TATA COMMUNICATIONS

TATA

# HOW TO PROTECT KUBERNETES CLUSTERS WITH GATEKEEPER POLICIES

*Shashank Sharma and Winnerson Kharsunai*


**Shashank Sharma**
Principal

Shashank Sharma has been following cloud native OSS for more than five years, with a particular focus on security, observability and serverless, and other platform engineering trends. At Tata Communications, he is working as a lead for Kubernetes and PaaS services.
*https://www.linkedin.com/in/shashank-sharma-ft9/*

**Winnerson Kharsunai**
Lead

Lead Engineer at Tata Communications with more than four and a half years of expertise in designing, developing, and delivering high-performance software solutions. Currently, driving cloud-native transformation by integrating cutting-edge tools and technologies to provide secure and enhanced user experience.
*https://linkedin.com/in/winnerson*


vayu | Cloud Platform Services

# TABLE OF CONTENTS

## 1. SECURITY VECTORS IN KUBERNETES

Like in any software, there are a number of ways a false actor can perform harmful activities in a Kubernetes cluster, and it can be difficult to find tools that can safeguard your cluster against all of them. To understand these security vectors in general, a good place to start is reading Mandiant reports (or M-Trends) published every year. We believe that some of the issues reported can directly be translated into some broad security categories from the perspective of a Kubernetes cluster and these are:

- Static validation policies
- Software supply chain security
- Runtime policies
- Compliance scanners

**Static validation policies:** These policies are enforced in a Kubernetes cluster to make sure that all the native resources or custom resources follow the configuration standard set by the cluster administrator or the service provider. One example could be to enforce a policy that makes sure that a Pod can't run in the cluster if it's using host path as volume.

There are many OSS tools that allow this way of policy enforcement, such as Gatekeeper, Kyverno, Kubewarden and more! These tools also have libraries of policies which can be applied in a Kubernetes cluster without much hassle.

**Software supply chain security:** In a Kubernetes cluster, software supply chain security comes into play when we talk about the OCI image that's going to run as a container – hardened images like distroless or images that maintain zero CVEs (like chainguard images), signed images with SBOM, whitelisted registries, schema validations.

**Runtime policies:** While the above two vectors address the prevention of shipping bad configurations and code, there is some security to be enforced at runtime – what happens if the bad configuration and code is already running in the Kubernetes cluster, how do you make sure that the problem is fixed without redeploying or getting downtime? Some tools that help address these problems are kubearmor, tetragon and more.

**Compliance scanners:** Apart from the policies, there is an emerging need for a Kubernetes cluster to follow certain compliances either set by the community or in many places, even the government. Some compliance examples are CSI, CISA and more. You can also create your own environment-based compliances and scan Kubernetes clusters against them. There are some tools like Kube-Bench, Trivy that help addressing the compliance scanning space. In addition to compliance checks, you can also use tools like Kube-Hunter to perform penetration testing and generate reports based on the findings.

In this blog, we are going to talk in detail about the static validation policies, be sure to follow along, and we will talk about security vectors in future blogs!

## 2. OPA GATEKEEPER

A great tool to work with static validation policies in a Kubernetes cluster could be OPA Gatekeeper. It helps with writing these policies in Rego language and then ship them to a Kubernetes cluster using a CRD – since they are stored in a cluster in the form of CRs, it's very easy to maintain them. Gatekeeper helps with auditing violations, mutating resources and much more that we are going to talk about at length in the following sections.

## 2.1 Gatekeeper Architecture:

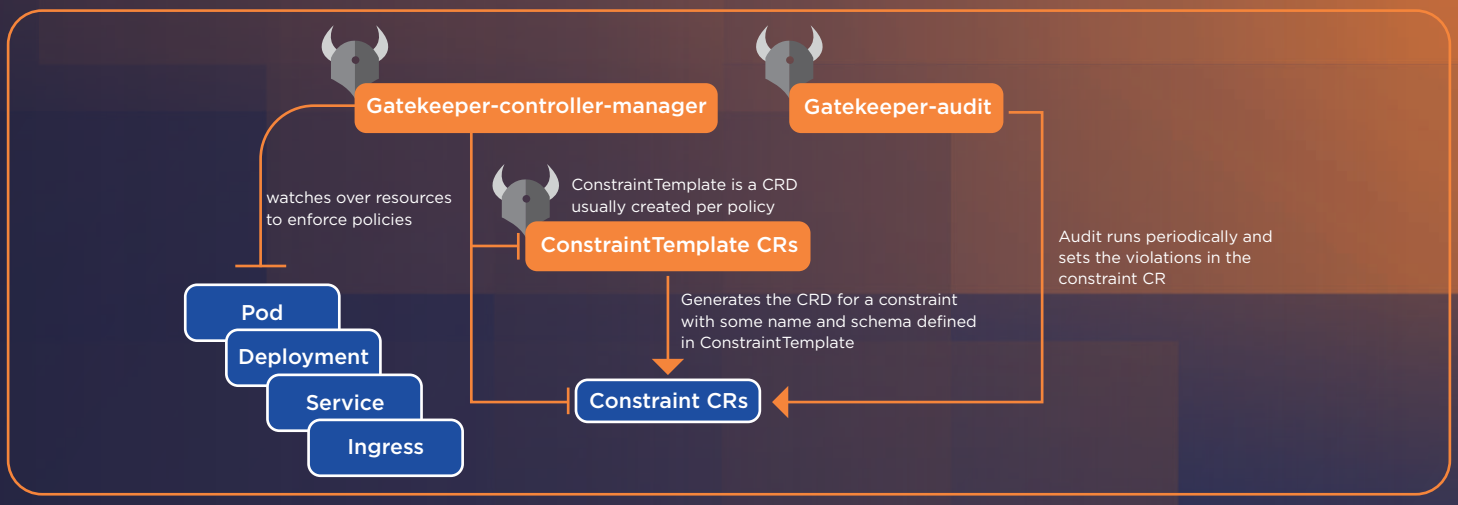


*Figure 1: Components installed with OPA Gatekeeper*

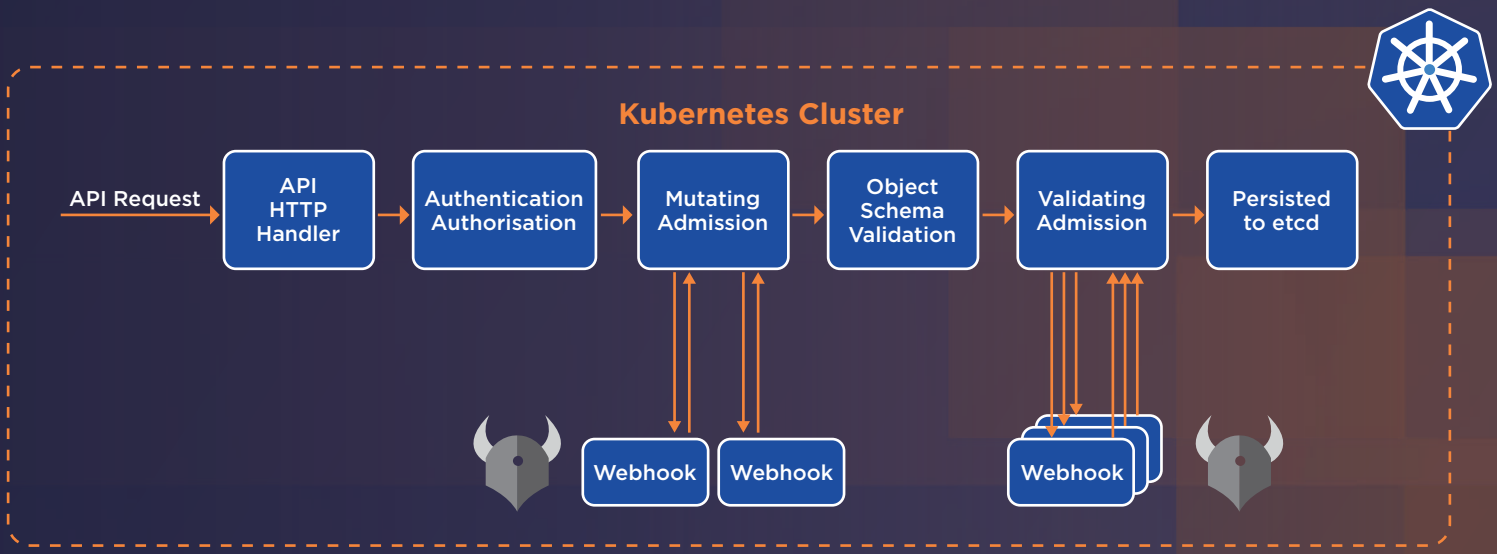


*Figure 2: Validating and Mutating webhook flow*

Here, Figure 1 depicts the two deployments that are deployed when you install OPA Gatekeeper in a Kubernetes cluster – gatekeeper-controller-manager and gatekeeper-audit. It shows the significance of the ConstraintTemplate CRD through which you can define the policy name and schema, which further generates a new constraint CRD. You can then define your Rego policies in the CRs. This follows the same format as any CRD in Kubernetes does, where you first create a CRD and then CRs that follow the schema defined in that CRD.

And Figure 2 talks about how a request reaches the API server and how it is being handled from the perspective of two very important Kubernetes resources – validating and mutating webhook configurations.

A validating webhook configuration can validate a resource (native or custom) against some rule defined. In OPA Gatekeeper, you can define such rules through the constraint CRs. Once this is done, every new Kubernetes resource (subjected to the validating webhook configuration) will only be created successfully if it does not violate the rules.

A mutating webhook configuration helps with mutating a resource(native or custom) based on the mutating rules defined. In OPA Gatekeeper, this can be achieved by making use of their Assign CRD.

## 2.2 Why we use Gatekeeper:

Even though there are lots of great tools in the CNCF landscape that help with the same use case, we use OPA Gatekeeper because of the following reasons:

- It has a rich past with its parent OSS project OPA.
- The UX is cloud-native with CRDs and integrations with monitoring tools.
- It has a huge library of production grade policies already available to be used.
- The ability to write complex queries in Rego which can be easily maintained and tested using their CLI.

But this doesn't mean that other tools don't come with their advantages, and we certainly think that tools like Kyverno where you can write policies in yaml format and Kubewarden where you can use webassembly to write policies do come really close.

We categorise the policies we deploy into "upstream policies" and "custom policies". Before we dive deep into what these policies are and how they are written or customised, it's important that we discuss in brief what these two categories are - upstream policies are slightly flavoured with source of truth being the OPA Gatekeeper policy library and custom policies are completely derived out of the customer and platform requirements where we run our Kubernetes Service (IKS).

## 2.3 Upstream Policies

These policies adhere to the standard set by the policies in the OPA Gatekeeper policy library but add a few more things on top for better automation and UX. These include:

- We already add certain namespaces and pods in the constraint that need exemption from a particular constraint. Example: to exempt pods of Istio, Prometheus etc., from a policy that blocks pods that mount a token to API Server. Its implementation looks something like this:

```
spec:
  enforcementAction: deny
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
    labelSelector:
      matchExpressions:
        - operator: "NotIn"
          key: "app"
          values:
          - istiod
          - prometheus-node-exporter
          - kube-prometheus-stack-operator
          - prometheus
          - alertmanager
```

*Figure 3: Already added namespaces and pods in constraint CR*

- We add a new parameter in the constraint through which the complete namespace can be exempted from either all the constraints or that particular constraint by labelling that namespace with the appropriate key-value pair. Example: to exempt pods created in namespace "test" from the constraint or the constraint that blocks running the container in privileged mode. Its implementation looks something like this:

```
parameters:
  namespace:
    labels:
    - tcl-opa-exclude-all # label to exclude all policies
    - tcl-opa-exclude-psp-privileged-container # label to exclude this policy
```

*Figure 4: Labels to improve the UX of exemption from constraints*

- We changed a few things in the constraint that restricts usage of volume types in a pod where we do allow usage of hostpath volumes but in a limited capacity. Example: a Kubernetes cluster that has a disk attached to the nodes at path /nfs/foo.

  To use this as a volume, a cluster administrator will have to create a configmap in "gatekeeper-system" namespace with either the absolute path or prefix path, which will look something like this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: host-paths-cm
  namespace: gatekeeper-system
data:
  paths: |
    /path/absolute
    /path/prefix/*
```

*Figure 5: Allowing hostpath volumes*

## We package the following policies:

- k8s-automount-serviceaccount-token: Controls the ability of any Pod to enable automountServiceAccountToken.
- k8s-block-clusterip-services-with-externalip: Disallows all Services with type ClusterIP to have ExternalIPs.
- k8s-block-loadbalancer-services: Disallows all Services with type LoadBalancer. This is done because IKS does not support these types of services, and if not blocked it can be misused.
- k8s-block-nodeport-services: Disallows all Services with type NodePort.
- psp-allow-privilege-escalation: Controls escalation to root privileges.
- psp-capabilities: Controls access to linux capabilities in containers.
- psp-forbidden-sysctls: Controls the `sysctl` profile used by containers.
- psp-host-network-ports: Controls usage of host network and ports by pod containers.
- psp-privileged-containers: Controls the ability of any container to run in privileged mode.
- psp-volume-types: Restricts the mountable volume types on containers.

## 2.4 Custom Policies

These policies have the automation and UX improvements of upstream policies already baked in. The main objective for creating these policies is to cover gaps and use cases that arise due to the platform variability and the consumption of a Kubernetes cluster by a customer.

### We package the following policies:

- tcl-allow-namespace-labels: Restricts the ability to label namespaces with certain key-value pairs to privileged users/groups only. Here, the key-value pair, privileged users/groups are specified in the constraint.
- tcl-block-cluster-admin-role: Restricts all ClusterRoleBindings with a `cluster-admin` roleRef unless they are annotated with a certain key-value pair. This addresses a kube-bench test. Here, the key-value pair is specified in the constraint.
- tcl-block-delete-resource: Blocks deletion of some very crucial resources unless they are annotated with a certain key-value pair. However, if the users/groups belong to a privileged group, the annotation is not necessary, this is done to not block deletion of resources by their controllers (owners). Here, the key-value pair and the privileged users/groups are specified in the constraint.
- tcl-block-deny-group: Restricts unprivileged users/groups from performing operations such as create, update, and delete in the specified namespaces. Here, unprivileged users/groups and the namespaces are specified in the constraint.
- tcl-block-deny-storageclass-use: Denies unprivileged users/groups from creating PVC/PV with storageClass having labels with certain key-value pair. Also, if it's some other user/group which is not in the list of restricted users/groups, check in which namespace the PVC is being created with storageClass having that label, it should be in the list of allowed namespaces. Here, the key-value pair, namespaces and unprivileged users/groups are specified in the constraint.
- tcl-block-wildcard-cluster-role: Restricts the use of wildcard (*) in ClusterRole's apiGroups unless the ClusterRole is annotated with a certain key-value pair. This addresses a kube-bench test. Here, the key-value pair is specified in the constraint.
- tcl-block-wildcard-ingress: Blocks the ability to create Istio VirtualServices and Gateways with a blank or wildcard (*) in hostname, since that would enable bad actors to intercept traffic for other services in the cluster, even if they don't have access to those services.
- tcl-unique-ingress-host: Enforces that all Istio VirtualServices and Gateways unique hostnames. Does not handle hostname wildcards.

Now that we have seen the vast variety of policies that can be written, it's important that we also understand what's the recipe of these policies, how we empower our security with it and how you can create more policies.

# 3. WRITING A REGO POLICY

Creating a new policy in OPA Gatekeeper involves several steps. Here's a detailed guide to help you through the process:

## 3.1 Step 1: Define a ConstraintTemplate

Before you can define a constraint, you must first define a ConstraintTemplate, which describes both the Rego that enforces the constraint and the schema of the constraint. The schema of the constraint allows an admin to fine-tune the behaviour of a constraint, much like arguments to a function.

**The most important pieces of the below ConstraintTemplate YAML are:**

- **validation,** which provides the schema for the parameters field for the constraint.
- **targets,** which specifies what "target" (defined later) the constraint applies to. Note that currently constraints can only apply to one target.
- **rego,** which defines the logic that enforces the constraint.
- **libs,** which is a list of all library functions that will be available to the Rego package. Note that all packages in libs must have lib as a prefix (e.g. package lib.<something>).

```yaml
apiVersion: templates.gatekeeper.sh/v1
kind: ConstraintTemplate
metadata:
  name: tclvolumesizerestriction
spec:
  crd:
    spec:
      names:
        kind: TCLVolumeSizeRestriction
      validation:
        # Schema for the `parameters` field
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego:
        # rego rule
      libs:
       # custom library
```

*Figure 6: ConstraintTemplate template*

### 3.1.1 Rego semantics for constraints

There are a few rules for the Rego constraint source code:

1. Everything is contained in one package
2. Limited external data access

a. No imports

b. Only certain subfields of the data object can be accessed:

   i. **data.inventory** allows access to the cached objects for the current target

c. Full access to the input object

3. Specific rule signature schema (described below)

### 3.1.2 Rule Schema

While template authors are free to include whatever rules and functions they wish to support their constraint, the main entry point called by the framework has a specific signature:

```
violation[{"msg": msg, "details": {}}] {
  # rule body
}
```

*Figure 7: Rule entrypoint*

- The rule name must be **violation.**
- **msg i**s the string message returned to the violator. It is required.
- **details** allow for custom values to be returned. This helps support uses like automated remediation. There is no predefined schema for the details object. Returning details is optional.

### 3.1.3 What is a target?

Target is an abstract concept. It represents a coherent set of objects sharing a common identification and/or selection scheme, generic purpose, and can be analysed in the same validation context.

### 3.1.4 Input review

The data that's passed to Gatekeeper for review is in the form of an input.review object that stores the admission request under evaluation. It follows a structure that contains the object being created, and in the case of update operations the old object being updated. It has the following fields:

- **dryRun:** Describes if the request was invoked by kubectl-dry-run. This cannot be populated by Kubernetes for audit.
- **kind:** The resource kind, group, version of the request object under evaluation.
- **name:** The name of the request object under evaluation. It may be empty if the deployment expects the API server to generate a name for the requested resource.
- **namespace:** The namespace of the request object under evaluation. Empty for cluster scoped objects.
- **object:** The request object under evaluation to be created or modified.

- **oldObject:** The original state of the request object under evaluation. This is only available for UPDATE operations.
- **operation:** The operation for the request (e.g. CREATE, UPDATE). This cannot be populated by Kubernetes for audit.
- **uid:** The request's unique identifier. This cannot be populated by Kubernetes for audit.
- **userInfo:** The request's user's information such as username, uid, groups, extra. This cannot be populated by Kubernetes for audit.

Here is an example ConstraintTemplate that restrict PersistentVolume and PersistentVolumeClaim to declare storage capacity beyond the allowed capacity described by the constraint to be present:

```
spec:
  crd:
    spec:
      names:
        kind: TCLVolumeSizeRestriction
      validation:
        # Schema for the `parameters` field
        openAPIV3Schema:
          type: object
          properties:
            limit:
              type: string
              pattern: "^[0-9]+(\\.[0-9]+)?[KMGTP]i$"
            annotation:
                type: object
                properties:
                  key:
                    type: string
                  value:
                    type: string
```

*Figure 8: Constraint's schema*

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

      import data.lib.exclude_delete.is_delete
      import data.lib.memory_size_utils.mem_gt

      violation[{"msg": msg}] {
        not is_delete(input.review)
        is_storage_limit_exceeded
        result := validate_annotation
        msg = result.message
      }

      # Checks if storage limit is exceeded for PersistentVolumes
      is_storage_limit_exceeded {
        mem_gt(input.review.object.spec.capacity.storage, input.parameters.limit)
      }

      # Checks if storage limit is exceeded for PersistentVolumeClaims
      is_storage_limit_exceeded {
        mem_gt(input.review.object.spec.resources.requests.storage, input.parameters.limit)
      }

      # generate error message if required annotation key is not found
      validate_annotation = {"message": message} {
        not input.review.object.metadata.annotations[input.parameters.annotation.key]
        message := sprintf("Storage capacity above '%s' only allowed once appropriate
annotation is set, contact TCL Kubernetes team", [input.parameters.limit])
      }

      # generate error message if required annotation key and value is not valid
      validate_annotation = {"message": message} {
        input.review.object.metadata.annotations[input.parameters.annotation.key]
        input.review.object.metadata.annotations[input.parameters.annotation.key] !=
input.parameters.annotation.value
        message := sprintf("Annotation key '%s' has an invalid value, contact TCL
Kubernetes team", [input.parameters.annotation.key])
      }
```

*Figure 9: Rego validation rule*

```
libs:
- |
  package lib.exclude_delete

  is_delete(review) {
      review.operation == "DELETE"
  }
- |
  package lib.memory_size_utils

  # Conversion factors for Kubernetes memory units to bytes
  memory_units = {
      "Ki": 1024,
      "Mi": 1048576,  # 1024 * 1024
      "Gi": 1073741824,  # 1024 * 1024 * 1024
      "Ti": 1099511627776,  # 1024 * 1024 * 1024 * 1024
      "Pi": 1125899906842624,  # 1024 * 1024 * 1024 * 1024 * 1024
      "Ei": 1152921504606846976  # 1024 * 1024 * 1024 * 1024 * 1024 * 1024
  }

  # Parse memory string (e.g., "512Mi") and convert it to bytes
  memory_in_bytes(mem) = bytes {
      re_match("^[0-9]+(\\.[0-9]+)?[KMGTP]i$", mem)
      unit_start := count(mem) - 2  # Calculate where the unit part starts
      unit := substring(mem, unit_start, count(mem))  # Extract the unit (e.g., "Mi")
      value := to_number(substring(mem, 0, unit_start))  # Extract the numeric part
      bytes := value * memory_units[unit]
  }

  # Comparison rule: returns true if memory size `a` is greater than `b`
  mem_gt(a, b) {
      memory_in_bytes(a) > memory_in_bytes(b)
  }
```

*Figure 10: Library function*

**You can install this constraint template with the following command:**

kubectl apply -f gatekeeper/custom-policy/tcl-volume-size-restriction/template.yaml

## 3.2 Step 2: Create a constraint

Constraints are then used to inform Gatekeeper that the admin wants a ConstraintTemplate to be enforced, and how. This constraint uses the **TCLVolumeSizeRestriction** constraint template above to make sure all PersistentVolume and PersistentVolumeClaim are restricted to use storage capacity beyond the allowed limit:

```yaml
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: TCLVolumeSizeRestriction
metadata:
  name: tcl-volume-size-restriction
spec:
  enforcementAction: deny
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["PersistentVolume", "PersistentVolumeClaim"]
  parameters:
    limit: 100Mi
    annotation:
      key: "tcl-opa-allow-storage"
      value: IKBE6QKA55
```

*Figure 11: Constraint*

### 3.2.1 The match field

The match field defines which resources the constraint will be applied to. It supports the following types of matchers:

- **kinds** accepts a list of objects with apiGroups and kinds fields that list the groups/kinds of objects to which the constraint will apply. If multiple groups/kinds objects are specified, only one match is needed for the resource to be in scope.
- **scope** determines if cluster-scoped and/or namespaced-scoped resources are matched. Accepts *, Cluster, or Namespaced. (defaults to *)
- **namespaces** is a list of namespace names. If defined, a constraint only applies to resources in a listed namespace. Namespaces also supports a prefix-based glob. For example, namespaces: [kube-*] matches both kube-system and kube-public.
- **excludedNamespaces** is a list of namespace names. If defined, a constraint only applies to resources not in a listed namespace. ExcludedNamespaces also supports a prefix-based glob. For example, excludedNamespaces: [kube-*] matches both kube-system and kube-public.
- **labelSelector** is the combination of two optional fields: matchLabels and matchExpressions. These two fields provide different methods of selecting or excluding k8s objects based on the label keys and values included in object metadata. All selection expressions are ANDed to determine if an object meets the cumulative requirements of the selector.
- **namespaceSelector** is a label selector against an object's containing namespace or the object itself, if the object is a namespace.
- **name** is the name of a Kubernetes object. If defined, it matches against objects with the specified name. Name also supports a prefix-based glob. For example, name: pod-* matches both pod-a and pod-b.

Note that if multiple matchers are specified, a resource must satisfy each top-level matcher (kinds, namespaces, etc.) to be in scope. Each top-level matcher has its own semantics for what qualifies as a match. An empty matcher, a undefined match field, is deemed to be inclusive (matches everything). Also understand namespaces, excludedNamespaces, and namespaceSelector will match on cluster scoped resources which are not namespaced. To avoid this, adjust the scope to Namespaced.

### 3.2.2 The parameters field

The parameters field describes the intent of a constraint. It can be referenced as **input.parameters** by the ConstraintTemplate's Rego source code. Gatekeeper populates **input.parameters** with values passed into the parameters field in the Constraint.

### 3.2.3 The enforcementAction field

The **enforcementAction** field defines the action for handling Constraint violations. By default, enforcementAction is set to **deny** as the default behaviour is to deny admission requests with any violation. Other supported enforcementActions include **dryrun** and **warn.** Refer to Handling Constraint Violations for more details.

**You can install this Constraint with the following command:**

kubectl apply -f gatekeeper/custom-policy/tcl-volume-size-restriction/constraint.yaml

Note that if multiple matchers are specified, a resource must satisfy each top-level matcher (kinds, namespaces, etc.) to be in scope. Each top-level matcher has its own semantics for what qualifies as a match. An empty matcher, an undefined match field, is deemed to be inclusive (matches everything). Also understand namespaces, excludedNamespaces, and namespaceSelector will match on cluster scoped resources which are not namespaced. To avoid this, adjust the scope to Namespaced.

## 4. VULNERABILITY FIXES REPORTED BY SCANNERS

When we previously discussed about the security vectors, we briefly mentioned about compliance scanners, this is one area where having OPA gatekeeper helps us. There have been instances where we have patched the vulnerabilities by creating new policies and we will talk about some of them below.

Kube-bench 5.1.1: This compliance test recommends against using “cluster-admin” role in ClusterRoleBindings. To comply with this, we created a policy from scratch, which we have already discussed in the custom policies subsection.

Kube-bench 5.1.3: This compliance test recommends against using “*” wildcards in Roles and ClusterRoles. To comply with this, we created a policy that addresses the wildcard usage in apiGroups only for now, although it can easily be enhanced to support verbs and resources too. This has been discussed in the custom policies subsection.

Kube-hunter avd-ksv-0119: This particular vulnerability reported by penetration testing recommends against adding "cap_net_raw" capability in a Pod. To comply with this, we created a policy that only allows very few capabilities to be added in a Pod, we have discussed about this policy in the upstream policies section.

## 5 AUTOMATION AND A BETTER UX:

Although OPA Gatekeeper is nicely packaged and baked with all the important features, as a Kubernetes service provider, we have a slightly opinionated approach about how we package it in the Kubernetes clusters deployed using IKS. It’s important to note that the changes we have made are just wrappers on top of OSS. It ain’t much but it’s honest work.


## 5.1 Policy Exception:

In Kubernetes, managing policies is crucial for maintaining security and compliance. However, there are scenarios where certain namespaces need exemptions from these policies temporarily or . This is where the **PolicyException** (in-house developed Kubernetes CRD) comes into play. It allows specific namespaces to bypass gatekeeper policies for a predefined duration.

### 5.1.1 What can be done with PolicyException?

1. Temporary Policy Exemptions: You can temporarily disable specific policies in a namespace. This is useful for testing, development, or any situation where you need to bypass certain restrictions without permanently altering your policy configurations.
2. Scheduled Policy Exemptions: PolicyException allows you to schedule when a policy exemption should start and end. This is particularly useful for planned maintenance windows or specific project timelines.

### 5.1.2 Available options

1. Namespace: Specify the namespace where the policy exception will be applied. This field is immutable once set.
2. Enforcement: Define the duration or the specific time window for the policy exemption.
    - Duration: Use this option to specify how long the exception will be active. The format is XmXdXh (e.g., 5m6d8h for 5 months, 6 days, and 8 hours).
    - Time: Use this option to set a start and end time for the exemption. The format follows the UTC time standard (e.g., 2023-12-01T00:00:00Z).
1. Policy Names: List the policies from which the namespace needs exemptions.

### 5.1.3 Examples

1. Disable Service Account Automount Policy for 1 Hour:

```yaml
apiVersion: tatacommunications.com/v1alpha1
kind: PolicyException
metadata:
  name: disable-automount-serviceaccount
spec:
  namespace: demo
  enforcement:
    duration: "0m0d1h"
  policyNames:
  - k8s-automount-serviceaccount-token-pod
```

*Figure12: PolicyException sample1*

2. Disable Service Account Automount Policy for 1 Hour:

```yaml
apiVersion: tatacommunications.com/v1alpha1
kind: PolicyException
metadata:
  name: disable-loadbalancer-nodeport
spec:
  namespace: demo
  enforcement:
    time:
      startTime: "2023-12-01T00:00:00Z"
      endTime: "2023-12-31T00:00:00Z"
  policyNames:
  - k8s-block-load-balancer
  - k8s-block-node-port
```

*Figure13: PolicyException sample2*

### 5.1.4 Checking PolicyException status

To monitor the status of your PolicyException resources, use:

```
kubectl get policyexceptions
```

This command provides details such as the name, namespace, start and end times, duration, active status, and reason. For example:

```
NAME                                 NAMESPACE  START                 END                   DURATION  ACTIVE  REASON
disable-automount-serviceaccount     demo       2023-11-01T05:00:00Z  2023-11-01T06:00:00Z  1h0m0s    true    Ready
disable-loadbalancer-nodeport        demo       2023-11-01T00:00:00Z  2023-11-30T00:00:00Z  720h0m0s  false   Waiting
```

*Figure14: PolicyException status*

- Active: Indicates if the exemption is currently active.
- Reason: Provides the status of the exemption (e.g., Ready, Waiting, Failed).

By leveraging PolicyException, you can manage temporary policy exemptions effectively, ensuring your Kubernetes environment remains both flexible and secure.

## 5.2 Observability

Maintaining visibility into policy enforcement is crucial. To achieve this, we integrated opa-exporter with Gatekeeper and created a comprehensive Grafana dashboard. This setup allows us to monitor policy violations and ensure compliance effectively.

The Grafana dashboard includes several panels that display key metrics, to quickly identify and address compliance issues. List of few important insights that can be gained from this dashboard:

- violations per namespace
- violations per policy
- list of policies applied in the cluster
- status of the gatekeeper control plane pods
- list of policies by action – deny or dryrun

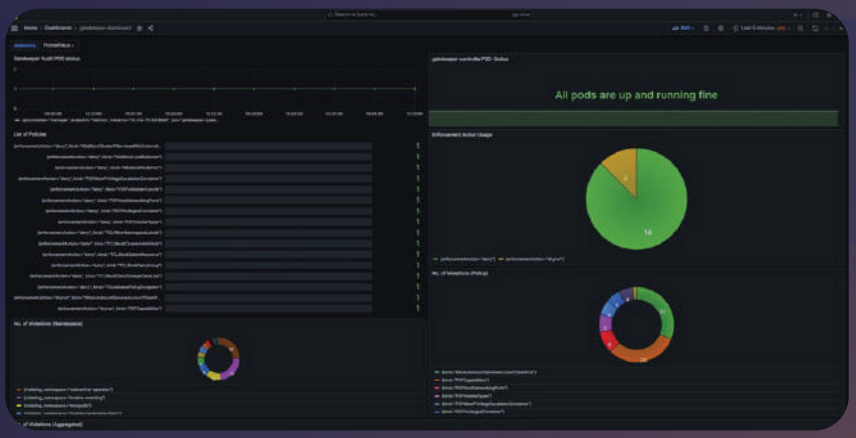


*Figure15: Grafana dashboard panels view1*

*Figure16: Grafana dashboard panels view2*

## 5.3 Ticketing

We have written some prometheus rules that work on top of the observability features talked above, for example – GatekeeperViolationDeny, GatekeeperAuditPodDown and GatekeeperControllerManagerPodDown; whenever these alerts are in firing state, a ticket is automatically created on ticketing systems like zendesk and servicenow. Once a ticket is created, it can be worked upon by either someone from operations or a dedicated application devops team.

This is not just the case for Gatekeeper violations, this is a general ticketing mechanism for the IZO Kubernetes Service.

# 6. WHAT'S NEXT:

### We have seen OPA Gatekeeper in its full glory, but what's next?

The roadmap ahead is a conjunction of new features that are available in OSS and what IKS provides on top of it. Although this roadmap is still work-in-progress, each item has its own significance and we can't stress enough how important the activity of continuously enhancing the security framework is, to deal with the ever-growing list of vulnerabilities. Following are the features that we are currently invested in:

- **Gatekeeper external data provider:** We have already discussed how we use gatekeeper to fix vulnerabilities and make Kubernetes cluster compliant here, but this particular feature makes for an excellent segue into the discussion of using gatekeeper to address another security vector of software supply chain security.
- **Validating admission policy:** This is a feature that became stable only in Kubernetes v1.30, but it offers a native way of writing validating policy using CEL. It's important that we evaluate how we can provide a great UX through gatekeeper where policies can be written in both rego and CEL.
- **Structured authorisation configuration:** We evaluated the Kubernetes authoriser flow towards the end of 2022 where we wanted to add a custom authoriser in the request flow to API Server, but there were some restrictions on achieving our use case back then. Since then, the authoriser flow has changed quite a lot, and the changes became beta in Kubernetes v1.30 that's our queue to start our work on it again and make use of the OPA project (not Gatekeeper) here.
- **Integration with OTEL:** In order to remove the dependency on opa-scorecard, we are evaluating an integration through which, we can use the logs of "gatekeeper-audit" Pod and translate them into grafana dashboards.
- Runtime security with Apparmor and Seccomp profile, and integration with tools like **KubeArmor** that can work with BPF LSM.