied before CRD.	CRDs before policies, for example with the annotation `argocd.argoproj.io/sync-wave`.

Stages

Roll policies out across environments: `dev` in `Deny`, `prod` still in `Audit`, and only then everywhere. Kustomize overlays work well for changing `validationActions` per environment.

> **PITFALL**
>
> If Kyverno is updated in the same Argo CD sync as workloads, a failure can block both. Argo CD itself needs Pods to repair things. Keep an emergency path ready that works without GitOps (chapter 33).

IN THE INTERVIEW

How do you get policies into the cluster?

Through GitOps: policies live in the repo, pull requests go through `kyverno test` and Chainsaw, Argo CD syncs. Environments differ by overlay, for example `Audit` in production and `Deny` in development.

KEY TAKEAWAYS

- Pipeline: lint, unit, E2E, review, sync.
- Shift Left: check manifests in CI already.
- Mutation creates drift. Plan for it.

28

High Availability

If Kyverno is in the path, it has to stay up.

Controller	Replicas	Scaling
Admission	At least 3	All handle webhooks. More replicas bring more performance.
Background	2	One leader. More replicas only for failover.
Reports	2	One leader. More replicas only for failover.
Cleanup	2	Multiple replicas increase throughput.

For the admission controller, three is the minimum Kyverno supports as highly available.

values-ha.yaml

```yaml
admissionController:
  replicas: 3
  podDisruptionBudget:
    enabled: true
    minAvailable: 2
  topologySpreadConstraints:
    - maxSkew: 1
      topologyKey: kubernetes.io/hostname
      whenUnsatisfiable: ScheduleAnyway
      labelSelector:
        matchLabels:
          app.kubernetes.io/component: admission-controller
backgroundController:
  replicas: 2
cleanupController:
  replicas: 2
reportsController:
  replicas: 2
config:
  webhooks:
    namespaceSelector:
      matchExpressions:
        - key: kubernetes.io/metadata.name
          operator: NotIn
          values: [kube-system]
```

> **PICTURE THIS**
>
> *Three counters, three clerks, spread across different buildings. A power outage in one building must not shut down the service.*

Webhook Exclusions

By default, the chart excludes `kube-system` in the `namespaceSelector`, and the Kyverno Namespace is excluded as well. These exceptions save you when the cluster restarts. But they come at a price: no policy applies there. So protect these Namespaces with RBAC.

Critical Rules Without a Webhook

You can additionally generate a `ValidatingPolicy` as a native `ValidatingAdmissionPolicy`. It runs in the API server and works even when Kyverno goes down:

```
spec:
  autogen:
    validatingAdmissionPolicy:
      enabled: true
```

This is mutually exclusive with `autogen.podControllers`, because a VAP can't carry the Pod controller variants. `MutatingPolicy` has a similar option too.

> **PITFALL**
>
> Two replicas look like HA, but they aren't. According to Kyverno, three is the minimum. Without a PodDisruptionBudget, a node upgrade can also evict all replicas at the same time.

> **IN THE INTERVIEW**
>
> **How do you make Kyverno highly available?**
>
> Three admission replicas on different nodes with a PodDisruptionBudget, two for the other controllers. `kube-system` and `kyverno` are excluded. I also generate critical rules as native VAPs. On top of that come alerts on latency and availability.

IN THE INTERVIEW

Why don't more reports replicas bring more performance?

The controller uses leader election. Only one replica works actively. The others are standbys. The only thing that adds performance there is more CPU and RAM.

KEY TAKEAWAYS

- Admission 3, the rest 2.
- PDB and Topology Spread.
- Exclude `kube-system` and `kyverno`.
- Critical rules also as VAPs.

29

Performance

Every millisecond sits in the API server's path.

The Levers

Lever	Effect
Narrow `matchConstraints`	Kyverno gets only what it needs.
No wildcards	No `resources: ["*"]`. That sends almost everything to Kyverno.
`config.resourceFilters`	Ignores noise such as Events.
`namespaceSelector`	Saves calls at the webhook level.
Native VAP	Runs in the API server, no network hop.
`GlobalContextEntry`	Cache for external data.
More admission replicas	More parallel requests.

Expensive Things

- `resource.Get` or `apiCall` per request calls the API server.
- Image verification talks to the registry.
- Any policy matching on `*` also affects frequent objects such as Leases.

For comparison: a Lease is updated constantly. If a policy hits it, the cluster pays on every renewal.

Global Context

`GlobalContextEntry` (short name `gctxentry`) fetches data regularly in the background. The policy reads from the cache instead of the API server. That stays in place with CEL policies too; only older API versions are deprecated.

> **PICTURE THIS**
>
> *Instead of running to the warehouse for every customer, you keep the most common parts at the register.*

Limiting Reports

In large clusters, reports put a load on etcd and the API server. `features.admissionReports.enabled` can be turned off if background results are enough. The scan interval is controlled by `features.backgroundScan.backgroundScanInterval`.

Measuring

The most important value is admission latency:

```
histogram_quantile(0.99, sum(rate(
  kyverno_admission_review_duration_seconds_bucket[5m]))
  by (le))
```

> **PITFALL**
>
> A policy on `*` resources looks harmless in the lab. In a cluster with thousands of Pods and constant Leases, every change lands at Kyverno. Check latency after every rollout.

> **IN THE INTERVIEW**
>
> **Kyverno is slowing down the API server. What do you do?**
>
> I measure `kyverno_admission_review_duration_seconds` and find the expensive policy via `kyverno_policy_execution_duration_seconds`. Then I narrow the matches, remove wildcards, cache external data, and increase the admission replicas.

KEY TAKEAWAYS

- A narrow match is the biggest lever.
- `resource.Get` and image verification are costly.
- Measure latency, don't guess.

30

Monitoring

Four numbers and an emergency plan.

Kyverno exposes Prometheus metrics. The chart optionally includes a Grafana dashboard (`grafana.enabled`).

Metric	Meaning
`kyverno_admission_review_duration_seconds`	Latency of admission reviews overall.
`kyverno_policy_execution_duration_seconds`	Latency per rule.
`kyverno_admission_requests_total`	Number of requests.
`kyverno_policy_results_total`	Results by `rule_result` and trigger.

Good Alerts

- Fewer than two admission replicas ready.
- p99 latency of admission reviews above your threshold.
- Many `fail` or `error` results in `kyverno_policy_results_total`.
- `UpdateRequest`s stuck in the `Failed` or `Pending` state.
- Webhook errors in the API server, there `apiserver_admission_webhook_rejection_count`.

Checking That Rules Still Apply

Monitor the `failurePolicy` of your webhooks. It can change through configuration, for example via `features.forceFailurePolicyIgnore`. Then your rules are suddenly soft during an outage, and nobody notices.

IN THE INTERVIEW

What do you monitor for Kyverno?

Replica availability, admission latency, results per policy, webhook errors in the API server, and stuck `UpdateRequest`s. Plus the number of reports as a load indicator.

KEY TAKEAWAYS

- Latency, availability, results, errors.
- API server metrics see what Kyverno doesn't.
- Check `failurePolicy` for drift.

31

Upgrades

The jump to 1.20 is not a normal upgrade.

The Normal Upgrade

1. Read the release notes.
2. Practice in a test cluster with real policies.
3. Look at the difference: compare `helm template` old and new.
4. Bump the CLI version, run the tests.
5. Upgrade, watch reports and latency.

```
helm upgrade kyverno kyverno/kyverno -n kyverno --version
3.9.2 -f values-ha.yaml
```

Memory aid: Always a pinned version, always your values.

```
kyverno migrate --resource policyexceptions.kyverno.io
```

Memory aid: Rewrites stored objects to the current storage version. This is part of the upgrade to 1.19.

The Upgrade to 1.20

As of the chart docs for 1.20 (still a release candidate):

- Old policies from `kyverno.io` are **no longer enforced**. This affects `ClusterPolicy`, `Policy`, `CleanupPolicy`, `ClusterCleanupPolicy`, and the old `PolicyException`.
- Their CRDs remain so you can read, export, and delete them.
- Creates and updates are rejected, even a `kubectl apply` with no changes and pure label patches. Reads and deletes stay allowed. As a result, GitOps tools report permanent Out-of-Sync until the old manifest is removed from the repository.
- The chart **blocks** installation and upgrades as long as old policies are in the cluster.

The block has two stages: a check at render time and a hook job for GitOps tools.

```
helm upgrade --install kyverno kyverno/kyverno -n kyverno \
  --set upgrade.acknowledgeLegacyPoliciesNotEnforced=true
```

PITFALL

This value doesn't bring the old policies back. It only confirms that they no longer apply. If you set it to "get the upgrade through", you silently turn off your rules.

REMEMBER

At the time of writing, 1.20 was available only as a release candidate. This description follows the code and chart README of `v1.20.0-rc.1`. The older tracking issue #17214 still mentions enforcement continuing, so it's either outdated or not updated. Check the final release notes before you plan.

IN THE INTERVIEW

What changes with Kyverno 1.20?

The old types such as `ClusterPolicy` are no longer enforced. The chart blocks the upgrade as long as such policies exist. So I migrate to CEL beforehand and stay on 1.19 until then.

KEY TAKEAWAYS

- Practice, diff, tests, then upgrade.
- 1.20 no longer enforces old policies.
- `acknowledgeLegacyPoliciesNotEnforced` is not a solution.

PART VI

Troubleshooting

This is where the interview separates the wheat from the chaff.

32

The Method

Look first, then think, and only then act.

Five Steps

1. **Symptom:** What exactly is happening? What does the error message say?
2. **Scope:** One object, one Namespace, the whole cluster?
3. **Data:** Policy status, events, reports, logs.
4. **Hypothesis:** What could the cause be?
5. **Test:** The smallest safe check.

Who Rejected It?

The message from `kubectl apply` reveals a lot:

```
admission webhook "vpol.validate.kyverno.svc-fail"
denied the request: ...
```

- A webhook name with `kyverno` points to Kyverno. The exact name depends on version and configuration, so don't check for it in a script.
- `failed calling webhook` means Kyverno isn't answering. That's an outage, not a violation.
- `context deadline exceeded` means a timeout.
- A message from `ValidatingAdmissionPolicy` points to a native policy.

The First Minute

```
kubectl -n kyverno get pods
```

Memory aid: Is Kyverno alive?

```
kubectl get vpol,mpol,gpol,ivpol,dpol
```

Memory aid: Which policies exist, and are they ready?

```
kubectl get validatingwebhookconfigurations | grep kyverno
```

Memory aid: Are the webhooks there?

```
kubectl -n kyverno logs deploy/kyverno-admission-controller
--tail=50
```

Memory aid: What does the clerk at the counter say?

```
kubectl get events -A --field-selector reason=PolicyViola-
tion
```

Memory aid: Violations as events.

> **REMEMBER**
>
> Many apparent Kyverno problems aren't Kyverno problems at all. Check the other admission building blocks: PSA, VAPs, and other webhooks.

IN THE INTERVIEW

A request is rejected. Where's it coming from?

I read the message: it names the webhook or the policy. With `failed calling webhook` it's an outage, with a custom message it's a violation. Then I check the policy, the report, and the logs.

KEY TAKEAWAYS

- Read the error message carefully.
- `denied` is a violation. `failed calling` is an outage.
- Status, events, logs.

33

The Cluster Is Locked Out

The scenario that comes up in every senior interview.

Symptom: No more Pods can be created. `kubectl apply` runs into timeouts. The cluster seems broken.

Cause: All Kyverno Pods are gone, for example after an outage or the replacement of a node group. The policies are set to `Fail` and hit Pods. The API server is calling into the void.

```
Error from server (InternalError): failed calling webhook
"validate.kyverno.svc-fail": ... context deadline exceeded
```

The circle is especially nasty: no Pods, no Kyverno; no Kyverno, no Pods.

> **PICTURE THIS**
>
> *The fuse in the fuse box has blown, and the box only opens electrically.*

The Rescue

You delete the webhooks that send normal resources to Kyverno:

```
kubectl delete validatingwebhookconfiguration kyverno-resource-validating-webhook-cfg
```

Memory aid: Cut the line to the inspector.

```
kubectl delete mutatingwebhookconfiguration kyverno-resource-mutating-webhook-cfg
```

Memory aid: The same for mutation.

Only these two. The other webhooks are there for Kyverno's internal objects.

```
kubectl -n kyverno rollout restart deploy kyverno-admission-
controller
```

Memory aid: Usually unnecessary, but useful if Kyverno doesn't start on its own.

```
kubectl -n kyverno get pods -w
```

Memory aid: Wait until the Pods are ready.

As soon as Kyverno is running again, it recreates its webhooks on its own, matching the installed policies.

> **PITFALL**
>
> During the rescue, no policies apply. Afterward, check what came in during that time. The background scan shows the violations in the reports.

Prevention

- Three replicas, PodDisruptionBudget, spread across nodes.
- Keep `kube-system` and `kyverno` excluded.
- Critical rules also as native VAPs.
- `failurePolicy: Ignore` for rules that aren't critical.
- The emergency runbook must be close at hand and must not depend on GitOps.
- Never delete node groups without taking the PDB seriously.
- Practice the scenario regularly in the lab.

> **PITFALL**
>
> GitOps can set you back. The next sync may undo a manual rescue. Pause auto-sync while you operate on the webhook.

IN THE INTERVIEW

Kyverno is down, and nobody can deploy anymore. What do you do?

First I confirm the cause with `failed calling webhook` and look at why Kyverno is gone. Then I delete the two resource webhooks to unlock the cluster. Kyverno comes up and recreates them. After that I use reports to check what happened in the gap, and I fix the root cause: HA, PDB, exclusions.

KEY TAKEAWAYS

- `failed calling webhook`: outage.
- Delete two webhooks, Kyverno recreates them.
- Check what got through unchecked.

34

The Policy Doesn't Take Effect

Everything's running, but the bad Pod gets through.

Check from the outside in.

```
kubectl get vpol require-app-label -o yaml
```

Memory aid: Look in `status`: `conditionStatus.ready` must be `true`.

If the policy is ready, go through this list:

1. **What's `validationActions` set to?** `[Audit]` blocks nothing.
2. **Does it match?** Group, version, plural, operation.
3. **Is something filtering it out first?** `namespaceSelector`, `resourceFilters`, webhook exclusions.
4. **Is admission evaluation on?** `evaluation.admission.enabled`.
5. **Does an exception apply?** Check the `PolicyException`s and `skip` in reports.
6. **Was the Pod created by a controller?** Then check Autogen.

Symptom	Cause
Only works on `CREATE`	`UPDATE` is missing from `operations`.
Doesn't work on Deployments	`autogen.podControllers` is missing.
Doesn't work anywhere in a Namespace	Label missing for `namespaceSelector`.
`error` in the report	CEL runtime error: field missing.
Doesn't work in `kube-system`	Webhook excludes the Namespace.
`skip` in the report	`matchConditions` or an exception.

Reproducing It Locally

```
kubectl get pod test -n demo -o yaml > pod.yaml
```

Memory aid: Grab the real object.

```
kyverno apply policy.yaml --resource pod.yaml
```

Memory aid: Reproduce the case without a cluster.

PITFALL

Kyverno only evaluates what arrives. If the webhook doesn't match, you see no log entry, no report, and no error. Silence is no proof of health.

IN THE INTERVIEW

A policy isn't blocking. How do you troubleshoot?

I check the status, then `validationActions`, then the match, then selectors and exceptions. With `kyverno apply` I reproduce the case using the real object. It's often `Audit`, missing Autogen, or a wrong selector.

KEY TAKEAWAYS

- Status, action, match, filter, exceptions, Autogen.
- `[Audit]` doesn't block.
- Test the real object locally.

35

Generate and Background Are Stuck

The order is there, the result is missing.

This isn't a job for the admission controller, but for the background controller.

```
kubectl get updaterequests -A
```

Memory aid: The order list. Read the `STATUS` column.

```
kubectl -n kyverno logs deploy/kyverno-background-controller
--tail=50
```

Memory aid: What does the back office say?

Clue	Cause
`Failed` with `forbidden`	The background controller lacks RBAC permissions for the target type.
`Pending` persists	Controller is stuck or being throttled.
No `UpdateRequest`	The policy doesn't match the event.
Object missing in old Namespace	`generateExisting` is off.
Change gets reverted	`synchronize` is on.

Adding Permissions

Kyverno works with aggregated ClusterRoles. So you give the controller a role of its own:

`background-rbac.yaml`

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: kyverno:background:networkpolicies
  labels:
    rbac.kyverno.io/aggregate-to-background-controller: "true"
rules:
```

```
  - apiGroups: [networking.k8s.io]
    resources: [networkpolicies]
    verbs: [create, update, patch, delete, get, list, watch]
```

The label is the switch: the controller's role collects all roles with this label. For the cleanup controller it's `rbac.kyverno.io/aggregate-to-cleanup-controller`.

IN THE INTERVIEW

A `GeneratingPolicy` creates nothing. Where do you look?

At the `UpdateRequest`s and in the background controller's logs. Typical causes are missing RBAC permissions or a trigger that doesn't match. For old Namespaces, `generateExisting` is often missing.

KEY TAKEAWAYS

- `kubectl get updaterequests -A`.
- `forbidden` means RBAC.
- The background controller does the work, not the user.

36

Image Verification Fails

It's signed, and it still won't get through.

Message	Cause
Timeout	Registry is slow. Raise `timeoutSeconds`.
`unauthorized`	Pull secret missing for Kyverno.
`x509`	The registry's own CA isn't trusted.
No signature found	Wrong key, wrong identity, or different repo.
Attestation missing	Wrong `type` or no signature on the bundle.

Checking by Hand

```
cosign verify --key cosign.pub registry.example.com/shop/
api:1.0
```

Memory aid: Does Cosign see the signature? If not, the problem isn't Kyverno.

```
crane digest registry.example.com/shop/api:1.0
```

Memory aid: What's the digest? Signature and image must match it.

Keyless Pitfalls

- `subject` must match the workflow path **and** the ref.
- `issuer` must match exactly.
- With your own Sigstore instances, you need your own trust material (`trustedRoot`, `tuf`).

> **REMEMBER**
>
> In the companion material for "Kyverno & NetworkPolicies zum Anfassen" (German title), Kyverno 1.19.1 didn't find cosign v3 signatures on a plain HTTP registry, not even with `allowInsecure`. The lab registry therefore ran with TLS.

IN THE INTERVIEW

Verification of a signed image fails. What do you check?

I use `cosign verify` outside of Kyverno to check whether signature and key match. Then Kyverno's access to the registry: pull secret, CA, timeout. For keyless, I compare subject and issuer exactly.

KEY TAKEAWAYS

- Check with `cosign` itself first.
- Pull secret, CA, timeout.
- Keyless: subject and issuer exactly.

PART VII

The Interview

Comparisons, questions, exercises, and a cheat sheet for the last evening.

37

Kyverno, Gatekeeper, VAP

Three routes to the same goal. When do you take which?

	Kyverno	OPA Gatekeeper	VAP
Language	CEL, formerly JMESPath	Rego, also CEL	CEL
Runs as	Webhook	Webhook	In the API server
Validate	Yes	Yes	Yes
Mutate	Yes	Yes, separate types such as `Assign`	Separate type: `MutatingAdmissionPolicy`, stable since Kubernetes 1.36
Generate	Yes	No	No
Verify images	Built in	Via External Data Provider (beta)	No
Scan existing	Yes, reports	Yes, audit	No
Exceptions	`PolicyException`	Via constraints	Via bindings
Learning curve	Low to medium	High	Low

OPA Gatekeeper

Gatekeeper separates two things:

- `ConstraintTemplate`: the rule in Rego or, stable since Gatekeeper 3.18, in CEL.
- `Constraint`: its application with parameters.

Rego is powerful and also usable outside Kubernetes, for example for Terraform or APIs. That's OPA's strength: one language for everything.

ValidatingAdmissionPolicy

VAP is built in and has been stable since Kubernetes 1.30. It needs no webhook, so it doesn't go down. A `ValidatingAdmissionPolicyBinding` binds the rule to a scope.

> **PICTURE THIS**
>
> *VAP is the lock in the door: always there, can't do much. Kyverno is the security service: can do a lot, but has to be there. OPA is the lawyer: can do anything, but you need to speak the language.*

How to Decide

- Only simple validation and minimal dependencies: **VAP**.
- Kubernetes platform with mutation, generation, and signatures: **Kyverno**.
- A unified policy language across many systems, with Rego know-how in place: **OPA**.
- Often a combination: VAP for the critical stuff, Kyverno for the rest.

Kyverno can roll out a `ValidatingPolicy` as a VAP itself. That combines the advantages.

IN THE INTERVIEW

Kyverno or OPA Gatekeeper?

Kyverno for Kubernetes teams: policies are YAML and CEL, plus mutation, generation, and image verification. OPA when policy spans many systems and Rego is already there. Both are mature. What decides it is team know-how and need.

IN THE INTERVIEW

Why not just `ValidatingAdmissionPolicy`?

VAP validates very robustly, but has no generating, no image signatures, no reports, and no deletion policies. For critical, simple rules it's ideal. Kyverno builds on it and can even generate it.

KEY TAKEAWAYS

- VAP: native. Kyverno: platform. OPA: cross-system.
- Rego is the big hurdle with OPA.
- Combining them is allowed.

38

The Questions

Short questions, short answers. For review before the interview.

Fundamentals

What is Kyverno?

A policy engine for Kubernetes that runs as an admission webhook and manages policies as resources.

What can Kyverno do?

Validate, mutate, generate, verify images, delete, report.

Why not just RBAC?

RBAC checks who may do something, but not what the object looks like.

What does the admission chain look like?

AuthN, AuthZ, Mutating, Schema, Validating, etcd.

Why mutation first?

Validation should see the final result.

What controllers are there?

Admission, Background, Reports, Cleanup.

Which is the most critical?

Admission, because it sits in the API server's path.

What is an `UpdateRequest`?

The order for asynchronous work such as Generate.

Policies

What policy types are there?

`ValidatingPolicy`, `MutatingPolicy`, `GeneratingPolicy`, `ImageValidatingPolicy`, `DeletingPolicy`, each with a `Namespaced` variant.

What happens to `ClusterPolicy`?

Deprecated since 1.19, no longer enforced from 1.20 on.

What is CEL?

The Common Expression Language, a safe expression language that the API server also uses.

How do you access fields safely?

`.?field.orValue(default)`.

What is Autogen?

Pod rules become rules for Pod controllers.

`matchConstraints` **or** `matchConditions`**?**

Coarse filter or fine-grained CEL filter.

`Deny`, `Audit`, `Warn`**?**

Reject, report only, warn only.

`failureAction` **or** `failurePolicy`**?**

Violation of the rule or outage of the webhook.

What's the background scan for?

To check existing resources.

What about `request.userInfo` **in the background?**

There's no user there. Turn background off for that rule.

Default for `failurePolicy`**?**

`Fail`.

Default timeout?

10 seconds; 1 to 30 are allowed.

Tools

`ApplyConfiguration` **or** `JSONPatch`**?**

Setting fields or precise operations.

What does `mutateExisting` **do?**

Changes existing resources via the background controller.

What does `synchronize` **do?**

Keeps generated objects up to date and restores them.

What does `generateExisting` **do?**

Applies Generate to existing triggers too.

How do you verify signatures?

`ImageValidatingPolicy` with Cosign or Notary attestors.

What is keyless?

Signing with an OIDC identity instead of your own key.

Why digests?

Tags are mutable, digests aren't.

What is an attestation?

A signed statement about an image, such as an SBOM or provenance.

PSA or Kyverno?

PSA as the base, Kyverno for fine-grained exceptions and reports.

How do exceptions work?

`PolicyException` with `policyRefs`, `matchConditions`, `expiresAt`. Off by default.

Where are violations recorded?

In `PolicyReport` and `ClusterPolicyReport`.

What does the label `cleanup.kyverno.io/ttl` do?

It gives an object an expiration date.

Operations

How many admission replicas?

At least three.

Why don't more reports replicas help?

Leader election. Only one works.

What do you exclude from the webhook?

`kube-system` and `kyverno`, plus whatever the platform needs.

How do you make rules outage-proof?

Generate them as VAPs.

What do you do in a lockout?

Delete the two resource webhooks, restore Kyverno, check the gap.

How do you test policies?

`kyverno test`, `kyverno apply`, Chainsaw.

What is `--dry-run=server`?

A run through the admission chain without saving.

How do you introduce policies?

Audit, Warn, Deny, via GitOps and stages.

Why do Argo CD and mutation fight?

The object in the cluster differs from Git. `ignoreDifferences` helps.

How do you find expensive policies?

`kyverno_policy_execution_duration_seconds`.

What do you do about latency?

Narrow matches, remove wildcards, cache, scale.

Who creates generated objects?

The background controller, with its RBAC permissions.

What do you watch for when upgrading to 1.20?

Migrate old policies first, otherwise the chart blocks.

When You Don’t Know Something

Say so. Then think out loud: “I haven’t run it that way. I suspect X, because Y. I’d test it in `Audit` first.” With technology in the API server’s path, exactly this caution is a good sign.

KEY TAKEAWAYS

- Short answer first. Details only if someone asks.
- Always explain what happens during an outage.
- Also say what you would **not** do.

39

Exercises

Four scenarios and two live policies from the senior interview.

> **REMEMBER**
>
> Flow in the interview: ask about requirements, name risks, plan the rollout, walk through an outage. Questions first: How many teams? What compliance? Who may grant exceptions? What happens during an outage?

Exercise 1: Multi-Tenant Platform

Situation: 30 teams share one cluster.

- **Generate:** For each Namespace with the `tenant` label: default-deny, ResourceQuota, LimitRange.
- **Validate:** PSS `restricted`, required labels, allowed registries, no host paths.
- **Mutate:** secure defaults in `securityContext`.
- **Exceptions:** central Namespace, review, `expiresAt`.
- **Visibility:** Policy Reporter per team.

Exercise 2: Secure Supply Chain

Situation: Only images from your own CI may run.

- CI signs keyless with Cosign and attaches an SBOM and provenance.
- `ImageValidatingPolicy` verifies identity and attestations.
- Raise the timeout, `Fail` in production, roll out with `Audit` first.
- Keep the registry highly available, because it's now in the path.

Exercise 3: 200 ClusterPolicies Before 1.20

- Inventory, remove duplicates, use PSA where it's enough.
- Translate rules to CEL one by one, reuse tests.
- New policies in parallel in `Audit`, compare reports.
- Switch to `Deny`, remove the old ones, and only then upgrade.

Exercise 4: The Postmortem

Situation: A node upgrade took out all Kyverno Pods. No deployments for three hours.

- **Immediately:** Delete webhooks, restore Kyverno, check the gap.
- **Cause:** No PDBs, replicas on one node.
- **Long term:** three replicas, PDB, spread, VAP for the critical stuff, runbook, practice.

Live 1: No Pods Without Limits

require-limits.yaml

```yaml
apiVersion: policies.kyverno.io/v1
kind: ValidatingPolicy
metadata:
  name: require-limits
spec:
  validationActions: [Audit]
  autogen:
    podControllers:
      controllers: [deployments, statefulsets, jobs, cronjobs]
  matchConstraints:
    resourceRules:
      - apiGroups: [""]
        apiVersions: [v1]
        operations: [CREATE, UPDATE]
        resources: [pods]
  variables:
    - name: missing
      expression: >-
        object.spec.containers
          .filter(c, !('memory' in
            c.?resources.?limits.orValue({})))
          .map(c, c.name)
  validations:
    - expression: "size(variables.missing) == 0"
      messageExpression: >-
        'limits.memory missing for: ' +
variables.missing.join(', ')
```

What's happening here:

- `Audit` first, because the rule immediately hits many workloads.
- `filter` collects the offenders so the message can name them.
- `orValue({})` prevents runtime errors on missing fields.
- Autogen puts the error right on the Deployment.

Live 2: Allowed Registries Only

allowed-registries.yaml

```yaml
apiVersion: policies.kyverno.io/v1
kind: ValidatingPolicy
metadata:
  name: allowed-registries
spec:
  validationActions: [Deny]
  matchConstraints:
    resourceRules:
      - apiGroups: [""]
        apiVersions: [v1]
        operations: [CREATE, UPDATE]
        resources: [pods]
  variables:
    - name: allowed
      expression: "['registry.example.com', 'ghcr.io']"
    - name: foreign
      expression: >-
        (object.spec.containers
          + object.spec.?initContainers.orValue([]))
        .map(c, c.image)
        .filter(i, !(image(i).registry() in variables.allowed))
  validations:
    - expression: "size(variables.foreign) == 0"
      messageExpression: >-
        'Registry not allowed: ' + variables.foreign.join(', ')
```

What's happening here:

- `image(i).registry()` parses the reference. A `startsWith` would be attackable, for example with `ghcr.io.evil.example`.
- `initContainers` count too.

IN THE INTERVIEW

How do you approach a policy live?

I clarify what exactly should apply, and write the test case first: one good and one bad object. Then the policy, starting with `Audit`. I talk out loud about fields that might be missing.

KEY TAKEAWAYS

- Ask first, then think through rollout and outage.
- Always mention `Audit` as the starting point.
- `image()` instead of text comparison.

40

Cheat Sheet

Everything important on a few pages, for the last evening.

One Sentence for Everything

Term	Key phrase
Admission	Mutate first, then validate.
`failurePolicy`	Kyverno goes down.
`validationActions`	A rule is violated.
Audit	Observe first.
Autogen	Error at the Deployment, not the Pod.
`orValue`	No crash on a missing field.
`synchronize`	Generated things stay up to date.
`PolicyException`	With an expiration date.
VAP	Runs even without Kyverno.
1.20	Old policies no longer apply.

Status and Diagnostics

```
kubectl -n kyverno get pods
kubectl api-resources | grep -i kyverno
kubectl get vpol,mpol,gpol,ivpol,dpol
kubectl get cpol,pol -A
kubectl get polr -A
kubectl get cpolr
kubectl get updaterequests -A
kubectl get events -A --field-selector reason=PolicyViolation
kubectl -n kyverno logs deploy/kyverno-admission-controller
kubectl -n kyverno logs deploy/kyverno-background-controller
```

Emergency

```
kubectl delete validatingwebhookconfiguration \
  kyverno-resource-validating-webhook-cfg
kubectl delete mutatingwebhookconfiguration \
  kyverno-resource-mutating-webhook-cfg
```

Testing

```
kyverno apply policy.yaml --resource pod.yaml
kyverno apply policy.yaml --cluster
kyverno test tests/
chainsaw test e2e/
kubectl apply -f pod.yaml --dry-run=server
```

Installation

```
helm repo add kyverno https://kyverno.github.io/kyverno/
helm install kyverno kyverno/kyverno -n kyverno \
  --create-namespace --version 3.9.2 -f values-ha.yaml
```

The Skeleton

```
apiVersion: policies.kyverno.io/v1
kind: ValidatingPolicy
metadata:
  name: NAME
spec:
  validationActions: [Audit]
  matchConstraints:
    resourceRules:
      - apiGroups: [""]
        apiVersions: [v1]
        operations: [CREATE, UPDATE]
        resources: [pods]
  validations:
    - expression: "..."
      message: "..."
```

CEL Patterns

```
object.spec.?a.?b.orValue(false)
object.spec.containers.all(c, ...)
object.spec.containers.filter(c, ...).map(c, c.name)
object.spec.containers
  + object.spec.?initContainers.orValue([])
image(i).registry()
image(i).tag()
image(i).containsDigest()
resource.Get("v1", "configmaps", "ns", "name")
```

Numbers You Need to Know

Value	Meaning
10 s	Default timeout of a webhook.
1–30 s	Allowed range for `timeoutSeconds`.
1 h	Default interval of the background scan.
3	Minimum admission replicas for HA.
1.17	CEL policies stable, `ClusterPolicy` marked for deprecation.
1.19	Last version with full support for old types.
1.20	Old policies are no longer enforced.

Never Without Thinking

- Switching a new policy straight to `Deny`.
- `resources: ["*"]` with `failurePolicy: Fail`.
- Removing the `kyverno` or `kube-system` exclusion.
- Admission controller with one replica in production.
- `DeletingPolicy` without a dry run.
- `upgrade.acknowledgeLegacyPoliciesNotEnforced=true` without migrating.
- `PolicyException` without an expiration date.

KEY TAKEAWAYS

- First `Audit`, then `Deny`.
- Kyverno belongs to the control plane.
- Always think through the outage.

Index

Admission Control 13
Admission Controller 15
Anchor 61
anyPattern 60
apiserver_admission_webhook_rejection_count 87
ApplyConfiguration 38
Argo CD 79
Attestation 44
Audit 32
Autogen 25
Autogen VAP 83
Background Controller 15
Background Scan 32
CEL 22
Chainsaw 76
Cleanup Controller 15
cleanup.kyverno.io/ttl 49
CleanupPolicy 68
ClusterCleanupPolicy 68
ClusterPolicy 58
ClusterPolicyReport 54
context 62
Cosign 44
DeletingPolicy 22
Deny 32
deny 60
expiresAt 52
failed calling webhook 94
failureAction 58
failurePolicy 34
foreach 64
generateExisting 42
GeneratingPolicy 22
generator.Apply 41
GitOps 79
GlobalContextEntry 85
HA 82
Helm 18
ImageValidatingPolicy 22
JMESPath 62
JSONPatch 38
kind 18
Kyverno 10
kyverno apply 74
Kyverno CLI 74
kyverno test 76
Lockout 94
matchConditions 27
matchConstraints 27
Metrics 87
Migration 70
mutateExisting 39
MutatingPolicy 22
Notary 44
OPA Gatekeeper 104
OpenReports 55
orValue 29
pattern 60
Playground 75
Pod Security Admission 50
Pod Security Standards 50
PodDisruptionBudget 82
Policy as Code 10
Policy Reporter 54
PolicyException 52
PolicyReport 54
preconditions 64
Rego 104
reinvocationPolicy 39
Reports Controller 15
resource.Get 29
resourceFilters 85
SBOM 44
schedule 48
Shift Left 80
synchronize 41
UpdateRequest 16
Upgrade 89
ValidatingAdmissionPolicy 14
ValidatingPolicy 22
validationActions 32
variables 29
Webhook 13