

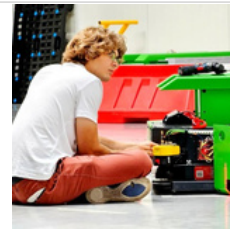
# Enterprise Kubernetes EKS Security: Automated mTLS with Linkerd, cert-manager, and Vault. Comparing OPA and Kyverno for Policy Enforcement

Check GitHub for helpful DevOps tools:

## Michael Robotics

Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats information overload by adhering to the set of principles: simplify, prioritize, and execute.

 <https://github.com/MichaelRobotics>



Ask Personal AI Document assistant to learn interactively (FASTER)!

1

Download PDF

2

Go to website

3

Browse file

4

Chat with Document

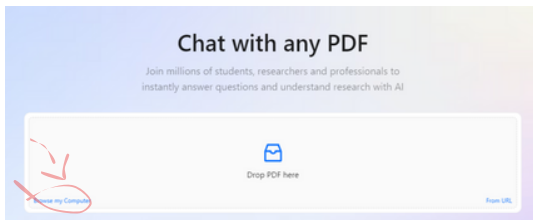
1

<https://github.com/MichaelRobotics/DevOpsTools/blob/main/KubernetesSecurity.pdf>

2

[Click there to go to ChatPdf website](#)

3

The image shows the ChatPdf website interface. It has a light blue header with the text "Chat with any PDF" and a sub-header "Join millions of students, researchers and professionals to instantly answer questions and understand research with AI". Below this is a large white box with a red arrow pointing to a "Drop PDF here" button. At the bottom right of the white box, it says "From URL".

4

Ask questions about document!

# Completly new to Linux and Networking?

Essential for this PDF is a thorough knowledge of networking. I highly recommend the HTB platform's networking module, which offers extensive information to help build a comprehensive understanding.

HTB - Your Cyber Performance Center

We provide a human-first platform creating and maintaining high performing cybersecurity individuals and organizations.

 <https://www.hackthebox.com/>



## What is Kubernetes?

Kubernetes is an open-source platform that automates the deployment, scaling, and management of containerized applications. It helps manage clusters of nodes running containers, ensuring efficient and reliable operation.

## How Kubernetes clusters are made?

Kubernetes clusters consist of a control plane and multiple worker nodes. The control plane manages cluster operations, while worker nodes run the actual container workloads.

# Why and When use Kubernetes

Kubernetes is ideal for deploying scalable, resilient, and automated containerized applications. It is used when managing multiple containers across different environments is necessary.

Example: Running a microservices-based e-commerce platform that scales up during peak hours.

## System Requirements

- RAM: 2 GB per node (1 GB can work for testing but may lead to limited performance)
- 10 GB free storage
- Ubuntu

## Kubernetes: Main components & packages

- **kube-apiserver:** Central management component that exposes the Kubernetes API; acts as the front-end for the cluster.
- **etcd:** Distributed key-value store for storing all cluster data, ensuring data consistency across nodes.
- **kube-scheduler:** Assigns pods to available nodes based on resource requirements and policies.
- **kube-controller-manager:** Manages core controllers that handle various functions like node status, replication, and endpoints.
- **kubelet:** Agent that runs on each node, responsible for managing pods and their containers.
- **kube-proxy:** Manages networking on each node, ensuring communication between pods and services within the cluster.

# Kubernetes EKS: TLS vs mTLS

## 1) TLS

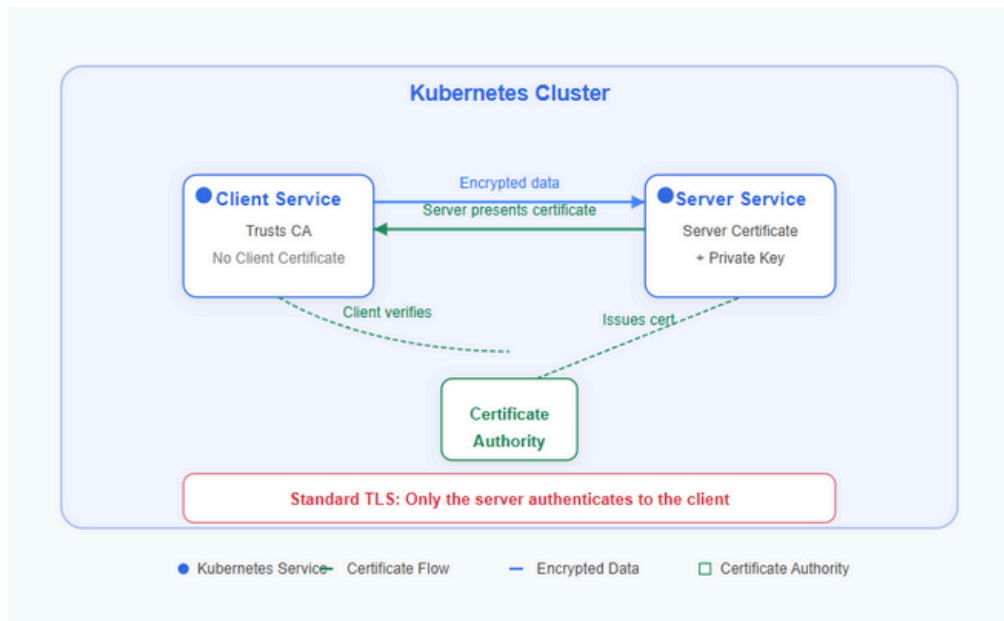
Standard TLS in Kubernetes involves one-way authentication where only the server proves its identity to the client. Key Components of Standard TLS are:

**Server Authentication Only:** The server presents its certificate to the client, but the client doesn't need to authenticate itself with a certificate.

**Certificate Authority (CA):** Issues and signs certificates for servers only.

### TLS Handshake Process:

- Client service initiates connection to server service
- Server presents its certificate
- Client verifies the server's certificate against trusted CA
- After server verification, encrypted communication begins



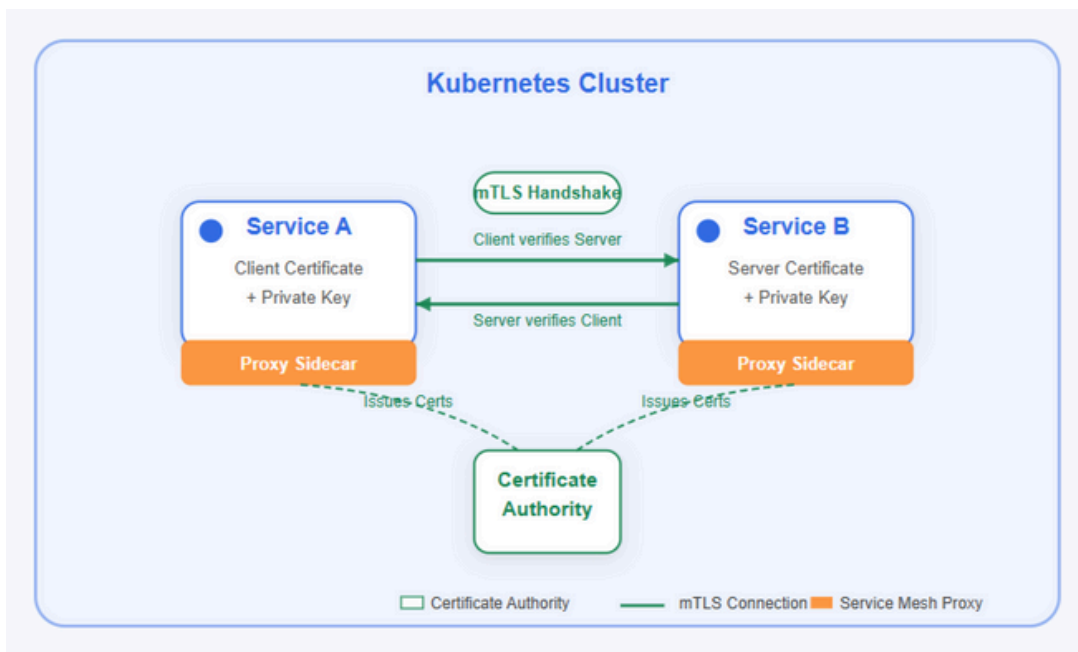
## 2) mTLS

Mutual TLS (mTLS) extends traditional TLS by requiring both the client and server to authenticate each other with certificates, creating a secure bidirectional trust relationship.

In mTLS, both parties must present valid certificates, ensuring that:

1. The connection is encrypted in both directions
2. Both the client and server have verified identities
3. Unauthorized services cannot communicate within the cluster

mTLS implementation with service meshes protects against on-path attacks, spoofing, and credential stuffing through mutual authentication. The only vulnerability—a compromised pod in a meshed namespace—can be mitigated with Kubernetes network policies that restrict communications based on identity.



# Kubernetes EKS: Why mTLS with Linkerd, cert-manager and Vault

Rather than struggling with manual certificate management or relying solely on cert-manager, combining Linkerd, cert-manager and HashiCorp Vault creates a fully automated security system that eliminates human error while maintaining strong security boundaries.

## Advantages Over Using cert-manager Alone

Linkerd provides transparent mTLS for all services while maintaining proper security isolation with trust anchor keys safely confined to the cert-manager namespace and short-lived 24-hour workload certificates that dramatically reduce security risks.

## Benefits of Adding HashiCorp Vault

HashiCorp Vault enhances your security posture by connecting to existing organizational certificate authorities, supporting HSM integration with sophisticated access controls, and providing comprehensive audit trails to meet regulatory compliance requirements.

## Why The Combined Approach Works Best

This integrated solution creates a secure, automated certificate hierarchy with Vault protecting the root CA, cert-manager handling intermediate certificates, and Linkerd managing workload certificates—all with proper security boundaries.

This approach delivers enterprise-grade security with minimal operational overhead, letting teams focus on applications rather than certificate management.

# Kubernetes EKS: Implement mTLS with Linkerd, cert-manager and Vault

Suggested mTLS PKI infrastructure consists of four interconnected components:

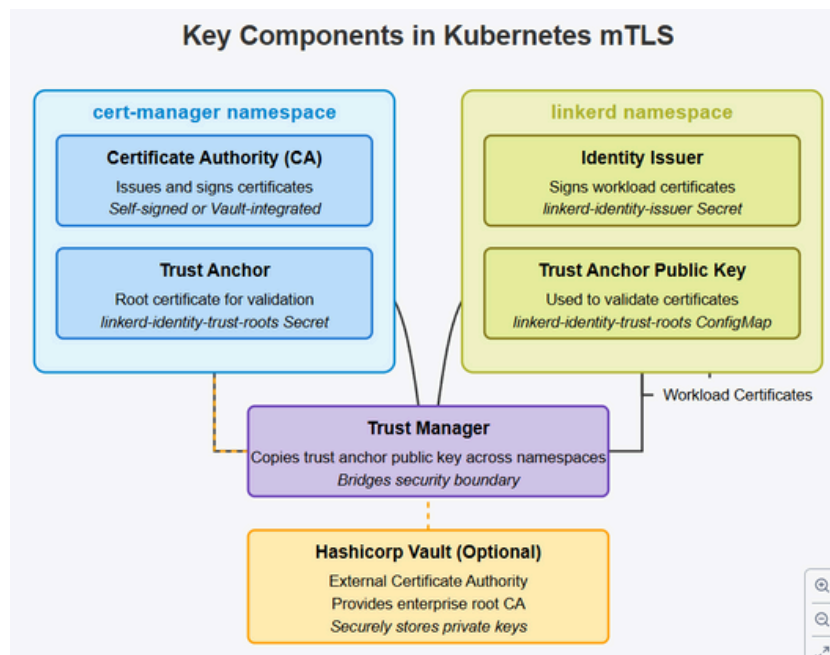
**Certificate Authority (CA):** Issues and signs digital certificates, either self-signed or integrated with your corporate CA via Vault.

**Trust Anchor:** The root certificate establishing the chain of trust, securely stored in the cert-manager namespace.

**Identity Issuer:** Signs all workload certificates, residing in the linkerd namespace for service mesh access.

**Trust Manager:** Copies only the trust anchor's public key between namespaces, maintaining security boundaries.

**How it works together:** The CA signs the Trust Anchor, which remains isolated in the cert-manager namespace with only its public key exposed to Linkerd via Trust Manager. The Identity Issuer, signed by the Trust Anchor, operates in the Linkerd namespace to authenticate and sign individual workloads. These workloads authenticate the Issuer against the Anchor's public key. This creates a robust chain of trust with proper separation between certificate management and service mesh components.



# Kubernetes EKS: Implement mTLS with Linkerd, cert-manager and Vault

## 1) Configure EKS infrastructure

Download repo

```
git clone https://github.com/MichaelRobotics/Kubernetes.git
cd Kubernetes/mTLS/eks-install
```

Install backend. Tweak s3 bucket name. I assume you have aws CLI installed and configured.

```
cd backend
terraform init
terraform plan
terraform apply
```

Deploy your EKS cluster by configuring the cluster name, adjusting node count, or selecting optimal instance types based on your specific performance and scaling requirements.

```
cd ..
terraform init
terraform plan
terraform apply
```

connect to eks

```
aws eks update-kubeconfig --name your-cluster-name --region your-aws-region
```



## 2) Install cert-manager and trust-manager

Switch repos:

```
cd ../cert-manager-workshop
```

add cert-manager and linkerd repos

```
helm repo add linkerd https://helm.linkerd.io/stable  
helm repo add jetstack https://charts.jetstack.io  
helm repo update
```

Install cert manager in cert-manager namespace

```
helm install cert-manager jetstack/cert-manager \\  
--namespace cert-manager --create-namespace \\  
--set installCRDs=true --version v1.14.4 \\  
--wait
```

install trust manager in cert-manager namespace

```
helm upgrade --install cert-manager-trust jetstack/cert-manager-trust \\  
--namespace cert-manager \\  
--wait
```

create linkerd namespace

```
kubectl create ns linkerd
```

Now you can choose to store your root CA in cert-manager or hashicorp vault.

## 2.1) Configure root CA in cert manager (demo solution, self-signed cert)

set up the self-signed root CA

```
kubectl apply -f manifests/cert-manager-root-ca.yaml
```

tell `cert-manager` how to create a trust anchor for Linkerd:

```
kubectl apply -f manifests/cert-manager-trust-anchor.yaml
```

At this point, `cert-manager` should have created the trust anchor for us:

```
kubectl describe secret -n cert-manager linkerd-identity-trust-roots
```

## 2.2) Configure root CA in hashicorp vault (Enterprise solution)

Install Hashicorp Vault using Helm

```
helm upgrade vault vault \
--install \
--create-namespace \
--wait \
--namespace vault \
--repo https://helm.releases.hashicorp.com \
--values manifests/hashicorp-vault.values.yaml
```

Mount the PKI engine at `pki/`

```
kubectl exec -n vault pods/vault-0 -- \
```

Allow certificates valid for up to 10 years:

```
kubectl exec -n vault pods/vault-0 -- \
```

Create a root certificate:

```
kubectl exec -n vault pods/vault-0 -- \
  vault write pki/root/generate/internal \
    common_name="Example Corp Root" \
    organization="Example Corp" \
    country="US" \
    issuer_name="root-2023" \
    ttl=87600h
```

Store it in a Kubernetes Secret:

```
kubectcl create secret generic vault-token \  
  --namespace=cert-manager \  
  --from-literal=token=root-token-1234
```

Create a ClusterIssuer, configured to connect to Hashicorp Vault:

```
kubectcl apply -f manifests/cert-manager-root-ca-vault.yaml
```

if you previously tried with demo self-signed cert solution, patch the Certificate resource 'linkerd-trust-anchor' to use the new Vault ClusterIssuer:

```
kubectcl patch certificate linkerd-trust-anchor \  
  --namespace cert-manager \  
  --type merge \  
  --patch '{"spec":{"issuerRef":{"name":"linkerd-vault-issuer"}}}'
```

If you skipped the self-signed certificate step and proceeded directly with Vault CA solution, manually update the secretName field in cert-manager-trust-anchor.yaml to "linkerd-vault-issuer" instead and apply changes

```
---  
# Given the trust anchor certificate, we can defined a second  
# that will use the Linkerd trust anchor secret to sign Linkerd  
# certificates.  
apiVersion: cert-manager.io/v1  
kind: ClusterIssuer  
metadata:  
  name: linkerd-trust-anchor  
spec:  
  # Rather than just saying "selfSigned", we use the "ca" issuer  
  # cert-manager that this issuer will use a CA certificate stored  
  # Secret.  
  ca:  
    secretName: linkerd-identity-trust-roots
```

This will cause the `linkerd-identity-trust-roots` Secret to be updated with a new TLS certificate, which in turn will cause `trust-manager` to copy the new `tls.crt` file to the `linkerd-identity-trust-roots` ConfigMap.

You can see this for your self by decoding the `ca-bundle.crt` file, as follows:

```
kubectl get configmap linkerd-identity-trust-roots \
  --namespace linkerd \
  --output jsonpath='{.data.ca-bundle.crt}'
```

### 3) Set up the identity issuer certificate

We have a trust anchor certificate and a corresponding CA, so we can tell `cert-manager` how to create the identity issuer certificate as well. A very important note here: the identity issuer certificate must be created in the `linkerd` namespace, so that Linkerd can use it to issue workload certificates.

create the identity issuer for linkerd:

```
kubectl apply -f manifests/cert-manager-identity-issuer.yaml
```

Linkerd pods obtain certificates from the identity issuer. For enhanced security, communication is verified through a two-layer validation process: first checking compatibility with the issuer certificate, then validating the issuer certificate against the trust anchor's public key. This ensures the integrity of the certificate chain and protects against issuer compromise. Copy trust anchor public key to linkerd namespace

```
kubectl apply -f manifests/trust-manager-ca-bundle.yaml
```

## 4) Install linkerd

Install linkerd crds

```
helm install linkerd-crds linkerd/linkerd-crds --namespace linkerd
linkerd upgrade --crds | kubectl apply --filename -
```

install the control plane with proper identity issuer and external CA

```
helm install linkerd-control-plane linkerd/linkerd-control-plane \
--namespace linkerd \
--set identity.externalCA=true \
--set identity.issuer.scheme=kubernetes.io/tls
```

install linkerd CLI

```
curl -sL https://run.linkerd.io/install | sh
```

export cli binaries

```
export PATH=$PATH:/home/laptopdev/.linkerd2/bin
```

check if linkerd is installed properly

linkerd check

```
linkerd-extension-checks
-----
✓ namespace configuration for extensions

Status check results are ✓
laptopdev@laptopdev2:~/Kubernetes/mTLS/cert-manager-workshop$ linkerd
```

# Kubernetes EKS: Test mTLS with Linkerd, cert-manager and Vault

Service mesh is deployed, cluster secured with mTLS. Linkerd have functionality to automatically deploy observability stack with grafana&prometheus

```
linkerd viz install | kubectl apply --filename -
```

You will have access to Grafana dashboard with pre-configured Linkerd metrics, showing things like:

- Request success rates
- Request volumes
- Latency statistics
- Resource utilization for the mesh and its workloads

Now its time to deploy demo app and test if mTLS works. Go to:

```
cd mtls-demo/
```

create namespace

```
kubectl apply --filename namespace.yaml
```

create app

```
kubectl --namespace demo apply --kustomize kustomize/base
```

```
kubectl --namespace demo run other-app --image alpine \
  --restart Never --rm --stdin --tty -- sh
```

```
apk add -U curl
```

```
curl "http://silly-demo:8080"
```

Output should be:

```
(9/9) Installing curl (8.12.1-r10)
Executing busybox-1.37.0-r12.trigger
OK: 12 MiB in 24 packages
/ # curl "http://silly-demo:8080"
This is a silly demo version 1.1.3
/ #
```

```
exit
```

The application is functioning properly. Now we'll enhance it by injecting Linkerd sidecar proxies to implement service mesh capabilities with mutual TLS (mTLS) encryption.

```
cat namespace-meshed.yaml
```

```
apiVersion: v1
kind: Namespace
metadata:
  annotations:
    linkerd.io/inject: enabled
  name: demo
```

delete demo before applying injection configuration:

```
kubectl --namespace demo delete --kustomize kustomize/base
```



apply injection

```
kubectl apply --filename namespace-meshed.yaml
```

create app

```
kubectl --namespace demo apply --kustomize kustomize/base
```

check if app deployed with success:

```
kubectl --namespace demo get pods
```

check if mTLS is working

```
linkerd viz edges pod --namespace demo
```

This command uses the Linkerd Viz extension to show the connections (or "edges") between pods in the "demo" namespace.

```
No edges found.
laptopdev@laptopdev2:~/Kubernetes/mTLS/cert-manager-workshop/mtls-demo$ kubectl --namespace demo get pods
NAME                                READY   STATUS    RESTARTS   AGE
silly-demo-7b5cbd95b9-82l78         2/2     Running   0           18s
laptopdev@laptopdev2:~/Kubernetes/mTLS/cert-manager-workshop/mtls-demo$ linkerd viz edges pod --namespace demo
SRC                                DST                                SRC NS    DST NS    SECURED
prometheus-85b856985f-w65x6        silly-demo-7b5cbd95b9-82l78      linkerd-viz  demo      ✓
laptopdev@laptopdev2:~/Kubernetes/mTLS/cert-manager-workshop/mtls-demo$ linkerd viz edges pod --namespace demo
```

There is connection between prometheus and pod which makes sense, since prometheus collects metrics data. Look at SECURED parameter, it have nice green check. It means, that connection is secured with TLS. Now create another pod:

```
kubectl --namespace demo run other-app --image alpine \
-- sleep 100000
```

```
linkerd viz edges pod --namespace demo
```

```
laptopdev@laptopdev2:~/Kubernetes/mTLS/cert-manager-workshop/mTLS-demo$ linkerd viz edges pod --namespace demo
SRC          DST          SRC NS      DST NS      SECURED
prometheus-85b856985f-w65x6 silly-demo-7b5cbd95b9-82l78 linkerd-viz demo      ✓
laptopdev@laptopdev2:~/Kubernetes/mTLS/cert-manager-workshop/mTLS-demo$ linkerd viz edges pod --namespace demo
SRC          DST          SRC NS      DST NS      SECURED
prometheus-85b856985f-w65x6 other-app    linkerd-viz demo      ✓
prometheus-85b856985f-w65x6 silly-demo-7b5cbd95b9-82l78 linkerd-viz demo      ✓
```

Everything works as expected! Now check mTLS with another command:

```
linkerd viz tap pod --namespace demo
```

This command uses the Linkerd Viz extension to perform real-time traffic monitoring

Navigate to another tab and exec into pod:

```
kubectl --namespace demo exec --stdin --tty other-app \
--container other-app -- sh
```

Then use curl to generate some traffic:

```
apk add -U curl
curl "http://silly-demo:8080"
```

```
laptopdev@laptopdev2:~/Kubernetes/mTLS/cert-manager-workshop/mTLS-demo$ linkerd viz tap pod --namespace demo
req id=0:0 proxy=in  src=10.0.3.59:44956 dst=10.0.3.137:8080 tls=no tls from remote :method=GET :authority=10.0.3.137:8080 :path=/ :status=200 latency=685µs
rsp id=0:1 proxy=in  src=10.0.3.59:44962 dst=10.0.3.137:8080 tls=no tls from remote :method=GET :authority=10.0.3.137:8080 :path=/ :status=200 latency=1294µs
end id=0:0 proxy=in  src=10.0.3.59:44956 dst=10.0.3.137:8080 tls=no tls from remote :duration=25µs response-length=35B
req id=0:0 proxy=in  src=10.0.3.59:44956 dst=10.0.3.137:8080 tls=no tls from remote :method=GET :authority=silly-demo:8080 :path=/ :status=200 latency=491µs
rsp id=0:1 proxy=in  src=10.0.3.59:44962 dst=10.0.3.137:8080 tls=no tls from remote :method=GET :authority=silly-demo:8080 :path=/ :status=200 latency=38µs
end id=0:0 proxy=in  src=10.0.3.59:44956 dst=10.0.3.137:8080 tls=no tls from remote :duration=15µs response-length=35B
req id=0:2 proxy=in  src=10.0.2.18:40276 dst=10.0.3.137:8080 tls=true :method=GET :authority=silly-demo:8080 :path=/ :status=200 latency=2074µs
rsp id=0:0 proxy=in  src=10.0.2.18:40276 dst=10.0.3.137:8080 tls=true :method=GET :authority=silly-demo:8080 :path=/ :status=200 latency=35B
end id=0:0 proxy=in  src=10.0.2.18:40276 dst=10.0.3.137:8080 tls=true :duration=31µs response-length=35B
req id=0:2 proxy=in  src=10.0.2.18:51088 dst=10.0.3.137:8080 tls=true :method=GET :authority=silly-demo:8080 :path=/ :status=200 latency=491µs
rsp id=0:0 proxy=in  src=10.0.2.18:51088 dst=10.0.3.137:8080 tls=true :method=GET :authority=silly-demo:8080 :path=/ :status=200 latency=38µs
end id=0:2 proxy=in  src=10.0.2.18:51088 dst=10.0.3.137:8080 tls=true :duration=38µs response-length=35B
```

Great! You can see packets related to GET request from silly demo with tls parameter check as “true”. That means, connection was secured! Other traffic is health check related.

# Kubernetes EKS: OPA vs Kyverno

These are the two leading policy management tools in the Kubernetes ecosystem. This guide demonstrates their implementation using your EKS cluster.

```
git clone https://github.com/MichaelRobotics/Kubernetes.git
cd Kubernetes/gatekeeper-vs-kyverno-demo
```

## 1) Policy Libraries

Both tools offer predefined policies, but with significant differences. Gatekeeper (OPA) provides an extensive library of reusable constraint templates through its library repository.

### Template

```
apiVersion: templates.gatekeeper.sh/v1
kind: ConstraintTemplate
metadata:
  name: k8sblocknodeport
  annotations:
    metadata.gatekeeper.sh/title: "Block NodePort"
    metadata.gatekeeper.sh/version: 1.0.0
  description: >-
    Disallows all Services with type NodePort.
    https://kubernetes.io/docs/concepts/services-networking/service/#nodeport
spec:
  crd:
    spec:
      names:
        kind: K8sBlockNodePort
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8sblocknodeport

        violation[{"msg": msg}] {
          input.review.kind.kind == "Service"
          input.review.object.spec.type == "NodePort"
          msg := "User is not allowed to create service of type NodePort"
        }
```

This template creates a Custom Resource Definition that serves as the foundation for our policy. We must first install this CRD in our cluster before applying any policies based on it.

The corresponding policy implementation looks like this:

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sBlockNodePort
metadata:
  name: block-node-port
spec:
  match:
    scope: Namespaced
    kinds:
      - apiGroups: [""]
        kinds: ["Service"]
    namespaces:
      - prod-gatekeeper
```

With Kyverno, the process is more streamlined - you simply apply the policy directly without needing to install prerequisite CRDs.

```
apiVersion: kyverno.io/v1
kind: Policy
metadata:
  name: restrict-nodeport
  annotations:
    policies.kyverno.io/description: >-
      A Kubernetes service of type NodePort uses a host port to receive traffic from
      any source. A 'NetworkPolicy' resource cannot be used to control traffic to host ports.
      Although 'NodePort' services can be useful, their use must be limited to services
      with additional upstream security checks.
  namespace: prod-kyverno
spec:
  validationFailureAction: enforce
  rules:
    - name: block-node-port
      match:
        resources:
          kinds:
            - Service
      validate:
        message: "Services of type NodePort are not allowed."
        pattern:
          spec:
            type: "!NodePort"
```

Let's start by setting up OPA Gatekeeper. First create gatekeeper namespace:

```
kubectl create namespace prod-gatekeeper
```

Install OPA using yaml on remote repo:

```
kubectl apply -f https://raw.githubusercontent.com/open-policy-agent/gatekeeper/release-3.15/deploy/gatekeeper.yaml
```

install all crds used to generate OPA policies

```
kubectl kustomize github.com/open-policy-agent/gatekeeper-library/library | kubectl apply -f -
```

```
laptopdev@laptopdev2:~/gatekeeper-vs-kyverno-demo$ kubectl kustomize github.com/open-policy-agent/gatekeeper-library/library
constrainttemplate.templates.gatekeeper.sh/k8sallowedrepos created
constrainttemplate.templates.gatekeeper.sh/k8sallowedreposv2 created
constrainttemplate.templates.gatekeeper.sh/k8sblockendpoiniteditdefault created
constrainttemplate.templates.gatekeeper.sh/k8sblockloadbalancer created
constrainttemplate.templates.gatekeeper.sh/k8sblocknodeport created
constrainttemplate.templates.gatekeeper.sh/k8sblockwildcardingress created
constrainttemplate.templates.gatekeeper.sh/k8scontainerephemeralstoragelimit created
constrainttemplate.templates.gatekeeper.sh/k8scontainerlimits created
constrainttemplate.templates.gatekeeper.sh/k8scontainerratios created
constrainttemplate.templates.gatekeeper.sh/k8scontainerrequests created
constrainttemplate.templates.gatekeeper.sh/k8sdisallowanonymous created
constrainttemplate.templates.gatekeeper.sh/k8sdisallowedrepos created
constrainttemplate.templates.gatekeeper.sh/k8sdisallowedtags created
constrainttemplate.templates.gatekeeper.sh/k8sdisallowinteractivetty created
constrainttemplate.templates.gatekeeper.sh/k8sexternalips created
constrainttemplate.templates.gatekeeper.sh/k8shorizontalpodautoscaler created
constrainttemplate.templates.gatekeeper.sh/k8shttpsonly created
constrainttemplate.templates.gatekeeper.sh/k8simage digests created
```

Now apply policies from gatekeeper folder

```
kubectl apply --filename gatekeeper
```

```
laptopdev@laptopdev2:~/gatekeeper-vs-kyverno-demo$ kubectl apply --filename gatekeeper
k8sblocknodeport.constraints.gatekeeper.sh/block-node-port created
k8scontainerlimits.constraints.gatekeeper.sh/container-must-have-limits created
k8sdisallowedtags.constraints.gatekeeper.sh/image-not-be-latest created
```

clear policies we dont need them now

```
kubectl delete --filename gatekeeper
```

Lets manage same process in kyverno. First install kyverno

```
kubectl create namespace prod-kyverno
```

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

```
helm repo update
```

```
helm install kyverno kyverno/kyverno --namespace kyverno --create-namespace
```

## Apply Kyverno Policies

```
kubectl apply --filename kyverno
```

```
laptopdev@laptopdev2:~/Kubernetes/gatekeeper-vs-kyverno-demo$ kubectl apply --filename kyverno
Warning: Validation failure actions enforce/audit are deprecated, use Enforce/Audit instead.
policy.kyverno.io/restrict-nodeport created
policy.kyverno.io/require-requests-limits created
policy.kyverno.io/disallow-latest-tag created
```

With Kyverno, you simply install the controller and apply policies directly. OPA Gatekeeper requires an extra step of creating constraint templates before applying policies. While this makes Kyverno more straightforward to manage, OPA offers extensive pre-defined templates that can be advantageous when implementing standard governance requirements but only in cases when Kyverno doesn't have those in their smaller library.

clear policies, we don't need them now

```
kubectl delete --filename kyverno
```

## 3) Writing policies

Let's begin with Gatekeeper's approach to blocking NodePort services. Creating effective policies requires understanding both the underlying CRD structure and the Rego policy language, which forms the foundation of Gatekeeper's enforcement capabilities.

```
rego: |
  package k8sblocknodeport

  violation[{"msg": msg}] {
    input.review.kind.kind == "Service"
    input.review.object.spec.type == "NodePort"
    msg := "User is not allowed to create service of type NodePort"
  }
```

This CRD creates a policy preventing NodePort services and returns violation messages. While this example is simple, complex requirements demand much more sophisticated Rego code.

```

targets:
- target: admission.k8s.gatekeeper.sh
  rego: |
    package k8scontainerlimits

    import data.lib.exempt_container.is_exempt

    missing(obj, field) = true {
      not obj[field]
    }

    missing(obj, field) = true {
      obj[field] == ""
    }

    canonifyv cou(orig) = new {

```

That means, you need to learn rego language what will take some time. In case of Kyverno, policies are defined with yamls

```

resources:
  kinds:
    - Service
  validate:
    message: "Services of type NodePort are not allowed."
    pattern:
      spec:
        type: "!NodePort"

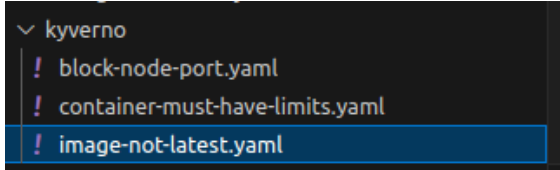
```

Policy logic is defined under 'pattern' block. That means its simpler but on the other hand, we have much less controll over how specific our policies should be

Kyverno excels in straightforward scenarios and typically satisfies most cluster governance requirements. OPA is often excessive, requiring its specialized language that can interact with policies beyond Kubernetes. For most compliance needs that aren't highly complex, Kyverno is the more practical choice.

## 4) Enforcing Policies

Let's test Kyverno's enforcement with a deployment that deliberately violates all three policies: using NodePort services, deploying 'latest' image tags, and omitting container resource limits.



apply those policies

```
kubectl apply --filename kyverno
```

Apply simple deployment

```
kubectl --namespace prod-kyverno \  
  apply --filename app.yaml
```

```
laptopdev@laptopdev2:~/Kubernetes/gatekeeper-vs-kyverno-demo$ kubectl --namespace prod-kyverno \  
  apply --filename app.yaml  
ingress.networking.k8s.io/devops-toolkit created  
Error from server: error when creating "app.yaml": admission webhook "validate.kyverno.svc-fail" denied the request:  
  
resource Deployment/prod-kyverno/devops-toolkit was blocked due to the following policies  
  
disallow-latest-tag:  
  autogen-validate-image-tag: 'validation error: Using a mutable image tag e.g. ''latest''  
    is not allowed. rule autogen-validate-image-tag failed at path /spec/template/spec/containers/0/image/'  
require-requests-limits:  
  autogen-required-resources: 'validation error: CPU and memory resource requests  
    and limits are required. rule autogen-required-resources failed at path /spec/template/spec/containers/0/resources/limits/'  
  
Error from server: error when creating "app.yaml": admission webhook "validate.kyverno.svc-fail" denied the request:  
  
resource Service/prod-kyverno/devops-toolkit was blocked due to the following policies  
  
restrict-nodeport:  
  block-node-port: 'validation error: Services of type NodePort are not allowed. rule  
    block-node-port failed at path /spec/type/'
```

It works! Kyverno showed all violations and prevented deployment creation

Now test OPA. Delete kyverno policies

```
kubectl delete --filename kyverno
```



create OPA policies

```
kubectl apply --filename gatekeeper
```

create deployment in gatekeeper-prod namespace

```
kubectl --namespace prod-gatekeeper \  
  apply --filename app.yaml
```

```
laptopdev@laptopdev2:~/Kubernetes/gatekeeper-vs-kyverno-demo$ kubectl --namespace prod-gatekeeper \  
  apply --filename app.yaml  
deployment.apps/devops-toolkit created  
ingress.networking.k8s.io/devops-toolkit created  
Error from server (Forbidden): error when creating "app.yaml": admission webhook "validation.gatekeeper.  
d to create service of type NodePort
```

OPA only reports the NodePort policy violation, missing the other issues with our deployment.

Let's examine the failed deployment details to understand what actually happened and why all violations weren't displayed.

```
kubectl --namespace prod-gatekeeper \  
  describe replicaset \  
  --selector app=devops-toolkit
```

```
Conditions:
  Type             Status  Reason
  ----             -
  ReplicaFailure   True    FailedCreate
Events:
  Type    Reason      Age    From          Message
  ----    -
  Warning FailedCreate 41s (x15 over 2m4s) replicaset-controller Error creating: admission webhook "validation.gatekeeper.sh" denied the <devops-toolkit> container <devops-toolkit> has no resource limits
  Warning FailedCreate 41s (x15 over 2m4s) replicaset-controller Error creating: admission webhook "validation.gatekeeper.sh" denied the <devops-toolkit> container <devops-toolkit> uses a disallowed tag <vfarctic/devops-toolkit-series:latest>; disallowed tags are ["latest"]
```

The deployment failed for multiple reasons, yet OPA only reported the NodePort violation. This demonstrates OPA's limitation in effectively identifying all policy violations at once, as it lacks Kyverno's deeper understanding of Kubernetes workload relationships.

## 5) Summary

### **Choose Kyverno when:**

- You need to implement policies quickly
- You have straightforward validation requirements
- Clear error reporting is important
- You want to minimize the learning curve

### **Choose OPA Gatekeeper when:**

- You need complex, highly conditional policy logic
- Your governance requirements extend beyond basic validation
- You're already using OPA elsewhere in your stack
- Your team has Rego experience or time to learn it
- You need maximum flexibility in policy definition

# common troubleshooting

## 1) Linkerd mTLS Certificate Issues

**Cause:** Trust anchor certificate expiration or misconfigured cert-manager issuer.

**Solution:** Verify certificate status with `linkerd check --proxy` and renew expired certificates using cert-manager. Ensure Vault issuer is properly connected by checking `kubectl get clusterissuers`.

## 2) Policy Enforcement Failures

**Cause:** Misconfigured OPA/Kyverno policies or incorrect namespaces.

**Solution:** Check policy status with `kubectl get constraints` (for OPA) or `kubectl get policy -A` (for Kyverno). Review admission controller logs to identify specific validation failures.

## 3) EKS Identity Management Errors

**Cause:** IAM role permissions insufficient for service account integration.

**Solution:** Verify IRSA (IAM Roles for Service Accounts) configuration with `aws eks describe-addon` and correct IAM policies to ensure proper access for cert-manager and Vault components.

## 4) Vault Integration Problems

**Cause:** Incorrect Vault authentication or unauthorized service accounts.

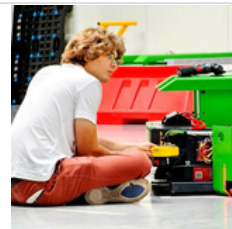
**Solution:** Validate Kubernetes auth method in Vault with `vault auth list`. Ensure service account has proper binding to Vault roles with `kubectl get clusterrolebinding`.

## 5) Check my Kubernetes Troubleshooting series:

Michael Robotics

Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats skill information overload by adhering to the set of principles: simplify, prioritize, and execute.

<https://github.com/MichaelRobotics>




## Learn more about Kubernetes

**Check Kubernetes and piyushsachdeva - great docs!**

Setup a Multi Node Kubernetes Cluster

kubeadm is a tool to bootstrap the Kubernetes cluster

 <https://github.com/piyushsachdeva/CKA-2024/tree/main/Resources/Day27>



Kubernetes Documentation

This section lists the different ways to set up and run Kubernetes

 <https://kubernetes.io/docs/setup/>



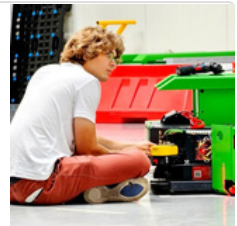
**Share, comment, DM and check GitHub for scripts & playbooks created to automate process.**

**Check my GitHub**

Michael Robotics

Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats skill information overload by adhering to the set of principles: simplify, prioritize, and execute.

<https://github.com/MichaelRobotics>



*PS.*

*If you need a playbook or bash script to manage KVM on a specific Linux distribution, feel free to ask me in the comments or send a direct message!*