Solve Real-Time Problem: Enforce Kubernetes Security with Kyverno | RealTime #Kubernetes Project
give me heading- solve real time rpoblem-yes i face real time problem in while working with K8S an di build real time solution to solve this problme by msyself,Enforce Kubernetes Security with Kyverno | RealTime #kubernetes project-watch video od k8s once..he explained well ly ABHISHEK VEERMALLA

Enforce Automated k8s cluster security using kyverno policy generator and argocd

In this project we will learn how to enforce policies, governence and compliance on your kubernetes cluster. Whether your kubernetes cluster is on AWS, Azure, GCP or on-premises, this project will work without any additional changes.

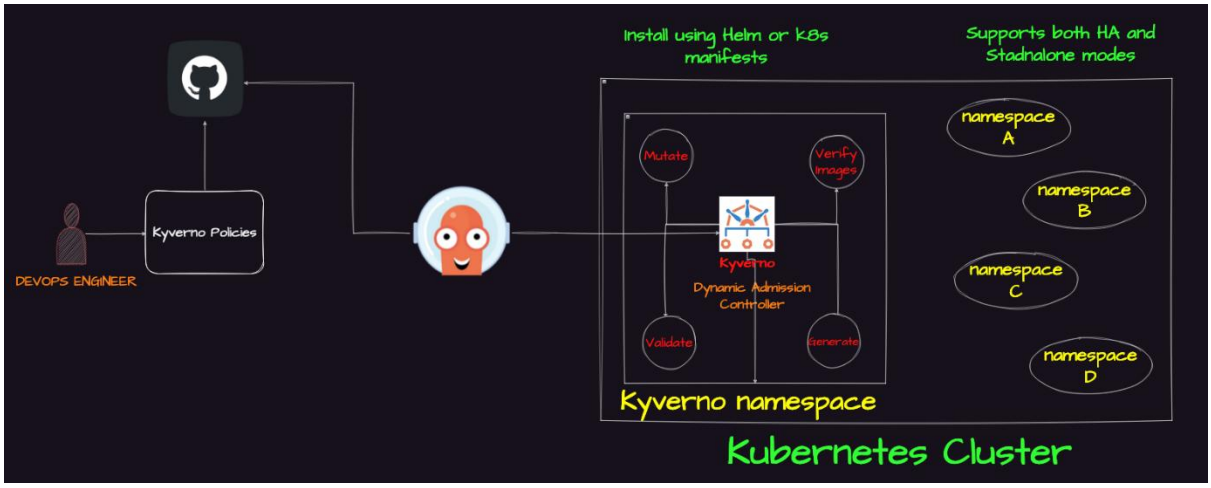
To explain the project with examples, using this configuration you can

1. Generate -> For example, Create a default network policy whenever a namespace is created.
2. Validate -> For example, Block users from using latest tag in the deployment or pod resources.
3. Mutate -> For example, Attach pod security policy for a pod that is created without any pod security policy configuration.
4. Verify Images -> For example, Verify if the Images used in the pod resources are properly signed and verified images.

High Level Design

On a very high level, A DevOps Engineer will write the required Kyverno Policy custom resource and commits it to a Git repository. Argo CD which is pre configured with auto-sync to watch for

resources in the git repo, deploys the Kyverno Policies on to the Kubernetes cluster.



Installation

To setup this project you need to install Argo CD controller and Kyverno controller, Assuming you have Kubernetes installed.

Installation of both Kyverno and Argo CD are pretty straight forward as both of them support Helm charts and also provide a consolidated installation yaml files.

Kyverno

There are two easy ways to install kyverno:

1. Using Helm
2. Using the kubernetes manifest files

Using Helm

Add helm repo for kyverno

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

helm repo update

Install kyverno in HA mode

helm install kyverno kyverno/kyverno -n kyverno --create-namespace --set replicaCount=3

(or)

Install kyverno in Standalone mode

helm install kyverno kyverno/kyverno -n kyverno --create-namespace --set replicaCount=1

Install a specific version of kyverno

helm search repo kyverno -l | head -n 10

helm install kyverno kyverno/kyverno -n kyverno --create-namespace --version 2.6.5

Using Kubernetes manifest yaml files

kubectl create -f https://github.com/kyverno/kyverno/releases/download/v1.8.5/install.yaml

Argo CD

There are three ways to install Argo CD

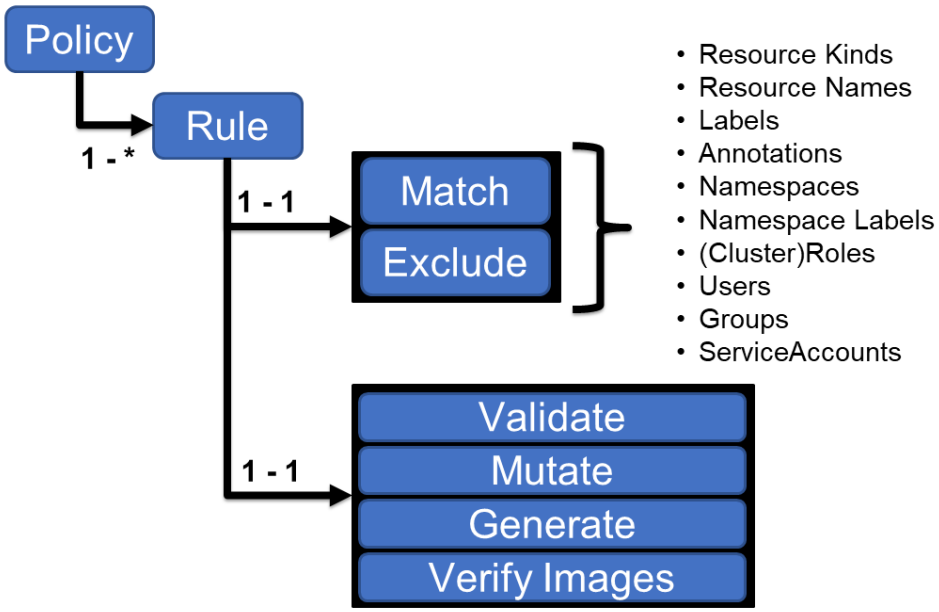
1. kubectl apply -f https://raw.githubusercontent.com/argoproj/argo-cd/master/manifests/install.yaml
2. Helm Charts, Follow the link
3. Using the Argo CD Operator, Follow the link

Demystifying Kyverno & Kyverno Policies

Kyverno is a policy engine designed for Kubernetes

A Kyverno policy is a collection of rules. Each rule consists of a match declaration, an optional exclude declaration, and one of a validate,

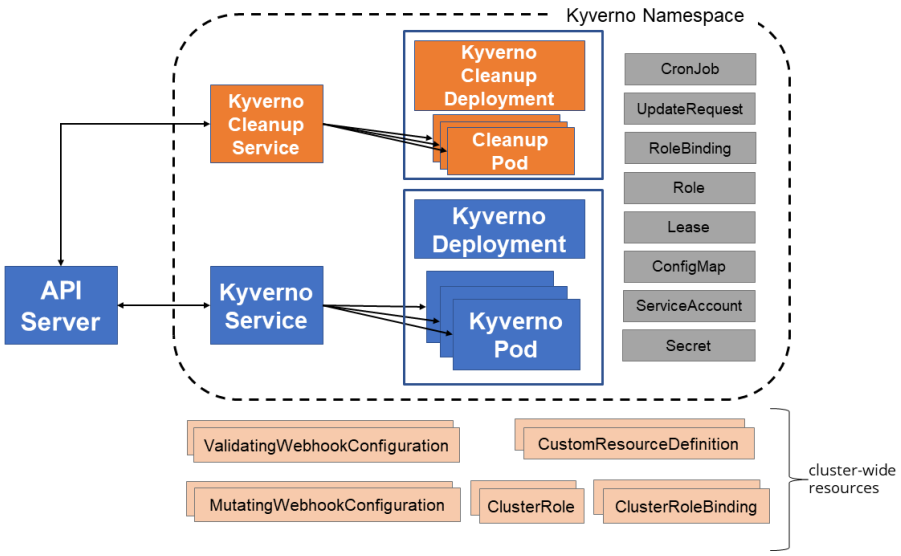
mutate, generate, or verifyImages declaration. Each rule can contain only a single validate, mutate, generate, or verifyImages child declaration.



Policies can be defined as cluster-wide resources (using the kind ClusterPolicy) or namespaced resources (using the kind Policy.) As expected, namespaced policies will only apply to resources within the namespace in which they are defined while cluster-wide policies are applied to matching resources across all namespaces. Otherwise, there is no difference between the two types.

Additional policy types include PolicyException and (Cluster)CleanupPolicy which are separate resources and described further in the documentation.

Architecture



Quick: what is .raw on GitHub?

A **raw** URL returns the file content directly (not the HTML page). Example raw URL pattern:

https://raw.githubusercontent.com/<user>/<repo>/<branch-or-commit>/<path-to-file>

Use it when you want to apply a YAML directly to your cluster:

kubectl apply -f https://raw.githubusercontent.com/<user>/<repo>/<branch>/path/policy.yaml

kubectl downloads the file content from raw.githubusercontent.com and applies it — convenient for demos, but prefer storing policies in Git for real GitOps.

How Kyverno picks up policies when you kubectl apply them

- Kyverno runs **inside your cluster** (in kyverno namespace by default) as a controller + admission webhook.

- When you kubectl apply -f policy.yaml (ClusterPolicy or Policy), Kubernetes stores that resource in etcd. Kyverno watches the API and **loads** the policy automatically.
- If you use GitOps (Argo CD), Argo CD writes the same resource to the cluster — same effect. Both are valid; GitOps gives versioning / code review.

Admission flow (simple)

1. You run kubectl create / kubectl apply.
2. API server receives the request and sends it to configured admission webhooks (Kyverno validating/mutating webhooks).
3. Kyverno evaluates matching ClusterPolicy/Policy rules for that resource:
    - mutate may change the object (add defaults)
    - validate may allow or deny
    - generate may create additional resources
    - verifyImages may check image signatures
4. Kyverno returns Admit / Deny (or allow + mutate).
5. If Deny → API server returns an error and the resource is not persisted (not created).
   If Audit mode → resource is created but Kyverno logs/records the violation.

Why your pod was NOT created (short)

If a Kyverno policy is validationFailureAction: enforce, and the incoming Pod/Deployment **does not match** the validate rule, Kyverno's validating webhook **denies** the request. kubectl shows an error like:

Error from server: admission webhook "validate.kyverno.svc" denied the request: <policy-name> failed: <reason>

So the pod/Deployment is never persisted to etcd and never scheduled.

---

validationFailureAction: audit vs enforce

- audit (lowercase): **log & report violations**, but **allow** the resource (useful for discovery & gradual rollout).
- enforce (lowercase): **block** non-compliant creations/updates at admission time.

Note: Kyverno requires the lowercase strings audit or enforce. Audit (capital A) will error (as you saw).

Annotations — what they are & how they're used

- Annotations = arbitrary key/value metadata on K8s objects (not used for selection).
- Useful for: policy exceptions, storing SBOM identifiers, tooling metadata.
- Kyverno policies can match or exclude resources based on annotations, e.g., skip policy for an object annotated policy-exception: "true".

Step-by-step from scratch — practical minikube demo (audit → enforce)

Run the following (copy / paste). I'll show both audit and enforce modes and how to observe results.

0. Start minikube

minikube start --driver=dockerkubectl config use-context minikube

1. Install Kyverno (using official manifest)

kubectl create ns kyvernokubectl apply -f https://github.com/kyverno/kyverno/releases/latest/download/kyverno.yaml# wait for kyverno pods to be ready:kubectl -n kyverno wait --for=condition=ready pod --all --timeout=120skubectl get pods -n kyverno

2. Create a ClusterPolicy that *requires* cpu/memory requests & limits

File: require-requests-limits-audit.yaml (audit mode)

apiVersion: kyverno.io/v1kind: ClusterPolicymetadata: name: require-requests-limitsspec: validationFailureAction: audit # <- audit vs enforce background: true rules: - name: validate-requests-limits match: resources: kinds: - Pod - Deployment - StatefulSet - DaemonSet validate: message: "Each container must have cpu/memory requests and limits." anyPattern: - spec: template: spec: containers: - resources: requests: cpu: "?*" memory: "?*" limits: cpu: "?*" memory: "?*"

Apply it:

kubectl apply -f require-requests-limits-audit.yamlkubectl get clusterpolicy require-requests-limits -o yaml

3. Test — Audit behavior (policy *logs* violations but allows resources)

Create a **bad** deployment (no resources):

kubectl create ns demo-audit || truekubectl -n demo-audit create deploy bad-deploy --image=nginx --replicas=1# check resource existskubectl -n demo-audit get deploy bad-deploy

Expected: the Deployment is CREATED because the policy is audit. Kyverno will detect the violation and record it.

Where to see the violation:

- Kyverno controller logs:

kubectl -n kyverno logs deploy/kyverno --tail=200 | grep -i require-requests-limits -C3

- Kubernetes events (look for Kyverno messages):

kubectl get events -n demo-audit --sort-by=.metadata.creationTimestamp | tail -n 20kubectl get events --all-namespaces | grep -i require-requests-limits || true

You should see Kyverno emitting an event or log entry describing the violation (message text from message: in the policy).

---

4. Switch to enforce (block non-compliant resources)

Edit the policy file and change:

spec: validationFailureAction: enforce

Or patch it live:

kubectl patch clusterpolicy require-requests-limits --type='merge' -p '{"spec":{"validationFailureAction":"enforce"}}'

Now try to create another bad deployment:

kubectl -n demo-audit delete deploy bad-deploy --ignore-not-foundkubectl -n demo-audit create deploy blocked-deploy --image=nginx

Expected: the second kubectl create fails with an admission error from Kyverno:

Error from server: admission webhook "validate.kyverno.svc" denied the request: require-requests-limits failed: Each container must have cpu/memory requests and limits.

Confirm it was not created:

kubectl -n demo-audit get deploy blocked-deploy # should not exist

Check Kyverno logs to see the deny reason:

kubectl -n kyverno logs deploy/kyverno --tail=200 | grep -i "deny" -C3

Example: how a real deny looks to the user

When Kyverno *enforces*, kubectl apply or kubectl create will return an error containing:

- the policy name
- the rule name
- the message you provided in the policy

Example (exact text varies but similar):

Error from server: admission webhook "validate.kyverno.svc" denied the request: require-requests-limits validate-requests-limits: Each container must have cpu/memory requests and limits.

Namespaces: Kyverno scope

- ClusterPolicy (cluster-wide) applies across namespaces.
- Policy (namespaced) applies only to that namespace.
- Kyverno itself runs in the kyverno namespace — policies are stored in the cluster and Kyverno watches them.

If you want to **exclude** a namespace from policies, you can write an exclude section in the policy, e.g.:

match: resources: kinds: ["Pod"]exclude: resources: namespaces: ["kube-system", "kyverno", "argocd"]

Example: exclude by annotation (policy exception)

You can write a match/exclude that checks annotations. Example: skip if policy-exception: "true":

exclude: resources: selector: annotations: policy-exception: "true"

Then patch a resource:

metadata: annotations: policy-exception: "true"

Kyverno will skip that resource (only if your policy has that exclude clause).

JSON version (same policy) — example snippet

(You can convert YAML to JSON; Kyverno accepts either format when you kubectl apply.)

{ "apiVersion": "kyverno.io/v1", "kind": "ClusterPolicy", "metadata": { "name": "require-requests-limits" }, "spec": { "validationFailureAction": "audit", "background": true, "rules": [{

"name": "validate-requests-limits", "match": { "resources": { "kinds": ["Pod","Deployment","StatefulSet","DaemonSet"] } }, "validate": { "message": "Each container must have cpu/memory requests and limits.", "anyPattern": [{ "spec": { "template": { "spec": { "containers": [{ "resources": { "requests": { "cpu": "?*", "memory": "?*" }, "limits": { "cpu": "?*", "memory": "?*" } } }] } } } }] } }] }}

Troubleshooting tips

- If Kyverno *doesn't* enforce:
    - kubectl get pods -n kyverno → ensure Kyverno pods are ready.
    - kubectl get validatingwebhookconfigurations → Kyverno validating webhook should be present.
    - kubectl logs -n kyverno deploy/kyverno → look for errors.
- If policy does not match resource shape: check match.resources.kinds and the YAML structure the policy expects (Deployment vs Pod spec shapes differ).
- Use kyverno test (kyverno CLI) locally to simulate policy evaluation (optional CLI).
- background: true enables background checks on existing resources (audit of live cluster).

Summary / Practical takeaway

- .raw lets kubectl apply straight from GitHub raw URLs.

- Kyverno is **dynamic admission controller**: you don't write custom webhook code for each rule — write a policy in YAML/JSON.
- audit = detect & report but allow; enforce = block non-compliant requests. Use audit first to discover, then flip to enforce.
- Policies are applied cluster-wide when using ClusterPolicy, and Kyverno loads them immediately when you kubectl apply or when Argo CD syncs them.
- Use minikube demo above to see the difference in practice.

Want me to:

- produce the full require-requests-limits-audit.yaml + require-requests-limits-enforce.yaml files (ready to download),

1. Why not write admission controllers for every rule?

- Kubernetes supports **Admission Controllers** to enforce compliance at the API server level.
- But if you want to enforce multiple org-specific policies (like disallow :latest, enforce network policies, verify images, etc.), you would have to **write and maintain custom admission controllers in Go**.
- That's heavy: coding, updating with new K8s versions, testing, etc.

- **Kyverno solves this**: it's a **dynamic admission controller** → you just write YAML policies (Validate / Mutate / Generate / Verify Images). No coding needed.

---

2. Governance & Compliance with Kyverno

- You define policies once → Kyverno enforces them cluster-wide.
- Kyverno sits as a **dynamic admission controller** and intercepts resource requests before they're persisted to etcd.
- Compliance becomes **policy as code** → versioned in Git, synced with ArgoCD, auditable.

Examples of governance/compliance rules:

- All Pods must have resource requests/limits.
- All containers must not run as root.
- Only signed images can run.
- Every namespace must have a NetworkPolicy.

Instead of custom admission controllers, you write these as **YAML Kyverno ClusterPolicies**.

---

3. DAC vs AC (Discretionary vs Admission Controllers)

- **DAC (Discretionary Access Control)** → Based on **RBAC**: who (user/serviceaccount) is allowed to perform what action (get/list/create/delete).
  Example: Dev cannot delete prod namespace.

- **AC (Admission Controller)** → Even if RBAC allows creation, admission controllers can **inspect the object** being created and block/modify it.
  Example: Dev has rights to create Pods, but admission controller (Kyverno) blocks if the Pod uses :latest tag.

So RBAC controls **who can act**, admission controllers control **what they can create/update**.

---

4. What does "Annotation" mean?

- **Annotation = metadata key/value pair on a K8s object.**
- Unlike labels (used for selection), annotations are used for **storing extra info** (tooling, governance, policy exceptions).
- Example:

metadata: annotations: owner: "team-frontend" policy-exception: "approved-by-security"

Kyverno can **match/exclude based on annotations**.

---

5. Example Kyverno JSON/YAML Policies (from best practices repo)

Let's see one in JSON (since you asked):

Block :latest image tags

{ "apiVersion": "kyverno.io/v1", "kind": "ClusterPolicy", "metadata": { "name": "disallow-latest-tag" }, "spec": { "validationFailureAction": "enforce", "rules": [{ "name": "block-latest-tag", "match": { "resources": { "kinds": ["Pod"] } }, "validate": { "message": "Using 'latest' tag is not allowed.", "pattern": { "spec": { "containers": [{ "image": "!*:latest" }] } } } }] }}

---

6. Governance Workflow with Kyverno + ArgoCD

17. Security/DevOps team writes policies (Validate, Mutate, etc.) as YAML/JSON.

18. Push to Git → Argo CD auto-syncs → policies installed in cluster.

19. Every Pod/Deployment creation request is **checked against these policies**.

20. If violation → request denied (compliance enforced).

21. Audit → Kyverno reports compliance status, so you prove governance.

✅ In short:

- **Kyverno** = "admission controller as a service" → you don't need to code one yourself.
- **DAC (RBAC)** = controls *who can act*.
- **Admission controllers (Kyverno)** = controls *what can be created/updated*.
- **Annotations** = metadata to drive governance decisions or exceptions.