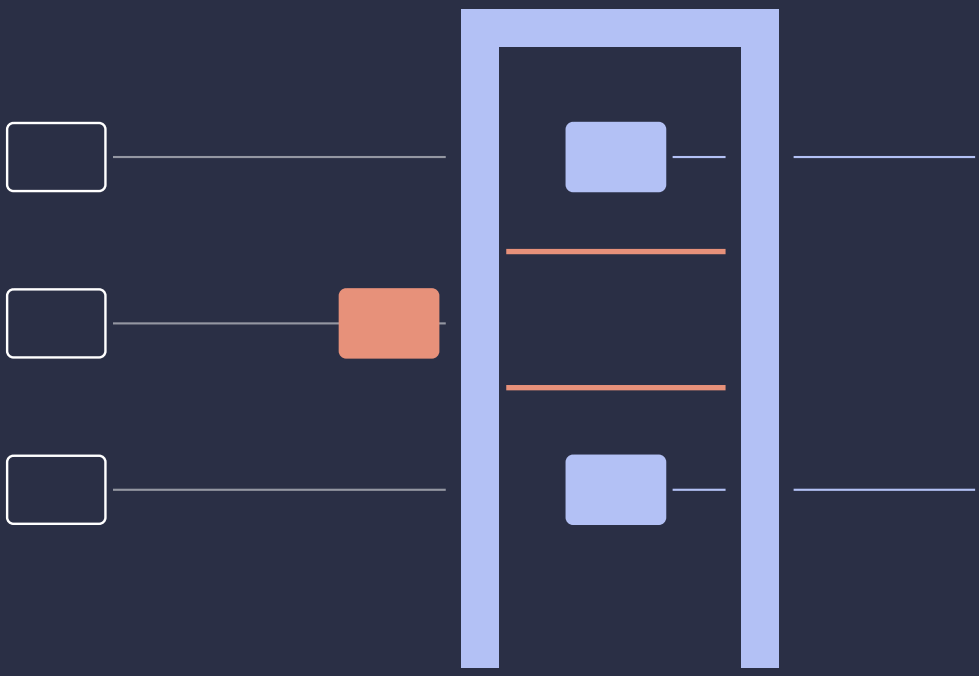
nyrvex.com

Kyverno

Hands-On

Policy as Code für Kubernetes:
prüfen, ändern, erzeugen, reparieren



Mit wenig Text fit fürs Interview

Kyverno

Hands-On

Policy as Code für Kubernetes: prüfen, ändern, erzeugen, reparieren

Mit wenig Text fit fürs Interview

Thomas Zachmann

Kyverno Hands-On. Policy as Code für Kubernetes: prüfen, ändern, erzeugen, reparieren.
Stand: 4. November 2026. © 2026 Thomas Zachmann. Lizenz: CC BY-ND 4.0.

Geschrieben für Kyverno 1.19 (Chart 3.9) mit Blick auf 1.20, das bei Redaktionsschluss als Release Candidate vorlag. Die Befehle, Policies und Beispielausgaben in diesem Buch wurden nicht auf einem Cluster und nicht mit der Kyverno CLI ausgeführt. Ausgaben sind beispielhaft, und Details können sich je nach Version unterscheiden. Probiere alles zuerst im Labor aus, nie auf einem produktiven Cluster.

Schriften: Charter, Fira Sans, JetBrains Mono.

Inhalt

So liest du dieses Buch 6

Der Start **9**
1 Was ist Kyverno? 10
2 Admission Control und Webhooks 13
3 Die Architektur 15
4 Das Labor 18

Policies denken **21**
5 Die Policy-Familie 22
6 Die erste ValidatingPolicy 24
7 Match: Was wird geprüft? 27
8 CEL in einer Stunde 29
9 Audit, Warn, Deny 32
10 Fail oder Ignore 34

Die Werkzeuge **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

Die klassische Welt **57**
18 ClusterPolicy lesen 58
19 Validate: pattern und deny 60
20 Variablen und JMESPath 62
21 Preconditions und foreach 64
22 Mutate, Generate, verifyImages 66
23 Alte Aufräumer und Ausnahmen 68
24 Migration nach CEL 70

Tests und Betrieb **73**
25 Kyverno CLI 74
26 Tests 76
27 CI und GitOps 79
28 Hochverfügbarkeit 82
29 Performance 85
30 Monitoring 87
31 Upgrades 89

Troubleshooting **91**
32 Die Methode 92
33 Der Cluster ist ausgesperrt 94
34 Die Policy greift nicht 97
35 Generate und Background hängen 99
36 Die Image-Prüfung scheitert 101

Das Interview **103**
37 Kyverno, Gatekeeper, VAP 104
38 Die Fragen 106
39 Aufgaben 110
40 Spickzettel 113

Stichwörter 116

So liest du dieses Buch

Du hast bald ein Interview und musst Kyverno können. Nicht irgendwann, sondern bald. Dieses Buch ist genau dafür gebaut: wenig Text, viele kleine Beispiele, und zu jedem Thema die Frage, die dir wirklich gestellt wird.

Die Regeln

- **Tippe alles selbst.** Nichts kopieren. Deine Finger lernen mit.
- **Jeder Befehl einzeln.** Es gibt keine Skripte. Du tippst, du schaust, was passiert.
- **Nur im Labor.** Eine falsche Policy sperrt einen Cluster aus. Ein kind-Cluster reicht, siehe Kapitel 4.
- **Verstehe zuerst das Warum.** Interviews prüfen selten YAML-Felder. Sie prüfen, ob du erklären kannst, was passiert, wenn Kyverno ausfällt.

> **MERKE**
>
> Die Befehle, Policies und Ausgaben in diesem Buch wurden nicht ausgeführt. Die Ausgaben sind gekürzte Beispiele. Deine Texte werden abweichen.

Zwei Welten

Kyverno steckt mitten im Umbau, und das ist selbst ein Interview-Thema:

- **Neu:** CEL-Policies wie `ValidatingPolicy`. Das ist die Zukunft und der Kern dieses Buchs.
- **Alt:** `ClusterPolicy` mit JMESPath. Seit 1.19 abgekündigt, in fast jedem bestehenden Cluster aber noch vorhanden.

Teil II und III zeigen die neue Welt. Teil IV zeigt die alte, damit du sie lesen und migrieren kannst.

Die Kästen

> **DAS BILD IM KOPF**
>
> *Ein Vergleich aus dem Alltag. Wenn du den behältst, kannst du das Thema herleiten.*

```
kubectl get vpol
```

Eselsbrücke: Ein Befehl zum Tippen, mit Merkhilfe darunter.

STOLPERFALLE

Der Fehler, den fast alle machen. Und den Interviewer gern abfragen.

IM INTERVIEW

Eine typische Frage?

Eine Antwort in zwei, drei Sätzen. So kurz darf sie sein.

Dein Lernplan

Du hast …	Dann lies …
1 Abend	Kapitel 1, 2, 3, 5, 6, 9 und 10, dann Kapitel 38 (die Fragen) und 40 (Spickzettel).
2 Tage	Tag 1: Teil I und II. Tag 2: Kapitel 11 bis 13, 16, 18, 33 und 38.
1 Woche	Tag 1: Teil I und II. Dann ein Teil pro Tag. Tag 6: die Aufgaben aus Kapitel 39. Tag 7: Fragen wiederholen.

TEIL I

Der Start

Was Kyverno ist, wo es im API-Server sitzt und wie du ein Labor aufbaust.

1

Was ist Kyverno?

Regeln für deinen Cluster, geschrieben als Kubernetes-Ressourcen.

Kyverno ist eine Policy Engine für Kubernetes. Der Name kommt aus dem Griechischen und bedeutet „regieren“. Der Kern ist eine einzige Idee:

Eine Regel ist eine Ressource wie jede andere.

Du schreibst Policies in YAML, legst sie mit `kubectl apply` an und verwaltest sie mit Git.

Das Problem

RBAC sagt, **wer** etwas anlegen darf. RBAC sagt nicht, **wie** es aussehen muss. Ein Entwickler mit dem Recht, Pods zu erstellen, kann also auch diesen Pod erstellen:

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

Kyverno schließt genau diese Lücke.

> **DAS BILD IM KOPF**
>
> *RBAC ist der Türsteher, der deinen Ausweis prüft. Kyverno ist die Kleiderordnung: Du darfst rein, aber nicht in diesem Aufzug.*

Was Kyverno kann

Fähigkeit	Beispiel
Validieren	Pods mit `:latest` ablehnen.
Mutieren	Fehlende Labels oder `securityContext` ergänzen.
Generieren	Zu jedem Namespace eine NetworkPolicy anlegen.
Images prüfen	Nur signierte Images zulassen.
Löschen	Fertige Pods nach einer Stunde entfernen.
Berichten	Verstöße bestehender Ressourcen als Report anzeigen.

Warum Kyverno so beliebt ist

- **Kubernetes-nativ.** Policies sind CRDs. `kubectl`, Helm und GitOps funktionieren wie immer.
- **Mehr als Prüfen.** Mutieren, Generieren und Aufräumen sind eingebaut.
- **CNCF.** Kyverno ist ein Projekt der Cloud Native Computing Foundation.
- **Eine Sprache.** Die neuen Policies nutzen CEL, dieselbe Sprache wie der API-Server selbst.

> **STOLPERFALLE**
>
> Kyverno prüft nur, was durch den API-Server geht. Es ersetzt weder RBAC noch NetworkPolicies noch eine Laufzeit-Sicherheit. Was in einem Container nach dem Start passiert, sieht es nicht.

IM INTERVIEW

Was ist Kyverno?

Eine Policy Engine für Kubernetes. Sie läuft als Admission Webhook und kann Ressourcen validieren, mutieren und generieren, Image-Signaturen prüfen und Ressourcen aufräumen. Policies sind normale Kubernetes-Ressourcen.

IM INTERVIEW

Warum reicht RBAC nicht?

RBAC entscheidet über Verben und Ressourcenarten, also wer was tun darf. Den Inhalt eines Objekts kann es nicht prüfen. Kyverno prüft genau diesen Inhalt, etwa `privileged: true`.

DAS NIMMST DU MIT

- Policies sind Ressourcen.
- RBAC: wer. Kyverno: wie.
- Validieren, mutieren, generieren, Images prüfen, löschen, berichten.

2

Admission Control und Webhooks

Die letzte Kontrolle, bevor etwas in etcd landet.

Jede Änderung läuft durch den API-Server. Dort gibt es eine feste Kette:

```
kubectl apply
  -> Authentifizierung     Wer bist du?
  -> Autorisierung         Darfst du das? (RBAC)
  -> Mutating Admission    Objekt ändern
  -> Schema-Validierung    Ist das Objekt gültig?
  -> Validating Admission  Objekt prüfen
  -> etcd
```

Merke dir die Reihenfolge: **erst mutieren, dann validieren**. So prüft die Validierung das Endergebnis, nicht den ersten Entwurf.

Webhooks

Der API-Server kann Admission-Entscheidungen auslagern. Über eine `ValidatingWebhookConfiguration` oder `MutatingWebhookConfiguration` sagt man ihm: „Schick mir diese Anfragen." Er sendet dann einen `AdmissionReview` per HTTPS an den Dienst und wartet auf die Antwort.

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

Ein mutierender Webhook antwortet zusätzlich mit einem JSON Patch.

> **DAS BILD IM KOPF**
>
> *Der API-Server ist eine Passkontrolle. Bei bestimmten Reisenden ruft der Beamte eine externe Behörde an und wartet auf die Antwort. Ist die Behörde nicht erreichbar, muss er entscheiden: durchwinken oder alle aufhalten.*

Die wichtigsten Felder

Feld	Bedeutung
`rules`	Welche Ressourcen und Operationen werden geschickt?
`namespaceSelector`	Nur Namespaces mit bestimmten Labels.
`failurePolicy`	`Fail` oder `Ignore`, falls der Webhook nicht antwortet.
`timeoutSeconds`	Wie lange der API-Server wartet.

Der letzte Satz im Bild ist die wichtigste Frage des Buchs. Sie kommt in Kapitel 10.

Eingebaute Alternative

Seit Kubernetes 1.30 gibt es `ValidatingAdmissionPolicy` (VAP) als stabile Funktion. Der API-Server wertet dabei CEL-Ausdrücke selbst aus, ohne Webhook. Mehr dazu in Kapitel 37.

> **STOLPERFALLE**
>
> Webhooks liegen im Pfad **jeder** passenden Anfrage. Ein langsamer Webhook macht den ganzen API-Server langsam. Ein toter Webhook kann ihn blockieren. Webhooks sind also Teil deiner Control Plane, nicht bloß eine Anwendung.

IM INTERVIEW

Was passiert zwischen `kubectl apply` und etcd?

Authentifizierung, Autorisierung, Mutating Admission, Schema-Validierung, Validating Admission, dann Speichern in etcd. Kyverno hängt sich als Webhook in die beiden Admission-Phasen ein.

IM INTERVIEW

Warum läuft Mutation vor Validierung?

Die Validierung soll das endgültige Objekt sehen. Würde sie vorher laufen, könnte eine spätere Mutation ein geprüftes Objekt wieder unzulässig machen.

DAS NIMMST DU MIT

- AuthN, AuthZ, Mutating, Schema, Validating, etcd.
- Der API-Server fragt Kyverno per `AdmissionReview`.
- Ein Webhook ist Teil der Control Plane.

3

Die Architektur

Vier Controller, jeder mit genau einer Aufgabe.

Controller	Aufgabe	Skaliert?
Admission	Beantwortet Webhooks: validieren, mutieren, Images prüfen.	Ja, alle Replikate arbeiten
Background	Generieren und bestehende Ressourcen mutieren.	Nein, ein Leader
Reports	Erstellt Policy Reports.	Nein, ein Leader
Cleanup	Führt Lösch-Policies aus.	Ja

> **DAS BILD IM KOPF**
>
> *Der Admission Controller ist der Schalterbeamte: Er muss sofort antworten. Der Background Controller ist das Backoffice: Er arbeitet Aufträge ab. Der Reports Controller führt die Akten. Der Cleanup Controller ist die Müllabfuhr mit festem Fahrplan.*

Der Weg einer Anfrage

```
API-Server --> Webhook --> Engine --> Antwort
                  ^
                  | Policy Cache
                  |
            Policies (CRDs)
```

- Der **Webhook** nimmt den `AdmissionReview` an.
- Die **Engine** wendet die passenden Policies an.
- Der **Webhook Controller** passt die Webhook-Konfiguration an die installierten Policies an. So bekommt Kyverno nur die Ressourcen zugeschickt, die eine Policy wirklich braucht.
- Der **Cert Renewer** erneuert die Zertifikate für die Webhooks.

UpdateRequests

Generieren dauert und darf eine Anfrage nicht aufhalten. Darum legt der Admission Controller einen Auftrag an, einen `UpdateRequest`. Der Background Controller arbeitet ihn danach ab.

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

Webhooks von Kyverno

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

Eselsbrücke: Zeig alle Telefonleitungen vom API-Server zu Kyverno.

Die wichtigsten beiden heißen `kyverno-resource-validating-webhook-cfg` und `kyverno-resource-mutating-webhook-cfg`. Sie leiten die normalen Ressourcen an Kyverno. Die anderen schützen Kyvernos eigene Objekte, etwa Policies und Ausnahmen.

> **STOLPERFALLE**
>
> Ein Backend-Ausfall trifft die Controller unterschiedlich. Fällt der Background Controller aus, entstehen keine generierten Ressourcen, aber Validierung läuft. Fällt der Admission Controller aus, kann der ganze API-Server betroffen sein. Plane nach dieser Rangfolge.

IM INTERVIEW

Welche Controller hat Kyverno?

Admission für Webhooks, Background für Generieren und Mutate-Existing, Reports für Policy Reports, Cleanup für Löschen. Der Admission Controller ist Pflicht und der kritischste. Er kann horizontal skalieren, Background und Reports arbeiten mit einem Leader.

IM INTERVIEW

Wie weiß der API-Server, was er an Kyverno schicken soll?

Über die Webhook-Konfigurationen. Kyverno pflegt sie automatisch anhand der installierten Policies, sodass nur passende Ressourcen ankommen.

DAS NIMMST DU MIT

- Admission, Background, Reports, Cleanup.
- Generieren läuft asynchron über `UpdateRequest`.
- Kyverno pflegt seine Webhooks selbst.

4

Das Labor

Ein kind-Cluster, ein Helm-Chart, und du kannst loslegen.

Was du brauchst

Docker, `kind`, `kubectl` und Helm. Dazu später die Kyverno CLI (Kapitel 25).

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

Eselsbrücke: **kind** heißt **K**ubernetes **in** **D**ocker: ein Cluster als Container.

```
kubectl get nodes
```

Eselsbrücke: Lebt der Cluster?

Installation

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

Eselsbrücke: Dem Helm-Regal ein neues Fach hinzufügen.

```
helm repo update
```

Eselsbrücke: Den Katalog aktualisieren.

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

Eselsbrücke: Name, Chart, Namespace, feste Version. Chart 3.9.2 entspricht Kyverno 1.19.2.

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

Eselsbrücke: Warte, bis alle vier Controller `Running` sind.

```
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
```

> **MERKE**
>
> Diese Installation hat je ein Replikat und ist nur für das Labor gedacht. Produktiv brauchst du mindestens drei Admission-Replikate, siehe Kapitel 28.

Was installiert wurde

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

Eselsbrücke: Zeigt alle neuen Ressourcenarten und ihre Kurznamen.

Ein Namespace zum Spielen

```
kubectl create namespace demo
```

Eselsbrücke: Hier leben unsere Testobjekte.

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

Eselsbrücke: Mit diesem Label wählen wir Namespaces später gezielt aus.

IM INTERVIEW

Wie installierst du Kyverno?

Mit dem offiziellen Helm-Chart, in einem eigenen Namespace und mit fester Version. Produktiv setze ich drei Admission-Replikate und je zwei für die übrigen Controller.

DAS NIMMST DU MIT

- `kind create cluster`, `helm install`.
- Immer eine feste Chart-Version.
- Vier Pods müssen `Running` sein.

TEIL II

Policies denken

Policy-Typen, CEL, Match, Audit und Fail. Das Fundament jedes Interviews.

5

Die Policy-Familie

Für jede Aufgabe ein eigener Typ.

Früher konnte eine `ClusterPolicy` alles. Heute hat jede Aufgabe ihren eigenen Typ. Alle gehören zur API-Gruppe `policies.kyverno.io` und nutzen **CEL** (Common Expression Language).

Typ	Kurzname	Aufgabe
`ValidatingPolicy`	`vpol`	Ressourcen prüfen.
`MutatingPolicy`	`mpol`	Ressourcen ändern.
`GeneratingPolicy`	`gpol`	Ressourcen erzeugen.
`ImageValidatingPolicy`	`ivpol`	Images und Signaturen prüfen.
`DeletingPolicy`	`dpol`	Ressourcen nach Zeitplan löschen.

Jeder Typ wirkt clusterweit. Zu jedem gibt es eine Variante für einen einzelnen Namespace: `NamespacedValidatingPolicy` (`nvpol`) und so weiter.

> **DAS BILD IM KOPF**
>
> *Früher ein Schweizer Taschenmesser, heute ein Werkzeugkasten. Jedes Werkzeug kann weniger, dafür kann es seine Sache richtig.*

Alt gegen neu

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

Der Zeitplan

Version	Datum	Was geschah
1.15	2025	CEL-Policies erscheinen.
1.17	Feb. 2026	`ValidatingPolicy` und andere werden stabil (`v1`). `ClusterPolicy` und `CleanupPolicy` werden zur Abkündigung vorgemerkt.
1.19	Aug. 2026	Gleichstand der Funktionen. Alte Typen offiziell abgekündigt, letzte Version mit vollem Support.
1.20	Als RC verfügbar	Alte Policies werden nicht mehr durchgesetzt.

Stand dieses Buchs: 1.19.2 ist die aktuelle stabile Version, 1.20.0-rc.1 erschien am 2. November 2026.

> **STOLPERFALLE**
>
> Das Upgrade auf 1.20 ist eine Klippe. Eine alte `ClusterPolicy` bleibt zwar lesbar, wird aber nicht mehr durchgesetzt. Wer blind aktualisiert, verliert still seine Sicherheitsregeln. Details stehen in Kapitel 24 und 31.

IM INTERVIEW

`ClusterPolicy` oder `ValidatingPolicy`?

Für alles Neue `ValidatingPolicy` und ihre Geschwister. `ClusterPolicy` ist seit 1.19 abgekündigt, und ab 1.20 wird sie nicht mehr durchgesetzt. Bestehende Policies migriere ich vorher.

IM INTERVIEW

Warum hat Kyverno auf CEL umgestellt?

CEL ist der Kubernetes-Standard, den auch `ValidatingAdmissionPolicy` nutzt. Dadurch teilt Kyverno Sprache und Struktur mit dem API-Server, und Policies lassen sich sogar als native VAP erzeugen.

DAS NIMMST DU MIT

- `vpol`, `mpol`, `gpol`, `ivpol`, `dpol`.
- Jede Aufgabe ein Typ, optional `Namespaced`.
- `ClusterPolicy` ist ein Auslaufmodell.

6

Die erste ValidatingPolicy

Pods ohne Label `app` kommen nicht mehr durch.

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: "Das Label app fehlt."
```

Drei Blöcke tragen alles:

- `validationActions`: Was passiert bei einem Verstoß? `Deny` lehnt ab.
- `matchConstraints`: Für welche Ressourcen gilt die Policy?
- `validations`: Welche Bedingung muss wahr sein?

> **MERKE**
>
> Der Ausdruck beschreibt, was **gelten muss**. Ist er `true`, ist alles gut. Bei der alten `deny`-Regel war es umgekehrt. Das verwechseln fast alle beim Migrieren.

Ausprobieren

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

Eselsbrücke: Eine Policy ist eine Ressource. Also ganz normal anlegen.

```
kubectl get vpol
```

Eselsbrücke: Kurzname. Warte, bis die Policy bereit ist.

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

Eselsbrücke: `kubectl run` setzt von selbst das Label `run=test`, aber kein `app`.

```
Error from server: admission webhook
"vpol.validate.kyverno.svc-fail" denied the request:
Das Label app fehlt.
```

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

Eselsbrücke: Mit Label kommt der Pod durch.

Der Fall mit dem Deployment

Die Policy matcht `pods`. Was passiert bei einem Deployment ohne Label im Template? Das Deployment selbst wird angenommen. Erst der ReplicaSet-Controller scheitert beim Erstellen der Pods. Der Anwender sieht nur ein Deployment, das nie bereit wird.

Autogen behebt das. Kyverno erzeugt daraus Regeln für die Pod-Controller:

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

So schlägt schon das `kubectl apply` des Deployments fehl, mit einer klaren Meldung.

> **STOLPERFALLE**
>
> Eine Policy auf Pods ohne Autogen meldet den Fehler an der falschen Stelle. Der Entwickler sieht ein grünes `apply`, und die Pods bleiben aus. Die Ursache steckt in den Events des ReplicaSets.

IM INTERVIEW

Was ist Autogen?

Kyverno leitet aus einer Pod-Regel automatisch Regeln für Deployments, StatefulSets, Jobs und CronJobs ab. So schlägt ein Verstoß direkt beim Anlegen des Controllers fehl, nicht erst später beim Pod.

DAS NIMMST DU MIT

- `validationActions`, `matchConstraints`, `validations`.
- Der Ausdruck beschreibt, was gelten muss.
- Autogen bringt Fehler an die richtige Stelle.

7

Match: Was wird geprüft?

Der teuerste Fehler ist eine Policy, die zu viel trifft.

Der Filter hat zwei Stufen. Beide kommen aus der Welt von `ValidatingAdmissionPolicy`.

- `matchConstraints`: grob und schnell. Ressourcenart, Operation, Namespace.
- `matchConditions`: fein, mit CEL.

match-beispiel.yaml (Auszug)

```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: nur-produktion
      expression: >-
        object.metadata.?labels.?stage.orValue('') == 'prod'
```

Feld	Wirkung
`resourceRules`	Gruppe, Version, Operation, Ressource.
`excludeResourceRules`	Das Gegenstück: Diese Regeln fallen heraus.
`namespaceSelector`	Nach Labels des Namespace.
`objectSelector`	Nach Labels des Objekts selbst.
`matchConditions`	Beliebige CEL-Bedingung. Alle müssen wahr sein.

DAS BILD IM KOPF

Erst das grobe Sieb, dann das feine. Was das grobe Sieb aussortiert, kostet keine Rechenzeit.

`apiGroups` nicht vergessen

- Pods und Namespaces liegen in der Kerngruppe `""`.
- Deployments liegen in `apps`, NetworkPolicies in `networking.k8s.io`.
- Der Ressourcenname ist Plural und klein: `pods`, nicht `Pod`.

STOLPERFALLE

Bei `operations: [CREATE]` bleibt `UPDATE` ungeprüft. Wer erst einen harmlosen Pod anlegt und dann ein Feld ändert, umgeht die Regel. Prüfe `CREATE` und `UPDATE` zusammen, außer du hast einen guten Grund.

STOLPERFALLE

`namespaceSelector` schaut auf **Labels**. Der Name ist nur über das automatische Label `kubernetes.io/metadata.name` erreichbar. Wer Labels an Namespaces setzen darf, kann sich so aus einer Policy herauslabeln. Schütze diese Labels mit RBAC oder einer eigenen Policy.

IM INTERVIEW

Wie schließt du `kube-system` aus?

Im Webhook global über `config.webhooks` im Helm-Chart. Das ist der Standard. Für einzelne Policies per `namespaceSelector`. Der globale Ausschluss spart Aufrufe, aber dort greift dann gar keine Policy.

IM INTERVIEW

`matchConstraints` oder `matchConditions`?

`matchConstraints` für den groben Filter, denn sie steuern, was Kyverno überhaupt erreicht. `matchConditions` für alles, was nur CEL ausdrücken kann.

DAS NIMMST DU MIT

- Erst `matchConstraints`, dann `matchConditions`.
- `CREATE` und `UPDATE` zusammen prüfen.
- Labels am Namespace sind Sicherheitsgrenzen.

8

CEL in einer Stunde

Fünf Muster reichen für fast jede Policy.

CEL ist eine kleine Ausdruckssprache. Es gibt keine Schleifen und keine Zuweisungen, dafür Listenfunktionen. Ein Ausdruck ergibt einen Wert.

Was du zur Verfügung hast

Variable	Inhalt
`object`	Das neue Objekt.
`oldObject`	Das alte Objekt bei `UPDATE`.
`request`	Die Anfrage: Nutzer, Namespace, Operation.
`variables`	Deine eigenen Variablen.
`namespaceObject`	Der Namespace der Ressource.

Muster 1: Sicherer Zugriff

Fehlt ein Feld, gibt es einen Laufzeitfehler. Dagegen hilft `?`:

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

Lies es so: „Wenn es den Weg gibt, nimm den Wert, sonst `false`."

Muster 2: Listen prüfen

```
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)
```

Muster 3: Alle Container

Vergiss die Init-Container nicht:

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

Muster 4: Variablen

Variablen vermeiden Wiederholung und machen Meldungen besser:

image-hygiene.yaml (Auszug)

```yaml
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: >-
        'Kein :latest: ' + variables.latest.join(', ')
```

Muster 5: Bibliotheken von Kyverno

Kyverno erweitert CEL um eigene Funktionen:

Funktion	Wozu
`image(i)`	Zerlegt eine Image-Referenz: `registry()`, `tag()`, `containsDigest()`.
`resource.Get(...)`	Liest eine Ressource vom API-Server.
`generator.Apply(...)`	Erzeugt Ressourcen in einer `GeneratingPolicy`.

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

STOLPERFALLE

`has(object.spec.foo.bar)` schützt nur das letzte Feld. Fehlt schon `foo`, knallt es trotzdem. Nimm `object.spec.?foo.?bar` und vergib am Ende mit `orValue` einen Standard.

STOLPERFALLE

Ein Laufzeitfehler in CEL ist ein Fehler der Policy, kein Verstoß. Er löst `failurePolicy` aus. Bei `Fail` wird die Anfrage abgelehnt, auch wenn das Objekt völlig in Ordnung wäre.

IM INTERVIEW

Wie prüfst du alle Container eines Pods?

Ich verbinde `containers` und `initContainers` mit `orValue([])` und nutze `all` oder `filter`. Ephemeral Container gehören je nach Regel auch dazu. Die Prüfung soll keinen Container auslassen.

IM INTERVIEW

Warum ist CEL für Admission geeignet?

CEL ist nicht Turing-vollständig, es gibt keine Schleifen ohne Ende. Ausdrücke sind schnell, haben begrenzte Kosten und lassen sich vorab typprüfen. Das passt zu einem Pfad, der keine Zeit hat.

DAS NIMMST DU MIT

- `.?feld.orValue(...)` statt Abstürzen.
- `all`, `exists`, `filter`, `map`.
- Immer auch `initContainers`.
- Laufzeitfehler lösen `failurePolicy` aus.

9

Audit, Warn, Deny

Erst beobachten, dann blockieren.

Aktion	Wirkung bei Verstoß
`Deny`	Die Anfrage wird abgelehnt.
`Audit`	Die Anfrage geht durch. Der Verstoß landet im Report.
`Warn`	Die Anfrage geht durch. Der Client sieht eine Warnung.

`Audit` und `Warn` lassen sich kombinieren: `validationActions: [Audit, Warn]`.

> **DAS BILD IM KOPF**
>
> *`Deny` ist die Schranke, `Warn` das Schild am Straßenrand, `Audit` der Blitzer: Du kommst durch, aber es wird notiert.*

Der sichere Rollout

1. Policy mit `[Audit]` ausrollen.
2. Reports lesen: Wer verstößt?
3. Teams informieren, Ressourcen reparieren, Ausnahmen klären.
4. Auf `[Audit, Warn]` wechseln, damit es jeder im `kubectl` sieht.
5. Erst bei null Verstößen auf `[Deny]` schalten.

Admission und Background

Jede Policy kann zu zwei Zeitpunkten laufen:

- **Admission:** bei einer Anfrage.
- **Background:** als regelmäßiger Scan bestehender Ressourcen. Standardintervall ist eine Stunde.

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

Beide sind standardmäßig an. Ein neuer Verstoß von heute fällt bei Admission auf, ein alter Pod von letztem Jahr beim Background Scan.

STOLPERFALLE

`Deny` blockiert auch `UPDATE`s bestehender, unzulässiger Objekte. Ein Team kann dann ein altes Deployment nicht einmal mehr skalieren oder patchen, weil das Objekt die neue Regel nicht erfüllt. Darum zuerst `Audit`.

STOLPERFALLE

Der Background Scan kennt keinen Benutzer. Es gibt dort keinen echten Request. Regeln, die `request.userInfo` nutzen, müssen `evaluation.background.enabled: false` setzen.

IM INTERVIEW

Wie führst du eine neue Policy ein, ohne Teams zu stören?

Erst `Audit`, dann Reports lesen, Teams informieren und reparieren. Danach `Warn`, damit alle es sehen, und erst bei null Verstößen `Deny`. Das läuft über Git, mit Tests, in Stufen je Umgebung.

IM INTERVIEW

Wie findest du Verstöße in bestehenden Ressourcen?

Mit dem Background Scan. Er bewertet vorhandene Ressourcen regelmäßig und schreibt das Ergebnis in Policy Reports.

DAS NIMMST DU MIT

- `Deny`, `Audit`, `Warn`.
- Rollout: Audit, Warn, Deny.
- Admission für Neues, Background für Bestand.

10

Fail oder Ignore

Was passiert, wenn Kyverno nicht antwortet?

Die wichtigste Frage zu jedem Webhook.

`failurePolicy`	**Kyverno fällt aus**	**Risiko**
`Fail`	Anfragen werden abgelehnt.	Verfügbarkeit: Deployments stehen still.
`Ignore`	Anfragen gehen ungeprüft durch.	Sicherheit: Regeln gelten nicht.

Der Standard der neuen Policies ist `Fail`.

> **DAS BILD IM KOPF**
>
> *Ein Stromausfall an der Sicherheitsschleuse. `Fail` heißt: Tür bleibt zu, niemand kommt rein. `Ignore` heißt: Tür bleibt offen, jeder kommt rein. Die Bank wählt das eine, das Krankenhaus das andere.*

Pro Policy entscheiden

Die Wahl hängt am Zweck der Regel:

soft-labels.yaml (Auszug)

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

- **Sicherheitsregel** wie „keine privilegierten Pods“: `Fail`.
- **Komfort- oder Hygiene-Regel** wie ein Team-Label: `Ignore`.

Der Timeout

`webhookConfiguration.timeoutSeconds` liegt zwischen 1 und 30 Sekunden, Standard sind 10. Nach Ablauf greift die `failurePolicy`. Lange Prüfungen wie Image-Signaturen brauchen mehr Zeit, Hygiene-Regeln weniger.

STOLPERFALLE

`Fail` und ein einziges Replikat sind eine Falle. Ein Node-Neustart reicht, und niemand kann mehr Pods anlegen, auch nicht den neuen Kyverno-Pod, wenn sein Namespace nicht ausgeschlossen wäre. Darum ist der Kyverno-Namespace standardmäßig ausgeschlossen.

MERKE

Es gibt eine dritte Antwort: Kritische Regeln zusätzlich als `ValidatingAdmissionPolicy` erzeugen. Sie läuft im API-Server und braucht keinen Webhook. Das steht in Kapitel 28.

IM INTERVIEW

`failurePolicy Fail` oder `Ignore`?

Das hängt von der Regel ab. Sicherheitsregeln `Fail`, damit bei einem Ausfall niemand durchrutscht. Hygiene-Regeln `Ignore`, damit sie die Plattform nicht lahmlegen. `Fail` verlangt HA und getestete Ausschlüsse.

IM INTERVIEW

Was passiert, wenn ein CEL-Ausdruck einen Laufzeitfehler wirft?

Das gilt als Fehler der Policy, und die `failurePolicy` entscheidet. Bei `Fail` wird die Anfrage abgelehnt. Bei `Ignore` wird die Regel übersprungen. Darum schreibe ich Ausdrücke robust mit `orValue`.

DAS NIMMST DU MIT

- `Fail`: sicher, aber Risiko für die Verfügbarkeit.
- `Ignore`: verfügbar, aber Risiko für die Sicherheit.
- Pro Regel entscheiden, Standard ist `Fail`.

TEIL III

Die Werkzeuge

Mutieren, Generieren, Images, Löschen, Pod Security, Ausnahmen und Reports.

11

MutatingPolicy

Fehlendes ergänzen, statt abzulehnen.

Mutation macht das Leben der Teams leichter: Statt „Feld fehlt“ zu melden, setzt Kyverno einen sinnvollen Standard.

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: {"sicherheit/defaults": "gesetzt"}
            },
            spec: Object.spec{
              securityContext: Object.spec.securityContext{
                runAsNonRoot: object.spec.?securityContext
                  .?runAsNonRoot.orValue(true)
              }
            }
          }
```

Der Wert `orValue(true)` übernimmt, was der Nutzer gesetzt hat, und setzt sonst einen Standard.

Zwei Patch-Arten

patchType	Wann
ApplyConfiguration	Felder setzen. Liest sich wie das Zielobjekt.
JSONPatch	Präzise Operationen, etwa ein Listenelement ersetzen.

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

> **DAS BILD IM KOPF**
>
> *`ApplyConfiguration` ist ein Formular, in das du nur die gewünschten Felder einträgst. `JSONPatch` ist eine Arbeitsanweisung: Geh zu Zeile 3 und ersetze das Wort.*

Bestehendes mutieren

Admission sieht nur neue Anfragen. Mit `mutateExisting` ändert Kyverno auch vorhandene Ressourcen, und zwar über den Background Controller.

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

`matchConstraints` beschreibt den Auslöser, `targetMatchConstraints` das Ziel.

Webhook-Reihenfolge

Mehrere Mutationen können einander beeinflussen. `reinvocationPolicy` regelt, ob eine Policy nach späteren Änderungen noch einmal läuft.

STOLPERFALLE

Mutation ist unsichtbar. Das Objekt im Git und das Objekt im Cluster unterscheiden sich dann. GitOps-Werkzeuge melden das als Drift, und im schlimmsten Fall kämpfen Argo CD und Kyverno gegeneinander. Die Kyverno-Doku empfiehlt dafür Server-Side Diff. Alternativ konfigurierst du für solche Felder `ignoreDifferences`, oder du setzt die Felder direkt im Git.

STOLPERFALLE

Mutiere nie etwas, das der Nutzer bewusst gesetzt hat. Ein Überschreiben von `image` oder `resources` ist eine Falle für die Fehlersuche. Standardwerte ja, heimliche Änderungen nein.

IM INTERVIEW

Wann mutierst du, wann validierst du?

Mutieren, wenn es einen klaren Standard gibt, etwa `runAsNonRoot` oder ein Label. Validieren, wenn das Team selbst entscheiden muss, etwa bei Ressourcenlimits. Oft kombiniere ich beides: Mutation setzt den Standard, Validierung sichert ihn ab.

DAS NIMMST DU MIT

- `ApplyConfiguration` für Felder, `JSONPatch` für Präzision.
- `mutateExisting` plus `targetMatchConstraints`.
- Vorsicht bei GitOps-Drift.

12

GeneratingPolicy

Ein Ereignis löst neue Ressourcen aus.

Das klassische Beispiel: Jeder neue Namespace bekommt eine 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, liste)` legt die Objekte im Ziel-Namespace an. `dyn(...)` sagt CEL, dass die Map-Werte unterschiedliche Typen haben dürfen.

Klonen

Statt Daten im YAML zu tippen, kopierst du eine Vorlage:

`clone-monitoring.yaml (Auszug)`

```
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])
```

> **DAS BILD IM KOPF**
>
> *Eine Filialkette. Wird eine neue Filiale eröffnet, kommt automatisch die Grundausstattung: Feuerlöscher, Hausordnung, Kasse. Ändert die Zentrale die Hausordnung, wird sie überall getauscht.*

Die Schalter

`evaluation.…`	**Wirkung**
`synchronize`	Löscht oder ändert jemand das Ziel, stellt Kyverno es wieder her.
`generateExisting`	Gilt auch für bereits vorhandene Auslöser.
`orphanDownstreamOnPolicyDelete`	Das Ziel bleibt beim Löschen der Policy bestehen.

RBAC

Der Background Controller legt die Objekte an. Für alles außerhalb der Standardrechte braucht er zusätzliche Rechte. Sonst bleibt der `UpdateRequest` in einem Fehlerzustand hängen.

> **STOLPERFALLE**
>
> Ohne `generateExisting` gilt die Policy nur für künftige Namespaces. Bestehende Namespaces bleiben ungeschützt, und niemand bemerkt es. Prüfe nach dem Ausrollen, ob die Objekte wirklich überall existieren.

STOLPERFALLE

Mit `synchronize` gehört das Ziel faktisch Kyverno. Änderungen von Hand werden zurückgedreht. Ein Team, das seine NetworkPolicy „nur kurz“ anpasst, wundert sich. Dokumentiere das, oder wähle für solche Objekte bewusst `synchronize: false`.

IM INTERVIEW

Wofür nutzt du Generate?

Für Mandanten-Grundausstattung: Default-Deny-NetworkPolicy, ResourceQuota, LimitRange, Rollen. So bekommt jeder neue Namespace automatisch die Regeln, und `synchronize` hält sie aktuell.

IM INTERVIEW

Warum läuft Generate asynchron?

Generate könnte lange dauern und mehrere Objekte betreffen. Das darf eine Admission-Anfrage nicht aufhalten. Darum legt der Admission Controller einen `UpdateRequest` an, und der Background Controller arbeitet ihn ab.

DAS NIMMST DU MIT

- `generator.Apply` erzeugt, `resource.Get` liest die Vorlage.
- `synchronize` hält Ziele aktuell.
- `generateExisting` für den Bestand.
- Background Controller braucht passendes RBAC.

13

ImageValidatingPolicy

Nur signierte Images kommen in den Cluster.

Lieferkettensicherheit heißt: Ich prüfe, **wer** ein Image gebaut hat, nicht nur, **wie es heißt**. Ein Tag ist nur ein Name. Eine Signatur beweist die Herkunft.

Die Bausteine

- **Signatur:** Der Hersteller unterschreibt den Digest des Images.
- **Cosign:** Sigstore-Werkzeug zum Signieren, mit Schlüssel oder keyless über OIDC.
- **Notary:** Alternative mit X.509-Zertifikaten.
- **Attestation:** eine signierte Aussage über das Image, etwa SBOM oder SLSA-Provenance.

Eine Policy mit 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 ist nicht vom Team signiert."
      expression: >-
        images.containers.map(i,
          verifyImageSignatures(i, [attestors.team]))
        .all(n, n > 0)
```

`verifyImageSignatures` liefert die Zahl gültiger Signaturen. `matchImageReferences` begrenzt die Prüfung auf passende Images.

Keyless

Ohne eigenen Schlüssel prüfst du, wer signiert hat, etwa ein bestimmter GitHub-Workflow:

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

Attestations prüfen

```
attestations:
  - name: sbom
    intoto:
      type: https://spdx.dev/Document
validations:
  - message: "Keine signierte SBOM."
    expression: >-
      images.containers.map(i, verifyAttestationSignatures(i,
        attestations.sbom, [attestors.team])).all(n, n > 0)
```

Mit `extractPayload(i, attestations.sbom)` liest du den Inhalt der Attestation und prüfst ihn in CEL.

> **DAS BILD IM KOPF**
>
> *Ein Tag ist ein Etikett, das jeder kleben kann. Eine Signatur ist das Siegel der Firma, eine Attestation der beglaubigte Herkunftsnachweis.*

Tag oder Digest?

Tags sind veränderlich. Prüfst du `:1.2` und läuft danach ein anderes Bild mit demselben Tag, hilft die Prüfung nichts. Erzwinge darum Digests oder lass Kyverno die Referenz auflösen.

STOLPERFALLE

Bei `Fail` hängen Deployments an der Registry. Fällt die Registry aus oder ist sie langsam, scheitert die Prüfung. Erhöhe `timeoutSeconds` und halte Pull-Rechte aktuell.

STOLPERFALLE

Prüfe alle Container-Arten. `images.containers` allein lässt `initContainers` und `ephemeralContainers` aus, denn beide sind eigene Schlüssel in `images`. Sammle sie in einer Variable. Für Ephemeral Container muss die Policy zusätzlich die Subressource `pods/ephemeralcontainers` matchen. Ein Angreifer muss seinen Code nicht im Hauptcontainer starten.

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

IM INTERVIEW

Wie erzwingst du signierte Images?

Mit einer `ImageValidatingPolicy`. Sie matcht die Images per Glob und prüft mit einem Attestor, also Cosign-Schlüssel, Keyless-Identität oder Notary-Zertifikat, die Signatur. Attestations wie SBOM kann ich zusätzlich prüfen und auswerten.

IM INTERVIEW

Warum reicht eine Liste erlaubter Registries nicht?

Die Liste sagt nur, woher ein Image kommt, nicht, wer es gebaut hat. Wer in die Registry pushen kann, kommt durch. Eine Signatur belegt die Herkunft aus deiner CI.

DAS NIMMST DU MIT

- Signatur statt Name.
- `attestors`, `attestations`, `validations`.
- `verifyImageSignatures` und `verifyAttestationSignatures`.
- Hoher Timeout, und Digests statt Tags.

14

DeletingPolicy

Aufräumen nach Fahrplan.

Eine `DeletingPolicy` löscht Ressourcen nach einem Cron-Zeitplan.

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: fertig
      expression: >-
        object.status.?phase.orValue('')
          in ['Succeeded', 'Failed']
    - name: aelter-als-1h
      expression: >-
        time.now()
          - timestamp(object.metadata.creationTimestamp)
          > duration('1h')
```

Auch `deletionPropagationPolicy` lässt sich wählen: `Foreground`, `Background` oder `Orphan`.

Die TTL-Abkürzung

Für einzelne Objekte reicht ein Label:

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

Eselsbrücke: Ein Verfallsdatum auf der Packung. Nach zwei Stunden räumt Kyverno auf.

RBAC

Der Cleanup Controller braucht das Recht, die Ressourcen zu löschen. Das gibst du ihm über eine zusätzliche ClusterRole, sonst passiert nichts.

> **STOLPERFALLE**
>
> Eine falsche Bedingung löscht massenhaft. Teste die Bedingung erst als `ValidatingPolicy` im `Audit` oder mit `kubectl get` und `jq`. Eine `DeletingPolicy` kennt keinen Papierkorb.

IM INTERVIEW

Wofür nutzt du Lösch-Policies?

Für fertige Pods und Jobs, abgelaufene Ausnahmen oder Testumgebungen mit Ablaufdatum. Sie ersparen selbst geschriebene CronJobs und bringen klare Bedingungen in CEL mit.

DAS NIMMST DU MIT

- `schedule`, `matchConstraints`, `conditions`.
- Label `cleanup.kyverno.io/ttl` für einzelne Objekte.
- Bedingung vorher im Trockenlauf prüfen.

15

Pod Security Standards

Kubernetes hat die Regeln schon geschrieben.

Die **Pod Security Standards** (PSS) definieren drei Stufen:

Stufe	Inhalt
`privileged`	Keine Einschränkungen. Für Systemdienste.
`baseline`	Verhindert bekannte Privilegien-Ausweitung.
`restricted`	Strenge Härtung nach bewährten Regeln.

Eingebaut ist dafür **Pod Security Admission** (PSA), gesteuert über Labels am Namespace:

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

Eselsbrücke: Das Label ist die Regel. Stufe `baseline` für diesen Namespace.

PSA oder Kyverno?

	PSA	Kyverno
Ausnahmen	Grob: Namespace, Nutzer, RuntimeClass.	Fein: pro Workload.
Meldung	Fest.	Eigene Texte.
Abhängigkeit	Keine, im API-Server.	Webhook.
Regelumfang	Nur die drei Stufen.	Beliebig.

Oft kombiniert man beides: PSA als Sockel, Kyverno für die Feinheiten.

PSS mit CEL

Die neuen Policies kennen die alte Kurzform `podSecurity` nicht. Du schreibst jede Prüfung als Ausdruck:

pss-restricted.yaml (Auszug)

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

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

Fertige Regeln findest du in der Policy-Bibliothek unter `kyverno.io/policies`. Du musst sie nicht neu erfinden.

IM INTERVIEW

PSA oder Kyverno für Pod Security?

PSA ist eingebaut, ohne Webhook und robust, aber grob. Kyverno bietet feine Ausnahmen, eigene Meldungen und Reports. Ich nutze oft PSA als Grundschutz und Kyverno für die Details.

DAS NIMMST DU MIT

- `privileged`, `baseline`, `restricted`.
- PSA steuert per Namespace-Label.
- CEL-Policies prüfen jede Regel einzeln.

16

PolicyException

Ausnahmen sind normal. Unkontrollierte Ausnahmen nicht.

Früher schrieb man Ausnahmen in die Policy. Das macht die Policy unlesbar, und jedes Team braucht Schreibrechte darauf. Eine `PolicyException` ist darum eine eigene Ressource.

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: nur-legacy-batch
      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`: Für welche Policy gilt die Ausnahme?
- `matchConditions`: Für welche Objekte?
- `expiresAt`: Wann endet sie?

> **MERKE**
>
> Ausnahmen sind standardmäßig aus. Du schaltest sie im Helm-Chart ein: `features.policyExceptions.enabled=true` und `features.policyExceptions.namespace=policy-exceptions`.

DAS BILD IM KOPF

Ein Attest vom Arzt: mit Namen, Grund und Ablaufdatum. Es wird nicht in das Gesetz geschrieben, sondern liegt in einer eigenen Akte, und diese Akte hütet jemand.

Ausnahmen regieren

- Eigener Namespace, und nur das Sicherheitsteam darf dort schreiben.
- Ausnahmen laufen über Pull Requests mit Review.
- Jede Ausnahme hat `expiresAt`, Ticket und Besitzer. Das erzwingst du mit einer eigenen Policy.
- Eine `DeletingPolicy` räumt abgelaufene Ausnahmen weg.

STOLPERFALLE

Wer `PolicyException`s frei anlegen darf, umgeht jede Policy. Der Namespace für Ausnahmen ist eine Sicherheitsgrenze. Begrenze die Schreibrechte per RBAC, und setze nie den Wert `"*"` für alle Namespaces, außer du weißt genau, warum.

IM INTERVIEW

Wie gehst du mit Ausnahmen um?

Als eigene `PolicyException`-Ressource mit Ablaufdatum, Ticket und Besitzer, in einem geschützten Namespace und über Review. So bleibt die Policy sauber, und die Ausnahmen sind prüfbar.

DAS NIMMST DU MIT

- `PolicyException` ist standardmäßig aus.
- `policyRefs`, `matchConditions`, `expiresAt`.
- Der Ausnahme-Namespace ist eine Sicherheitsgrenze.

17

Policy Reports

Was verstößt gerade wogegen?

Policy Reports sind Ressourcen. Sie folgen einem Standard der Kubernetes Policy Working Group (`wgpolicyk8s.io`). Das bleibt auch in Kyverno 1.20 so.

```
kubectl get polr -A
```

Eselsbrücke: **polr** wie **pol**icy **r**eport. Ein Report pro Ressource, mit Zählern.

```
kubectl get cpolr
```

Eselsbrücke: Das Gleiche für clusterweite Ressourcen.

```
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
```

Eselsbrücke: Zeigt die einzelnen Ergebnisse. Name aus der Liste einsetzen.

```yaml
results:
  - policy: image-hygiene
    result: fail
    message: "Kein :latest: nginx:latest"
    source: KyvernoValidatingPolicy
    properties:
      process: background scan
```

Ergebnisse

Ergebnis	Bedeutung
`pass`	Regel erfüllt.
`fail`	Verstoß.
`warn`	Warnung.
`error`	Fehler bei der Auswertung.
`skip`	Übersprungen, etwa durch eine Ausnahme.

Darstellen

Für viele Reports ist `kubectl` zu mühsam. **Policy Reporter** zeigt sie in einer Oberfläche und sendet Meldungen an Slack, Teams oder Prometheus.

> **MERKE**
>
> Reports brauchen Speicher in etcd. Bei vielen Ressourcen kann das die Kapazität belasten. Beobachte ihre Zahl. Optional lassen sich Reports über OpenReports anders ablegen (`openreports.enabled`, standardmäßig aus).

IM INTERVIEW

Wie siehst du, wer gegen Policies verstößt?

Über Policy Reports, die Kyverno bei Admission und im Background Scan schreibt. `kubectl get polr -A` gibt den Überblick. Für den Alltag nutze ich Policy Reporter und Prometheus.

DAS NIMMST DU MIT

- `polr` im Namespace, `cpolr` clusterweit.
- `pass`, `fail`, `warn`, `error`, `skip`.
- Reports belegen etcd. Beobachte sie.

TEIL IV

Die klassische Welt

ClusterPolicy, JMESPath und die Migration. Du wirst sie noch lange antreffen.

18

ClusterPolicy lesen

Die alte Form, die in fast jedem Cluster noch steht.

> **MERKE**
>
> Dieser Teil ist für das Lesen, Betreiben und Migrieren. Neue Policies schreibst du als CEL-Typ. `ClusterPolicy` und `Policy` sind seit 1.19 abgekündigt.

Eine `ClusterPolicy` gilt clusterweit, eine `Policy` für einen Namespace. Beide enthalten eine Liste von **Regeln**. Jede Regel macht genau eines: `validate`, `mutate`, `generate` oder `verifyImages`.

require-app-label-alt.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: "Das Label app fehlt."
        pattern:
          metadata:
            labels:
              app: "?*"
```

`match` und `exclude`

- `any`: Eine der Bedingungen reicht (ODER).
- `all`: Alle müssen passen (UND).

- Filter gibt es für `kinds`, `names`, `namespaces`, `selector`, `operations`, `namespaceSelector` und für Nutzer über `subjects`, `roles`, `clusterRoles`.

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

`failureAction`

Alt	Neu
`Enforce`	`validationActions: [Deny]`
`Audit`	`validationActions: [Audit]`
`emitWarning: true`	`validationActions: [Warn]`

Ältere Policies setzen das Feld noch auf Policy-Ebene als `spec.validationFailureAction`.

> **STOLPERFALLE**
>
> Die Namen täuschen: `failurePolicy` regelt den **Ausfall von Kyverno**. `failureAction` regelt den **Verstoß gegen die Regel**. Das sind zwei getrennte Schalter, und das ist eine beliebte Interviewfrage.

> **IM INTERVIEW**
>
> **`failureAction` oder `failurePolicy`?**
>
> `failureAction` sagt, was bei einem Verstoß passiert: `Enforce` oder `Audit`. `failurePolicy` sagt, was passiert, wenn der Webhook nicht antwortet: `Fail` oder `Ignore`. Bei den neuen Policies heißt das erste `validationActions`.

DAS NIMMST DU MIT

- Eine Policy, viele Regeln, jede Regel eine Art.
- `match` und `exclude` mit `any` und `all`.
- `failureAction` ungleich `failurePolicy`.

19

Validate: pattern und deny

Zwei Wege, eine Regel auszudrücken.

`pattern`: So muss es aussehen

Ein `pattern` ist ein Stück YAML, das zum Objekt passen muss.

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

Zeichen	Bedeutung
`*`	Beliebig viele Zeichen.
`?`	Genau ein Zeichen. `?*` heißt: nicht leer.
`!`	Nicht.
`|`	Oder.
`>`, `<`	Vergleich, etwa `>=2`.

Eine Liste im Pattern gilt für **alle** Elemente der Liste.

Anker

Mit Klammern um den Feldnamen änderst du die Logik:

Anker	Bedeutung
`(feld)`	Bedingung: Prüfe den Rest nur, wenn das hier passt.
`=(feld)`	Wenn das Feld existiert, muss es passen.
`^(feld)`	Mindestens ein Listenelement muss passen.
`X(feld)`	Das Feld darf nicht existieren.
`+(feld)`	Mutation: Nur setzen, wenn es fehlt.

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

> **DAS BILD IM KOPF**
>
> *Ein Pattern ist eine Schablone. Liegt sie auf dem Objekt und passt, ist alles gut. Anker sind Löcher in der Schablone mit Bedingungen.*

`deny`: Das ist verboten

`deny` dreht die Logik um. Sind die Bedingungen wahr, wird abgelehnt.

```
validate:
  message: "Löschen nur für Admins."
  deny:
    conditions:
      all:
        - key: "{{ request.operation }}"
          operator: Equals
          value: DELETE
```

Wichtige Operatoren sind `Equals`, `NotEquals`, `AnyIn`, `AllIn`, `AnyNotIn`, `AllNotIn`, `GreaterThan` und `LessThan`.

IM INTERVIEW

`pattern` oder `deny`?

`pattern` beschreibt, wie ein Objekt aussehen soll, und ist gut für Feldprüfungen. `deny` beschreibt verbotene Bedingungen und kann Variablen und Operationen wie `DELETE` prüfen. In CEL gibt es beides nur noch als Ausdruck, bei `deny` mit umgekehrter Logik.

DAS NIMMST DU MIT

- `pattern`: Schablone. `deny`: verbotene Lage.
- `?*` heißt nicht leer.
- Anker steuern Bedingungen.

20

Variablen und JMESPath

Doppelte geschweifte Klammern überall.

In der alten Welt steht alles in `{{ }}`. Innen steht ein **JMESPath**-Ausdruck, eine Abfragesprache für JSON.

Variable	Inhalt
`{{ request.object }}`	Das neue Objekt.
`{{ request.oldObject }}`	Das alte Objekt.
`{{ request.operation }}`	`CREATE`, `UPDATE`, `DELETE`.
`{{ request.namespace }}`	Der Namespace.
`{{ request.userInfo.username }}`	Der anfragende Nutzer.
`{{ serviceAccountName }}`	Der ServiceAccount.
`{{ images.containers }}`	Zerlegte Image-Referenzen.
`{{ element }}`	Das aktuelle Element in `foreach`.

JMESPath lesen

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

- `[]` macht eine Liste flach.
- `[?...]` filtert.
- `||` setzt einen Standard.
- `|` reicht das Ergebnis weiter, und `@` ist das aktuelle Element.

`context`: Daten von außen

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

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

Daneben gibt es `variable` für eigene Werte, `imageRegistry` für Image-Daten und `globalReference` für Kyvernos globalen Cache.

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

STOLPERFALLE

Fehlt ein Wert in `{{ }}`, schlägt die Ersetzung fehl, und die Regel liefert einen Fehler. Setze mit `||` einen Standard. Kyverno rechnet außerdem Variablen auch innerhalb von Texten aus. Wer buchstäblich `{{` braucht, muss es mit einem Backslash schützen, etwa in Helm-Templates.

IM INTERVIEW

Wie liest du eine ConfigMap in einer Policy?

In der `ClusterPolicy` über `context` mit `configMap`. In der neuen Welt über `variables` mit `resource.Get`. So bleiben Listen wie erlaubte Registries außerhalb der Policy änderbar.

DAS NIMMST DU MIT

- `{{ request.object... }}` ist alt, `object` ist neu.
- `||` für Standardwerte, `|` zum Weiterreichen.
- `context` holt ConfigMaps und API-Daten.

21

Preconditions und foreach

Wann eine Regel gilt, und wie sie über Listen läuft.

Preconditions

`match` wählt nach Art und Namen. `preconditions` entscheiden mit Variablen, ob die Regel überhaupt läuft. Sind sie falsch, wird die Regel übersprungen (`skip`), nicht abgelehnt.

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

`any` heißt ODER, `all` heißt UND. In CEL wird daraus `spec.matchConditions`.

`foreach`

`foreach` läuft über eine Liste. Das Element heißt `element`, die Position `elementIndex`.

```
validate:
  message: "Image {{ element.image }} ist nicht erlaubt."
  foreach:
    - list: request.object.spec.containers
      deny:
        conditions:
          any:
            - key: "{{ element.image }}"
              operator: Equals
              value: "*:latest"
```

Auch in `mutate` und `generate` gibt es `foreach`:

```
mutate:
  foreach:
```

```
    - list: request.object.spec.containers
      patchStrategicMerge:
        spec:
          containers:
            - name: "{{ element.name }}"
              image: "registry.io/{{ element.image }}"
```

DAS BILD IM KOPF

`preconditions` ist die Frage „Bin ich überhaupt zuständig?“. `foreach` ist der Prüfer, der jede Kiste auf der Palette einzeln öffnet.

In CEL entfällt `foreach`. Du nutzt `all()`, `exists()`, `filter()` und `map()`.

STOLPERFALLE

Mutationen mit `foreach` über Listen sind heikel. Ein Index im JSON Patch kann sich durch frühere Änderungen verschieben. Bei Löschungen läuft die Liste darum rückwärts (`order: Descending`). In verschachtelten Schleifen teste das besonders gründlich.

IM INTERVIEW

Was ist der Unterschied zwischen `match` und `preconditions`?

`match` filtert, welche Ressourcen überhaupt infrage kommen, und steuert die Webhooks. `preconditions` wertet danach Variablen aus und überspringt die Regel bei `false`. In CEL heißen sie `matchConstraints` und `matchConditions`.

DAS NIMMST DU MIT

- `preconditions` falsch: `skip`, nicht Ablehnung.
- `foreach` mit `element`.
- In CEL: `matchConditions` und Listenfunktionen.

22

Mutate, Generate, verifyImages

Die anderen drei Regelarten in Kurzform.

`mutate`

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

- `+(team)`: nur setzen, wenn es fehlt.
- `(name): "*"`: für jeden Container.

Für Präzision gibt es `patchesJson6902`. Für bestehende Ressourcen `mutateExistingOnPolicyUpdate` und `targets`. In den neuen Policies treten `patchType: ApplyConfiguration` und `JSONPatch` an ihre Stelle.

`generate`

```
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]
```

Statt `data` geht auch `clone` mit `namespace` und `name` der Vorlage.

`verifyImages`

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

`mutateDigest` ersetzt den Tag durch den Digest. So läuft genau das geprüfte Image.

Autogen in der alten Welt

Autogen ist bei `ClusterPolicy` standardmäßig an. Du siehst die erzeugten Regeln im Status als `autogen-...` und schaltest sie mit einer Annotation ab:

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

IM INTERVIEW

Warum setzt `verifyImages` den Digest ein?

Damit das geprüfte Image auch das ist, das läuft. Ein Tag kann sich zwischen Prüfung und Pull ändern. Mit Digest gibt es diese Lücke nicht.

DAS NIMMST DU MIT

- `+(feld)` nur wenn es fehlt.
- `generate` mit `data` oder `clone`.
- `mutateDigest` schließt die Tag-Lücke.

23

Alte Aufräumer und Ausnahmen

CleanupPolicy und PolicyException aus `kyverno.io`.

`ClusterCleanupPolicy`

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

Das Objekt selbst heißt hier `target`. Intern erzeugt der Cleanup Controller dafür CronJobs. Die neue Form ist die `DeletingPolicy` aus Kapitel 14.

Alte `PolicyException`

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

Beachte `autogen-...`: Die Ausnahme muss auch die erzeugten Regeln nennen.

Alt	Neu
`kyverno.io PolicyException`	`policies.kyverno.io PolicyException`
`exceptions[].policyName`	`policyRefs[].name`, `kind`
`match`	`matchConditions` mit CEL
Zeit über Cleanup	`expiresAt`

DAS NIMMST DU MIT

- `CleanupPolicy` wird `DeletingPolicy`.
- Bei alten Ausnahmen die `autogen-`-Regeln nicht vergessen.
- Neue Ausnahmen nennen Policy und Typ.

24

Migration nach CEL

Wie du 200 ClusterPolicies sicher umstellst.

Warum jetzt?

Ab 1.20 wird eine alte Policy nicht mehr durchgesetzt. Sie bleibt lesbar, aber sie schützt nicht mehr.

Die Übersetzung

`ClusterPolicy`	**CEL-Policy**
`rules[].match`	`matchConstraints`, `matchConditions`
`rules[].exclude`	`matchConstraints.excludeResourceRules`
`rules[].context`	`variables`
`preconditions`	`matchConditions`
`validate.pattern`	`validations` mit `expression`
`validate.deny`	`validations`, Logik umkehren
`validate.foreach`	`all()`, `exists()`, `exists_one()`
`validate.podSecurity`	Nicht verfügbar. Jede Regel einzeln.
`validate.manifests`	Nicht verfügbar.
`allowExistingViolations`	Nicht verfügbar. Ausnahmen nutzen.
`failureActionOverrides`	Nicht verfügbar. Ausnahmen nutzen.
`mutate.patchStrategicMerge`	`patchType: ApplyConfiguration` oder `JSONPatch`
`generate.data`, `clone`	`generate` mit CEL
`verifyImages`	`ImageValidatingPolicy`
`background`	`evaluation.background.enabled`

Eine Regel, eine Policy

Eine `ClusterPolicy` mit fünf Regeln wird zu mehreren Policies. Das `applyRules` fällt weg. Du teilst die Regeln auf und nutzt `matchConditions`.

Der Plan

1. **Inventar:** `kubectl get cpol,pol -A` und die Policies in Git zählen.
2. **Aufräumen:** Was ist überflüssig? Was deckt PSA schon ab?

3. **Übersetzen:** mit dem Migration Guide, eine Policy nach der anderen.
4. **Testen:** Die bestehenden CLI-Tests laufen weiter. Prüfe die neue Policy gegen dieselben Beispiele.
5. **Parallel fahren:** Neue Policy im `Audit` neben der alten. Reports vergleichen.
6. **Umschalten:** Neue Policy auf `Deny`, alte entfernen.
7. **Upgraden:** Erst wenn keine alte Policy mehr aktiv ist.

DAS BILD IM KOPF

Der Umzug von Brücke zu Brücke. Du sperrst die alte erst, wenn der Verkehr nachweislich über die neue läuft.

STOLPERFALLE

Bei `deny` kehrt sich die Logik um. Was vorher „verboten, wenn wahr“ hieß, heißt jetzt „erlaubt, wenn wahr“. Ein vergessenes `!` verwandelt die Regel in ihr Gegenteil. Darum Negativ-Tests zuerst.

STOLPERFALLE

Autogen verhält sich anders. Bei `ClusterPolicy` war es automatisch an, bei den neuen Typen schaltest du `autogen.podControllers` gezielt ein. Vergisst du das, kommen Deployments wieder durch, und die Fehler tauchen erst bei den Pods auf.

IM INTERVIEW

Wie migrierst du von `ClusterPolicy` auf CEL?

Inventar, Aufräumen, Übersetzen mit dem Migration Guide, Tests wiederverwenden. Dann neue Policies im `Audit` parallel zur alten fahren und die Reports vergleichen. Erst danach umschalten und auf 1.20 aktualisieren. Bis dahin bleibe ich bewusst auf 1.19.

DAS NIMMST DU MIT

- Eine Regel wird zu einer Policy.
- `deny` umdrehen, Autogen einschalten.
- Parallel im `Audit` fahren, dann umschalten.

TEIL V

Tests und Betrieb

CLI, CI, GitOps, Hochverfügbarkeit, Leistung und Upgrades.

25

Kyverno CLI

Policies prüfen, ohne einen Cluster zu brauchen.

```
brew install kyverno
```

Eselsbrücke: Oder als Plugin: `kubectl krew install kyverno`.

```
kyverno version
```

Eselsbrücke: Passt die CLI zur Cluster-Version? Sie sollte es.

Eine Policy anwenden

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

Eselsbrücke: Lies es als Satz: „Wende diese Policy auf diese Ressource an." Alles passiert lokal.

```
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
```

Eselsbrücke: Gegen alles im aktuellen Cluster. Nur lesend, eine billige Vorschau auf den Rollout.

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

Eselsbrücke: Testet auch die Ausnahme. Erwartetes Ergebnis: `skip`.

Die Probe im Cluster

Auch `kubectl` hilft, wenn Policies schon im Cluster sind. Ein Server-Dry-Run läuft durch die Admission-Kette, speichert aber nichts:

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

Eselsbrücke: `server` statt `client`. Kyverno prüft, aber nichts landet in etcd.

> **MERKE**
>
> Für Experimente ohne Installation gibt es den Kyverno Playground im Browser (`playground.kyverno.io`). Er unterstützt auch die neuen Policy-Typen.

IM INTERVIEW

Wie testest du eine Policy, bevor sie in den Cluster kommt?

Lokal mit `kyverno apply` und `kyverno test` in der CI. Danach `--cluster` gegen den Bestand als Vorschau und `--dry-run=server` für einzelne Objekte. Im Cluster startet sie als `Audit`.

DAS NIMMST DU MIT

- `kyverno apply` geht ohne Cluster.
- `--cluster` zeigt, was im Bestand durchfallen würde.
- `--dry-run=server` läuft durch die Webhooks.

26

Tests

Eine Policy ohne Test ist eine Vermutung.

Unit-Tests mit `kyverno test`

Du legst neben die Policy eine Datei `kyverno-test.yaml`.

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: [gut]
    result: pass
  - isValidatingPolicy: true
    policy: image-hygiene
    kind: Pod
    resources: [latest, ohne-tag]
    result: fail
  - isValidatingPolicy: true
    policy: image-hygiene
    kind: Pod
    resources: [legacy-batch-1]
    result: skip
```

Die `values.yaml` liefert Daten, die lokal fehlen, etwa die Labels eines 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/
```

Eselsbrücke: Sucht alle Testdateien im Ordner und führt sie aus.

> **DAS BILD IM KOPF**
>
> *Jede Regel braucht zwei Testfälle: einen, der durchkommen muss, und einen, der scheitern muss. Eine Tür ist erst geprüft, wenn der Schlüssel öffnet und der falsche nicht.*

E2E mit Chainsaw

Chainsaw ist das Test-Werkzeug des Kyverno-Projekts. Es läuft gegen einen echten Cluster und prüft das Verhalten der Webhooks.

`e2e/image-hygiene/chainsaw-test.yaml (Auszug)`

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

```
chainsaw test e2e/
```

Eselsbrücke: Spielt die Schritte gegen den aktuellen Cluster ab.

	`kyverno test`	**Chainsaw**
Braucht	Keinen Cluster.	Echten Cluster.
Prüft	Die Logik der Policy.	Das ganze Zusammenspiel.
Tempo	Sekunden.	Minuten.

STOLPERFALLE

Teste auch `skip`-Fälle. Eine Policy, die nie matcht, liefert keine Fehler und sieht gesund aus. Prüfe also, dass die Regel wirklich etwas trifft.

IM INTERVIEW

Wie testest du Policies?

Unit-Tests mit `kyverno test`: Positiv-, Negativ- und Ausnahmefälle, in jeder Pull Request. Dazu E2E mit Chainsaw in einem kind-Cluster, weil nur das Webhooks, Autogen und RBAC wirklich prüft.

DAS NIMMST DU MIT

- Pro Regel ein `pass` und ein `fail`.
- `kyverno test` schnell, Chainsaw echt.
- Auch `skip` testen.

27

CI und GitOps

Policies leben in Git wie jeder andere Code.

Die Pipeline

```
Pull Request
  -> Lint und Schema
  -> kyverno test
  -> Chainsaw in kind
  -> Review
Merge
  -> Argo CD oder Flux synchronisiert
```

`.github/workflows/policies.yaml (Auszug)`

```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/
```

Workloads schon in der CI prüfen

`kyverno apply` lässt sich auch im Anwendungs-Repo gegen gerenderte Manifeste ausführen:

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

So sieht der Entwickler den Fehler im Pull Request, nicht erst beim Deployment. Das ist **Shift Left**.

GitOps-Fallen

Problem	Lösung
Argo CD meldet Drift nach Mutation.	Server-Side Diff mit `IncludeMutationWebhook=true`, sonst `ignoreDifferences`.
Drift bei aggregierten ClusterRoles.	Diese Rollen per `ignoreDifferences` ausnehmen.
Argo CD und Kyverno-Chart nutzen dasselbe Label.	In Argo CD eine andere Tracking-Methode wählen.
Post-Delete-Hook stört.	`config.preserve=false` setzen.
Generierte Objekte erscheinen als fremd.	Ownership klären: Nur einer verwaltet ein Objekt.
Kyverno-CRDs sind zu groß.	`ServerSideApply=true` nutzen, `Replace=true` vermeiden.
Reports stören in der UI.	Reports in Argo CD ausblenden.
Policy vor CRD angewendet.	CRDs vor Policies, etwa mit der Annotation `argocd.argoproj.io/sync-wave`.

Stufen

Rolle Policies über Umgebungen aus: `dev` im `Deny`, `prod` noch im `Audit`, und erst danach überall. Kustomize-Overlays eignen sich gut, um pro Umgebung `validationActions` zu ändern.

> **STOLPERFALLE**
>
> Wenn Kyverno im selben Argo-CD-Sync wie Workloads aktualisiert wird, kann ein Fehler beides blockieren. Argo CD braucht selbst Pods, um zu reparieren. Halte einen Notfallweg bereit, der ohne GitOps funktioniert (Kapitel 33).

IM INTERVIEW

Wie bringst du Policies in den Cluster?

Über GitOps: Policies liegen im Repo, Pull Requests laufen durch `kyverno test` und Chainsaw, Argo CD synchronisiert. Umgebungen unterscheiden sich per Overlay, etwa `Audit` in Produktion und `Deny` in Entwicklung.

DAS NIMMST DU MIT

- Pipeline: Lint, Unit, E2E, Review, Sync.
- Shift Left: Manifeste schon in der CI prüfen.
- Mutation erzeugt Drift. Plane das ein.

28

Hochverfügbarkeit

Wenn Kyverno im Pfad liegt, muss es stehen.

Controller	Replikate	Skalierung
Admission	Mindestens 3	Alle bearbeiten Webhooks. Mehr Replikate bringen mehr Leistung.
Background	2	Ein Leader. Mehr Replikate nur für Ausfallschutz.
Reports	2	Ein Leader. Mehr Replikate nur für Ausfallschutz.
Cleanup	2	Mehrere Replikate erhöhen den Durchsatz.

Drei ist für den Admission Controller das Minimum, das Kyverno als hochverfügbar unterstützt.

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]
```

> **DAS BILD IM KOPF**
>
> *Drei Schalter, drei Beamte, verteilt auf verschiedene Gebäude. Ein Stromausfall in einem Gebäude darf den Schalterbetrieb nicht beenden.*

Webhook-Ausschlüsse

Standardmäßig schließt das Chart `kube-system` im `namespaceSelector` aus, und der Kyverno-Namespace ist ebenfalls ausgeschlossen. Diese Ausnahmen retten dich beim Neustart des Clusters. Sie haben aber einen Preis: Dort greift keine Policy. Schütze diese Namespaces also mit RBAC.

Kritische Regeln ohne Webhook

Du kannst eine `ValidatingPolicy` zusätzlich als native `ValidatingAdmissionPolicy` erzeugen. Sie läuft im API-Server und wirkt auch, wenn Kyverno ausfällt:

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

Das schließt sich mit `autogen.podControllers` aus, denn eine VAP kann die Pod-Controller-Varianten nicht tragen. Auch für `MutatingPolicy` gibt es eine solche Option.

> **STOLPERFALLE**
>
> Zwei Replikate wirken wie HA, sind es aber nicht. Laut Kyverno ist drei das Minimum. Ohne PodDisruptionBudget kann außerdem ein Node-Upgrade alle Replikate gleichzeitig evakuieren.

> **IM INTERVIEW**
>
> **Wie machst du Kyverno hochverfügbar?**
>
> Drei Admission-Replikate auf verschiedenen Nodes mit PodDisruptionBudget, zwei für die anderen Controller. `kube-system` und `kyverno` sind ausgeschlossen. Kritische Regeln erzeuge ich zusätzlich als native VAP. Dazu kommen Alarme auf Latenz und Verfügbarkeit.

IM INTERVIEW

Warum bringen mehr Reports-Replikate keine Leistung?

Der Controller arbeitet mit Leader Election. Nur ein Replikat arbeitet aktiv. Die anderen sind Ersatz. Mehr Leistung bringen dort nur mehr CPU und RAM.

DAS NIMMST DU MIT

- Admission 3, Rest 2.
- PDB und Topology Spread.
- `kube-system` und `kyverno` ausnehmen.
- Kritische Regeln als VAP zusätzlich.

29

Performance

Jede Millisekunde liegt im Pfad des API-Servers.

Die Hebel

Hebel	Wirkung
Enges `matchConstraints`	Kyverno bekommt nur, was es braucht.
Keine Wildcards	Kein `resources: ["*"]`. Das schickt fast alles an Kyverno.
`config.resourceFilters`	Ignoriert Rauschen wie Events.
`namespaceSelector`	Spart Aufrufe auf Webhook-Ebene.
Native VAP	Läuft im API-Server, ohne Netzweg.
`GlobalContextEntry`	Cache für externe Daten.
Mehr Admission-Replikate	Mehr parallele Anfragen.

Teure Dinge

- `resource.Get` oder `apiCall` pro Anfrage ruft den API-Server auf.
- Image-Prüfung spricht mit der Registry.
- Jede Policy, die auf `*` matcht, wirkt auch auf häufige Objekte wie Leases.

Ein Vergleich: Ein Lease wird ständig aktualisiert. Trifft ihn eine Policy, zahlt der Cluster bei jeder Erneuerung.

Globaler Kontext

`GlobalContextEntry` (Kurzname `gctxentry`) holt Daten regelmäßig im Hintergrund. Die Policy liest aus dem Cache statt aus dem API-Server. Das bleibt auch mit CEL-Policies erhalten, nur ältere API-Versionen sind abgekündigt.

> **DAS BILD IM KOPF**
>
> *Statt für jeden Kunden ins Lager zu laufen, stellst du die häufigsten Teile an die Kasse.*

Reports begrenzen

Bei großen Clustern belasten Reports etcd und den API-Server. `features.admissionReports.enabled` lässt sich abschalten, wenn Background-Ergebnisse genügen. Das Intervall für den Scan steuert `features.backgroundScan.backgroundScanInterval`.

Messen

Der wichtigste Wert ist die Admission-Latenz:

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

STOLPERFALLE

Eine Policy auf `*`-Ressourcen wirkt im Labor harmlos. In einem Cluster mit tausenden Pods und ständigen Leases landet jede Änderung bei Kyverno. Prüfe nach jedem Rollout die Latenz.

IM INTERVIEW

Kyverno macht den API-Server langsam. Was tust du?

Ich messe `kyverno_admission_review_duration_seconds` und suche die teure Policy über `kyverno_policy_execution_duration_seconds`. Dann verenge ich die Matches, entferne Wildcards, cache externe Daten und erhöhe die Admission-Replikate.

DAS NIMMST DU MIT

- Enger Match ist der größte Hebel.
- `resource.Get` und Image-Prüfung kosten.
- Latenz messen, nicht raten.

30

Monitoring

Vier Zahlen und ein Notfallplan.

Kyverno stellt Prometheus-Metriken bereit. Das Chart bringt optional ein Grafana-Dashboard mit (`grafana.enabled`).

Metrik	Bedeutung
`kyverno_admission_review_duration_seconds`	Latenz der Admission-Reviews insgesamt.
`kyverno_policy_execution_duration_seconds`	Latenz pro Regel.
`kyverno_admission_requests_total`	Anzahl der Anfragen.
`kyverno_policy_results_total`	Ergebnisse nach `rule_result` und Auslöser.

Gute Alerts

- Weniger als zwei Admission-Replikate bereit.
- p99-Latenz der Admission-Reviews über deiner Schwelle.
- Viele `fail`- oder `error`-Ergebnisse in `kyverno_policy_results_total`.
- `UpdateRequest`s bleiben im Zustand `Failed` oder `Pending`.
- Webhook-Fehler im API-Server, dort `apiserver_admission_webhook_rejection_count`.

Prüfen, ob Regeln noch gelten

Überwache die `failurePolicy` deiner Webhooks. Sie kann sich durch Konfiguration ändern, etwa durch `features.forceFailurePolicyIgnore`. Dann sind deine Regeln bei einem Ausfall plötzlich weich, und niemand merkt es.

IM INTERVIEW

Was überwachst du bei Kyverno?

Verfügbarkeit der Replikate, Admission-Latenz, Ergebnisse pro Policy, Webhook-Fehler im API-Server und hängende `UpdateRequest`s. Dazu die Anzahl der Reports als Lastindikator.

DAS NIMMST DU MIT

- Latenz, Verfügbarkeit, Ergebnisse, Fehler.
- API-Server-Metriken sehen, was Kyverno nicht sieht.
- `failurePolicy` auf Drift prüfen.

31

Upgrades

Der Sprung auf 1.20 ist kein normales Upgrade.

Das normale Upgrade

1. Release Notes lesen.
2. In einem Testcluster mit echten Policies üben.
3. Unterschied ansehen: `helm template` alt und neu vergleichen.
4. CLI-Version anheben, Tests laufen lassen.
5. Upgrade, Reports und Latenz beobachten.

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

Eselsbrücke: Immer feste Version, immer deine Werte.

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

Eselsbrücke: Schreibt gespeicherte Objekte in die aktuelle Storage-Version um. Das gehört zum Upgrade auf 1.19.

Das Upgrade auf 1.20

Stand der Chart-Doku für 1.20 (noch Release Candidate):

- Alte Policies aus `kyverno.io` werden **nicht mehr durchgesetzt**. Das betrifft `ClusterPolicy`, `Policy`, `CleanupPolicy`, `ClusterCleanupPolicy` und die alte `PolicyException`.
- Ihre CRDs bleiben, damit du sie lesen, exportieren und löschen kannst.
- Anlegen und Ändern werden abgelehnt, selbst ein `kubectl apply` ohne Änderung und reine Label-Patches. Lesen und Löschen bleiben erlaubt. GitOps-Werkzeuge melden dadurch dauerhaft Out-of-Sync, bis das alte Manifest aus dem Repository entfernt ist.
- Das Chart **blockiert** Installation und Upgrade, solange alte Policies im Cluster liegen.

Der Block hat zwei Stufen: eine Prüfung beim Rendern und einen Hook-Job für GitOps-Werkzeuge.

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

STOLPERFALLE

Dieser Wert bringt die alten Policies nicht zurück. Er bestätigt nur, dass sie nicht mehr gelten. Wer ihn setzt, um das Upgrade „durchzubekommen“, schaltet still seine Regeln ab.

MERKE

Bei Redaktionsschluss war 1.20 nur als Release Candidate verfügbar. Die Beschreibung folgt dem Code und Chart-README von `v1.20.0-rc.1`. Das ältere Tracking-Issue #17214 nennt noch ein Weiterlaufen der Durchsetzung, ist also überholt oder nicht aktualisiert. Prüfe die endgültigen Release Notes, bevor du planst.

IM INTERVIEW

Was ändert sich mit Kyverno 1.20?

Die alten Typen wie `ClusterPolicy` werden nicht mehr durchgesetzt. Das Chart blockiert das Upgrade, solange solche Policies existieren. Ich migriere also vorher nach CEL und bleibe bis dahin auf 1.19.

DAS NIMMST DU MIT

- Üben, Diff, Tests, dann Upgrade.
- 1.20 setzt alte Policies nicht mehr durch.
- `acknowledgeLegacyPoliciesNotEnforced` ist keine Lösung.

TEIL VI

Troubleshooting

Hier trennt sich im Interview die Spreu vom Weizen.

32

Die Methode

Erst schauen, dann denken, dann erst handeln.

Fünf Schritte

1. **Symptom:** Was genau passiert? Wie lautet die Fehlermeldung?
2. **Umfang:** Ein Objekt, ein Namespace, der ganze Cluster?
3. **Daten:** Policy-Status, Events, Reports, Logs.
4. **Hypothese:** Was könnte die Ursache sein?
5. **Test:** Die kleinste sichere Prüfung.

Wer hat abgelehnt?

Die Meldung beim `kubectl apply` verrät viel:

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

- Ein Webhook-Name mit `kyverno` zeigt auf Kyverno. Der genaue Name hängt von Version und Konfiguration ab, also prüfe im Skript nicht darauf.
- `failed calling webhook` heißt: Kyverno antwortet nicht. Das ist ein Ausfall, kein Verstoß.
- `context deadline exceeded` heißt: Timeout.
- Eine Meldung von `ValidatingAdmissionPolicy` zeigt auf eine native Policy.

Die erste Minute

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

Eselsbrücke: Lebt Kyverno?

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

Eselsbrücke: Welche Policies gibt es, und sind sie bereit?

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

Eselsbrücke: Sind die Webhooks da?

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

Eselsbrücke: Was sagt der Schalterbeamte?

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

Eselsbrücke: Verstöße als Events.

> **MERKE**
>
> Viele scheinbare Kyverno-Probleme sind gar keine. Prüfe die anderen Admission-Bausteine: PSA, VAPs und andere Webhooks.

IM INTERVIEW

Eine Anfrage wird abgelehnt. Woher kommt das?

Ich lese die Meldung: Sie nennt den Webhook oder die Policy. Bei `failed calling webhook` ist es ein Ausfall, bei einer eigenen Meldung ein Verstoß. Danach prüfe ich die Policy, den Report und die Logs.

DAS NIMMST DU MIT

- Fehlermeldung genau lesen.
- `denied` ist ein Verstoß. `failed calling` ist ein Ausfall.
- Status, Events, Logs.

33

Der Cluster ist ausgesperrt

Das Szenario, das in jedem Senior-Interview kommt.

Symptom: Keine Pods mehr anlegbar. `kubectl apply` läuft in Timeouts. Der Cluster wirkt kaputt.

Ursache: Alle Kyverno-Pods sind weg, etwa nach einem Ausfall oder dem Austausch einer Node-Gruppe. Die Policies stehen auf `Fail` und treffen Pods. Der API-Server fragt ins Leere.

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

Besonders gemein ist der Kreis: Ohne Pods kein Kyverno, und ohne Kyverno keine Pods.

> **DAS BILD IM KOPF**
>
> *Die Sicherung im Sicherungskasten ist raus, und der Kasten öffnet sich nur elektrisch.*

Die Rettung

Du löschst die Webhooks, die normale Ressourcen zu Kyverno schicken:

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

Eselsbrücke: Trenne die Leitung zum Prüfer.

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

Eselsbrücke: Das Gleiche für die Mutation.

Das sind nur diese beiden. Die übrigen Webhooks sind für Kyvernos interne Objekte da.

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

Eselsbrücke: Meist unnötig, aber sinnvoll, wenn Kyverno nicht von allein startet.

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

Eselsbrücke: Warte, bis die Pods bereit sind.

Sobald Kyverno wieder läuft, legt es seine Webhooks selbst neu an, passend zu den installierten Policies.

> **STOLPERFALLE**
>
> Während der Rettung gelten keine Policies. Danach prüfst du, was in dieser Zeit hereingekommen ist. Der Background Scan zeigt die Verstöße in den Reports.

Vorbeugen

- Drei Replikate, PodDisruptionBudget, Verteilung über Nodes.
- `kube-system` und `kyverno` ausgeschlossen lassen.
- Kritische Regeln zusätzlich als native VAP.
- `failurePolicy: Ignore` für Regeln, die nicht kritisch sind.
- Das Notfall-Runbook muss griffbereit sein und darf nicht an GitOps hängen.
- Lösche nie Node-Gruppen, ohne das PDB ernst zu nehmen.
- Übe das Szenario regelmäßig im Labor.

> **STOLPERFALLE**
>
> GitOps kann dich zurückwerfen. Eine Rettung per Hand wird vom nächsten Sync vielleicht rückgängig gemacht. Halte Sync-Automatik an, wenn du am Webhook operierst.

IM INTERVIEW

Kyverno ist down, und niemand kann mehr deployen. Was tust du?

Zuerst bestätige ich die Ursache mit `failed calling webhook` und schaue, warum Kyverno weg ist. Dann lösche ich die zwei Resource-Webhooks, um den Cluster zu entsperren. Kyverno kommt hoch und erzeugt sie neu. Anschließend prüfe ich mit Reports, was in der Lücke passiert ist, und schließe die Ursache: HA, PDB, Ausschlüsse.

DAS NIMMST DU MIT

- `failed calling webhook`: Ausfall.
- Zwei Webhooks löschen, Kyverno erzeugt sie neu.
- Nachprüfen, was ungeprüft durchkam.

34

Die Policy greift nicht

Alles läuft, aber der schlechte Pod kommt durch.

Prüfe von außen nach innen.

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

Eselsbrücke: Schau in `status`: `conditionStatus.ready` muss `true` sein.

Ist die Policy bereit, gehst du diese Liste durch:

1. **Wie steht `validationActions`?** `[Audit]` blockiert nichts.
2. **Matcht sie?** Gruppe, Version, Plural, Operation.
3. **Filtert etwas davor?** `namespaceSelector`, `resourceFilters`, Webhook-Ausschlüsse.
4. **Ist die Admission-Auswertung an?** `evaluation.admission.enabled`.
5. **Greift eine Ausnahme?** Prüfe die `PolicyException`s und `skip` in Reports.
6. **Wurde der Pod von einem Controller erzeugt?** Dann prüfe Autogen.

Symptom	Ursache
Wirkt nur bei `CREATE`	`UPDATE` fehlt in `operations`.
Wirkt nicht bei Deployments	`autogen.podControllers` fehlt.
Wirkt nirgends in einem Namespace	Label fehlt für `namespaceSelector`.
`error` im Report	CEL-Laufzeitfehler: Feld fehlt.
Wirkt nicht in `kube-system`	Webhook schließt den Namespace aus.
`skip` im Report	`matchConditions` oder eine Ausnahme.

Lokal nachstellen

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

Eselsbrücke: Hole das echte Objekt.

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

Eselsbrücke: Reproduziere den Fall ohne Cluster.

STOLPERFALLE

Kyverno wertet nur aus, was ankommt. Passt der Webhook nicht, siehst du keinen Log-Eintrag, keinen Report und keinen Fehler. Stille ist kein Beweis für Gesundheit.

IM INTERVIEW

Eine Policy blockiert nicht. Wie suchst du den Fehler?

Ich prüfe den Status, dann `validationActions`, dann den Match, dann Selektoren und Ausnahmen. Mit `kyverno apply` stelle ich den Fall mit dem echten Objekt nach. Oft sind es `Audit`, fehlendes Autogen oder ein falscher Selector.

DAS NIMMST DU MIT

- Status, Aktion, Match, Filter, Ausnahmen, Autogen.
- `[Audit]` blockiert nicht.
- Echtes Objekt lokal testen.

35

Generate und Background hängen

Der Auftrag ist da, das Ergebnis fehlt.

Hier ist nicht der Admission Controller gefragt, sondern der Background Controller.

```
kubectl get updaterequests -A
```

Eselsbrücke: Die Auftragsliste. Lies die Spalte `STATUS`.

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

Eselsbrücke: Was sagt das Backoffice?

Spur	Ursache
`Failed` mit `forbidden`	Dem Background Controller fehlen RBAC-Rechte für den Zieltyp.
`Pending` bleibt	Controller hängt oder wird gedrosselt.
Kein `UpdateRequest`	Die Policy matcht das Ereignis nicht.
Objekt fehlt bei altem Namespace	`generateExisting` ist aus.
Änderung wird zurückgedreht	`synchronize` ist an.

Rechte ergänzen

Kyverno arbeitet mit Aggregated ClusterRoles. Du gibst dem Controller also eine eigene Rolle:

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]
```

Das Label ist der Schalter: Die Rolle des Controllers sammelt alle Rollen mit diesem Label ein. Für den Cleanup Controller heißt es `rbac.kyverno.io/aggregate-to-cleanup-controller`.

IM INTERVIEW

Eine `GeneratingPolicy` erzeugt nichts. Wo schaust du?

Bei den `UpdateRequest`s und in den Logs des Background Controllers. Typisch sind fehlende RBAC-Rechte oder ein Auslöser, der nicht passt. Bei alten Namespaces fehlt oft `generateExisting`.

DAS NIMMST DU MIT

- `kubectl get updaterequests -A`.
- `forbidden` heißt RBAC.
- Der Background Controller führt aus, nicht der Nutzer.

36

Die Image-Prüfung scheitert

Signiert ist es, durch kommt es trotzdem nicht.

Meldung	Ursache
Timeout	Registry langsam. `timeoutSeconds` erhöhen.
`unauthorized`	Pull Secret fehlt für Kyverno.
`x509`	Eigene CA der Registry nicht vertraut.
Keine Signatur gefunden	Falscher Key, falsche Identität oder anderes Repo.
Attestation fehlt	Falscher `type` oder keine Signatur am Bundle.

Von Hand prüfen

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

Eselsbrücke: Sieht Cosign die Signatur? Wenn nicht, liegt das Problem nicht bei Kyverno.

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

Eselsbrücke: Wie lautet der Digest? Signatur und Image müssen dazu passen.

Keyless-Fallen

- `subject` muss zum Workflow-Pfad **und** zum Ref passen.
- `issuer` muss exakt stimmen.
- Bei eigenen Sigstore-Instanzen braucht es ein eigenes Vertrauensmaterial (`trustedRoot`, `tuf`).

> **MERKE**
>
> Im Begleitmaterial zu „Kyverno & NetworkPolicies zum Anfassen“ fand Kyverno 1.19.1 cosign-v3-Signaturen auf einer reinen HTTP-Registry nicht, selbst nicht mit `allowInsecure`. Die Lab-Registry lief deshalb mit TLS.

IM INTERVIEW

Eine signierte Image-Prüfung schlägt fehl. Was prüfst du?

Ich prüfe mit `cosign verify` außerhalb von Kyverno, ob Signatur und Key passen. Danach Zugriff von Kyverno auf die Registry: Pull Secret, CA, Timeout. Bei Keyless vergleiche ich Subject und Issuer exakt.

DAS NIMMST DU MIT

- Erst mit `cosign` selbst prüfen.
- Pull Secret, CA, Timeout.
- Keyless: Subject und Issuer exakt.

TEIL VII

Das Interview

Vergleiche, Fragen, Aufgaben und ein Spickzettel für den letzten Abend.

37

Kyverno, Gatekeeper, VAP

Drei Wege zum selben Ziel. Wann nimmst du welchen?

	Kyverno	OPA Gatekeeper	VAP
Sprache	CEL, früher JMESPath	Rego, auch CEL	CEL
Läuft als	Webhook	Webhook	Im API-Server
Validieren	Ja	Ja	Ja
Mutieren	Ja	Ja, eigene Typen wie `Assign`	Eigener Typ: `MutatingAdmissionPolicy`, stabil seit Kubernetes 1.36
Generieren	Ja	Nein	Nein
Images prüfen	Eingebaut	Über External Data Provider (Beta)	Nein
Bestand scannen	Ja, Reports	Ja, Audit	Nein
Ausnahmen	`PolicyException`	Über Constraints	Über Bindings
Lernkurve	Niedrig bis mittel	Hoch	Niedrig

OPA Gatekeeper

Gatekeeper trennt zwei Dinge:

- `ConstraintTemplate`: die Regel in Rego oder, seit Gatekeeper 3.18 stabil, in CEL.
- `Constraint`: ihre Anwendung mit Parametern.

Rego ist mächtig und auch außerhalb von Kubernetes einsetzbar, etwa für Terraform oder APIs. Das ist die Stärke von OPA: eine Sprache für alles.

ValidatingAdmissionPolicy

VAP ist eingebaut, seit Kubernetes 1.30 stabil. Sie braucht keinen Webhook und fällt also nicht aus. Eine `ValidatingAdmissionPolicyBinding` bindet die Regel an einen Geltungsbereich.

> **DAS BILD IM KOPF**
>
> *VAP ist das Schloss in der Tür: immer da, kann wenig. Kyverno ist der Sicherheitsdienst: kann viel, muss aber da sein. OPA ist der Jurist: Er kann alles, aber du brauchst seine Sprache.*

Wie du entscheidest

- Nur einfache Validierung und minimale Abhängigkeiten: **VAP**.
- Kubernetes-Plattform mit Mutation, Generierung und Signaturen: **Kyverno**.
- Einheitliche Policy-Sprache über viele Systeme, Rego-Wissen vorhanden: **OPA**.
- Oft eine Kombination: VAP für das Kritische, Kyverno für den Rest.

Kyverno kann eine `ValidatingPolicy` selbst als VAP ausrollen. Das verbindet die Vorteile.

IM INTERVIEW

Kyverno oder OPA Gatekeeper?

Kyverno für Kubernetes-Teams: Policies sind YAML und CEL, dazu Mutation, Generierung und Image-Prüfung. OPA, wenn Policy über viele Systeme geht und Rego schon da ist. Beides ist ausgereift. Entscheidend sind Team-Wissen und Bedarf.

IM INTERVIEW

Warum nicht nur `ValidatingAdmissionPolicy`?

VAP validiert sehr robust, kennt aber kein Generieren, keine Image-Signaturen, keine Reports und keine Lösch-Policies. Für kritische, einfache Regeln ist sie ideal. Kyverno baut darauf auf und kann sie sogar erzeugen.

DAS NIMMST DU MIT

- VAP: nativ. Kyverno: Plattform. OPA: übergreifend.
- Rego ist die große Hürde bei OPA.
- Kombinieren ist erlaubt.

38

Die Fragen

Kurze Fragen, kurze Antworten. Zum Wiederholen vor dem Gespräch.

Grundlagen

Was ist Kyverno?

Eine Policy Engine für Kubernetes, die als Admission Webhook läuft und Policies als Ressourcen verwaltet.

Was kann Kyverno?

Validieren, mutieren, generieren, Images prüfen, löschen, berichten.

Warum nicht nur RBAC?

RBAC prüft, wer etwas darf, aber nicht, wie das Objekt aussieht.

Wie sieht die Admission-Kette aus?

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

Warum erst Mutation?

Die Validierung soll das Endergebnis sehen.

Welche Controller gibt es?

Admission, Background, Reports, Cleanup.

Welcher ist der kritischste?

Admission, denn er liegt im Pfad des API-Servers.

Was ist ein `UpdateRequest`?

Der Auftrag für asynchrone Arbeit wie Generate.

Policies

Welche Policy-Typen gibt es?

`ValidatingPolicy`, `MutatingPolicy`, `GeneratingPolicy`, `ImageValidatingPolicy`, `DeletingPolicy`, je mit `Namespaced`-Variante.

Was passiert mit `ClusterPolicy`?

Seit 1.19 abgekündigt, ab 1.20 nicht mehr durchgesetzt.

Was ist CEL?

Die Common Expression Language, eine sichere Ausdruckssprache, die auch der API-Server nutzt.

Wie greifst du sicher auf Felder zu?

`.?feld.orValue(standard)`.

Was ist Autogen?

Aus Pod-Regeln werden Regeln für Pod-Controller.

`matchConstraints` **oder** `matchConditions`**?**

Grober Filter oder CEL-Feinfilter.

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

Ablehnen, nur berichten, nur warnen.

`failureAction` **oder** `failurePolicy`**?**

Verstoß gegen die Regel oder Ausfall des Webhooks.

Wofür Background Scan?

Um bestehende Ressourcen zu prüfen.

Was ist bei `request.userInfo` **im Background?**

Dort gibt es keinen Nutzer. Background für diese Regel aus.

Standard für `failurePolicy`**?**

`Fail`.

Standard-Timeout?

10 Sekunden, erlaubt sind 1 bis 30.

Werkzeuge

`ApplyConfiguration` **oder** `JSONPatch`**?**

Felder setzen oder präzise Operationen.

Was macht `mutateExisting`**?**

Ändert bestehende Ressourcen über den Background Controller.

Was macht `synchronize`**?**

Hält generierte Objekte aktuell und stellt sie wieder her.

Was macht `generateExisting`**?**

Wendet Generate auch auf vorhandene Auslöser an.

Wie prüfst du Signaturen?

`ImageValidatingPolicy` mit Cosign- oder Notary-Attestoren.

Was ist Keyless?

Signieren mit OIDC-Identität statt eigenem Schlüssel.

Warum Digests?

Tags sind veränderlich, Digests nicht.

Was ist eine Attestation?

Eine signierte Aussage über ein Image, etwa SBOM oder Provenance.

PSA oder Kyverno?

PSA als Sockel, Kyverno für feine Ausnahmen und Reports.

Wie funktionieren Ausnahmen?

`PolicyException` mit `policyRefs`, `matchConditions`, `expiresAt`. Standardmäßig aus.

Wo stehen Verstöße?

In `PolicyReport` und `ClusterPolicyReport`.

Was macht das Label `cleanup.kyverno.io/ttl`?

Es gibt einem Objekt ein Ablaufdatum.

Betrieb

Wie viele Admission-Replikate?

Mindestens drei.

Warum helfen mehr Reports-Replikate nicht?

Leader Election. Nur einer arbeitet.

Was schließt du vom Webhook aus?

`kube-system` und `kyverno`, dazu Bedarf je nach Plattform.

Wie machst du Regeln ausfallsicher?

Als VAP erzeugen.

Was tust du beim Lockout?

Die zwei Resource-Webhooks löschen, Kyverno wiederherstellen, Lücke prüfen.

Wie testest du Policies?

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

Was ist `--dry-run=server`?

Lauf durch die Admission-Kette, ohne zu speichern.

Wie führst du Policies ein?

Audit, Warn, Deny, über GitOps und Stufen.

Warum kämpfen Argo CD und Mutation?

Das Objekt im Cluster weicht von Git ab. `ignoreDifferences` hilft.

Wie findest du teure Policies?

`kyverno_policy_execution_duration_seconds`.

Was tust du gegen Latenz?

Matches verengen, Wildcards entfernen, cachen, skalieren.

Wer legt generierte Objekte an?

Der Background Controller, mit seinen RBAC-Rechten.

Worauf achtest du beim Upgrade auf 1.20?

Alte Policies vorher migrieren, das Chart blockiert sonst.

Wenn du etwas nicht weißt

Sag es. Dann denk laut: „Das habe ich so nicht betrieben. Ich vermute X, weil Y. Ich würde es erst im `Audit` testen.“ Bei einer Technik im Pfad des API-Servers ist genau diese Vorsicht ein gutes Zeichen.

DAS NIMMST DU MIT

- Kurze Antwort zuerst. Details nur, wenn jemand nachfragt.
- Erkläre immer, was bei einem Ausfall passiert.
- Sage auch, was du **nicht** tun würdest.

39

Aufgaben

Vier Szenarien und zwei Live-Policies aus dem Senior-Interview.

> **MERKE**
>
> Ablauf im Interview: Anforderungen nachfragen, Risiken nennen, Rollout planen, Ausfall durchspielen. Fragen zuerst: Wie viele Teams? Welche Compliance? Wer darf Ausnahmen? Was passiert bei einem Ausfall?

Aufgabe 1: Mandanten-Plattform

Situation: 30 Teams teilen sich einen Cluster.

- **Generieren:** Pro Namespace mit Label `tenant` Default-Deny, ResourceQuota, LimitRange.
- **Validieren:** PSS `restricted`, Pflicht-Labels, erlaubte Registries, keine Host-Pfade.
- **Mutieren:** sichere Standards im `securityContext`.
- **Ausnahmen:** zentraler Namespace, Review, `expiresAt`.
- **Sichtbarkeit:** Policy Reporter pro Team.

Aufgabe 2: Sichere Lieferkette

Situation: Nur Images aus der eigenen CI dürfen laufen.

- CI signiert keyless mit Cosign und hängt SBOM und Provenance an.
- `ImageValidatingPolicy` prüft Identität und Attestations.
- Timeout erhöhen, `Fail` in Produktion, Rollout erst mit `Audit`.
- Registry hochverfügbar halten, denn sie ist jetzt im Pfad.

Aufgabe 3: 200 ClusterPolicies vor 1.20

- Inventar, Dubletten streichen, PSA nutzen, wo es reicht.
- Regeln einzeln nach CEL übersetzen, Tests wiederverwenden.
- Neue Policies parallel im `Audit`, Reports vergleichen.
- Auf `Deny` schalten, alte entfernen, dann erst Upgrade.

Aufgabe 4: Das Postmortem

Situation: Ein Node-Upgrade legte alle Kyverno-Pods lahm. Drei Stunden lang kein Deployment.

- **Sofort:** Webhooks löschen, Kyverno wiederherstellen, Lücke prüfen.
- **Ursache:** Keine PDBs, Replikate auf einem Node.
- **Dauerhaft:** drei Replikate, PDB, Spread, VAP für Kritisches, Runbook, Übung.

Live 1: Keine Pods ohne 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: ohne
      expression: >-
        object.spec.containers
          .filter(c, !('memory' in
            c.?resources.?limits.orValue({})))
          .map(c, c.name)
  validations:
    - expression: "size(variables.ohne) == 0"
      messageExpression: >-
        'limits.memory fehlt bei: ' + variables.ohne.join(', ')
```

Das passiert hier:

- `Audit` zuerst, denn die Regel trifft sofort viele Workloads.
- `filter` sammelt die Sünder, damit die Meldung sie nennen kann.
- `orValue({})` verhindert Laufzeitfehler bei fehlenden Feldern.
- Autogen bringt den Fehler direkt ans Deployment.

Live 2: Nur erlaubte Registries

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: erlaubt
      expression: "['registry.example.com', 'ghcr.io']"
    - name: fremd
      expression: >-
        (object.spec.containers
          + object.spec.?initContainers.orValue([]))
        .map(c, c.image)
        .filter(i, !(image(i).registry() in variables.erlaubt))
  validations:
    - expression: "size(variables.fremd) == 0"
      messageExpression: >-
        'Registry nicht erlaubt: ' + variables.fremd.join(', ')
```

Das passiert hier:

- `image(i).registry()` zerlegt die Referenz. Ein `startsWith` wäre angreifbar, etwa mit `ghcr.io.boese.example`.
- `initContainers` zählen mit.

IM INTERVIEW

Wie gehst du eine Policy live an?

Ich kläre, was genau gelten soll, und schreibe erst den Testfall: ein gutes und ein schlechtes Objekt. Dann die Policy, mit `Audit` als Start. Ich spreche laut über Felder, die fehlen können.

DAS NIMMST DU MIT

- Erst fragen, dann Rollout und Ausfall denken.
- Immer `Audit` als Start nennen.
- `image()` statt Textvergleich.

40

Spickzettel

Alles Wichtige auf ein paar Seiten, für den letzten Abend.

Der eine Satz zu allem

Begriff	Merksatz
Admission	Erst mutieren, dann validieren.
`failurePolicy`	Kyverno fällt aus.
`validationActions`	Regel wird verletzt.
Audit	Erst beobachten.
Autogen	Fehler am Deployment, nicht am Pod.
`orValue`	Kein Absturz bei fehlendem Feld.
`synchronize`	Generiertes bleibt aktuell.
`PolicyException`	Mit Ablaufdatum.
VAP	Läuft auch ohne Kyverno.
1.20	Alte Policies gelten nicht mehr.

Status und Diagnose

```
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
```

Notfall

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

Testen

```
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
```

Das Grundgerüst

```
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-Muster

```
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")
```

Zahlen, die du kennen musst

Wert	Bedeutung
10 s	Standard-Timeout eines Webhooks.
1–30 s	Erlaubter Bereich für `timeoutSeconds`.
1 h	Standardintervall des Background Scans.
3	Minimum der Admission-Replikate für HA.
1.17	CEL-Policies stabil, `ClusterPolicy` zur Abkündigung vorgemerkt.
1.19	Letzte Version mit vollem Support für alte Typen.
1.20	Alte Policies werden nicht mehr durchgesetzt.

Niemals ohne Nachdenken

- Neue Policy direkt auf `Deny` schalten.
- `resources: ["*"]` mit `failurePolicy: Fail`.
- `kyverno`- oder `kube-system`-Ausschluss entfernen.
- Admission Controller mit einem Replikat in Produktion.
- `DeletingPolicy` ohne Probelauf.
- `upgrade.acknowledgeLegacyPoliciesNotEnforced=true` ohne Migration.
- `PolicyException` ohne Ablaufdatum.

DAS NIMMST DU MIT

- Erst `Audit`, dann `Deny`.
- Kyverno gehört zur Control Plane.
- Ausfall immer mitdenken.

Stichwörter

Admission Control 13
Admission Controller 15
Anker 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
Metriken 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 40
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