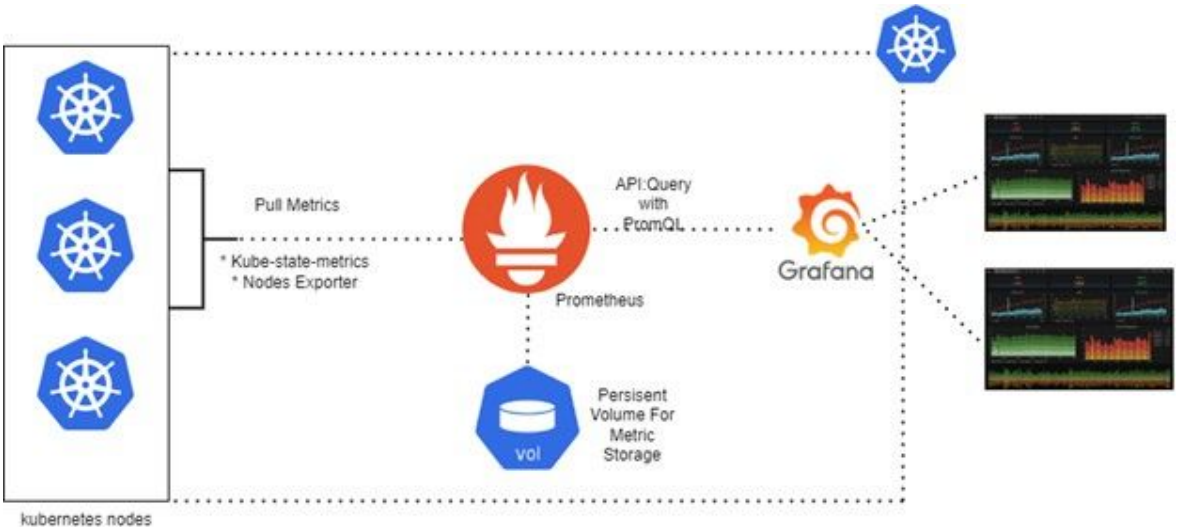
# MONITORING DOCUMENTATION

## Monitoring Architecture.




### Kubernetes Nodes

**Nodes**: The architecture comprises multiple Kubernetes nodes where applications and services are running.

### Metric Collection

**Node Exporter**: Deployed on each node to collect hardware and OS metrics.

**Kube-state-metrics**: Provides detailed information about the state of Kubernetes objects.

### Prometheus

**Prometheus Server**: Deployed as a pod within the Kubernetes cluster.

**Metrics Pulling**: Prometheus pulls metrics from the Node Exporter and Kube-state-metrics.

**Persistent Storage**: Metrics are stored in a Persistent Volume (PVC) to ensure data durability and persistence.

### Grafana

**Dashboarding**: Grafana is used to create, manage, and visualize dashboards.

**PromQL**: Utilizes Prometheus Query Language (PromQL) to query and retrieve metrics from Prometheus.

**ConfigMap**: Dashboards are stored and persisted using Kubernetes ConfigMaps for easy management and deployment.

## Monitoring Namespace Resources

```
root@YashPatil-VM:~# kubectl get po -n monitoring
NAME                                                     READY   STATUS      RESTARTS        AGE
alertmanager-prometheus-stack-kube-prom-alertmanager-0   2/2     Running     1 (7d20h ago)   7d20h
prometheus-prometheus-stack-kube-prom-prometheus-0       2/2     Running     0               7d20h
prometheus-stack-grafana-849bd6c4f9-9smk9                2/2     Running     0               2d19h
prometheus-stack-kube-prom-admission-create-4nmj7        0/1     Completed   0               7d20h
prometheus-stack-kube-prom-admission-patch-vfl4n         0/1     Completed   0               7d20h
prometheus-stack-kube-prom-operator-69ffc5cbb5-57f6d     1/1     Running     0               2d19h
prometheus-stack-kube-state-metrics-f7dc4d67d-rzlxw      1/1     Running     0               2d19h
prometheus-stack-prometheus-node-exporter-2rfnh          1/1     Running     0               7d20h
prometheus-stack-prometheus-node-exporter-4gh6v          1/1     Running     0               7d20h
prometheus-stack-prometheus-node-exporter-4grr9          1/1     Running     0               7d20h
prometheus-stack-prometheus-node-exporter-5cwsp          1/1     Running     0               7d20h
prometheus-stack-prometheus-node-exporter-k97ww          1/1     Running     0               7d20h
prometheus-stack-prometheus-node-exporter-pc7bx          1/1     Running     0               7d20h
prometheus-stack-prometheus-node-exporter-s4ns7          1/1     Running     0               7d20h
prometheus-stack-prometheus-node-exporter-vlv2g          1/1     Running     0               7d20h
prometheus-stack-prometheus-node-exporter-vt6rr          1/1     Running     0               7d20h
```

To get all information regarding the monitoring resources in the `monitoring` namespace, use the following command:

```
kubectl get all -n monitoring
```

### Pods in the Monitoring Namespace

- alertmanager-prometheus-stack-kube-prom-alertmanager-0

**Description**: This pod is related to Alert manager, which handles alerts.

- prometheus-prometheus-stack-kube-prom-prometheus-0

**Description**: This pod is related to Prometheus, which is responsible for scraping and storing metrics.

- prometheus-stack-grafana-849bdc64f9-9smk9

**Description**: This pod is related to Grafana, which is used for visualizing metrics stored in Prometheus.

- prometheus-stack-kube-prom-operator-69ff5cbb5-57f6d

**Description**: This pod manages the entire Prometheus stack, including configurations and updates.

- prometheus-stack-kube-state-metrics-f7dc4d67dr-rzlxw

**Description**: This pod collects metrics about the state of Kubernetes resources.

- **prometheus-stack-Prometheus-node-exporter**

**Description**: Node exporter pod for collecting hardware and OS metrics from the nodes.

### Prometheus StatefulSet and Grafana Deployments

- **Prometheus**

**Type**: StatefulSet

**Reason**: Prometheus uses StatefulSet to store TSDB data, ensuring data persistence.

**Management**: Managed by its Custom Resource Definition (CRD) with kind `Prometheus`.

**View CRD**:

```
kubectl get prometheus -n monitoring
```

**Edit CRD**:

```
kubectl edit prometheus -n monitoring
```

- **Grafana**

**Type**: Deployment

**Edit Deployment:**

```
kubectl edit deployment/prometheus-stack-grafana -n monitoring
```

### Key Parameters in Prometheus CRD

- **Name**: `prometheus-stack-kube-prom-prometheus`

**Description**: Name of the prometheus instance.

- **Alerting**:

**Alertmanagers**: List of Alertmanager instances that Prometheus can send alerts to.

**Instance**:

**API Version**: `v2`

**Name**: `prometheus-stack-kube-prom-alertmanager`

**Namespace**: `monitoring`

**Path Prefix**: `/`

**Port**: `http-web`

- **Evaluation Interval**: `30s`

**Description**: How frequently rules are evaluated.

- **Image**: `quay.io/katonic/prometheus:v2.40.5`

**Description**: Container image used for Prometheus.

- **Replicas**: `1`

**Description**: Number of Prometheus replicas.

- **Retention Size**: `19GB`

**Description**: Maximum size of retained data.

- **Scrape Interval**: `30s`

**Description**: How frequently to scrape targets.

- **Service Monitor Selector**:

**Match Labels**:

**Release**: `prometheus-stack`

**Description**: Conditions to match ServiceMonitors by labels.

## Using ServiceMonitor for Automatic Metric Scraping

### What is a ServiceMonitor?

A **ServiceMonitor** is a Custom Resource Definition (CRD) used by the Prometheus Operator to automatically discover and scrape metrics from services running in a Kubernetes cluster. It tells Prometheus which endpoints to scrape for metrics.

### Why Use a ServiceMonitor?

1. **Automatic Discovery**: Allows Prometheus to automatically discover services and endpoints that expose metrics, eliminating the need for manual configuration.
2. **Dynamic Configurations**: Ensures Prometheus adapts dynamically as new services come online or change.
3. **Targeted Scraping**: Enables specific configuration of how metrics are scraped from services, such as the interval, path, and port.

### How Does a ServiceMonitor Work?

- **Selectors**: Uses label selectors to find the services it should monitor.
- **Endpoints**: Defines the endpoints (path and port) from which Prometheus should scrape the metrics.
- **NamespaceSelector**: Specifies whether the ServiceMonitor should look for services in all namespaces or specific ones.

- To get a list of ServiceMonitors in the monitoring namespace, use:

```
kubectl get servicemonitor -n monitoring
```

- Example of a ServiceMonitor Configuration

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    app: gd-7a59f704-e113-457d-b2ae-e68a1b55836c
    platform-component: chatbots
    release: prometheus-stack
  name: gd-7a59f704-e113-457d-b2ae-e68a1b55836c-servicemonitor-piyush-panchal-katonic-ai
  namespace: monitoring
spec:
  endpoints:
    - interval: 15s
      path: metrics
      port: app-endpoint-public
  namespaceSelector:
    any: true
  selector:
    matchLabels:
      app: gd-7a59f704-e113-457d-b2ae-e68a1b55836c
```

- **Labels**: Identify and group the ServiceMonitor for organizational and management purposes.
- **Name**: Unique name of the ServiceMonitor.
- **Namespace**: Namespace where the ServiceMonitor is deployed.
- **Endpoints**: Specifies the endpoints Prometheus should scrape.

**Interval**: Frequency of scraping (15 seconds).

**Path**: HTTP path to scrape for metrics ( `/metrics` ).

**Port**: Port on the service to scrape ( `app-endpoint-public` ).

- **NamespaceSelector**: If `true` , the ServiceMonitor can monitor services in any namespace.
- **Selector**: Used to select the services to monitor based on their labels.

**MatchLabels**: Only services with these labels will be monitored.

Example of a Service Configuration

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: gd-7a59f704-e113-457d-b2ae-e68a1b55836c
    kind: generative-ai
    platform-component: chatbots
  name: gd-7a59f704-e113-457d-b2ae-e68a1b55836c
  namespace: piyush-panchal-katonic-ai
spec:
  clusterIP: 10.43.106.249
  ports:
    - name: app-endpoint-secure
      port: 4180
      protocol: TCP
      targetPort: 4180
    - name: app-endpoint-public
      port: 8000
      protocol: TCP
      targetPort: 8000
    - name: app-gradio
      port: 4280
      protocol: TCP
      targetPort: 4280
  selector:
    app: gd-7a59f704-e113-457d-b2ae-e68a1b55836c
  sessionAffinity: None
  type: ClusterIP
```

- **Labels**: Used for service identification and selection by the ServiceMonitor.
- **Name**: Unique name of the service.
- **Namespace**: Namespace where the service is deployed.
- **Ports**: Defines the ports that the service exposes.

**Name**: Name of the port (e.g., `app-endpoint-secure` , `app-endpoint-public` , `app-gradio` ).

**Port**: Port number (e.g., 4180, 8000, 4280).

**TargetPort**: Port on the pod that the service forwards traffic to.

- **Selector**: Matches the pods with the specified labels ( `app: gd-7a59f704-e113-457d-b2ae-e68a1b55836c` ).

### How ServiceMonitor and Service Work Together

1. **Label Matching**: The ServiceMonitor uses the selector field to match services with the label `app: gd-7a59f704-e113-457d-b2ae-e68a1b55836c`. The service has this label, so it is selected for monitoring.
2. **Endpoint Scraping**: The ServiceMonitor's `spec.endpoints` defines that Prometheus should scrape the `/metrics` path on the port named `app-endpoint-public` (port 8000) every 15 seconds.
3. **Namespace Selection**: The ServiceMonitor's `namespaceSelector` allows it to monitor services in any namespace.

## Creating and Persisting Dashboards in Grafana

### Overview

Grafana dashboards are provisioned with the platform during cluster installation. While these pre-configured dashboards provide a good starting point, any modifications made directly within Grafana will not persist after a restart. To ensure changes are saved and new dashboards are added permanently, follow these steps.

### Steps to Modify and Persist Existing Dashboards

1. **Accessing the Existing Dashboard**:

When a cluster is installed, Grafana dashboards are provisioned automatically.

Direct modifications to these dashboards in the Grafana UI will not persist after a restart.

2. **Downloading the Dashboard JSON**:

To modify and persist changes to an existing dashboard:

1. Open the dashboard in Grafana.
2. Click on the "Share" icon and select the "Export" tab.
3. Download the JSON file for the dashboard.
4. **Editing the ConfigMap**:

   Edit the `katonic-dashboard` ConfigMap to include the downloaded JSON file:

```
kubectl edit cm/katonic-dashboard -n monitoring
```

   Add the JSON content to the appropriate section of the ConfigMap.

4. **Restarting Grafana Deployment**:

To apply the changes, restart the Grafana deployment:

```
kubectl rollout restart deployment/prometheus-stack-grafana -n monitoring
```

### Steps to Create and Persist a New Dashboard

1. **Creating a New Dashboard**:

From the Grafana UI:

a. Click on the "+" icon on the left-hand side and select "Dashboard".

b. Add the desired panels and metrics to create your dashboard.

c. Save the dashboard with a meaningful name.

2. **Setting a Unique UID**:

- When saving the new dashboard, ensure it has a unique UID for a meaningful and static URL.
- This UID is essential for sharing the dashboard URL with the UI team for integration.

3. **Downloading the Dashboard JSON**:

After creating the dashboard, download its JSON file:

a. Open the dashboard.

b. Click on the "Share" icon and select the "Export" tab.

c. Download the JSON file.

4. **Editing the ConfigMap**:

- Add the new dashboard JSON to the `katonic-dashboard` ConfigMap:

```
kubectl edit cm/katonic-dashboard -n monitoring
```

- Insert the JSON content for the new dashboard.

5. **Restarting Grafana Deployment**:

- Apply the changes by restarting the Grafana deployment.

```
kubectl rollout restart deployment/prometheus-stack-grafana -n monitoring
```

### Configuring Grafana

- Editing Grafana Configuration:
- All Grafana configurations are managed in the `prometheus-stack-grafana` ConfigMap.
- To make changes, edit this ConfigMap:

```

kubectl edit cm/prometheus-stack-grafana -n monitoring
```

- After making changes, restart the Grafana deployment to apply them:

```
kubectl rollout restart deployment/prometheus-stack-grafana -n monitoring
```

## Accessing Prometheus

- To access Prometheus via port forwarding:

```
kubectl port-forward svc/prometheus-stack-kube-prom-prometheus -n monitoring 8080:9090 --address 0.0.0.0
```

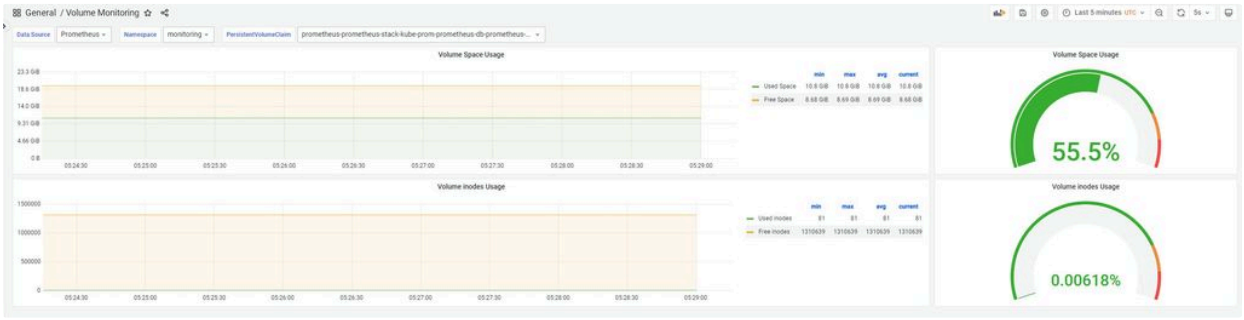
## Troubleshooting: Prometheus PVC Full Alerts

When monitoring Prometheus in a Kubernetes cluster, it's crucial to handle situations where the Persistent Volume Claim (PVC) for Prometheus gets filled up. This can lead to downtime and alerts. The following steps will guide you through troubleshooting and resolving this issue.

## Monitoring PVC Usage

1. **Accessing Volume Monitoring in Grafana**:

- Open Grafana UI.
- Navigate to the "Volume Monitoring" dashboard.
- Select the data source as "Prometheus" and the namespace as "monitoring".
- The "Volume Space Usage" panel shows the amount of data filled in the Prometheus PVC.




2. **Alerts Setup**:

- An alert is configured to trigger if the PVC usage exceeds 80%.
- Alerts are sent via email to notify administrators.

### Troubleshooting Steps

1. **Increasing PVC Size**:

- You can dynamically increase the PVC size to accommodate more data.
- Edit the Prometheus PVC configuration to increase the size from 20GB to 25GB.
- Modify the retention size in the Prometheus CRD accordingly.

2. **Manually Clearing Data**:

- If increasing the PVC size is not feasible, manually clear old data.
- Exec into the Prometheus StatefulSet pod and delete data:

```
kubectl exec -it -n monitoring prometheus-prometheus-stack-kube-prom-prometheus-0 -- sh
```

```
root@YashPatil-VM:~# kubectl exec -it -n monitoring prometheus-prometheus-stack-kube-prom-prometheus-0 -- sh
/prometheus $ ls
01J16W9JK4SD7983EP9GN8E51C  01J1JF28QMPR9H15H38WCD3Q1W  01J1P3RAQGXSAP3PBA2QV3VJAE  01J1PAMRVRQC7XAJACFFF2G43E  lock                    wal
01J1CNNP9ZV44X2HQKGGG1QYBA  01J1MSZ2H9X46XM9124VZTJJ1G  01J1PAM1ZHYDRAQY2M8GJ4KQQY  chunks_head                 queries.active
/prometheus $
```

- Delete the data using

```
/prometheus $ rm -rf *
```

- Delete and restart the Prometheus pod to apply changes:

```
kubectl delete pod prometheus-prometheus-stack-kube-prom-prometheus-0 -n monitoring
```

## Checking Logs

1. **Prometheus Logs**:

- If Prometheus is down, check the logs to diagnose the issue:

```
kubectl logs -f prometheus-prometheus-stack-kube-prom-prometheus-0 -n monitoring
```

2. **Grafana Logs**:

- If Grafana is down or not functioning correctly, check the logs:

```
kubectl logs -f prometheus-stack-grafana-849bd6c4f9-9smk9 -n monitoring -c grafana
```

### Verifying Prometheus Data Source

- To ensure the Prometheus data source is correctly configured in Grafana

1. Open the Grafana dashboard.
2. Go to "Configuration" and select "Data Sources".
3. Click on the Prometheus data source and click "Test".

## Alerting in Grafana

In our monitoring setup, we utilize two primary types of alerting: Node Alerting and Volume Alerting. This guide will explain the processes for creating alerts, managing them, and troubleshooting common issues related to Persistent Volume Claims (PVC).

### Volume Alerting

- For PVC alerting, we receive an email notification when the data usage exceeds 80% of the total PVC capacity. Prometheus handles data compression, and when the data usage falls below 80%, a resolution email is sent, restarting the 24-hour alert cycle.
- If the PVC usage exceeds 80% again within the same day, another alert will be triggered, even though the alert interval is set to 24 hours. Persistent issues can be resolved using the troubleshooting steps outlined in the `Troubleshooting: Prometheus PVC Full Alerts` section.

### Node Alerting

- Node alerting is designed to notify us when a node in the Kubernetes cluster goes down. By monitoring the status of nodes, we can ensure the health and availability of the cluster and respond promptly to any issues.

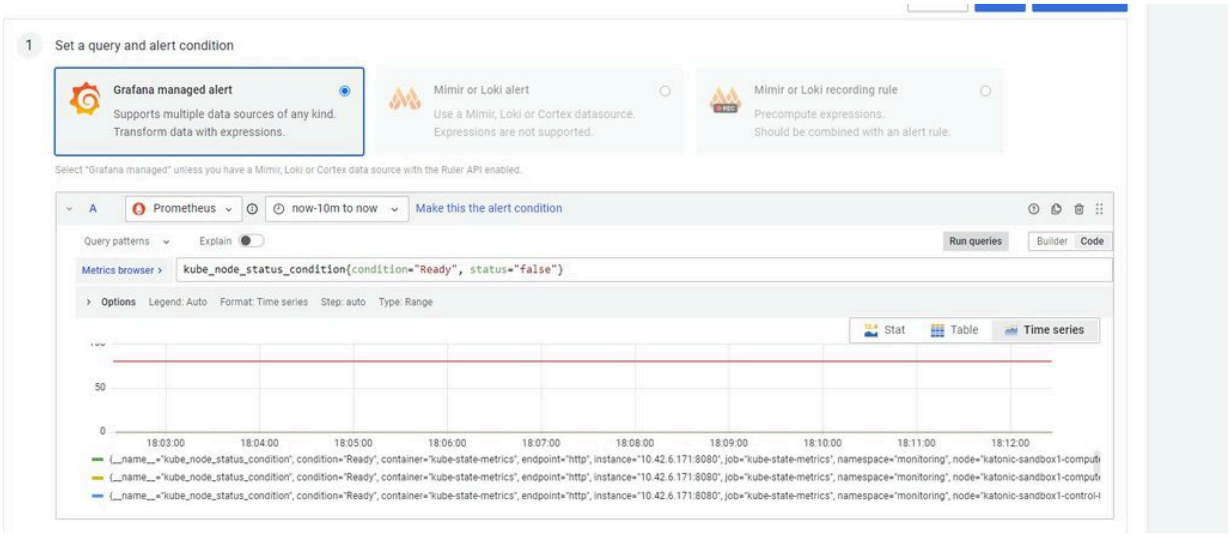
### Creating and Persisting Alerts in Grafana

- Now, we will demonstrate how to create an alert and ensure it persists in Grafana, using Node Alerting as an example. This process involves setting up a notification to alert us when a node is down.
- Step-by-Step Guide to Create and Persist Node Alerting

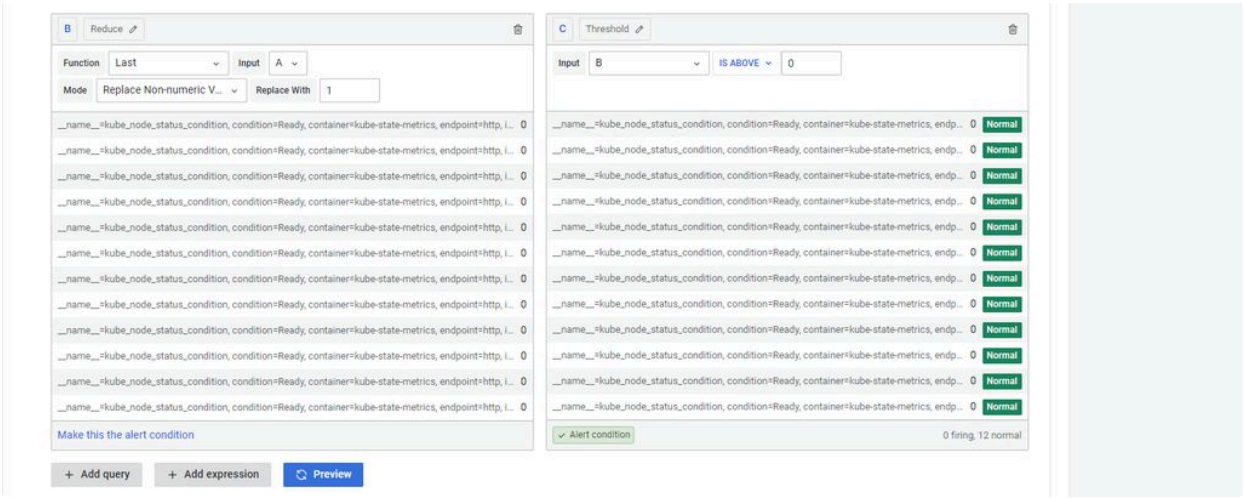
1. **Set Query and Alert Condition**:

- Navigate to the alerting section and select "Grafana Managed Alert".
- In Section A, set the query to monitor node status

```
kube_node_status_condition{condition="Ready", status="false"}
```




- In Section B, select the last value of the query in a 10-minute interval, replacing non-numeric values with 1.
- In Section C, set the threshold to trigger an alert when the value exceeds 0 (indicating a node is down). Click "Make this Alert condition".

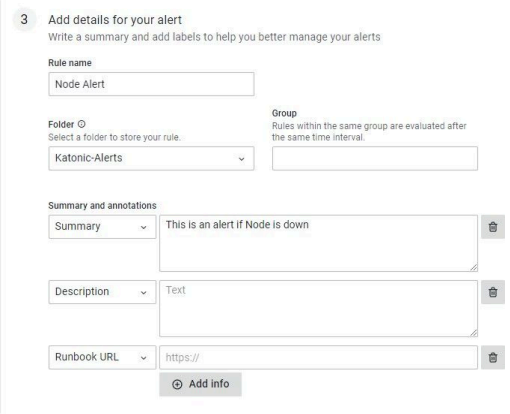


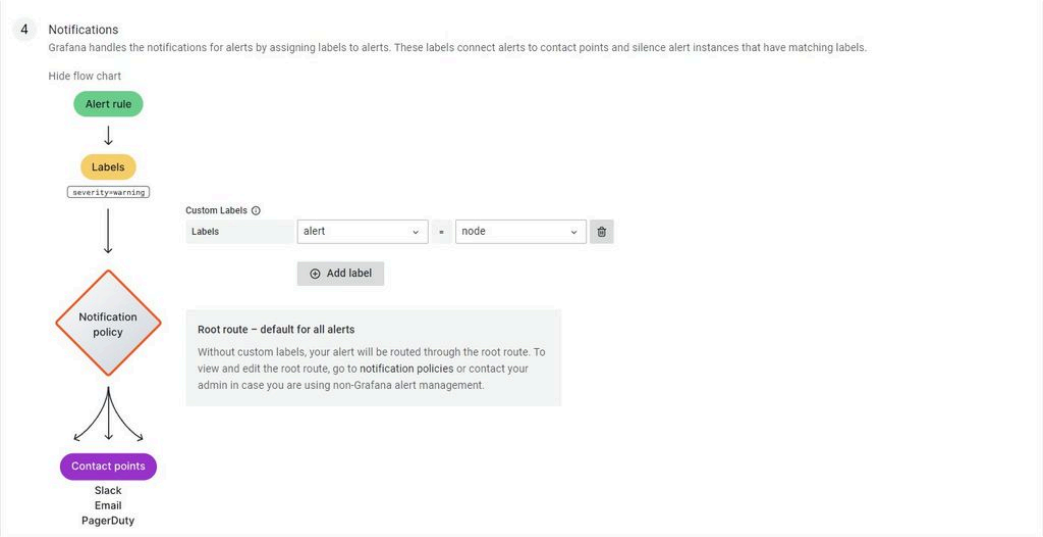

2. **Alert Evaluation**:

- The alert is evaluated every 5 minutes.
- If the condition persists for more than 15 minutes, the alert becomes a firing alert.


3. **Add Alert Details**:

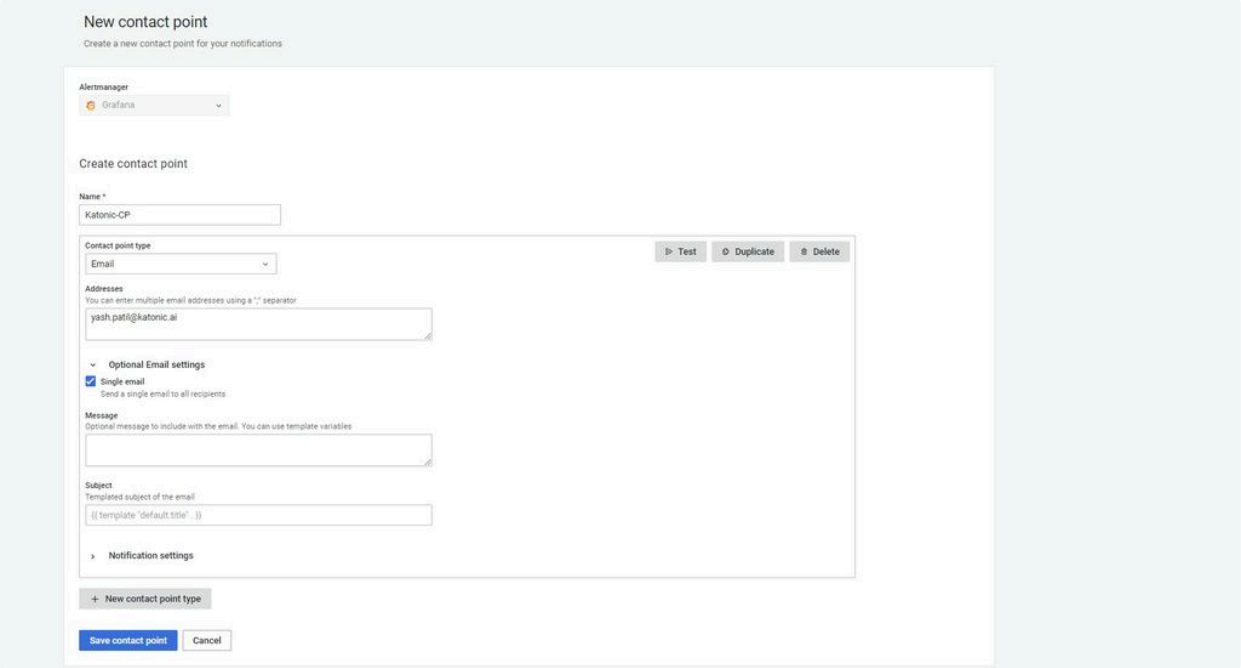
- Provide details such as rule name, folder, group, summary, and description.
- Assign labels to the alert and save.




4. **Create Contact Point**:

- Define whom to send alerts to and through which medium.
- Add the contact point in the `grafana-alerting` configmap.
- This is how we define contact point in `grafana-alerting` configmap.




```yaml
contactPoints:
  - orgId: 1
    name: katonic-email-point
    receivers:
      - uid: email
        type: email
        settings:
          addresses: santosh.shetkar@katonic.ai
```

5. **Set Notification Policy**:

- Define the notification policy in the `grafana-alerting` configmap:

```yaml
    policies:
      - orgId: 1
        receiver: katonic-email-point
        group_by: ['...']
        group_wait: 30s
        group_interval: 5m
        repeat_interval: 24h #<alert interval>
```

6. Persisting Alerts

- To ensure alerts persist even after Grafana restarts:

1. **Create API Key**:

- Generate an API key in Grafana configuration.

2. **Export Alert Configuration**:

- Use a curl request to export the alert configuration in JSON format.

3. **Convert JSON to YAML**:

- Convert the exported JSON to YAML format.

4. **Update ConfigMap**:

- Add the YAML configuration to the `grafana-alerting` configmap.

7. **Restart Grafana Deployment**:

```
kubectl rollout restart deployment/prometheus-stack-grafana -n monitoring

```

### Using Grafana Explorer

- The Grafana Explorer tab allows running Prometheus queries without accessing the Prometheus UI directly. This is useful for creating custom dashboards and verifying queries.

## Troubleshooting: Prometheus Pod Terminating and Restarting in Rackorp Clusters

- In Rackorp, we have two monitoring systems: Rancher Monitoring and our own Prometheus-stack-Grafana. This setup involves two namespaces: `cattle-monitoring-system` and `monitoring`, each with its own Prometheus Operator. A common issue arises because Prometheus Operators manage Prometheus Custom Resource Definitions (CRDs) across all namespaces, leading to conflicts. This guide explains how to configure each Prometheus Operator to monitor only its respective namespace, thereby avoiding conflicts and ensuring stable operation.

### Understanding the Issue

- **Namespaces**:
- `cattle-monitoring-system`: Contains Rancher Monitoring resources.
- `monitoring`: Contains our custom Prometheus-stack-Grafana resources.
- **Prometheus Operators**:
- Each namespace has its own Prometheus Operator to manage Prometheus CRDs.
- By default, a Prometheus Operator monitors Prometheus CRDs across all namespaces.
- Conflict
- Both Prometheus Operators detect Prometheus pods in both namespaces.
- If the Prometheus CRD in `cattle-monitoring-system` specifies `replicas: 1`, the Prometheus Operator may try to bring down the Prometheus pod in the `monitoring` namespace to maintain this configuration.

### Solution

- To resolve this issue, configure each Prometheus Operator to monitor only its respective namespace. This can be done using two approaches:

1. **Using `--namespaces` Argument**:

- Specify that the Prometheus Operator should monitor only its own namespace.

2. **Using `--deny-namespaces` Argument**:

- Specify that the Prometheus Operator should not monitor a specific namespace.

### Steps to Resolve the Issue

1. Configuring the Rancher Monitoring Operator

- Edit the deployment of the Rancher Monitoring Operator in the `cattle-monitoring-system` namespace to deny monitoring of the `monitoring` namespace:

```
kubectl edit deploy rancher-monitoring-operator -n cattle-monitoring-system
```

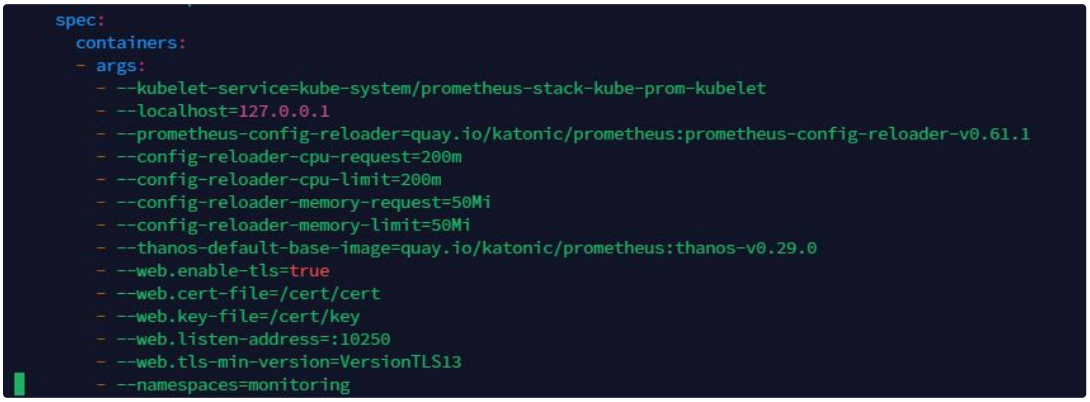
- Add the `--deny-namespaces=monitoring` argument to the container args. This ensures that the Prometheus Operator in the `cattle-monitoring-system` namespace does not monitor the Prometheus pod in the `monitoring` namespace.


2. Configuring the Prometheus Operator in the Monitoring Namespace

- Edit the deployment of the Prometheus Operator in the `monitoring` namespace to monitor only its own namespace:

```
kubectl edit deployment/prometheus-stack-kube-prom-operator -n monitoring
```

- Add the `--namespaces=monitoring` argument to the container args. This ensures that the Prometheus Operator in the `monitoring` namespace monitors only the Prometheus pod in its namespace.




### Verification and Troubleshooting

- **Check Pod Status**:
  - Ensure that the Prometheus pods in both namespaces are running without restarting or terminating issues.

## GPU Monitoring

- To monitor GPU nodes in your cluster and create a dashboard in Grafana, follow these steps. This guide outlines the process, including necessary configurations and importing a predefined dashboard.

Steps to Monitor GPU Nodes

1. **Verify GPU Nodes**:

- Ensure your GPU nodes are running:

```
kubectl get nodes
```

2. **Deploy NVIDIA DCGM Exporter**:

- The NVIDIA DCGM Exporter exposes GPU metrics. Ensure the `dcgm-exporter` pod is running in the `monitoring` namespace:

```
kubectl get pods -n monitoring
```

3. **Apply Configuration Files**:

- To deploy the DCGM Exporter, apply three configuration files: `daemonset.yaml`, `service.yaml`, and `servicemonitor.yaml`.
- DaemonSet Configuration (`daemonset.yaml`)

This file deploys the DCGM Exporter as a DaemonSet, ensuring it runs on each GPU node. It includes necessary settings such as image details, ports, environment variables, and volume mounts.

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: gpu-exporter-dcgm-exporter
  namespace: monitoring
  labels:
    app.kubernetes.io/name: dcgm-exporter
    app.kubernetes.io/version: 3.4.2
spec:
  updateStrategy:
    type: RollingUpdate
  selector:
    matchLabels:
      app.kubernetes.io/name: dcgm-exporter
      app.kubernetes.io/version: 3.4.2
  template:
    metadata:
      labels:
        app.kubernetes.io/name: dcgm-exporter
        app.kubernetes.io/version: 3.4.2
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: katonic.ai/node-pool
                operator: NotIn
                values:
                - platform
                - deployment
                - compute
      serviceAccountName: gpu-exporter-dcgm-exporter
      volumes:
```

```
      - name: "pod-gpu-resources"
        hostPath:
          path: "/var/lib/kubelet/pod-resources"
      - configMap:
          defaultMode: 420
          name: dcgm-metrics-cm
        name: dcgm-metrics
      tolerations:
      - effect: NoSchedule
        key: nvidia.com/gpu
        operator: Exists
      containers:
      - name: exporter
        securityContext:
          capabilities:
            add:
            - SYS_ADMIN
          runAsNonRoot: false
          runAsUser: 0
        image: "nvcr.io/nvidia/k8s/dcgm-exporter:3.3.6-3.4.2-ubuntu22.04"
        imagePullPolicy: "IfNotPresent"
        args:
        - -f
        - /etc/dcgm-exporter/dcp-metrics-included.csv
        env:
        - name: "DCGM_EXPORTER_KUBERNETES"
          value: "true"
        - name: "DCGM_EXPORTER_LISTEN"
          value: ":9400"
        ports:
        - name: "metrics"
          containerPort: 9400
        volumeMounts:
        - name: "pod-gpu-resources"
          readOnly: true
          mountPath: "/var/lib/kubelet/pod-resources"
        - mountPath: /etc/dcgm-exporter
          name: dcgm-metrics
        livenessProbe:
          httpGet:
            path: /health
            port: 9400
          initialDelaySeconds: 20
          periodSeconds: 5
        readinessProbe:
          httpGet:
            path: /health
            port: 9400
          initialDelaySeconds: 20
```

Service Configuration ( `service.yaml` )

- This file creates a service to expose the DCGM Exporter metrics within the cluster.

```
apiVersion: v1
kind: Service
metadata:
```

```
  name: gpu-exporter-dcgm-exporter
  namespace: monitoring
  labels:
    app.kubernetes.io/name: dcgm-exporter
    app.kubernetes.io/version: 3.4.2
spec:
  type: ClusterIP
  ports:
  - name: metrics
    port: 9400
    targetPort: 9400
    protocol: TCP
  selector:
    app.kubernetes.io/name: dcgm-exporter
    app.kubernetes.io/version: 3.4.2
```

ServiceMonitor Configuration (`servicemonitor.yaml`)

- This file creates a ServiceMonitor resource for Prometheus to scrape the DCGM Exporter metrics.

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: gpu-exporter-dcgm-exporter
  namespace: monitoring
  labels:
    app.kubernetes.io/name: dcgm-exporter
    release: prometheus-stack
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: dcgm-exporter
  namespaceSelector:
    matchNames:
    - "monitoring"
  endpoints:
  - port: metrics
    path: /metrics
    interval: 15s
    scheme: http
```

4. **Apply the Configuration Files**:

```
kubectl apply -f daemonset.yaml
kubectl apply -f service.yaml
kubectl apply -f servicemonitor.yaml
```

5. **Import GPU Monitoring Dashboard in Grafana**:

- In Grafana, import a predefined dashboard using ID `12239`. This dashboard provides GPU metrics visualization.

6. **Retrieve and Configure the Dashboard JSON**:

- After importing the dashboard, retrieve its JSON configuration and set the UID to `gpu-monitoring`. This provides a static URL for easy integration with the platform UI.

7. **Store Dashboard JSON in ConfigMap**:

- Store the dashboard JSON in a ConfigMap named `katonic-dashboard`

```
kubectl edit cm/katonic-dashboard -n monitoring
```

8. **Restart Grafana Deployment**:

- To apply the changes, restart the Grafana deployment

```
kubectl rollout restart deployment/prometheus-stack-grafana -n monitoring
```