Kyverno Certified Associate

This exam is an online, proctored, multiple-choice exam.

Resources

- https://kyverno.io/docs/

Topics

▶ Fundamentals of Kyverno (18%)

- Kyverno Policies & Rules
- YAML Manifests
- Admission Controllers
- OCI Images

Kyverno is **a cloud native policy engine**. Originally built for Kubernetes but can now also be used outside of Kubernetes.

Kyverno allows platform engineers to automate security, compliance and best practices validation and deliver secure self-service to application teams.

How Kyverno works

Installing

The simplest way of installing `kyverno` is to use: `kubectl create -f <https://github.com/kyverno/kyverno/releases/latest/download/install.yaml>`

The following components are installed in the `kyverno` namespace:

- Kyverno Admission Controller
- Kyverno Background Controller
- Kyverno Cleanup Controller
- Kyverno Report Controller

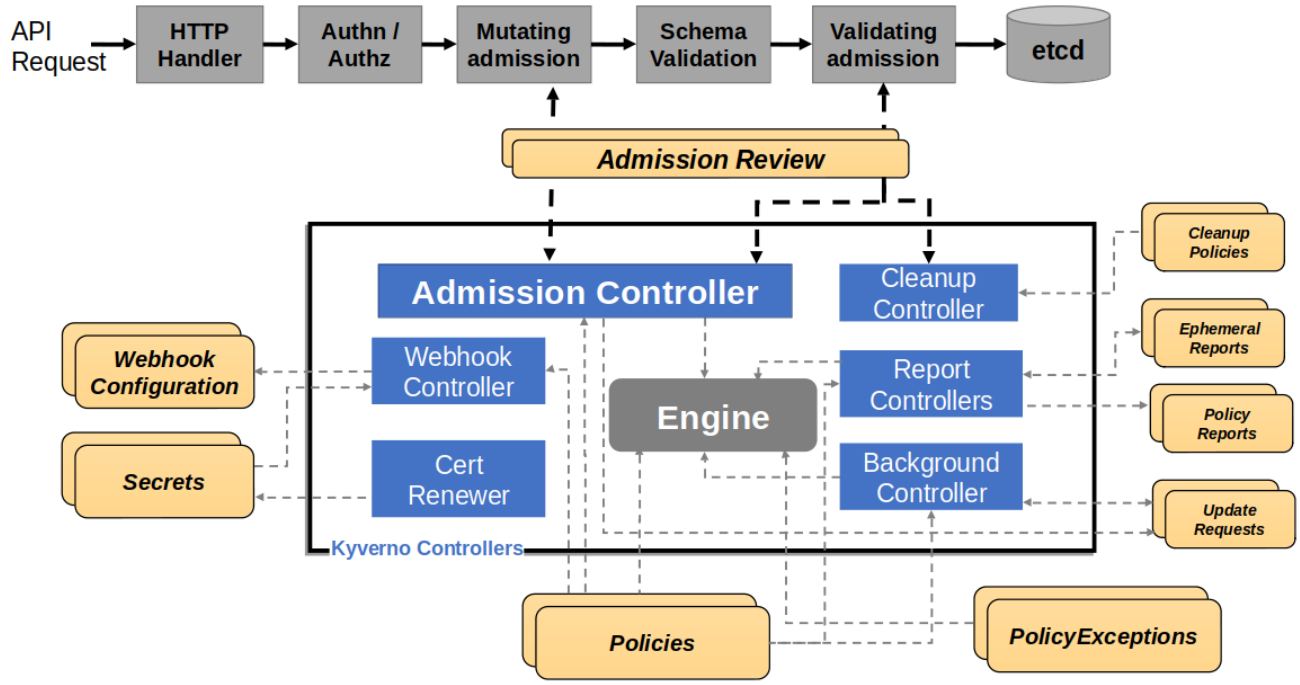
Admission Controls

Kyverno runs as a **dynamic admission controller** in a Kubernetes cluster. Kyverno receives **validating** and **mutating** admission **webhook HTTP callbacks** from the Kubernetes API server.

Mutating policies can be written as overlays (like Kustomize) OR as **JSON patches**. Validating policies als use an overlay style syntax, with support for pattern matching and conditional (if-then-else) processing.

Policy **enforcement** is captured using **Kubernetes** events. For requests that are either allowed or existed prior to introduction of a Kyverno policy then Kyverno creates **Policy Reports** in the cluster which contain a running list of resources matched by a policy, their status and more.

High-level architecture of Kyverno:



The **Webhook** is the server that handles incoming **AdmissionReview** requests from the Kubernetes API server and sends them to the **Engine** for processing.

The **Cert Renewer** is responsible for watching and renewing the certificates, stored as Kubernetes Secrets, needed by the webhook.

The **Background Controller** handles all generate and mutate-existing policies by reconciling `UpdateRequest`, which is a intermediary resource. And the **Report Controller** handle creaion and reconciliation of Policy Reports.

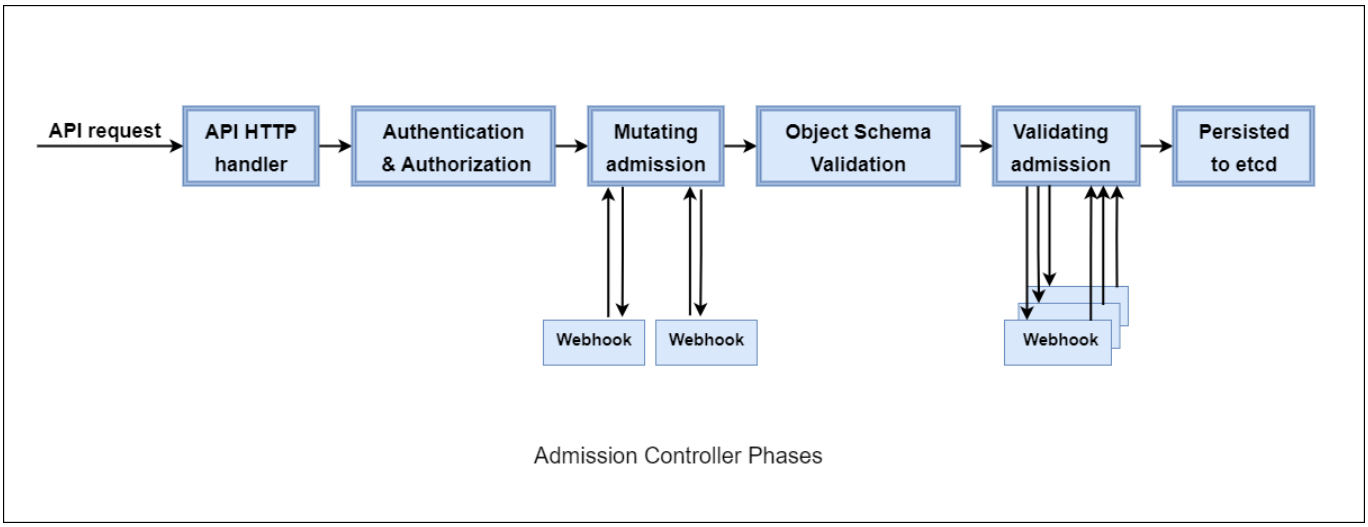
Kyverno supports **high-availability**, which means that controllers that are selected for installation are configured to run multile replicas.

Admission controllers are components responsilbe for:

- Validating
- Modifying

requests as part of the adminssions process. These components can be seen as **extensibility** points for Kubernetes. **Used to control the outcome when new resources are being created**.

Admission controller can be **validating** or **mutating**. A single controller *could* do both. **ALL admissions controller functions AFTER authn and authz** but BEFORE data is persisted to etcd.



Admission Controller Phases

Two types of admission controllers

- `MutatingAdmissionWebhook` - modify
- `ValidatingAdmissionWebhook` - validate whether the request should be allowed to be created or not.

These differ from the built-in admissions controllers, so they are known as **dynamic admission controllers**.

Use cases for dynamic admission controllers:

- Security
- Governance
- Config management
- Fine-grained RBAC
- Multi-tenancy
- Cost control
- Supply chain security

Dynamic Admission Controllers

A DAC are those which are implemented as part of the `MutatingAdmissionWebhook` and `ValidatingAdmissionWebhook` admissions controllers. These contain two parts:

- Webhook
  - What resources to send
  - Where they should be sent
  - What the response behavior should be
- Controller
  - Listens and responds to requests sent to it by **the API server**.

Webhook

Defines a "bridge" between the API server and a seperate piece of software. Can be:

- `ValidatingWebhookConfiguration` - contract governing validations
- `MutatingWebhookConfiguration` - contract governing mutations

Example:

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
  name: kyverno-resource-validating-webhook-cfg
webhooks:
  - name: validate.kyverno.svc-fail  ## The name of this webhook
    rules:                           ## What resources should be sent
      - apiGroups:
          - apps
        apiVersions:
          - v1
        operations:
          - CREATE
        resources:
          - deployments
    clientConfig:                    ## Where the resources should be sent
      caBundle: LS0t<snip>0tLS0K
      service:
        name: kyverno-svc
        namespace: kyverno
        path: /validate/fail
        port: 443
    timeoutSeconds: 10               ## How long should the API server wait
    failurePolicy: Fail              ## What should happen after the wait is
over
```

Controller

The controller is the software that listens on requests sent by the API server. Typically implemented by a controller running in the same cluster as a Pod.

The instruction on how to respond to requsets is provided in the form of a **policy**.

Controllers receiving reqeusts from the Kubernetes API server do so over HTTP/REST, the contents of that request are "packaging" or "wrapping" of the resource. This package is called an `AdmissionReview`.

The API server wraps the matching resource in another resource called an `AdmissionReview` which contains a number of descriptive data about that request:

- What **type** of operation is being performed (CREATE, UPDATE, DELETE, CONNECT)
- The user making the request
- The resource being acted upon (most important!)
- Other metadata

Example:

```
{
    "kind": "AdmissionReview",
```

```
    "apiVersion": "admission.k8s.io/v1",
    "request": {
        "uid": "3d4fc6c1-7906-47d9-b7da-fc2b22353643",
        "kind": {
            "group": "",
            "version": "v1",
            "kind": "Pod"
        },
        "resource": {
            "group": "",
            "version": "v1",
            "resource": "pods"
        },
        "requestKind": {
            "group": "",
            "version": "v1",
            "kind": "Pod"
        },
        "requestResource": {
            "group": "",
            "version": "v1",
            "resource": "pods"
        },
        "name": "mypod",
        "namespace": "foo",
        "operation": "CREATE",
        "userInfo": {
            "username": "thomas",
            "uid": "404d34c4-47ff-4d40-b25b-4ec4197cdf63"
        },
        "object": {
            "kind": "Pod",
            "apiVersion": "v1",
            "metadata": {
                "name": "mypod",
                "creationTimestamp": null
            },
            "spec": {
                "containers": [
                    {
                        "name": "busybox",
                        "image": "busybox",
                        "resources": {}
                    }
                ]
            },
            "status": {}
        },
        "oldObject": null,
        "dryRun": false,
        "options": {
            "kind": "CreateOptions",
            "apiVersion": "meta.k8s.io/v1"
        }
```

```
        }
    }
```

Note the various fields in the request!

Kyverno Policies & Rules

Kyverno can match resources using the resource's fields such as:

- kind
- name
- label
- selectors

Policies are defined using `Custom Resource Definitions` (CRDs) in Kubernetes. Each policy contains one or more rules.

A `ClusterPolicy` contains a list of rules.

Policies with validation rules can be used to block insecure or non-compliant configurations by setting the:

- `failureAction` to `Enforce`

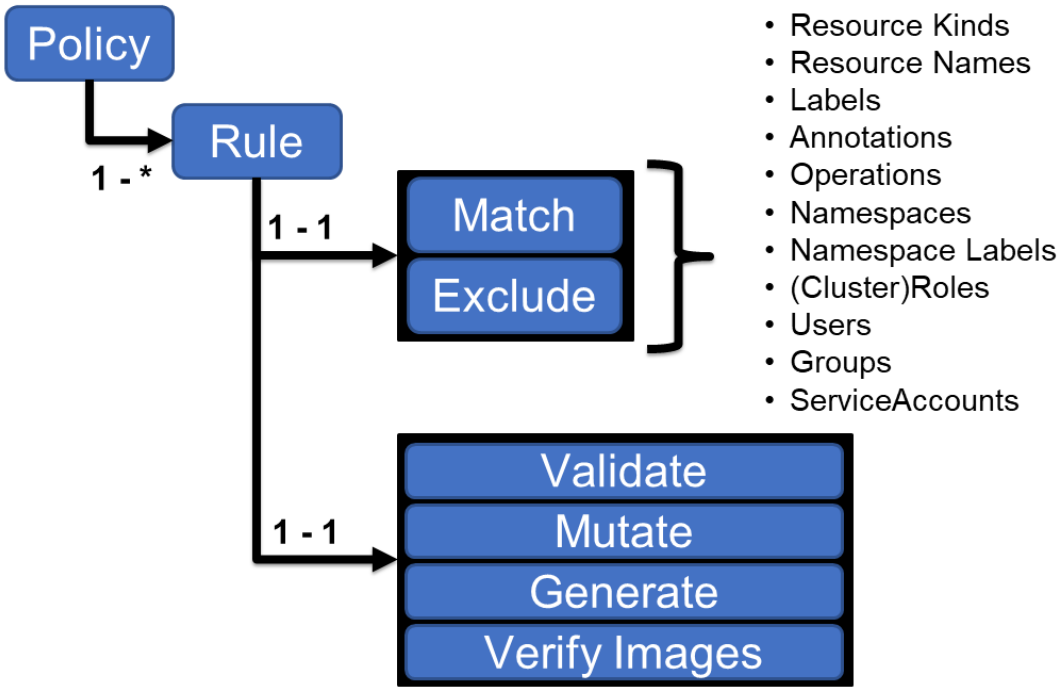
Or validation rules can be applied using periodic scans with results available as `policy reports`.

Rules in a policy are applied in the order of *definition*.

During admission control **mutation rules** are applied **BEFORE** validation rules - this allows validation of changes made during mutation.

Note that **ALL** mutation rules are applied **first** across all policies before any validation rules are applied.

Structure of a policy:



The rules are:

- Each rule consists of a `match` and an optional `exclude` declaration.
- One of the following types:
  - `validate`
  - `mutate`
  - `generate`
  - `verifyImages` Each rule can only have a single child declaration of these types.

Policies can be defined as cluster-wide resources using the `ClusterPolicy` resource or namespaced resource called `Policy`.

Additinal policy types are:

- Policy Exception
- Cleanup Policies

▶ Installation, Configuration and Upgrades (18%)

- Helm-based Installation and Configuration
- Kyverno Custom Resource Definitions (CRDs)
- Controller Configuration with Flags
- Configuring Kyverno RBAC, roles, and permissions
- High Availability Installations
- Upgrading Kyverno

Helm-based Installation and Configuration

The Helm chart is the recommended way to install Kyverno in a Kubernetes cluster, in production.

There's two installation methods:

- Standalone installation with Helm, using default values.
- High-availability installation with Helm, setting primarily the replicas counts to at least 2.
    - admission controller = 3
    - background controller = 2
    - cleanup controller = 2
    - report controller = 2

By default the Kyverno Namespace will be **excluded** using a `namespaceSelector` configured with the immutable label `kubernetes.io/metadata.name`. Additional namespaces can be excluded.

To install a pre-release (RC) you can use `--devel` flag with Helm to install e.g. the latest pre-release version available.

Kyverno Custom Resource Definitions (CRDs)

Category	CRD	Scope	Description
Core Policies	`policies.kyverno.io`	Namespaced	The basic Kyverno policy type. Defines validation, mutation, generation, and other rules scoped to a namespace.
	`clusterpolicies.kyverno.io`	Cluster-wide	Same as above, but applies cluster-wide (not bound to any namespace).
Cleanup Policies	`cleanuppolicies.kyverno.io`	Namespaced	Defines rules for automatically deleting resources in a namespace.
	`clustercleanuppolicies.kyverno.io`	Cluster-wide	Same as above, but applies cluster-wide. Useful for global cleanup of CRDs, nodes, etc.

Category	CRD	Scope	Description
Reporting	`ephemeralreports.reports.kyverno.io`	Namespaced	Temporary reports summarizing policy evaluations on namespaced resources.
	`clusterephemeralreports.reports.kyverno.io`	Cluster-wide	Temporary reports summarizing evaluations of cluster-scoped resources.
Specialized Policy Subtypes	`validatingpolicies.policies.kyverno.io`	Namespaced	Represents the validation rules (like admission control checks).
	`mutatingpolicies.policies.kyverno.io`	Namespaced	Represents mutation logic (e.g., defaulting fields).
	`generatingpolicies.policies.kyverno.io`	Namespaced	Represents resource generation rules (creating new objects from a policy).
	`deletingpolicies.policies.kyverno.io`	Namespaced	Represents cleanup/deletion-type logic.
	`imagevalidatingpolicies.policies.kyverno.io`	Namespaced	Specialized policies for image verification (signatures, attestations).
Policy Exceptions	`policyexceptions.kyverno.io`	Namespaced	Defines which resources or rules are exempt from which policies.

Category	CRD	Scope	Description
	`policyexceptions.policies.kyverno.io`	Cluster-wide	Cluster-level equivalent of above.
Global Context	`globalcontextentries.kyverno.io`	Cluster-wide	Allows adding global, reusable context data for use in CEL or JMESPath expressions in policies.
Internal Processing	`updaterequests.kyverno.io`	Cluster-wide	Internal Kyverno CRD used to track and queue background processing of “generate” rules.

Notes

- 🧩 **You usually define only `Policy` or `ClusterPolicy`.**
  The other CRDs are created and managed automatically by Kyverno.
- 🧹 **Cleanup Policies** let Kyverno automatically delete resources matching criteria.
- 📊 **Ephemeral Reports** store short-lived evaluation results for dashboards or `kyverno-cli`.
- 🔐 **Policy Exceptions** provide per-resource or per-namespace exclusions.
- 🧠 **Global Context** is used to inject reusable context into all policy evaluations.
- ⚙️ **UpdateRequests** are Kyverno’s background job queue for generate/sync rules.

Controller Configuration with Flags

Container flags, the following flags can be used to control the **advanced bhavior** of the various Kyverno controllers and should be set on the main container in the form of arguments:

Reports Controller:

- `aggregationWorkers`, number of internal worker threads.
- `backgroundScan`, enable/disable background scans.
- `policyReports`, enables policy reports system.

Admission Controller:

- `dumpPatches`, toggles debug mode.

Background Controller:

- `genWorkers`, number of generation worker threads, for concurrent processing.

For ABR:

- `enablePolicyException`, set to true to enable PolicyException capability.

Configuring Kyverno RBAC, roles, and permissions

Kyverno creates several Roles, ClusterRoles, RoleBindings and ClusterRoleBindings, some of may need to be customized depending on additional functionality required.

The following roles are installed in the `kyverno` namespace:

- `kyverno:admission-controller`
- `kyverno:background-controller`
- `kyverno:cleanup-controller`
- `kyverno:reports-controller`

Kyverno uses aggregated `ClusterRoles` to search for and combine `ClusterRoles` which apply to Kyverno. Each controller has its own `ClusterRoles`.

Those ending in **core** are the aggregate ClusterRoles which are then aggregated by the top-level role without the **core** suffix.

The Kyverno:

- Admission
- Background
- Reports

controllers have a role binding to the built-in `view` role. This allows these Kyverno controllers view access to most namespaced resources. Thes are possible to customize using:

```
admissionController.rbac.viewRoleName
backgroundController.rbac.viewRoleName
reportsController.rbac.viewRoleName
```

Kyvernos default permissions are designed to cover commonly used and security non-critical resources. To extend a controllers permission, add a **new role** with one or more of the following labels:

admission-controller: `rbac.kyverno.io/aggregate-to-admission-controller: "true"` background-controller: `rbac.kyverno.io/aggregate-to-background-controller: "true"` reports-controller: `rbac.kyverno.io/aggregate-to-reports-controller: "true"` clean-up-controller: `rbac.kyverno.io/aggregate-to-cleanup-controller: "true"`

To **avoid upgrade issues** it is **highly recommended** that default roles are not modified but new roles are used to extend them!

Example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: kyverno:create-deployments
```

```
    labels:
      rbac.kyverno.io/aggregate-to-background-controller: "true"
rules:
- apiGroups:
  - apps
  resources:
  - deployments
  verbs:
  - create
  - update
```

If a Kyverno validate an mutate policies operates on a cumstom resource the **background** and **reports** controllers needs to be provided permissions to manage the resource:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: kyverno:virtualservers:edit
  labels:
    rbac.kyverno.io/aggregate-to-background-controller: "true"
rules:
- apiGroups:
  - cis.f5.com
  resources:
  - virtualservers
  verbs:
  - update
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: kyverno:virtualservers:view
  labels:
    rbac.kyverno.io/aggregate-to-background-controller: "true"
    rbac.kyverno.io/aggregate-to-reports-controller: "true"
rules:
- apiGroups:
  - cis.f5.com
  resources:
  - virtualservers
  verbs:
  - get
  - list
  - watch
```

A ClusterRole binding is required for the Admission controller to grant it to generate the ValidatingAdmissionPolicies:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
```

```
  metadata:
    name: kyverno:generate-validatingadmissionpolicy
    labels:
      rbac.kyverno.io/aggregate-to-admission-controller: "true"
  rules:
  - apiGroups:
    - admissionregistration.k8s.io
    resources:
    - validatingadmissionpolicies
    - validatingadmissionpolicybindings
    verbs:
    - get
    - list
    - watch
    - create
    - update
    - delete
```

High Availability Installations

High-Availability installations of Kyverno involves:

- Set replica count to at least 2 for each controller.
- Its per-controller basis!

Multiple replicas does not mean higher scale or perfoermance across all controllers.

The different capabilities of Kyverno are decoupled into seperate controllers. Each controller runs in it's own **Kubernetes Deployment**.

Admission Controller:

- Receives AdmissionReview requests from the API server, via validation and mutating webhooks.
- Processes validation, mutatation and image validating rules.
- Policy validation
- Process policy exceptions
- Generates EphemeralReport and ClusterEphemeralReport for processing by the Reports Controller.

Reports Controller:

- Responsible for creation and reconciliation of the final PolicyReport and ClusterPolicyReport custom resources.
- Performs background scans and generates, processes and converts EphemeralReports

Background Controller:

- Processes generate and mutate existing rules of Policy or ClusterPolicy.
- Processes policy add, update and delete events.
- Has no relationship to the Reports Controller for background scans.

Cleanup Controller

- Processes CleanupPolicy and DeletingPolicy resources

- Manages and renews certificates as Kubernetes Secrets for use in the webhook
- Performs policy validation for the CleanupPolicy and ClusterCleanupPolicy resources

How HA works in Kyverno

Admission Controller

Always required.

Does not use leader election for inbound webhook requests which means AdmissionReview requests can be distributed across all replicas.

Minimum supported replica count: 3

Extra replicas **can** be used for both availability and scale! Vertical scaling is also possible.

Reports Controller

Responsible for **all** report processing logic.

Stateful, so it requires leader election, regardless of the number of replicas. Only a single replica will handle repots processing at any given time.

Multiple replicas can only be used for avaialbility.

Background Controller

Responsible for handling of generate and mutate-existing rules.

Also stateful and requires leader election.

Multiple replicas configured for the background controller can only be used for availability. Vertical scaling is possible. Also increasing the number of internal workers with `--genWorkers` flag.

Cleanup Controller

Responsible for handling cleanup policies via creation of CronJobs.

It has components which require leader election.

Multiple replicas configured for the clleanup controller can be used for both availability and scale.

Clusters with many concurrent cleanup invocations will see increased throughput when multiple replicas are configured. Vertical scaling is also possible.

CronJobs are created and managed on a 1:1 relationship.

Upgrading Kyverno

Helm and YAML.

Upgrade Steps

1. Backup your existing configuration.
2. Update the Helm repository:

```
helm repo update
```

3. Upgrade Kyverno using Helm:

```
helm upgrade kyverno kyverno/kyverno --namespace kyverno --reuse-values
```

4. Verify the upgrade:

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

▶ Kyverno CLI (12%)

Apply

The `apply` command is used to **perform a dry run** on one or more policies with a given set of input resources. This can be useful to determine a policy effectiveness prior to committing to a cluster. The apply command can show the mutated resources as an output.

The input resources can either be resource manifests or can be taken from a running cluster.

It supports files from:

- URLs both as policies and resources

Examples:

```
kyverno apply /path/to/policy.yaml --resource /path/to/resource.yaml # Dry
run a policy against a resource file
```

Point a dir to run all!

Use `--exceptions-with-resources` to apply PolicyExceptions from the provided resources.

Use `-o <dir|file>` to output mutated resources to a directory or file.

Note that:

A PolicyException is a Namespaced Custom Resource which allows a resource(s) to be allowed past a given policy and rule combination. It can be used to exempt any resource from any Kyverno rule type although it is primarily intended for use with validate rules. A PolicyException encapsulates the familiar match/exclude statements used in Policy and ClusterPolicy resources but adds an exceptions{} object to select the policy and rule name(s) used to form the exception.

Use `-f` or `--values-file` for applying multiple policies to multiple resources while passing a file containing variables and their values. Variables specified can be of various types include AdmissionReview fields, ConfigMap context data, API call context data, and Global Context Entries.

Use `-u` or `--userinfo` for applying policies while passing an optional user_info.yaml file which contains necessary admission request data made during the request.

```
apiVersion: cli.kyverno.io/v1alpha1
kind: Values
metadata:
  name: values
policies:

- name: <policy1 name>
    rules:
  - name: <rule1 name>
      values:
        <context variable1 in policy1 rule1>: <value>
        <context variable2 in policy1 rule1>: <value>
```

If a resource-specific value and a global value have the same variable name, the resource value takes precedence over the global value!

The `apply` command can be used to apply native Kubernetes policies and their corresponding bindings to resources, allowing you to test them locally without a cluster.

Test

The `test` command is used to test a given set of resources against one or more policies to check desired results, declared in advance in a seperate test manifest file.

Useful when you wish to declare what your expected results should be by defining the intent which the assist with locating discrepancies should those results change.

`test` works by **scanning** a given location:

- Git repository
- Local directory

and executing the tests defined within. The rule types:

- validate
- mutate
- generate

are currently supported. The command looks **recursively** for YAML files with policy test declarations and execute those.

In every test there are four desired results which can be tested for:

- Pass
- Skip
- Fail
- Warn

The test declaration consists of the following parts:

- The `policies` element which lists one or more policies to be applied.
- The `resources` element which lists one or more resources to which the policies are applied.
- The `exceptions` element which lists one or more policy exceptions. Cannot be used with ValidatingAdmissionPolicy. Optional.
- The `variables` element which defines a file in which variables and their values are stored for use in the policy test. Optional depending on policy content.
- The `userinfo` element which declares admission request data for subjects and roles. Optional depending on policy content.
- The `results` element which declares the expected results. Depending on the type of rule being tested, this section may vary.
- The `checks` element which declares the assertions to be evaluated against the results (see Working with Assertion Trees).

jp

The Kyverno CLI has a `jp` subcommand which makes it possible to test not only the custom filters endemic to Kyverno but also the full array of capabilities of JSMESPath.

Auto-Gen rules

Kubernetes has many higher-level controllers that directly or indirectly manage Pods, namely the Deployment, DaemonSet, StatefulSet, Job, and CronJob resources. Third-party custom resources may also “wrap” or leverage the Pod template as part of their resources as well.

Kyverno solves this issue by supporting automatic generation of policy rules for higher-level controllers from a rule written exclusively for a Pod. For rules which match on Pods in addition to other kinds, auto-generation is not activated.

▶ Applying Policies (10%)

- Applying Policy in Cluster
- Resource Selection
- Common Policy Settings for Kyverno Rules

Applying Policy in Cluster

You can apply policies in three different ways:

- In Clusters, when installing Kyverno it runs as a dynamic admission controller in a Kubernetes cluster. You can create policies directly with e.g. `kubectl apply -f policy.yaml`. Exceptions to polcies shall be added as `PolicyException` resources. Cleanup policies is another resource used to remove existing resources based upon a definition and schedule.
- In Pipelines, use the `kyverno` CLI to apply polcies to YAML resource manifest files as part of a software delivery pipeline. Allows for integrating Kyverno into GitOps style workflows and checks for policy compliance of resource manifests before they are committed to version control and applied to clusters.
- Via APIs, Kyverno JSON policies and the new ValidatingPolicy and ImageValidatingPolicy types can be applied to *any* JSON payloads. Policies can be applied via a Goland SDK or web service.

Resource Selection

There's a couple of ways to identify and filter resources for rule evaluation. The `match` and `exclude` filter control the **scope** to which rules are applied. They have the **same** structure and can each contain **ONLY ONE** of the two elements:

- `any` - specify filters which are **OR**ed together. ANYORED.
- `all` - specify filters which are **AND**ed together. ALLANDED.

Resource filters

The following resource filters can be specified under an `any` or `all`clause:

- `resources` - select by name, namespaces, kinds
- `subjects` - select users, groups and service accounts
- `roles` - namespaced roles
- `clusterRoles` - cluster-wide roles

At least one element must be specified in a `match.(any|all).resources.kinds` or `exclude` block.

Wildcards are supported in the `resources.kinds` and `subject` fields.

Supported formats:

- Group/Version/Kind
- Version/Kind
- Kind

Wildcard examples in the `kinds` field:

- `Group/*/Kind`
- `Group/*/*`
- `*/Kind`
- `*`

Use a `/` or `.` as a seprator between parent and subresource.

Using parent resources followed by its subresource is **neccessary** to be explicit in the matching decision.

In the AdmissionReview request flow:

validation/mutation webhook -> Check if resource and user information matches OR should be excluded from processing -> Process and apply logic to mutate, validate or generate resources.

Match statements

IN EVERY RULE THERE MUST BE A SINGLE `match` STATEMENT to function as the filter to which the rule will apply!

Example:

```
match:
  any:
  - resources:
      kinds:
      - Service
      names:
      - staging
      operations:
      - CREATE
  - resources:
      kinds:
      - Service
      namespaces:
      - prod
      operations:
      - CREATE
```

This match statement matches all resources that EITHER have the kind Service with name `staging` OR have the kind Service and being created in the `prod` namespace.

The `operations[]` field are optimal but **recommeneded**.

By combining multiple elements in the `match` statement you can be more selective as to which resources you wish to process.

```
match:
  any:
  - resources:
      names:
      - "prod-*"
      - "staging"
      kinds:
      - Service
      operations:
      - CREATE
  - resources:
      kinds:
      - Service
      operations:
      - CREATE
    subjects:
```

```
    - kind: User
      name: dave
```

Here we've filtered out Services that begin with the text `prod-` OR have the name `staging`. The second block matches Servies being created by the `dave` user regardless of the of the Servbice.

With GVK you can:

```
match:
  any:
  - resources:
      kinds:
      - networking.k8s.io/v1/NetworkPolicy

OR

match:
  any:
  - resources:
      kinds:
      - v1/NetworkPolicy
```

Supported formats:

- `*`
- `*pattern*`
- `*pattern`
- `pattern?`
- `patte?rn`

Example:

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: require-labels
spec:
  background: false
  rules:
  - name: check-for-labels
    match:
      any:
      - resources:
          kinds:
          - "*"
          operations:
          - CREATE
    validate:
      failureAction: Audit
      message: "The label `app.kubernetes.io/name` is required."
```

```
      pattern:
        metadata:
          labels:
            app.kubernetes.io/name: "?*"
```

All resources kind are checked for the existence of a label having key `app.kubernetes.io/name` during CREATE operations. The **problem** with this is that it will send all resources to Kyverno for evaluation which can be a performance issue.

You can also use a `namspaceSelector` to select/exclude namespaces based on labels:

```
match:
  any:
  - resources:
      kinds:
      - Pod
      namespaceSelector:
        matchLabels:
          organization: engineering
```

Match a Deployment or StatefulSet with a specific label

```
match:
  any:
  # AND across kinds and namespaceSelector
  - resources:
      # OR inside list of kinds
      kinds:
      - Deployment
      - StatefulSet
      operations:
      - CREATE
      - UPDATE
      selector:
        matchLabels:
          app: critical
```

Uses the following logic: **"AND across types but an OR within list types"**.

Remember this one: *RULES ARE APPLIED IN THE ORDER OF DEFINITION!*

Combining `match` and `exclude`

In some cases where a subset of resources selected in a `match` block need to be omitted from processing, you may optionally use an `exclude` block.

ALL MATCH AND EXCLUDE CONDITIONS MUST BE SATISFIED FOR A RESOURCE TO BE SELECTED FOR THE POLICY RULE!

The default operations for validating resources are:

- CONNECT
- CREATE
- UPDATE
- DELETE

For mutating resources:

- CREATE
- UPDATE

▶ Writing Policies (32%)

- Validation Rules
- Preconditions
- Background Scans
- Mutation Rules
- Generation Rules
- VerifyImage Rules
- Variables & API Calls in Policies
- JSON Patches
- Autogen Rules
- Cleanup Policies
- Common Expression Language (CEL)

Tips and Tricks

- Use `kubectl explain` to explain and explore the various parts and fields of a Kyverno policy
- Use the VS Code OpenAPIV3 integration to get hints and autocompletion when writing policies
- Make use of `kyverno` CLI to test policies out in advance
- When developing your `validate` polcies, it's easiest to set `failureAction` to `Enforce` so when testing you can see the results immediately
- Before deploying to production, change `failureAction` to `Audit` to avoid disruption

Validation Rules

These are probably the most common and the main use case for admission controllers such as Kyverno.

If agreed == resource is created. Otherwise blocked.

These rules responds determined by the `failureAction` field:

- `Enforce` - block.
- `Audit` - allowed and recorded in a policy report.

Validation rules in audit mode can be used to get a report on matching resources which violate the rule(s) both on:

- Initial creation
- During periodic background scans

To validate resource data, define a pattern in the validation rule, for more advanced processing define a `deny` element with conditions that controls when to allow or deny the request.

Failure Action

REMEMBER: *For preexisting resources, which violates a newly created policy, Kyverno will allow subsequent updates to those resources. However, if an update to a resource makes it compliant (allowe), subsequent updates that violates the rule after that will be blocked!*

This beahviour can be disabled using:

- Set `validate.allowExistingViolations` to `false` in an `Enforce` rule.

Tip: Set `spec.emitWarning` to `true` to show audit policy violations in admissions responses.

Failure Action Overrides

Use `failureActionOverrides` to specify which actions to apply per Namespace. Only available in `ClusterPolicy`.

Example:

```
      validate:
        failureAction: Audit
        failureActionOverrides:
          - action: Enforce     # Action to apply
            namespaces:       # List of affected namespaces
              - default
          - action: Audit
            namespaces:
              - test
```

Special overrides for specific Namespaces.

Patterns

Following rules are followed when processing the overlay pattern:

1. Validation will fail if a field is deined in pattern and does not exist in the config.
2. Undefined fields are treated as wildcards
3. A validation pattern field with the wildcard `*` will match zero or more alphanumeric characters. Empty values are matched. Missing fields are not matched.
4. A validation pattern field with the wildcard `?` will match any single alphanumeric character. Empty values are not matched. Missing fields are not matched.
5. A validation pattern field with the wildcard `?*` will match any single alphanumeric character. Empty or missing fields are not matched.
6. A validation pattern field with the value `null` or `""` requires that the field not be defined or has no value.
7. Validation of child values is only performed if the parent matches the pattern.

Anchors

- Conditional `()`
- Equality `=()`
- Existence - `^()` - only list/arrays
- Negation - `X()`
- Global - `<()`

`anyPattern`

In some cases `securityContext` can be defined at the Pod or Container level. anyPattern is a logical OR across multiple patterns.

Do not use negated conditions.

Deny rules

Example:

```
validate:
  message: Main message is here.
  deny:
    conditions:
      any:
      - key: "{{ request.object.data.team }}"
        operator: Equals
        value: eng
        message: The expression team = eng failed.
      - key: "{{ request.object.data.unit }}"
        operator: Equals
        value: green
        message: The expression unit = green failed.
```

Deny rules are more powerful and expressive than simple patterns but are also more **complex** to write, use deny rules WHEN:

- You need advanced selection logic with multiple "if" conditions
- You need access to the full contents or the AdmissionReview
- You need access to more built-in variables
- You need access to the compelte JMESPath filtering system

The expressions looks like the ones used in Kubernetes selectors and `matchExpressions`.

Pod Security

There's a subrule type called `podSecurity`. Which eases the pain of writing and applying PSP profiles.

For example, this policy enforces the latest version of the Pod Security Standards baseline profile in a single rule across the entire cluster:

```

```

```yaml
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: psa
spec:
  background: true
  rules:
  - name: baseline
    match:
      any:
      - resources:
          kinds:
          - Pod
    validate:
      failureAction: Enforce
      podSecurity:
        level: baseline
        version: latest
```

ValidatingAdmissionPolicies

Provides a declarative, in-process option for validating admission webhooks using the CEL to perform validation checks **directly in the Kubernetes API server**.

Designed to perform basic validation checks for an admission request.

Kyverno policies can be used to generate and manage lifecycle of Kubernetes ValidatingAdmissionPolcies. Extends Kyvernos reporting and testing capabilities.

Example:

```yaml
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: disallow-host-path
spec:
  background: false
  rules:
    - name: host-path
      match:
        any:
        - resources:
            kinds:
              - Deployment
      validate:
        failureAction: Enforce
        cel:
          expressions:
            - expression: "!has(object.spec.template.spec.volumes) ||
object.spec.template.spec.volumes.all(volume, !has(volume.hostPath))"
```

```
                message: "HostPath volumes are forbidden. The field
spec.template.spec.volumes[*].hostPath must be unset."
```

Check the `status` object to see if theres a corresponding VAP generated.

Only ClusterPolicy can create these, since VAPs are cluster scoped.

Kyverno JSON Assertion

`assert` is a subrule type that allows users to use Kyverno JSON assertion trees for resource validation.

Example of a policy that ensures that a Pod does not use the default service account:

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: disallow-default-sa
spec:
  validationFailureAction: Enforce
  rules:
  - match:
      any:
      - resources:
          kinds:
          - Pod
    name: disallow-default-sa
    validate:
      message: default ServiceAccount should not be used
      assert:
        object:
          spec:
            (serviceAccountName == 'default'): false
```

Preconditions

Preconditions allows for more fine-grained selection of resources than the options allowed by `match` and `exclude` statements.

Consists of one or more expressions which are evaluated after a resources has been **successfully matched** AND not **excluded by a rule**.

They're powerful since they allow you to access variables, JMESPath filters, operators and other constructs.

When preconditions are **evaluated to and overall TRUE** result processing of the body **begins**.

Similar to deny rules becuase they are built of the same type of expressions and have the same fields.

When used in rule types that **supports reporting** a result will be scored as a `skip` if a resources is matched by a rule but discarded by the combined preconditions.

Any and All Statements

Preconditions are evaluated by **nesting** the expressions under `any` and/or `all` statements. This gives you further power in building more precise logic for how the rule is triggered.

If any of the `any`/`all` statement blocks does not evaluate to TRUE, preconditions will not be satisfied.

Operators

Does not work on arrays:

- Equals
- NotEquals

Most commonly used:

- AnyIn
- AllIn
- AnyNotIn
- AllNotIn

Can be used with Kubernetes resources quantities:

- GreaterThan
- GreaterThanOrEquals
- LessThan
- LessThanOrEquals

Duratons operators can be used for things usch as validating an annotation that is a duration unit:

- DurationGreaterThan
- DurationGreaterThanOrEquals
- DurationLessThan
- DurationLessThanOrEquals

Example:

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: resource-quantities
spec:
  background: false
  rules:
  - name: memory-limit
    match:
      any:
      - resources:
          kinds:
          - Pod
    preconditions:
      any:
      - key: "
{{request.object.spec.containers[0].resources.requests.memory}}"
```

```
        operator: LessThan
        value: 1Gi
```

Wildcard Matches

Wildcard matches are possible!

Background Scans

Periodically reapply policies to exsting resources for reporting.

REMEMBER THAT: Background scans are handled by the **reports controller** and not the **background controller**!

Background scanning are enabled by default in `Policy` and `ClusterPolicy` resources through the `spec.background` field.

Default periodically, one hour by default. Change the `backgroundScanInterval` field for the Reports Controller.

Behavior:

| Setting | | New | Existing | |---|---|---| | `background: true` | `failureAction: Enforce` | Pass only | Report | | `background: true` | `failureAction: Audit` | Report | Report | | `background: false` | `failureAction: Enforce` | Pass only | None | | `background: false` | `failureAction: Audit` | Report | None |

Roles, ClusterRoles, and Subjects in `match` and `exclude` statements, cannot be applied to existing resources in the background scanning!

Mutation Rules

A `mutate` rule can be used to modify matching resources and is written as either:

- RFC 6902 JSON Patch
- Strategic merge patch

By using the JSONPatch RFC 6902 format you can make **precise** change to the resources. Kubernetes will not allow mutations of resource fields such as:

- name
- namespace
- uid
- kind
- apiVersion

REMEMBER: **Mutations occures BEFORE validation!**

JSON 6902 JSON patch

Example of a Json6902 patch mutation rule:

```yaml
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: policy-patch-cm
spec:
  rules:
    - name: pCM1
      match:
        any:
        - resources:
            names:
              - config-game
            kinds:
              - ConfigMap
      mutate:
        patchesJson6902: |-
          - path: "/data/ship.properties"
            op: add
            value: |
              type=starship
              owner=utany.corp
          - path: "/data/newKey1"
            op: add
            value: newValue1
```

REMEMBER: **Mutations using this patching is NOT translated int higher-level Pod controllers**.

Strategic Merge Patch

`kubectl` uses a strategic merge patch when you use the `kubectl patch` command.

Mutation rules written with this style, if the match on Pod, are subject to **auto-generation** rules for Pod controllers!

Anchors

There's three types of anchors supported in mutation overlay rules:

- Conditional: `()` - Use the tag and value as an “if” condition.
- Add if not present: `+()` - Add the tag value if the tag is not already present. Not to be used for arrays/lists unless inside a foreach statement.
- Global: `<()` - Add the pattern when the global anchor is true

Processing flow and combinations:

1. First, all conditial anchors are processed. Stops when returned `false`. Proceeds only if **all** cond anchors return `true`.
2. Next, all tag-values without anchors and all add anchor tags are processed to apply the mutation.

Mutate Existing resources

Can be patched with `patchStrategicMerge` and `patchesJson6902`.

Mutation of existing policies are applied in the background (via the Background Controller).

Two important implications:

1. Mutation for existing resources is an asynchronous process.
2. Custom permissions are almost always required.

REMEMBER: **Mutation of existing Pods is limited to mutable fields only.**

REMEMBER: **If you set `mutateExistingOnPolicyUpdate` to `true`, Kyverno will mutate the existing secret on policy CREATE and UPDATE AdmissionReview events.**

When `name` and/or `namespace` fields are omitted in the `targets` list, **it implies `*`**.

Mutate Rule Ordering (Cascading)

In some cases it might be desired to have multiple levels of mutation rules apply on incoming resources.

Example:

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: database-type-labeling
spec:
  rules:
    - name: assign-type-database
      match:
        any:
        - resources:
            kinds:
            - Pod
      mutate:
        patchStrategicMerge:
          metadata:
            labels:
              type: database
          spec:
            (containers):
            - (image): "*cassandra* | *mongo*"
```

Generation Rules

Create new Kubernetes resources based on a policy and optionally keep them in-sync.

A generate rule can be used to create new Kubernetes resources in response to some other event such as resource:

- Creation
- Update
- Or updating a policy itself

Useful to create supporting resources such as new `RoleBindings` or `NetworkPolicies`.

Common use cases for generate rules:

- Namespace provisioner
- Retroactive creation of NetworkPolicies

Generate rules come in **two** flavors:

1. Apply to admission events that occur across the cluster
2. Apply to existing resources

Supports `match` and `exclude`.

Can keep generated resources in sync to prevent tampering by use of `synchornize` property. When `true` the generated resources is kept in-sync with the source resource.

Kyverno can optionally use `server-side apply` when generating the resource.

Data Source

The **source** of a generated resource may be defined in the Kyverno policy/rule directly.

The `orphanDownStreamOnPolicyDelete` property can be used to preserve generated resources on policy/rule deletion. Used to preserve or delete orphaned resources of generate rules.

Clone Source

When a generate policy should take the source from a resource which already exists in the cluster, a `clone` object is used instead of a `data` object.

Kyverno needs to add labels to the clone source in order to track changes.

VerifyImage Rules

Check container image signatures and attestations for software chain security.

Each rule contains the following common configurations attributes:

- `type` - signature type
- `imageReferences` - image reference pattern to match
- `skipImageReferences` - list of image reference patterns
- `required` - enforce that all matching images are verified
- `mutateDigest`: converts tags to digests for matching images
- `verifyDigest`: enforces that digests are used for matching images
- `repository`: use a different repository for fetching signatures
- `imageRegistryCredentials`: use specific registry credentials for this policy.

Contains a list of attestors to check the attached image signature. Depends on the tool used to sign the image e.g. Cosign.

Attestatations are signed metadata.

The rule mutates matching images to add the image digest when `mutateDigest` is set to `true`.

Enable TTL cache for verified images: `imageVerifyCacheEnabled`

Variables & API Calls in Policies

Defining and using variables in policies from multiple sources!

Variables **makes policies smarter and reusable** by enabling references to data in the policy definition.

Example:

```
- name: mountpath
  variable:
    jmesPath: request.object.metadata.annotations.optional ||
'/custom/string/{{request.object.metadata.annotations.mandatory"}}'
```

Pre-defined variables

Kyverno automatically creates a few usefule variables and makes them available within rules:

1. `serviceAccountName`
2. `serviceAccountNamespace`
3. `request.roles`
4. `request.clusterRoles`
5. `images`

Variables from policy definitions

```
  validate:
    failureAction: Enforce
    message: "Port number for the livenessProbe must be less than that of
the readinessProbe."
    pattern:
      spec:
        ^(containers):
        - livenessProbe:
            tcpSocket:
              port: "$(./../../../readinessProbe/tcpSocket/port)"
```

Escaping Variables

In some cases you wish to write a rule containing a variable for action on by another program or process flow. Use leading `\`, JMESPAth notationcan alsobe escaped usingthe same syntax.

Variables from external data sources

Some policy decisions require access to cluster resources and data managed by other Kubernetes controllers or external applications. For these types of policies, Kyverno allows HTTP calls to the Kubernetes API server

and the use of ConfigMaps.

Evaluation Order

- Rule context
- Rule preconditions
- Rule definitions:
  - Validation patterns
  - Validation deny rules
  - Mutate strategic merge patches (patchesStrategicMerge)
  - Generate resource data definitions
  - verifyImages definitions

Variables are not supported in the `match` and `exclude` elements!

JSON Patches

Autogen Rules

Automatically generate rules for Pod controllers.

Pod are one of the most common object types in Kubernetes and as such are the focus of most types of validation rules. But creation of Pod directly is almost never done as it is considered an anti-pattern.

REMEMBER: Auto-gen rules also cover ReplicaSet and ReplicationControllers. This is important for understanding the scope of auto-generated policies.

Rule auto-generation behavior is controlled by the policy annotation `pod-policies.kyverno.io/autogen-controllers`.

Example:

```
pod-policies.kyverno.io/autogen-controllers=Deployment,Job.
```

Kyverno skips generating Pod controller rules whenever the following `resources` fields/objects are specified in a `match` or `exclude` block:

- names
- selector
- annotations

Exclusion by Metadata

```
rules:
  - name: validate-resources
    match:
      any:
      - resources:
          kinds:
```

```
            - Pod
    exclude:
      any:
      - resources:
          annotations:
            policy.test/require-requests-limits: skip
```

When Kyverno sees these types of fields as mentioned above it skips auto-generation for the rule.

Cleanup Policies

Delete matching resources based on a schedule!

Kyverno has the ability to cleanup existing resources in a cluster in two different ways:

- Declarative policy definition in either a `ClusterPolicy` or `ClusterCleanupPolicy`
- Reserved time-to-live (TTL) label added to a resource

Example:

```
apiVersion: kyverno.io/v2
kind: ClusterCleanupPolicy
metadata:
  name: cleandeploy
spec:
  match:
    any:
      - resources:
          kinds:
            - Deployment
          selector:
            matchLabels:
              canremove: "true"
  conditions:
    any:
      - key: "{{ target.spec.replicas }}"
        operator: LessThan
        value: 2
  schedule: "*/5 * * * *"
```

Common Expression Language (CEL)

`cel` is a subrule type in Kyverno. This subrule type allows users to write CEL expressions for resource validation. This was introduced in Kubernetes originally to write validation rules for CRDs.

It's used in `ValidatingAdmissionPolicies`. Standard `match` and `exclude` processing is available just like with other rules. This subrule type is enabled when a validate rule is written with a `cel` object.

Example:

```yaml
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: check-deployment-replicas
spec:
  background: false
  rules:
    - name: check-deployment-replicas
      match:
        any:
        - resources:
            kinds:
              - Deployment
      validate:
        failureAction: Enforce
        cel:
          expressions:
            - expression: "object.spec.replicas < 4"
              message:  "Deployment spec.replicas must be less than 4."
```

`validate.cel` subrules also supports `autogen` rules for higher-level controllers that directly or indirectly manage Pods: Deployment, DaemonSet, StatefulSet, Job, and CronJob resources.

A policy can define `cel.paramKind`, which outlines the GVK of the parameter resource, and then associate the policy with a specific parameter resource via `cel.paramRef`:

```yaml
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: check-deployment-replicas
spec:
  background: false
  rules:
    - name: check-deployment-replicas
      match:
        any:
        - resources:
            kinds:
              - Deployment
      validate:
        failureAction: Enforce
        cel:
          paramKind:
            apiVersion: rules.example.com/v1
            kind: ReplicaLimit
          paramRef:
            name: "replica-limit"
            parameterNotFoundAction: "Deny"
          expressions:
            - expression: "object.spec.replicas < params.maxReplicas"
```

```
                messageExpression:  "'Deployment spec.replicas must be less
than ' + string(params.maxReplicas)"
```

CEL Preconditions

Preconditions allows for more fine-grained selection of resources than the optiosn allowed by match and exclude statements.

For example, if you wished to apply policy to all Kubernetes Services which were of type NodePort, since neither the match/exclude blocks provide access to fields within a resource's spec, a CEL precondition could be used.

Example:

```
rules:
  - name: validate-nodeport-trafficpolicy
    match:
      any:
      - resources:
          kinds:
            - Service
    celPreconditions:
        - name: check-service-type
          expression: "object.spec.type.matches('NodePort')"
    validate:
      cel:
        expressions:
        - expression: "object.spec.externalTrafficPolicy.matches('Local')"
          message: "All NodePort Services must use an externalTrafficPolicy
of Local."
```

CEL Variables

A variable is a named expression that can be referred later as variables in other expressions.

▶ Policy Management (10%)

- Policy Reports
- PolicyExceptions
- Kyverno Metrics

Policy Reports

Policy reports are CRDs, generated and managed automatically by Kyverno. Contains results of applying matching Kubernetes resource to Kyverno ClusterPolicy or Policy resources.

Generated are created based on two different triggers:

- An admission event, CREATE, UPDATE or DELETE.
- The result of a background scan

Kyverno uses a standard and open format published by the Kubernetes Policy working group which proposes a common policy report format across Kubernetes tools.

```
--enableReporting=validate,mutate,mutateExisting,generate,imageVerify
```

Result logic:

Result:

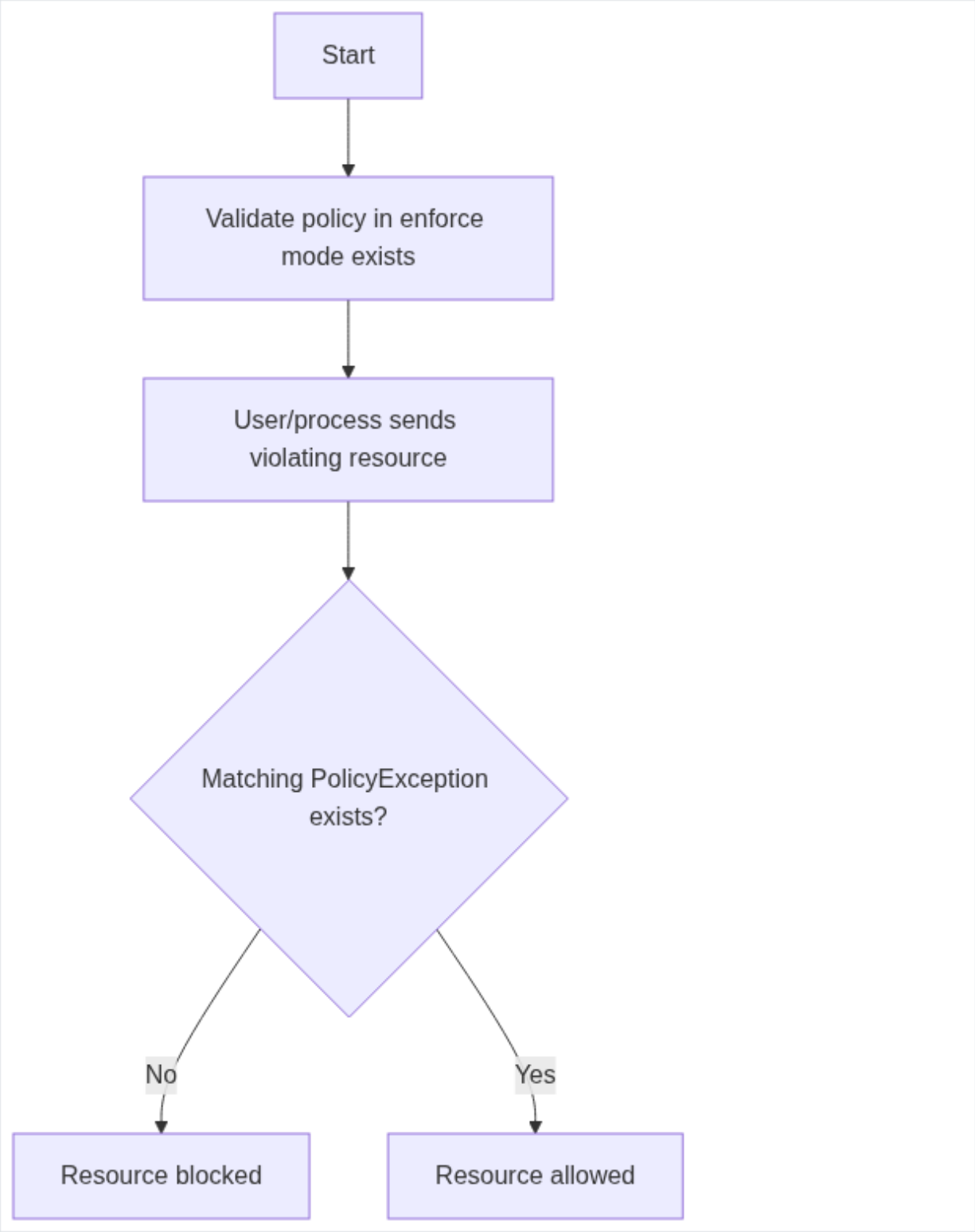
- `pass` - resource was applicable to a rule and the pattern passed evaluation
- `skip` - Preconditions were not satisifed
- `fail` - The resource failed the pattern evaluation
- `warn` - Annotation `scored` has been set to `false`.
- `error` - Variable substitution failed outside of preconditions

Common scenarios resulting in a `skip`:

1. Preconditions Not Met
2. Policy Exceptions
3. Conditional Anchors `()` with Unmet Conditions
4. Global Anchors `<()` with Unmet Conditions
5. Anchor Logic Resulting in Skip

PolicyExceptions

PolicyExceptions are disabled by default. To enable them, set `enablePolicyException` flag to `true`. You also must set the `exceptionNamespace` flag.



Example:

```
apiVersion: kyverno.io/v2
kind: PolicyException
metadata:
  name: delta-exception
  namespace: delta
spec:
  exceptions:
  - policyName: disallow-host-namespaces
    ruleNames:
```

```
      - host-namespaces
      - autogen-host-namespaces
    match:
      any:
      - resources:
          kinds:
          - Pod
          - Deployment
          namespaces:
          - delta
          names:
          - important-tool*
    conditions:
      any:
      - key: "{{ request.object.metadata.labels.app || '' }}"
        operator: Equals
        value: busybox
```

Since these are just another CRD their use can and SHOULD be controlled by a number of different mechanisms:

- Kubernetes RBAC
- Specific Namespace for PolicyExceptions (see Container Flags)
- Existing GitOps governance processes
- Kyverno validate rules
- YAML manifest validation

Kyverno Metrics

Kyverno can expose metrics on port 8000. In Helm that is:

```
<controller>:
  metricsService:
    create: true
```

You can congiure ports and type of service via the same values.

Configuring the metrics

You can tweak which namespaces to exclude from exposure of metrics via `metricsExposure` section in `metricsConfig`.