Kubernetes Monitoring

Agenda

1. Vorbereitung
    - Self-Service Cluster ausrollen
2. Kubernetes Monitoring Grundlagen
    - Abgrenzung zu Observeability
3. Kubernetes Monitoring (Single Cluster / Instance of Prometheus)
    - Prometheus Monitoring Server (Overview)
    - Prometheus - Achtung bitte kein prometheus agent
    - Prometheus / Grafana Stack installieren (advanced)
    - Prometheus / blackbox exporter
4. Prometheus Praxis
    - Nginx mit ServiceMonitor und export konfigurieren (sidecar)
5. Grafana - Dashboards
    - Bestehendes Dashboard anpassen
    - Pod und Container Dashboard
6. Grafana - Alterting and Notifications
    - Grafana neuen alert anlegen
    - Grafana absence alert konfigurieren - d.h. Service hat keine Pods mehr
    - Grafana alert, >= pod aus replicaset nicht erreichbar
    - Grafana Notifications/Contact points
7. Kubernetes Multi-Cluster (Types of setups including disadvantags/advantages)
    - Recommended: Variant 1: prometheus agent + thanos/grafana stack
    - Variant 2: Full prometheus in each cluster with thanos sidecar
8. Grafana Loki
    - Installation von Grafana Loki - Single Instance - für Testing
    - Datasource in Grafana bereitstellen per helm
    - Wo finde ich Loki in Grafana ?

Backlog / Sammlung

1. Prometheus
    - Prometheus-Metriktypen (engl. metric types)
2. Kubernetes Multi-Cluster (using Thanos)
    - Prerequisites: What is Thanos
    - Components
    - Thanos Compactor
3. Kubernetes Multi-Cluster (using Cortex - multi-tenant tsdb's)

Vorbereitung

Self-Service Cluster ausrollen

- ausgerollt mit terraform (binary ist installiert) - snap install --classic terraform
- beinhaltet
    1. 1 controlplane
    2. 2 worker nodes
    3. metallb mit ips der Nodes (hacky but works)
    4. ingress mit wildcard-domain: *.tlnx.do.t3isp.de

Walktrough

```
cd
git clone https://github.com/jmetzger/training-kubernets-monitoring-stack-do-terraform.git install
cd install
cat /tmp/.env
source /tmp/.env
terraform init
terraform apply -auto-approve
```

Hinweis

```
## Sollte es nicht sauber durchlaufen
## einfach nochmal
terraform apply -auto-approve

## Wenn das nicht geht, einfach nochmal neu
terraform destroy -auto-approve
terraform apply -auto-approve
```

Kubernetes Monitoring Grundlagen

Abgrenzung zu Observeability

Der Unterschied zwischen **Kubernetes Monitoring** und **Observability** liegt vor allem im **Ziel**, **Umfang** und **Vorgehen**. Hier eine klare Gegenüberstellung:

Monitoring (Beobachtung)

Definition: Monitoring ist das **Sammeln und Anzeigen** von Metriken und Logs, um den **Zustand eines Systems zu überwachen**.

Merkmale:

- Fokus auf bekannte Metriken, Fehler und Schwellenwerte
- Alarme bei bekannten Problemen (z. B. CPU-Auslastung > 90 %)
- Reaktiv: Man erkennt, *dass* etwas nicht stimmt

Beispiele:

- CPU-, RAM-, Netzwerkverbrauch von Pods
- Anzahl von HTTP 500 Errors
- Alerts bei Pod-Crashes

Werkzeuge:

- Prometheus
- Grafana
- Alertmanager
- Metrics Server

Observability (Beobachtbarkeit)

Definition: Observability beschreibt die **Fähigkeit**, den **internen Zustand** eines Systems allein durch seine externen Outputs (Logs, Metriken, Traces) **verstehen** zu können.

Merkmale:

- Ermöglicht Ursachenanalyse auch für **unbekannte Probleme**
- Proaktiv: Man kann *warum* etwas passiert ist, herausfinden
- Nutzt drei Hauptsäulen:
  - **Metriken** (z. B. Requests/sec)
  - **Logs** (z. B. Stacktraces)
  - **Traces** (z. B. verteilte Aufrufe zwischen Microservices)

Beispiele:

- Warum ist ein Request langsam? (Trace-Analyse)
- Welche Kette von Services war beteiligt?
- Korrelation von Logs mit Metrik-Anomalien

Werkzeuge:

- OpenTelemetry
- Grafana Tempo / Jaeger (für Tracing)
- Loki (für Logs)
- Elastic Stack
- Honeycomb, Lightstep

⚖ Vergleich zusammengefasst

Aspekt	Monitoring	Observability
Ziel	Systemzustand überwachen	Systemverhalten verstehen
Fokus	Bekannte Probleme erkennen	Ursachen auch unbekannter Probleme analysieren
Methoden	Metriken, Schwellenwerte, Alarme	Logs, Metriken, Traces kombiniert
Reaktiv / Proaktiv	Reaktiv	Proaktiv / Diagnostisch
Typische Fragen	Ist etwas kaputt?	Warum ist etwas kaputt?

Kubernetes Monitoring (Single Cluster / Instance of Prometheus)

Prometheus Monitoring Server (Overview)

What does it do ?

- It monitors your system by collecting data
- Data is pulled from your system by defined endpoints (http) from your cluster
- To provide data on your system, a lot of exporters are available, that
  - collect the data and provide it in Prometheus

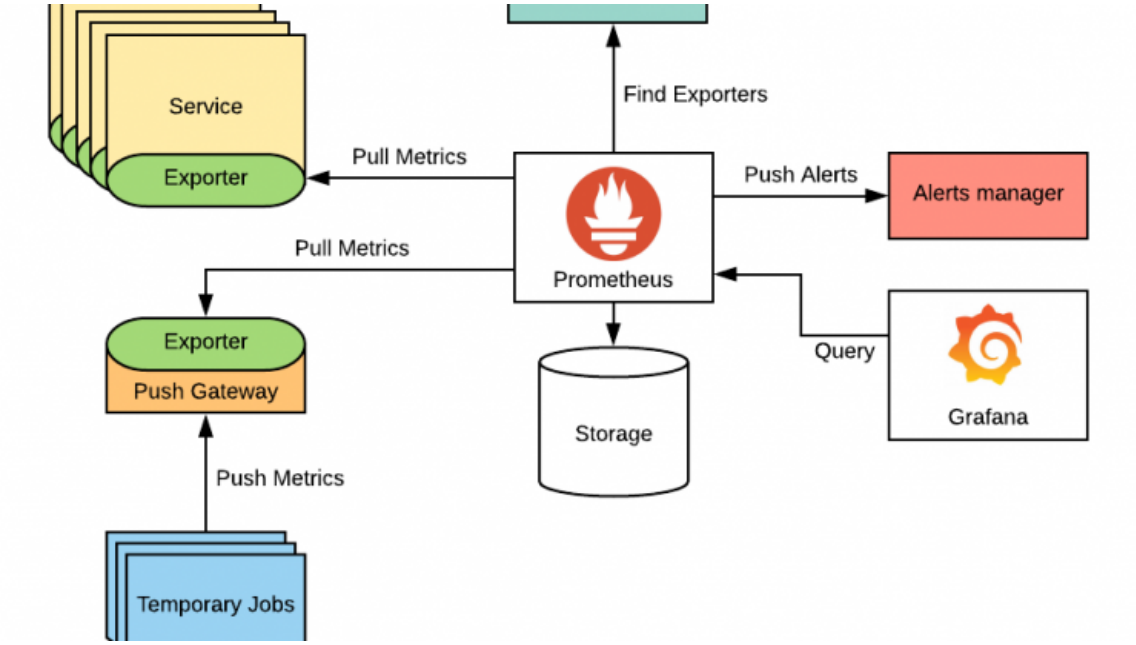
Technical

- Prometheus has a TDB (Time Series Database) and is good as storing time series with data
- Prometheus includes a local on-disk time series database, but also optionally integrates with remote storage systems.
- Prometheus's local time series database stores data in a custom, highly efficient format on local storage.
- Ref: https://prometheus.io/docs/prometheus/latest/storage/

What are time series ?

- A time series is a sequence of data points that occur in successive order over some period of time.
- Beispiel:
  - Du willst die täglichen Schlusspreise für eine Aktie für ein Jahr dokumentieren
  - Damit willst Du weitere Analysen machen
  - Du würdest das Paar Datum/Preis dann in der Datumsreihenfolge sortieren und so ausgeben
  - Dies wäre eine "time series"

Kompenenten von Prometheus



Quelle: https://www.devopsschool.com/

Prometheus Server

1. Retrieval (Sammeln)
   - Data Retrieval Worker
     - pull metrics data
2. Storage
   - Time Series Database (TDB)
     - stores metrics data
3. HTTP Server
   - Accepts PromQL - Queries (e.g. from Grafana)
     - accept queries

Grafana ?

- Grafana wird meist verwendet um die grafische Auswertung zu machen.
- Mit Grafana kann ich einfach Dashboards verwenden
- Ich kann sehr leicht festlegen (Durch Data Sources), wo meine Daten herkommen

Prometheus - Achtung bitte kein prometheus agent

Warum ?

- Coole Objekte wie PodMonitor, ServiceMonitor, PrometheusRules funktionieren
- Das ist schlecht und macht Dein unnötig schwer.
- Dann musst du nämlich die alten ScrapeConfigs verwenden (IHHHHH !)

Prometheus / Grafana Stack installieren (advanced)

- using the kube-prometheus-stack (recommended !: includes important metrics)

Attention: Upgrades and uninstall can be a bit tricky

- CRD's need to deleted manually after uninstall
- Before Upgrade update the CRD's
- https://github.com/prometheus-community/helm-charts/blob/main/charts/kube-prometheus-stack/UPGRADE.md

What do we want to do ?

- We want to protect prometheus with basic-auth
- We want to protect alertmanager with basic-auth
- We want to use letsencrypt

Prerequisites

```
## 1. With have setup ingress-controller Service type:LoadBalancer -> external
## 2. We have a subdomain
## 3. Already done for you
sudo apt install apache2-utils
```

Step 1: Create our project - folder (just to be organized)

```
cd
mkdir -p manifests
cd manifests
mkdir -p monitoring
cd monitoring
```

Step 2: Create basic-auth

```
kubectl create ns monitoring
htpasswd -c auth admin  # Enter your desired password
kubectl create secret generic prometheus-basic-auth --from-file=auth -n monitoring
```

Step 3: Install cert-manager

```
helm repo add jetstack https://charts.jetstack.io
```

```
nano cert-manager-values.yml
```

```
crds:
  enabled: true
```

```
helm upgrade --install cert-manager jetstack/cert-manager \
  --namespace cert-manager --create-namespace --version 1.17.2 -f cert-manager-values.yml
```

Step 4: Create ClusterIssuer

```
nano clusterissuer.yaml
```

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
spec:
```

```
  acme:
    email: training.tn1@t3company.de
    server: https://acme-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      name: letsencrypt-prod
    solvers:
      - http01:
          ingress:
            class: nginx
```

```
kubectl apply -f clusterissuer.yaml
```

Step 5: Prepare Monitoring Stack (values - file)

```
nano monitoring-values.yaml
```

```
grafana:
  fullnameOverride: grafana
  enabled: true
  adminUser: admin
  adminPassword: "yourStrongPassword"
  ingress:
    enabled: true
    annotations:
      kubernetes.io/ingress.class: nginx
      cert-manager.io/cluster-issuer: letsencrypt-prod
    hosts:
      - grafana.<du>.do.t3isp.de
    path: /
    pathType: Prefix
    tls:
      - hosts:
          - grafana.<du>.do.t3isp.de
        secretName: grafana-tls

prometheus:
  fullnameOverride: prometheus
  ingress:
    enabled: true
    annotations:
      kubernetes.io/ingress.class: nginx
      nginx.ingress.kubernetes.io/auth-type: basic
      nginx.ingress.kubernetes.io/auth-secret: prometheus-basic-auth
      nginx.ingress.kubernetes.io/auth-realm: "Authentication Required"
      cert-manager.io/cluster-issuer: letsencrypt-prod
    hosts:
      - prometheus.<du>.do.t3isp.de
    paths:
      - /
    pathType: Prefix
    tls:
      - hosts:
          - prometheus.<du>.do.t3isp.de
        secretName: prometheus-tls

## Optional: Persist data
prometheusOperator:
  admissionWebhooks:
```

```
    enabled: true

alertmanager:
  fullnameOverride: alertmanager
  ingress:
    enabled: true
    annotations:
      kubernetes.io/ingress.class: nginx
      nginx.ingress.kubernetes.io/auth-type: basic
      nginx.ingress.kubernetes.io/auth-secret: prometheus-basic-auth
      nginx.ingress.kubernetes.io/auth-realm: "Authentication Required"
      cert-manager.io/cluster-issuer: letsencrypt-prod
    hosts:
      - alertmanager.<du>.do.t3isp.de
    paths:
      - /
    pathType: Prefix
    tls:
      - hosts:
          - alertmanager.<du>.do.t3isp.de
        secretName: alertmanager-tls


kube-state-metrics:
  fullnameOverride: kube-state-metrics

prometheus-node-exporter:
  fullnameOverride: node-exporter
```

Step 6: Install with helm

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm upgrade --install prometheus prometheus-community/kube-prometheus-stack -f monitoring-
values.yaml --namespace monitoring --create-namespace --version 72.3.0
```

Step 6.5 Check, if everything works

```
kubectl -n monitoring get pods
kubectl -n cert-manager get pods
```

```
## ein neue Ressource cert-manager
## True ?
kubectl get clusterissuer
kubectl -n monitoring get certicaterequests
## Alertmanager has a problem
kubectl -n monitoring describe certificaterequests alertmanager-tls-1


kubectl -n monitoring get certificates
kubectl -n monitoring describe cert alertmanager-tls
```

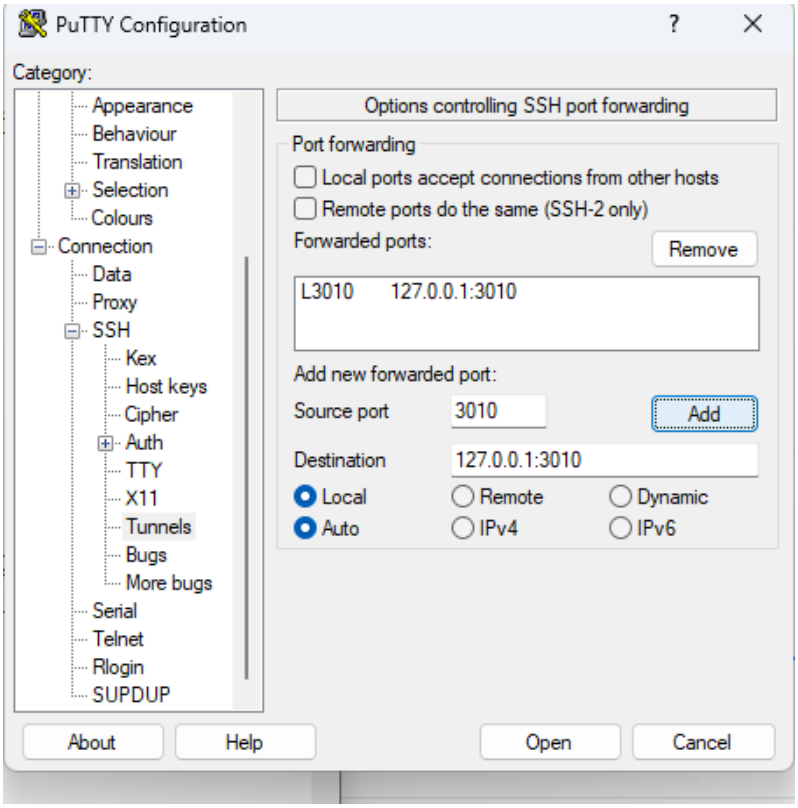
Step 7: Connect to prometheus from the outside world

```
https://prometheus.<du>.do.t3isp.de
```

Step 8: Connect to the grafana from the outside world

```
https://grafana.<du>.do.t3isp.de
```

```
## ändern in euer port + teilnehmer
## d.h. z.B. 3000 + tln1 = 3001 statt 3010
kubectl -n monitoring port-forward deploy/grafana 3010:3000 &
## if on remote - system do a ssh-tunnel
## ssh -L 3010:127.0.0.1:3010 user@remote-ip
```



Step 9: Connect to alertmanager from the outside world

```
https://alertmanager.<du>.do.t3isp.de
```

Attention: No persistent storage

- In this chart prometheus by default uses EmptyDir, only exists as long as pod runs
- Retention time: is 10d currenty, so this long will data be there

```
  prometheus-prometheus-kube-prometheus-prometheus-db:
    Type:       EmptyDir (a temporary directory that shares a pod's lifetime)
    Medium:
    SizeLimit:  <unset>
```

Set to storageclass

```
nano monitoring-values.yaml
```

```
grafana:
  fullnameOverride: grafana
  enabled: true
```

```
  adminUser: admin
  adminPassword: "yourStrongPassword"
  ingress:
    enabled: true
    annotations:
      kubernetes.io/ingress.class: nginx
      cert-manager.io/cluster-issuer: letsencrypt-prod
    hosts:
      - grafana.<du>.do.t3isp.de
    path: /
    pathType: Prefix
    tls:
      - hosts:
          - grafana.<du>.do.t3isp.de
        secretName: grafana-tls

prometheus:
  fullnameOverride: prometheus
### That is the storageclass part
  prometheusSpec:
    storageSpec:
      volumeClaimTemplate:
        spec:
          accessModes: ["ReadWriteOnce"]
          resources:
            requests:
              storage: 20Gi
          storageClassName: "standard"
#######
  ingress:
    enabled: true
    annotations:
      kubernetes.io/ingress.class: nginx
      nginx.ingress.kubernetes.io/auth-type: basic
      nginx.ingress.kubernetes.io/auth-secret: prometheus-basic-auth
      nginx.ingress.kubernetes.io/auth-realm: "Authentication Required"
      cert-manager.io/cluster-issuer: letsencrypt-prod
    hosts:
      - prometheus.<du>.do.t3isp.de
    paths:
      - /
    pathType: Prefix
    tls:
      - hosts:
          - prometheus.<du>.do.t3isp.de
        secretName: prometheus-tls

## Optional: Persist data
prometheusOperator:
  admissionWebhooks:
    enabled: true

alertmanager:
  fullnameOverride: alertmanager
  ingress:
    enabled: true
    annotations:
      kubernetes.io/ingress.class: nginx
      nginx.ingress.kubernetes.io/auth-type: basic
```

```
      nginx.ingress.kubernetes.io/auth-secret: prometheus-basic-auth
      nginx.ingress.kubernetes.io/auth-realm: "Authentication Required"
      cert-manager.io/cluster-issuer: letsencrypt-prod
    hosts:
      - alertmanager.<du>.do.t3isp.de
    paths:
      - /
    pathType: Prefix
    tls:
      - hosts:
          - alertmanager.<du>.t3isp.de
        secretName: alertmanager-tls


kube-state-metrics:
  fullnameOverride: kube-state-metrics

prometheus-node-exporter:
  fullnameOverride: node-exporter
```

```
## ausrollen
helm upgrade --install prometheus prometheus-community/kube-prometheus-stack -f monitoring-
values.yaml --namespace monitoring --create-namespace --version 72.3.0
```

References:

- https://github.com/prometheus-community/helm-charts/blob/main/charts/kube-prometheus-stack/README.md
- https://artifacthub.io/packages/helm/prometheus-community/prometheus

Prometheus / blackbox exporter

Prerequisites

- prometheus setup with helm

Step 1: Setup

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm install my-prometheus-blackbox-exporter prometheus-community/prometheus-blackbox-exporter --
version 8.17.0 --namespace monitoring --create-namespace
```

Step 2: Find SVC

```
kubectl -n monitoring get svc | grep blackbox
```

```
my-prometheus-blackbox-exporter   ClusterIP   10.245.183.66    <none>        9115/TCP
```

Step 3: Test with Curl

```
kubectl run -it --rm curltest --image=curlimages/curl -- sh
```

```
## Testen nach google in shell von curl
curl http://my-prometheus-blackbox-exporter.monitoring:9115/probe?
target=google.com&module=http_2xx
```

```
## Looking for metric
probe_http_status_code 200
```

Step 4: Test apple-service with Curl

```
## From within curlimages/curl pod
curl http://my-prometheus-blackbox-exporter.monitoring:9115/probe?target=apple-
service.app&module=http_2xx
```

Step 5: Scrape Config (We want to get all services being labeled example.io/should_be_probed = true

```
prometheus:
  prometheusSpec:
    additionalScrapeConfigs:
    - job_name: "blackbox-microservices"
      metrics_path: /probe
      params:
        module: [http_2xx]
      # Autodiscovery through kube-api-server
      #
https://prometheus.io/docs/prometheus/latest/configuration/configuration/#kubernetes_sd_config
      kubernetes_sd_configs:
      - role: service
      relabel_configs:
        # Example relabel to probe only some services that have "example.io/should_be_probed =
true" annotation
        - source_labels: [__meta_kubernetes_service_annotation_example_io_should_be_probed]
          action: keep
          regex: true
        - source_labels: [__address__]
          target_label: __param_target
        - target_label: __address__
          replacement:  my-prometheus-blackbox-exporter:9115
        - source_labels: [__param_target]
          target_label: instance
        - action: labelmap
          regex: __meta_kubernetes_service_label_(.+)
        - source_labels: [__meta_kubernetes_namespace]
          target_label: app
        - source_labels: [__meta_kubernetes_service_name]
          target_label: kubernetes_service_name
```

Step 6: Test with relabeler

- https://relabeler.promlabs.com

```

```

Step 7: Scrapeconfig einbauen

```
## von kube-prometheus-grafana in values und ugraden
 helm upgrade prometheus prometheus-community/kube-prometheus-stack -f values.yml --namespace
monitoring --create-namespace --version 61.3.1
```

Step 8: annotation in service einfügen

```
kind: Service
apiVersion: v1
metadata:
  name: apple-service
```

```
  annotations:
    example.io/should_be_probed: "true"

spec:
  selector:
    app: apple
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5678 # Default port for image
```

```
kubectl apply -f service.yml
```

Step 9: Look into Status -> Discovery Services and wait

- blackbox services should now appear under blackbox_microservices
- and not being dropped

Step 10: Unter http://64.227.125.201:30090/targets?search= gucken

- .. ob das funktioniert

Step 11: Hauptseite (status code 200)

- Metrik angekommen `?
- http://64.227.125.201:30090/graph?g0.expr=probe_http_status_code&g0.tab=1&g0.display_mode=lines&g0.show_exemplars=0&g0.range_input=1h

Step 12: pod vom service stoppen

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apple-deployment
spec:
  selector:
    matchLabels:
      app: apple
  replicas: 8
  template:
    metadata:
      labels:
        app: apple
    spec:
      containers:
      - name: apple-app
        image: hashicorp/http-echo
        args:
        - "-text=apple-<dein-name>"
```

```
kubectl apply -f apple.yml # (deployment)
```

Step 13: status_code 0

- Metrik angekommen `?
- http://64.227.125.201:30090/graph?g0.expr=probe_http_status_code&g0.tab=1&g0.display_mode=lines&g0.show_exemplars=0&g0.range_input=1h

Prometheus Praxis

Nginx mit ServiceMonitor und export konfigurieren (sidecar)

Voraussetzung:

- kube-prometheus-stack muss installiert sein -> Kube-Prometheus-Stack installieren

Vorbereitung: Verzeichnisstruktur anlegen

```
cd ~
mkdir -p manifests
cd manifests
mkdir svcm-nginx
cd svcm-nginx
```

> *Alle YAML-Dateien werden in diesem Verzeichnis erstellt und mit* `kubectl apply -f .` *angewendet.*

---

1. Namespace

```
nano 01-namespace.yaml
```

```
apiVersion: v1
kind: Namespace
metadata:
  name: web-demo
```

```
kubectl apply -f .
```

---

2. ConfigMap: Stub Status aktivieren

```
nano 02-nginx-stubstatus-configmap.yaml
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-stubstatus
  namespace: web-demo
data:
  default.conf: |
    server {
      listen 80;
      location / {
        root /usr/share/nginx/html;
        index index.html index.htm;
      }

      location /stub_status {
        stub_status;
        allow 127.0.0.1;
        deny all;
      }
    }
```

```
kubectl apply -f .
```

---

3. Deployment mit Sidecar (Exporter)

```
nano 03-nginx-deployment-metrics.yaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  namespace: web-demo
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:stable
        ports:
        - containerPort: 80
        volumeMounts:
        - name: nginx-conf
          mountPath: /etc/nginx/conf.d/default.conf
          subPath: default.conf
      - name: exporter
        image: nginx/nginx-prometheus-exporter:latest
        args:
        - "-nginx.scrape-uri=http://localhost:80/stub_status"
        ports:
        - containerPort: 9113
      volumes:
      - name: nginx-conf
        configMap:
          name: nginx-stubstatus
```

```
kubectl apply -f .
```

---

4. Service mit zusätzlichem Metrics-Port

```
nano 04-nginx-service-with-metrics.yaml
```

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: web-demo
  labels:
    app: nginx
spec:
```

```
  selector:
    app: nginx
  ports:
  - name: http
    port: 80
    targetPort: 80
  - name: metrics
    port: 9113
    targetPort: 9113
```

```
kubectl apply -f .
```

5. Verbindung testen

```
kubectl run -it --rm podtest --image=busybox
```

```
## in der bash
wget -O - http://nginx.web-demo:9113/metrics
exit
```

6. Ingress (Optional)

```
nano 06-nginx-ingress.yaml
```

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx
  namespace: web-demo
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
  - host: app.tln1.do.t3isp.de
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: nginx
            port:
              number: 80
```

```
kubectl apply -f .
```

7. ServiceMonitor

```
nano 05-nginx-servicemonitor.yaml
```

```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: nginx
  namespace: web-demo
  labels:
    release: prometheus  # muss zu Helm-Werten passen!
spec:
  selector:
    matchLabels:
      app: nginx
  namespaceSelector:
    matchNames:
    - web-demo
  endpoints:
  - port: metrics
    path: /metrics
    interval: 15s
```

```
kubectl apply -f .
```

```
## Welches Label prometheus hat, könnt ihr prüfen
kubectl -n monitoring get pods -l release=prometheus
```

```
## Ist der ServiceMonitor konfiguriert ?
kubectl -n web-demo get servicemonitor nginx
kubectl -n web-demo get smon nginx
kubectl -n web-demo describe smon nginx
```

8. Targets finden (in web gui)

```
## im Browser öffnen und nach web-demn suchen
https://prometheus.<du>.do.t3isp.de/targets

## Dann menü links oben ausklappen, ganz runter scrollen
## serviceMonitor/web-demo/nginx/0
## oder
https://prometheus.tln10.do.t3isp.de/targets?pool=serviceMonitor%2Fweb-demo%2Fnginx%2F0
```

9. mit promql abfragen

```
1. Zunächst finden wir heraus, welche labels diese pods haben (siehe Punkt 8)
das sieht nach job="nginx" aus

Jeder ServiceMonitor (z.B. unser, der nginx heisst), wird beim Scrapen als job="
<serviceMonitorName>"
automatisch von Kubernetes abgefragt.
```

```
d.h. wir können Fragen

## (gilt dann für alle pods)
up
## gilt für alle pods in einen für bestimmten job
up {job="nginx"}
## gilt für alle pods in einem bestimmten namespace
up {namespace="web-demo"}
```

```
## and combining all endpoints with job=nginx in the namespace web-demo
up {job="nginx",namespace="web-demo"}
```

```
## Regular Expressions also work:
up{namespace="web-demo", pod=~"nginx-.*"}
```

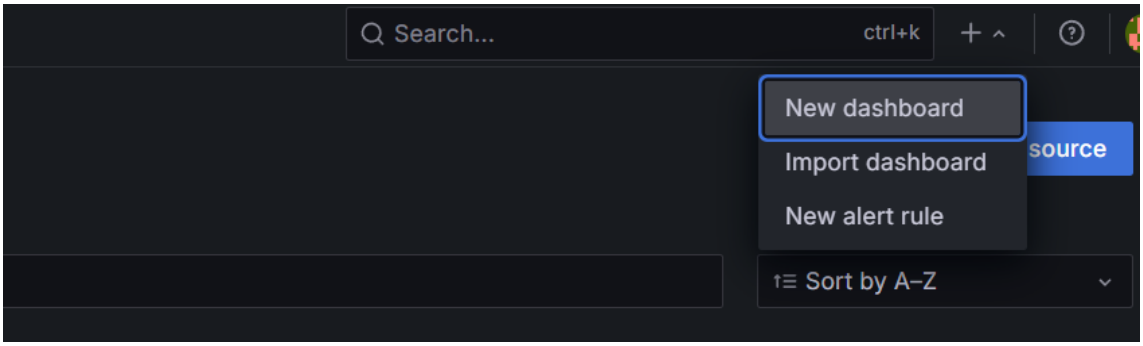
```
##Pratical - on: https://prometheus1.tln1.do.t3isp.de/query
+ enter:
up {job="nginx"} + Press "Execute"
```


```
You can not also click on Graph
```

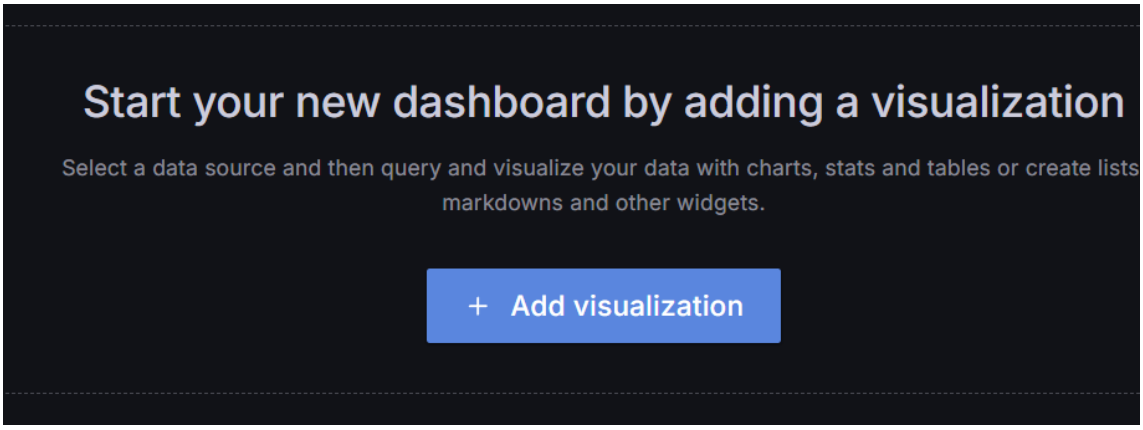
10. In Grafana ein Dashboard erstellen

Step 1: New Dashboard

```
Oben rechts auf neues Dashboard erstellen klicken
-->
```

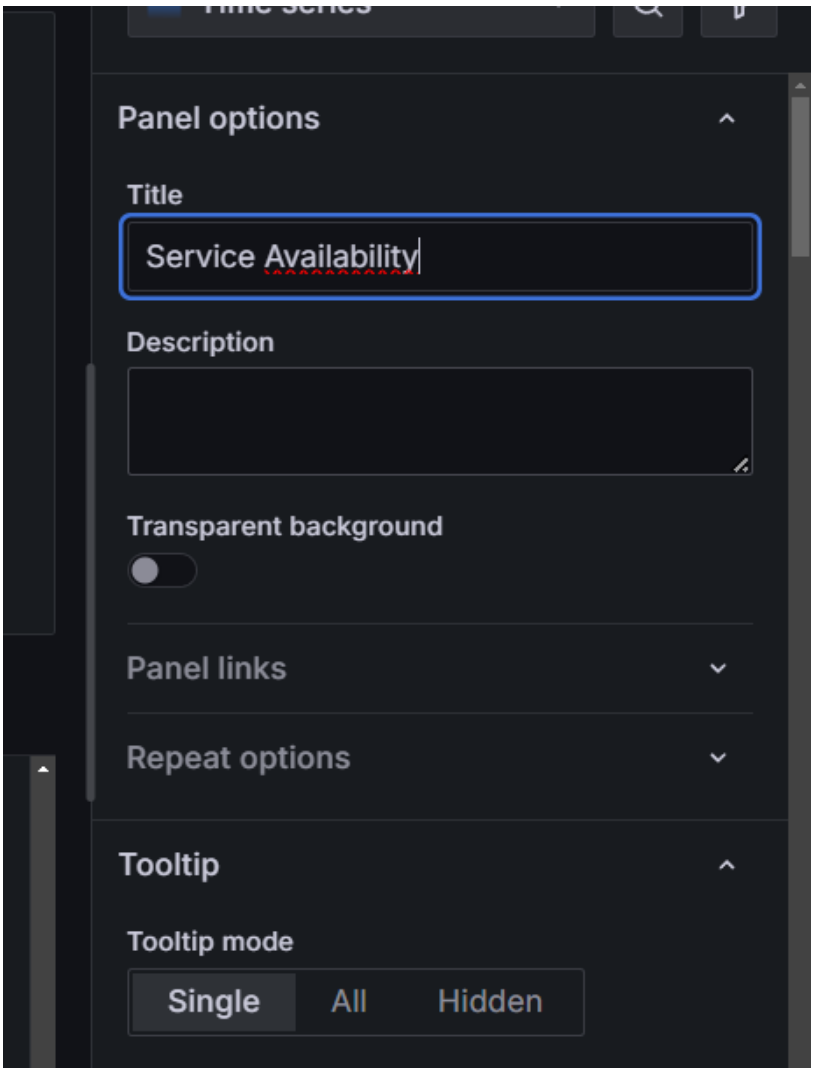


Step 2: Add Visualization



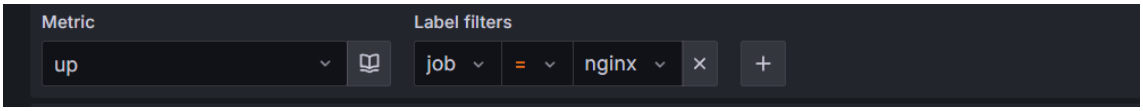
Step 3: Datasource -> Prometheus (Default) auswählen / Visualisation + Query definieren

- DataSource Prometheus is bereits vorkonfiguriert

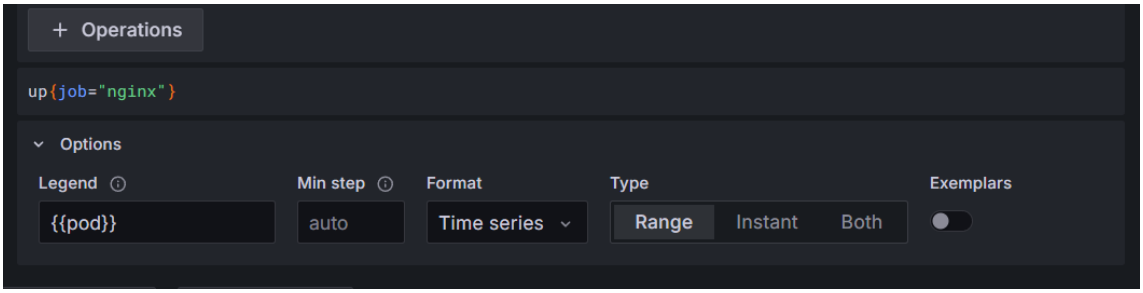


- Choose Visualisation (Stat, Gauge, or Bar Gauge)
- Set the query: up | job | nginx

- run query



- Damit immer der pod angezeigt wird, trage dies als custom label unter der query ein



Step 4: Save Dashboard (Button oben rechts)

Grafana - Dashboards

Bestehendes Dashboard anpassen

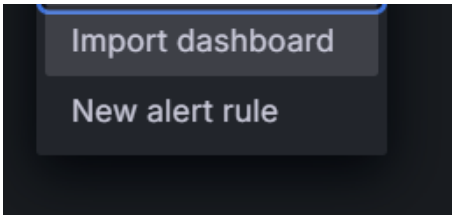
Schritt 1: bestehendes Dashboard (provisioned) - exportieren bzw. importieren

- Dashboard (Node Exporter -> Nodes) aufrufen
- Oben rechts: Export -> Export as JSON

Schritt 2: Neues Dashboard durch import erstellen

1. Oben rechts: + Zeichen: Import Dashboard

2. Copy to clipboard
3. - und import dashboard



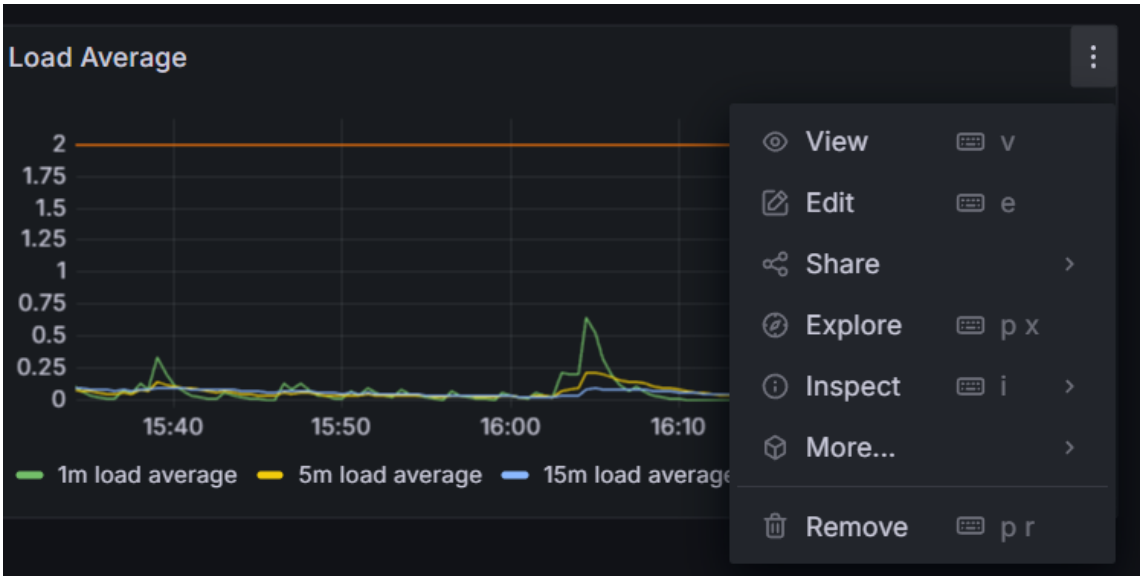
Schritt 3: Dashboard speichern

- Achtung: Beim Speichern des Dashboards anderen Namen und andere ID angeben (einfach die uid um +1 hochzählen), vorher: change uid anklicken
- dann **Import** - Button klicken

Schritt 4: Dashboard ausdünnen (alle nicht benötigte Panels raus)

- Wir löschen alles ausser das CPU - Panel

1. Oben rechts neben **Settings** auf **Edit** klicken
2. Neben dem jeweiligen Panel auf die DREI-PUNKTE klicken:

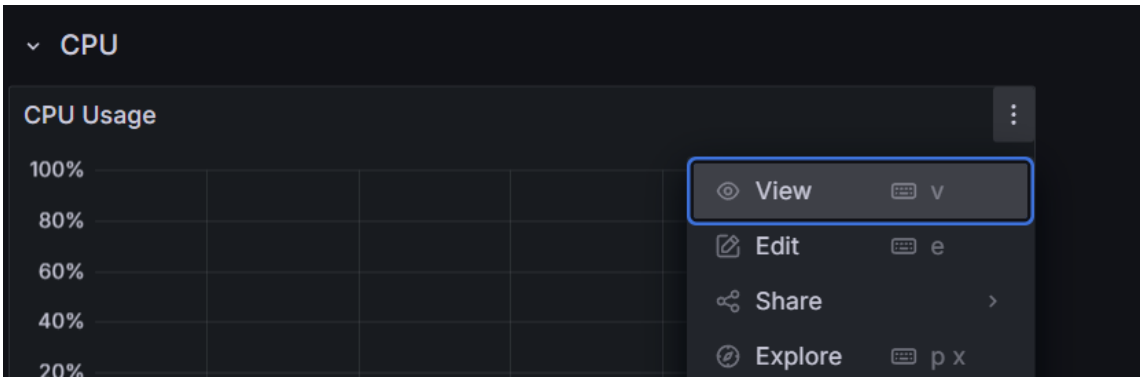


1. Auf Remove klicken
2. Alle nicht benötigen Rows löschen (Neben der Row auf den Papierkorb)

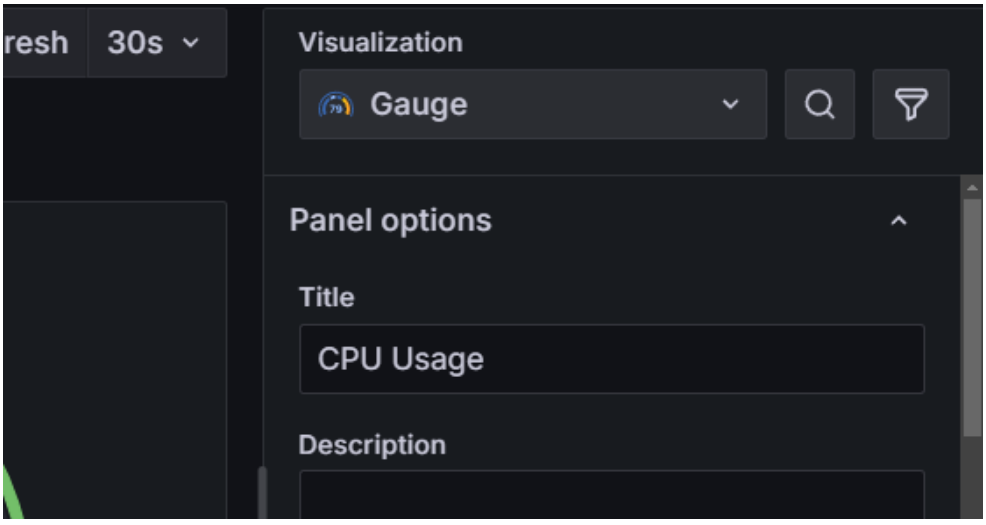
Schritt 5: Visualisierung von CPU ändern auf Graph

- Vorher:

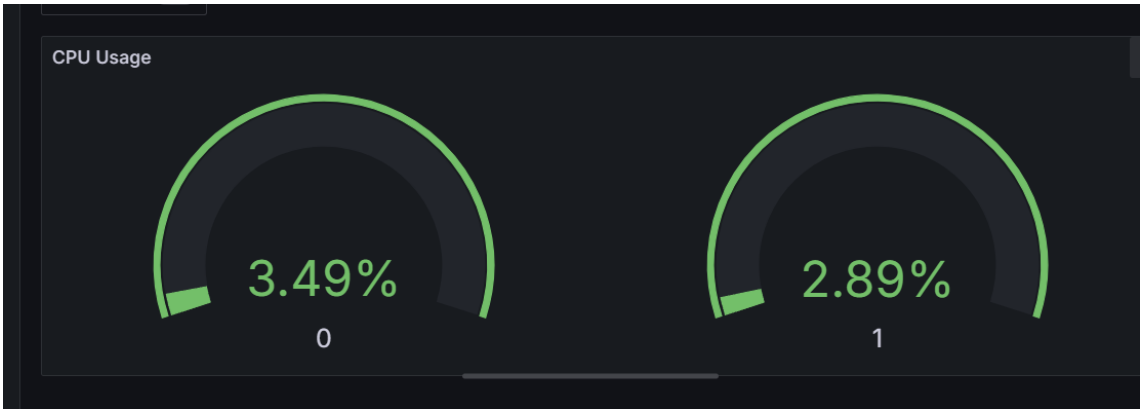
- dann: rechts auf die 3 Punkte oben rechts edit



- Dann Visualisierung ändern in ->

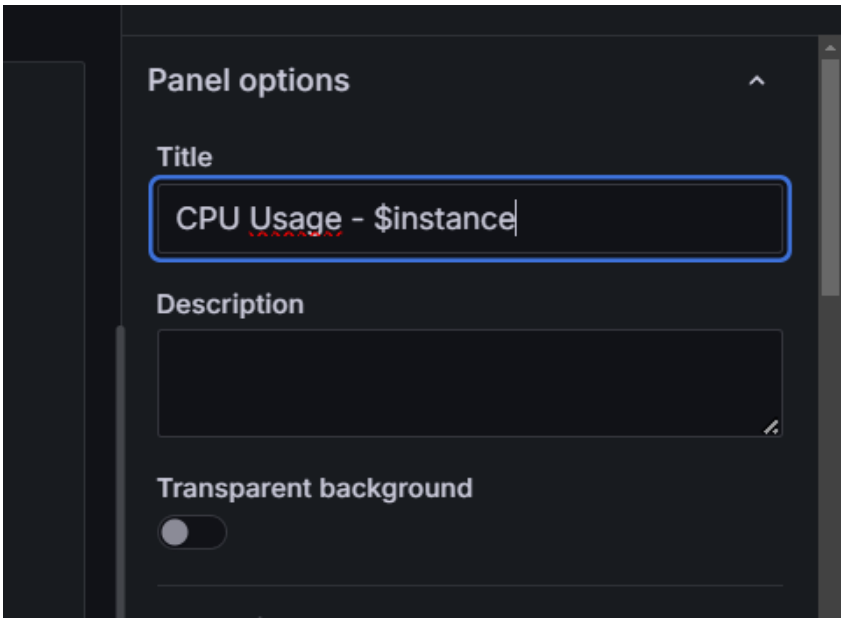


- Nachher:

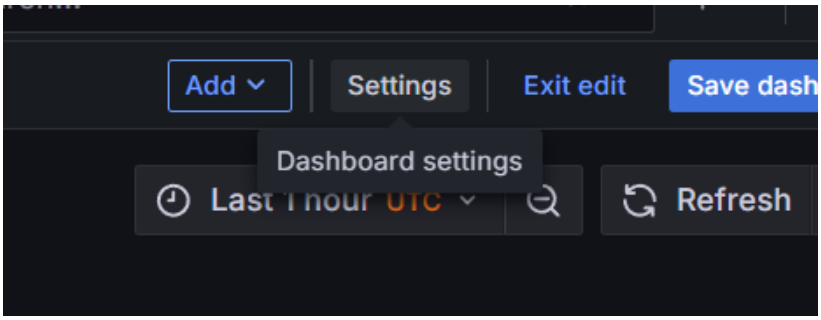


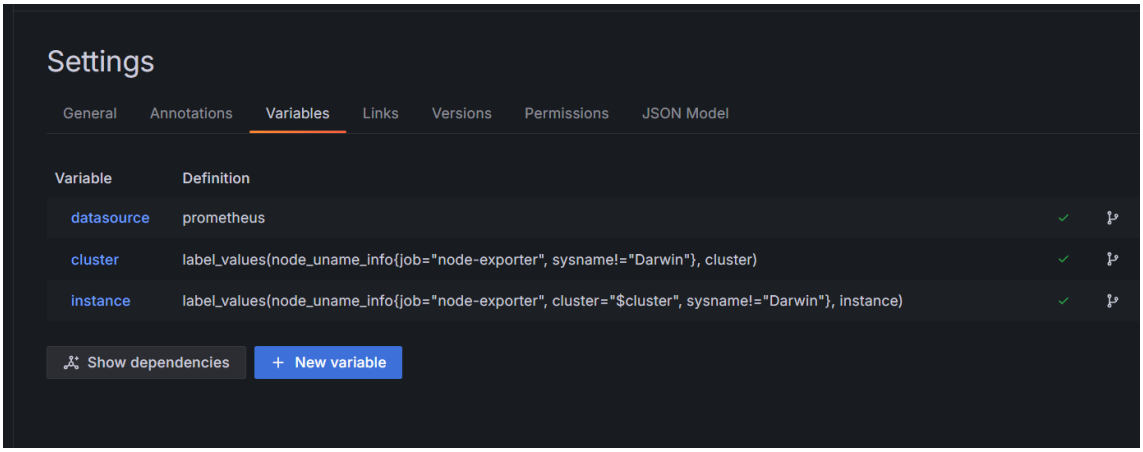
Schritt 6: Titel auf dynamisch ändern

- Wir setzen hier die Variable $instance ein



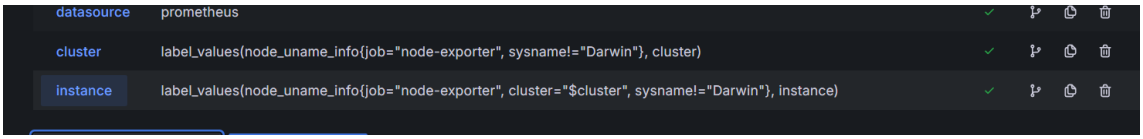
- Diese wurde bereits unter Settings -> Variablen definiert.



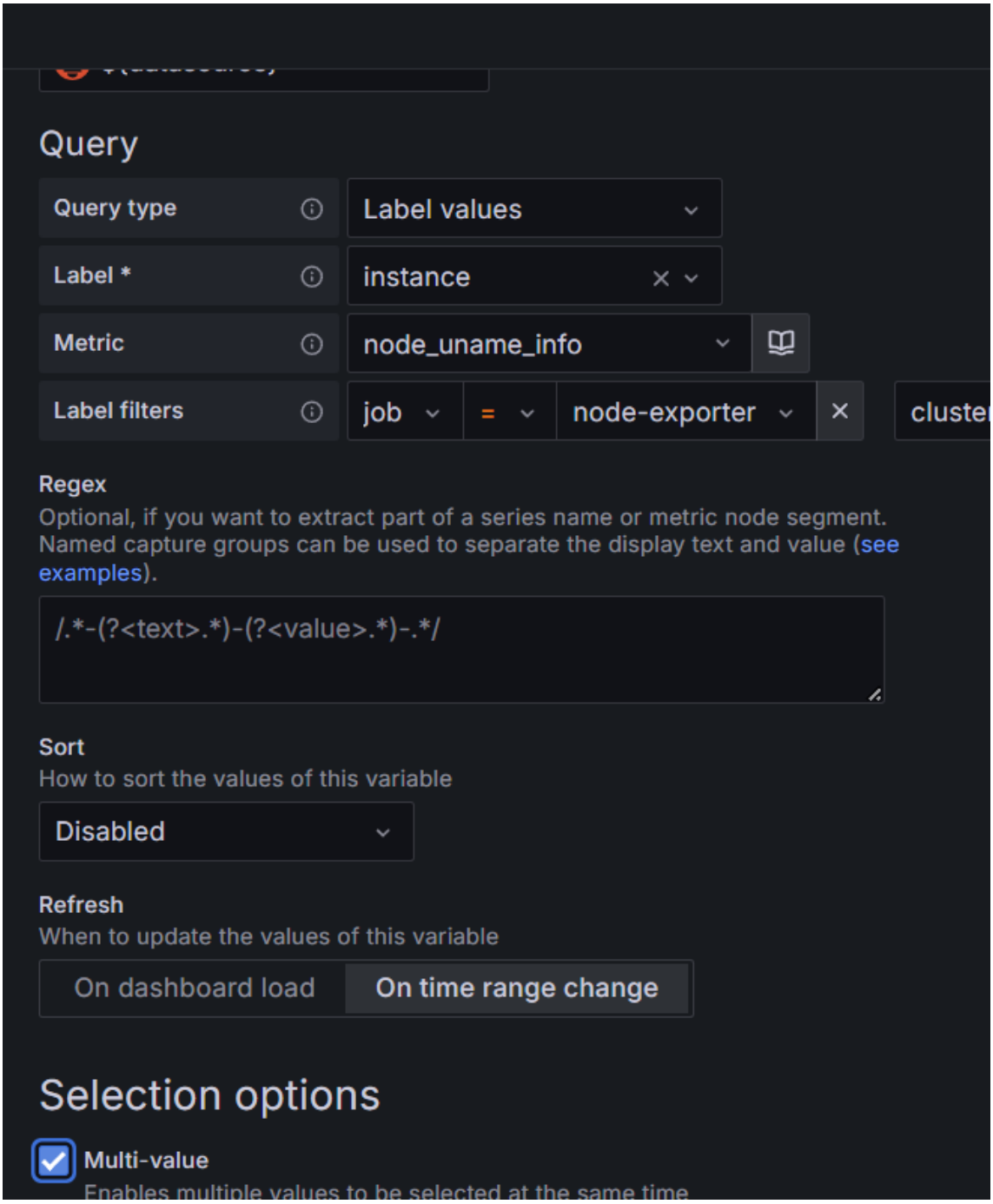


Schritt 7: Variable "instance" auf Multi-Auswahl ändern

1. Unter settings: siehe 6.
2. instance anklicken

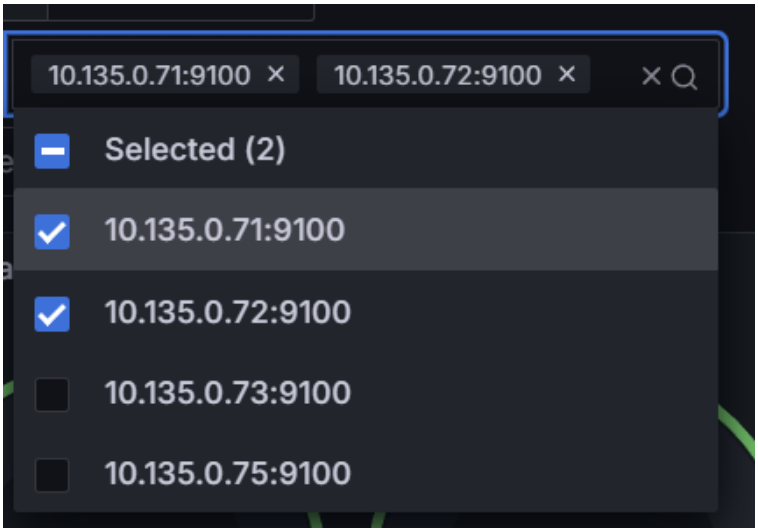


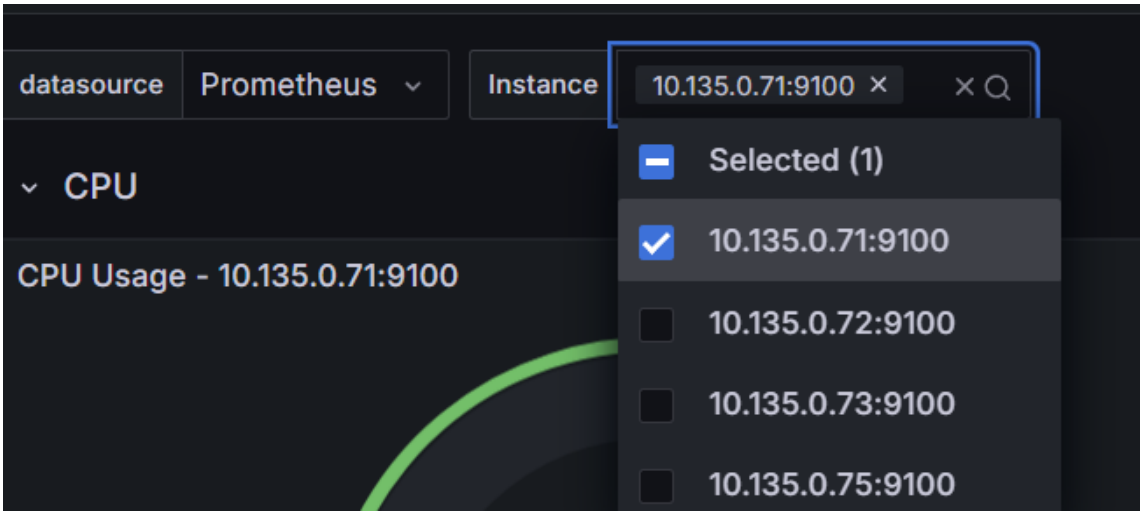
3. Multi-Value anklicken:



4. Wichtig: Dashboard speichern

```
Durch Multi-Value:
Dadurch lässt sich nicht nur eine Instance (ein Server), sondern mehrere auswählen
```





5. Problem - no data beheben

- Durch umstellung auf Multi-Value muss die Query geändert werden.
- Es darf nicht mehr explizit nach einer instance gefragt werden, sondern mit regex

```
## Vorher
(
  (1 - sum without (mode) (rate(node_cpu_seconds_total{job="node-exporter",
mode=~"idle|iowait|steal", instance="$instance", cluster="$cluster"}[$__rate_interval])))
/ ignoring(cpu) group_left
  count without (cpu, mode) (node_cpu_seconds_total{job="node-exporter", mode="idle",
instance="$instance", cluster="$cluster"})
)
```

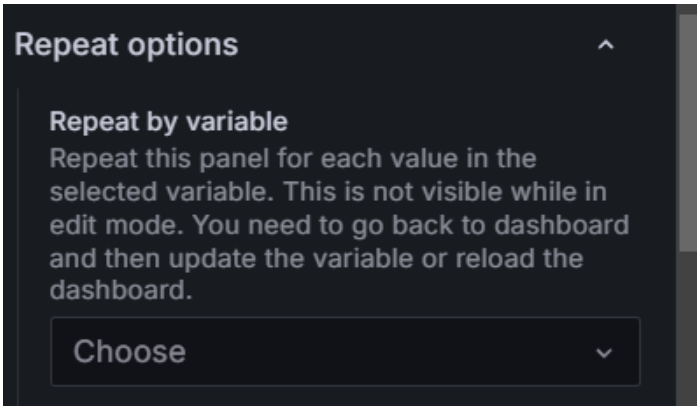
```
## Ändern in
(
  (1 - sum without (mode) (rate(node_cpu_seconds_total{job="node-exporter",
mode=~"idle|iowait|steal", instance=~"$instance", cluster="$cluster"}[$__rate_interval])))
/ ignoring(cpu) group_left
  count without (cpu, mode) (node_cpu_seconds_total{job="node-exporter", mode="idle",
```

```
instance=~"$instance", cluster="$cluster"})
)
```

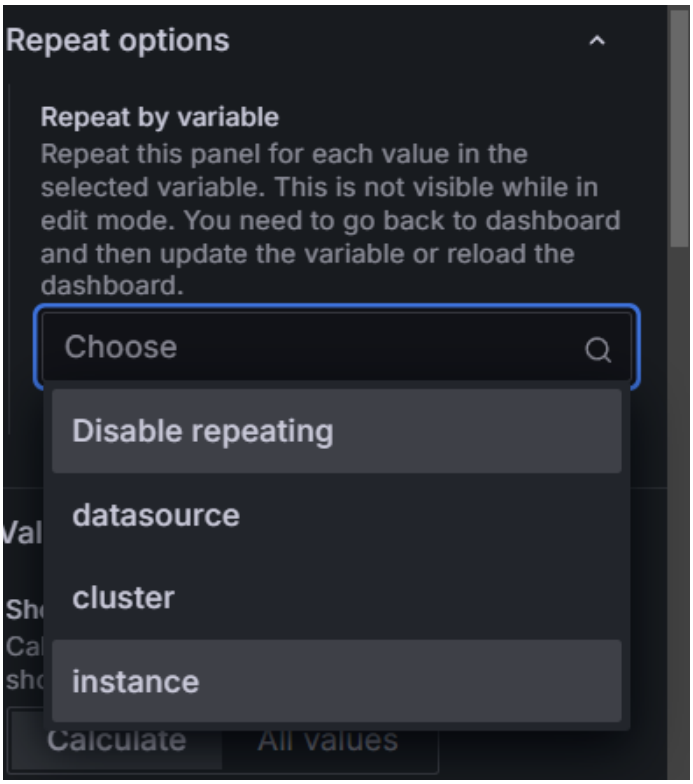
- Dashboard speichern

Schritt 5: Auf panel repetitions umstellen

1. Bei den Panel Settings runterscrollen



2. ... und instance auswählen



3. Max per row: auf 2 stellen
4. Dashboard speichern

Schritt 6: Testen: Dashboard muss nochmal neu geladen zu werden

1. Auf Back to Dashboard klicken

2. Die neue Ausgabe sollte erscheinen (evtl. oben bei instances nochmal alle auswählen)

Pod und Container Dashboard

Grafana GUI-Anleitung: Pod & Container Dashboard

1. Dashboard erstellen

1. In Grafana links auf "**Dashboards**" > "**New**" > "**New Dashboard**" klicken.
2. Auf "**Add a new panel**" klicken.

---

2. Panel 1: Nicht-laufende Pods pro Namespace

- **Titel:** „Fehlgeschlagene Pods je Namespace"
- **Abfrage (PromQL):**

```
count by (namespace) (kube_pod_status_phase{phase=~"Pending|Failed|Unknown"})
```

- **Panel-Typ: Bar chart**
- **X-Achse:** `namespace`
- **Y-Achse:** Anzahl Pods
- Speichern mit "**Apply**"

---

3. Panel 2: Top 5 Container mit höchsten Restarts

- Neues Panel > Titel: „Top 5 Container Restarts"
- **Abfrage:**

```
topk(5, sum by (pod, container) (kube_pod_container_status_restarts_total))
```

- **Panel-Typ: Table**
- Optional: Transformationen aktivieren für bessere Darstellung
- Apply

---

4. Panel 3: CPU-Nutzung pro Container

- Neues Panel > Titel: „CPU pro Container (millicores)"
- **Abfrage:**

```
sum by (namespace, pod, container) (
  rate(container_cpu_usage_seconds_total{container!=""}[5m])
) * 1000
```

- Panel-Typ: **Time series**
- Legende anzeigen: `{{namespace}} / {{pod}} / {{container}}`
- Apply

---

5. Panel 4: Memory-Nutzung pro Container

- Titel: „Speicher pro Container (MiB)"
- **Abfrage:**

```
sum by (namespace, pod, container) (
  container_memory_usage_bytes{container!=""}
) / (1024 * 1024)
```

- Panel-Typ: **Time series**
- Legende anzeigen: `{{namespace}} / {{pod}} / {{container}}`
- Einheit: **MiB**
- Apply

---

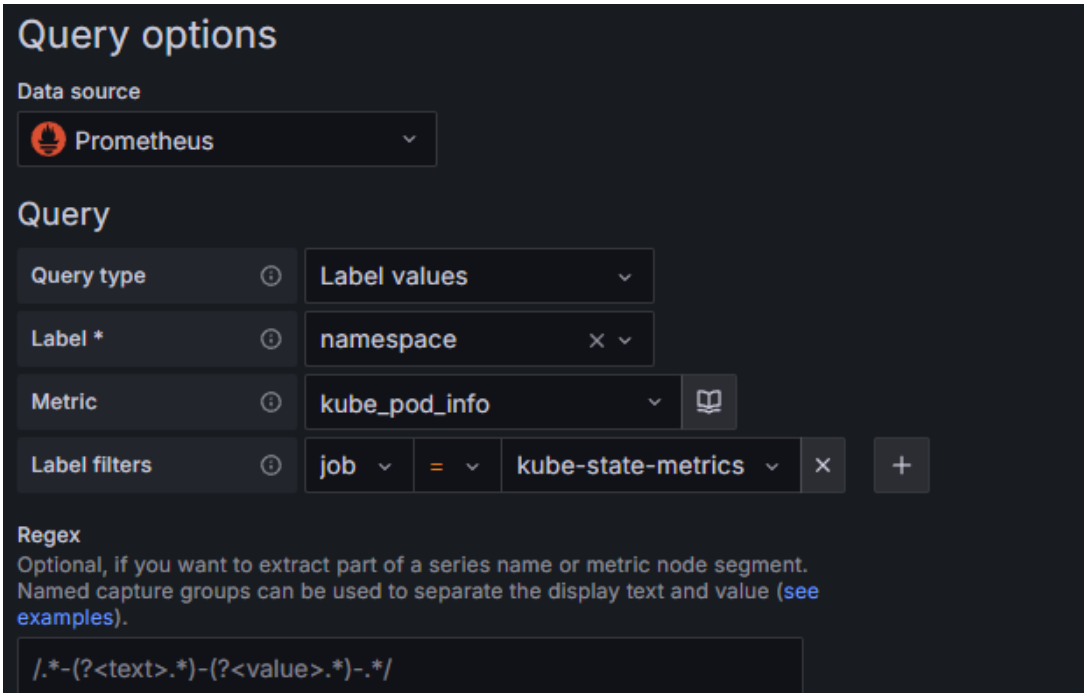
6. Panel 5: Container Ready Status als Gauge

- Titel: „Container bereit (Ready)“
- **Abfrage:**

```
max by (pod, container) (kube_pod_container_status_ready)
```

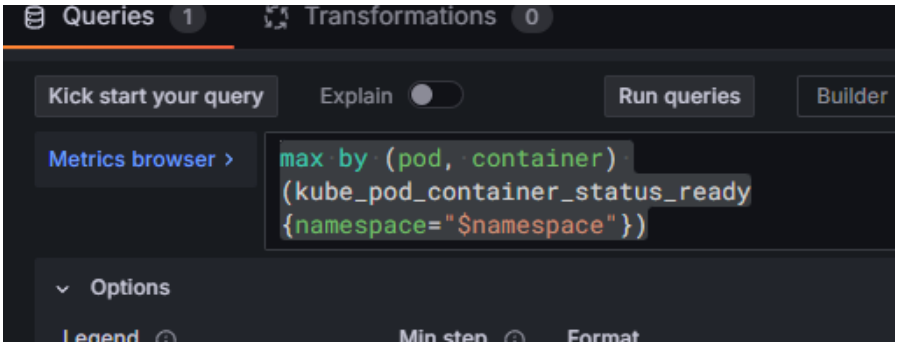
- **Panel-Typ: Gauge**
- **Thresholds:**
  - Min: `0` , Max: `1`
  - Thresholds definieren: `0 = rot` , `1 = grün`
    - `0.5` -> rot
    - `0.99` -> gelb
    - `1` -> grün
- Einheit: none
- Apply

- Dashboard bearbeiten (Zahnrad oben rechts) > **Variables** > „New“:
  - **Name:** `namespace`
  - **Type:** Query
  - **Datasource:** Prometheus
  - **Query:** `label_values(kube_pod_info, namespace)`
  - Gebe eine Liste von Werte (und zwar namespace) zurück aus der Metric kube_pod_info, bei der job = kube-state-metric ist



- Jetzt auch für die Query anpassen (Variable verwenden)

```
max by (pod, container) (kube_pod_container_status_ready{namespace="$namespace"})
```



---

8. Save & teilen

- Dashboard speichern (Diskette-Symbol oben)
- Optional: In Ordner verschieben oder als Start-Dashboard setzen

---

Möchtest du eines der Panels genauer als Screenshot oder mit JSON-Code sehen? Ich kann dir auch ein Demo-Dashboard JSON liefern.

Grafana - Alterting and Notifications

Grafana neuen alert anlegen

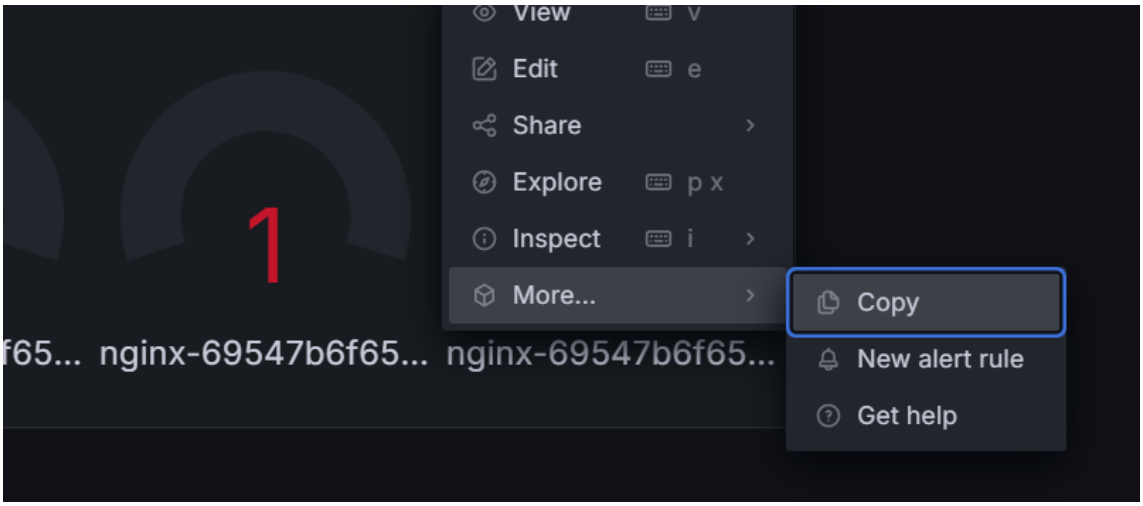
Voraussetzung

- ServiceMonitor eingerichtet: Nginx mit ServiceMonitor anlegen

Basics

- am besten im über das jeweilige Panel im Dashboard (more -> new alert rule
- Vorteil, die Query wird schon direkt übernommen

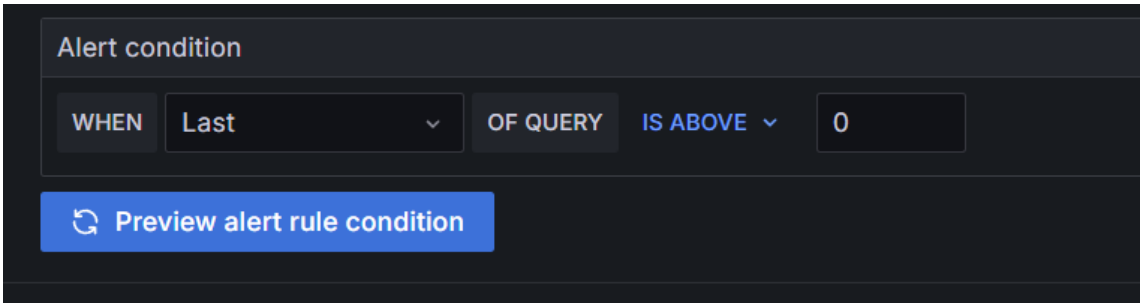
Schritt 1: Neues Alert - Formular aufrufen



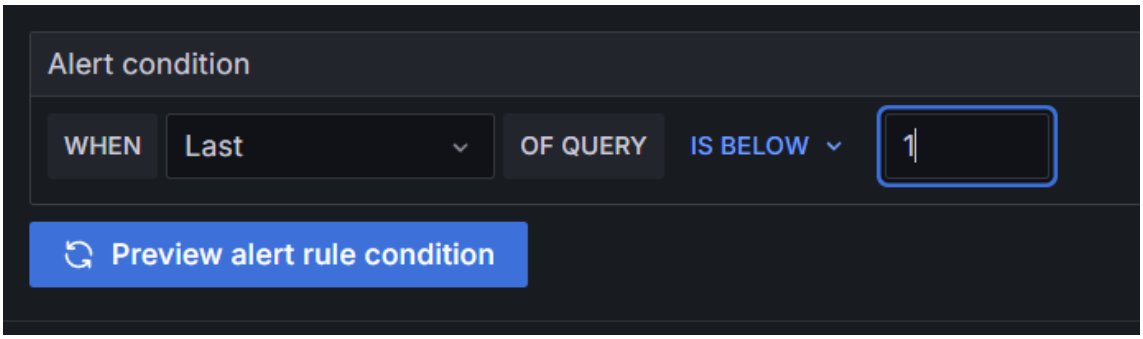
- Es wird immer zu einem Unified Alert

Schritt 2: Wichtig: Die Alert-Condition einstellen

- Diese ist aktuell falsch

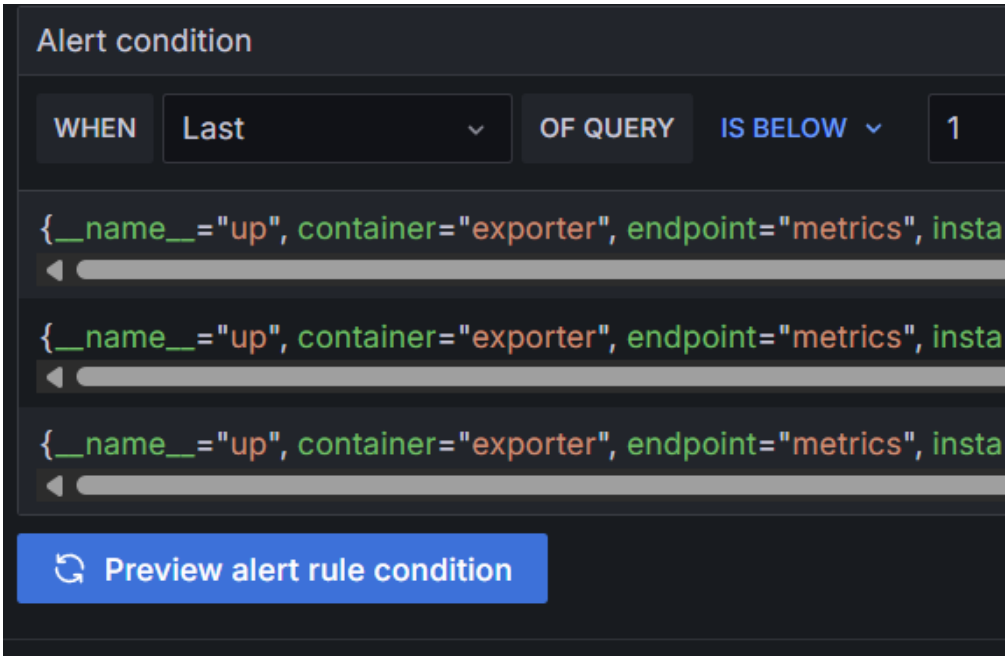


- So ist es richtig



Schritt 3: Preview alert rule condition

- Button klicken und gucken, wie es reagiert



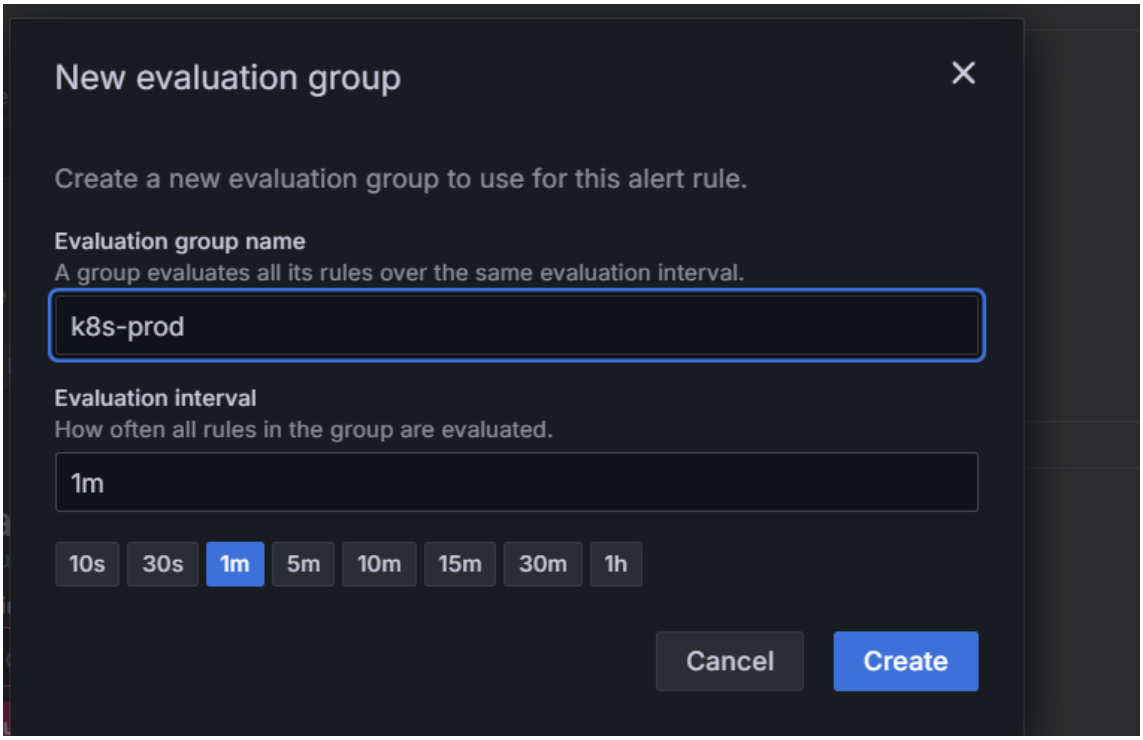
Schritt 4: Ein oder mehrere Labels setzen und Folder erstellen

- Wir erstellen einen neunen Folder: app1
- Labels sind u.a. wichtig für die Benachrichtigung (Die erfolgen in Form von labels)
- Wir nehmen hier

```
## label
team -> saas
```

Schritt 6: Set evaluation behaviour

- Wie oft soll er überprüfen ? (das kann man nach Umgebung machen, z.B)



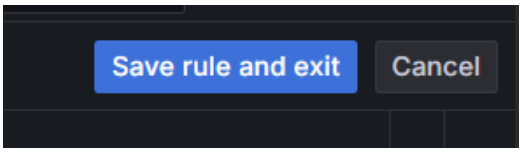
Schritt 7: Configure notifications

Contact Points

- Contact Point auswählen
- In unserem Beispiel nehmen wir Slack

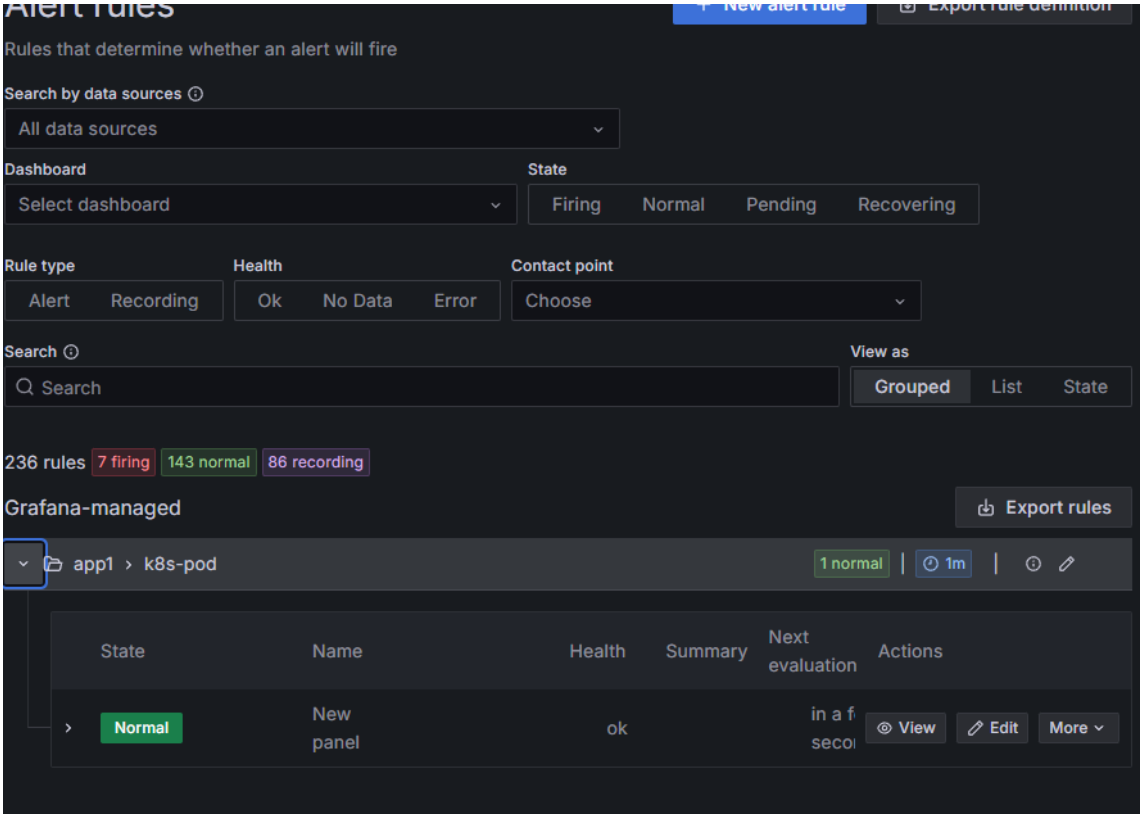
Schritt 8: Save rule and exit

- Button oben rechts klicken



Schritt 9: Wo ist der Alert ?

- alert rules -> app1 -> k8s-pod



Schritt 10: Testen:

- deployment löschen und im interface nachschauen

```
cd
cd manifests
cd svc-nginx
```

```
kubectl -n web-demo get deploy nginx
```

```
nano 03-nginx-deployment-metrics.yaml
```

```
## Readiness Check, einbauen, der nicht funktioniert
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  namespace: web-demo
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
```

```
      - name: nginx
        image: nginx:stable
        ports:
        - containerPort: 80
        volumeMounts:
        - name: nginx-conf
          mountPath: /etc/nginx/conf.d/default.conf
          subPath: default.conf
        readinessProbe:
          exec:
            command:
            - /bin/false
          initialDelaySeconds: 5
          periodSeconds: 10
      - name: exporter
        image: nginx/nginx-prometheus-exporter:latest
        args:
        - "-nginx.scrape-uri=http://localhost:80/stub_status"
        ports:
        - containerPort: 9113
      volumes:
      - name: nginx-conf
```

```
kubectl -n web-demo apply -f .
kubectl -n web-demo get pods
```

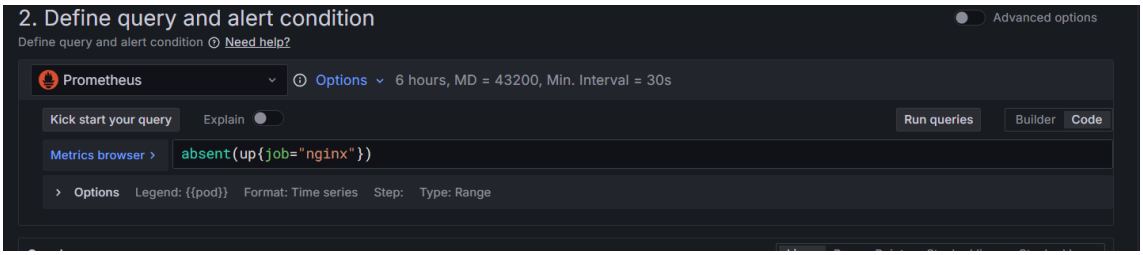
Grafana absence alert konfigurieren - d.h. Service hat keine Pods mehr

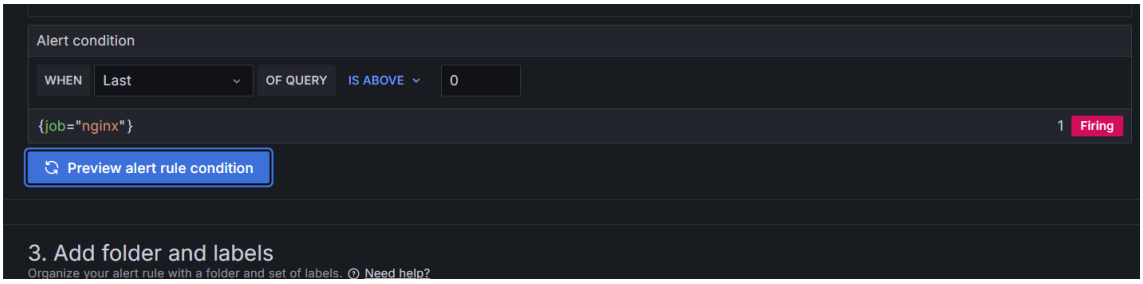
Prepare

- Delete deployment

```
kubectl -n web-demo get pods -l app=nginx
kubectl -n web-demo delete deploy nginx
```

Setup Alert





Click on

```
Preview and alert condition
```

Safe and exit rules

```
Safe rule and exit
```

Alert ausklappen und warten bis er feuert

1. Erst pending (dauert einen Moment)
2. Dann firing und es kommt ein Benachrichtigung per Slack

✅ Grafana Unified Alert Example: "No Data" for a Job

Use the `absent()` function

Grafana can alert when `absent(up{job="myjob"})` returns something — meaning no data is present.

Step-by-step (in Grafana UI)

1. **Go to Alerting → Alert Rules**
2. Click **"Create alert rule"**
3. Set **Data source** to your **Prometheus**
4. Add a **query** like this:

```
absent(up{job="myjob"})
```

5. In **Conditions**, set:
   - **WHEN**: `Query (A)` `returns a number`
   - **IS ABOVE**: `0`

   This works because `absent()` returns `1` if the series is absent.
6. Under **Alert Details**, give it a name like: `No data for job "myjob"`
7. Configure **Contact Points**, **Labels**, etc.

Grafana alert, >= pod aus replicaset nicht erreichbar

Hier ist ein vollständiges Beispiel für ein Kubernetes Deployment mit **absichtlich fehlschlagender Readiness-Probe**, das du nutzen kannst, um **Alerting zu testen**, wenn mindestens 3 Pods nicht "ready" sind:

Beispiel-Deployment `unready-demo.yaml`

```
mkdir manifests
cd manifests
```

```
mkdir unready
cd unready
```

```
nano unready-demo.yaml
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: unready-demo
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: unready-demo
  template:
    metadata:
      labels:
        app: unready-demo
    spec:
      containers:
        - name: demo
          image: busybox
          command: ["sh", "-c", "sleep 3600"]
          readinessProbe:
            exec:
              command:
                - /bin/false
            initialDelaySeconds: 5
            periodSeconds: 10
```

```
kubectl apply -f .
```

Erklärung:

- `sleep 3600` : Der Container läuft eigentlich stabil.
- `/bin/false` in der Readiness-Probe sorgt dafür, dass Kubernetes den Pod **niemals als „ready“** einstuft.
- Du kannst über `kube_pod_status_ready{condition="false"}` abfragen, wie viele Pods als „not ready“ gelten.

Test-Query für Alert in Grafana / Prometheus:

```
sum(kube_pod_status_ready{namespace="default", condition="false", pod=~"unready-demo-.*"}) >= 3
```

> ⚠ *Achte darauf, dass `kube-state-metrics` in deinem Cluster läuft und `kube_pod_status_ready` bereitstellt.*

✔ So nutzt du es:

1. YAML-Datei anwenden:

   ```
   kubectl apply -f unready-demo.yaml
   ```

2. Abfragen:

   ```
   kubectl get pods -l app=unready-demo
   ```

   Du wirst sehen, dass alle Pods im Status `0/1 Ready` sind.

3. **Grafana-Alert konfigurieren** auf Basis der obigen Query.

```
sum(kube_pod_status_ready{namespace="dein-namespace", condition="false", pod=~"dein-deployment-.*"}) >= 3
```

```
---

Möchtest du auch ein **komplettes Beispiel für den passenden Grafana Alert** (als YAML oder UI-Anleitung)?

### Grafana Notifications/Contact points


To configure which **contact point** to use for an alert in a typical **alerting system** (like Prometheus Alertmanager, Grafana, etc.), the process generally involves **creating routing rules** or **setting labels** that determine how alerts are matched to specific contact points (such as email, Slack, PagerDuty, etc.).

Here’s a general breakdown depending on the platform:

### **Grafana Alerting (Unified Alerting System)**

1. **Create Contact Points** via **Alerting > Contact Points**.

2. **Create Notification Policies** in **Alerting > Notification Policies**:

   * Define **routing rules** that match **labels or conditions**.
   * Attach contact points to each rule.

3. **In Alert Rules**, define **custom labels** that match the notification policies.

---

## Kubernetes Multi-Cluster (Types of setups including disadvantags/advantages)

### Recommended: Variant 1: prometheus agent + thanos/grafana stack


### Part 1: The agent side (on each cluster)

Awesome — here’s a **complete working setup** for deploying **Prometheus in `agent` mode with `remote_write`** using **Helm** in Kubernetes. We'll use the **`kube-prometheus-stack`** Helm chart, and assume you're pushing data to a **Thanos Receive** endpoint.

---

###   Goal
- Deploy Prometheus in **Agent mode**
- Use **`remote_write`** to send metrics to **Thanos Receive**
- Retain **no local TSDB**
- Scrape Kubernetes nodes (or other jobs)
- Run everything in a namespace like `monitoring`

---
```

```
###   Step-by-step Helm Setup

#### 1.   Add Helm Repo and Create Namespace
```bash
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update

kubectl create namespace monitoring
```

---

2. Create `values-agent.yaml`

```yaml
## values-agent.yaml

prometheus:
  prometheusSpec:
    enableRemoteWriteReceiver: false
    enableAdminAPI: false
    externalLabels:
      cluster: my-cluster

    #   Enable Agent mode
    extraArgs:
      enable-feature: agent

    #   No TSDB storage needed
    retention: 0h
    storageSpec: {}

    #   Remote write to Thanos Receive
    remoteWrite:
      - url: http://thanos-receive.monitoring.svc:19291/api/v1/receive
        write_relabel_configs:
          - source_labels: [__name__]
            regex: ".*"
            action: keep

    # ✅ Add your scrape jobs here
    additionalScrapeConfigs:
      - job_name: 'kubernetes-nodes'
        kubernetes_sd_configs:
          - role: node
        relabel_configs:
          - action: labelmap
            regex: __meta_kubernetes_node_label_(.+)
          - source_labels: [__address__]
            target_label: __address__
            regex: (.*):10250
            replacement: ${1}:9100
            action: replace

## Optional: disable default recording/alerting rules
kube-prometheus-stack:
  defaultRules:
    create: false # default rules can be set to true, but agent does not use them

## Optional: disable other components like alertmanager, grafana, etc.
alertmanager:
  enabled: false
```
```

```
grafana:
  enabled: false
```

> *You can also use `configMap` for `additionalScrapeConfigs` if you prefer that method.*

---

3. Install the chart with custom values

```
helm install prom-agent prometheus-community/kube-prometheus-stack \
  -n monitoring \
  -f values-agent.yaml
```

This will deploy:

- Prometheus running in **agent mode**
- Scraping Kubernetes nodes
- Streaming data to `remote_write` (e.g., Thanos Receive)

Part 2: The monitoring cluster side (once)

Perfect — here's a complete Helm-based setup to deploy **Thanos components** in Kubernetes:

- ✅ **Thanos Receive** — accepts remote_write from Prometheus agents
- ✅ **Thanos Store Gateway** — reads from object storage (e.g. S3 or MinIO)
- ✅ **Thanos Query (Querier)** — unified query layer
- ✅ **Grafana** — dashboards that query Thanos Query

We'll also deploy **MinIO** as an S3-compatible store.

Folder Structure (you'll get this as a zipped Helm lab)

```
thanos-lab/
├── values/
│   ├── minio.yaml
│   ├── thanos-receive.yaml
│   ├── thanos-store.yaml
│   ├── thanos-query.yaml
│   ├── grafana.yaml
├── install.sh   # helper script to install everything
```

✅ 1. MinIO (Object Storage)

`values/minio.yaml`

```
auth:
  rootUser: minioadmin
  rootPassword: minioadmin

defaultBuckets: thanos

resources:
  requests:
    memory: 256Mi
    cpu: 100m
```

Install:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm install minio bitnami/minio -n monitoring --create-namespace -f values/minio.yaml
```

✅ 2. Thanos Receive

`values/thanos-receive.yaml`

```yaml
objstoreConfig:
  type: s3
  config:
    bucket: "thanos"
    endpoint: "minio.monitoring.svc.cluster.local:9000"
    access_key: "minioadmin"
    secret_key: "minioadmin"
    insecure: true

receive:
  enabled: true
  tsdbRetention: 24h
  replicaLabel: replica
  hashPartitioning:
    replicationFactor: 1
    enableSharding: false

service:
  type: ClusterIP
```

Install:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm install thanos-receive bitnami/thanos -n monitoring -f values/thanos-receive.yaml
```

✅ 3. Thanos Store Gateway

`values/thanos-store.yaml`

```yaml
objstoreConfig:
  type: s3
  config:
    bucket: "thanos"
    endpoint: "minio.monitoring.svc.cluster.local:9000"
    access_key: "minioadmin"
    secret_key: "minioadmin"
    insecure: true

storegateway:
  enabled: true

service:
  type: ClusterIP
```

Install:

```
helm install thanos-store bitnami/thanos -n monitoring -f values/thanos-store.yaml
```

✅ 4. Thanos Query

`values/thanos-query.yaml`

```yaml
query:
  enabled: true
  replicaLabel: replica

  stores:
    - thanos-receive.monitoring.svc.cluster.local:10901
    - thanos-store.monitoring.svc.cluster.local:10901

service:
  type: ClusterIP
```

Install:

```
helm install thanos-query bitnami/thanos -n monitoring -f values/thanos-query.yaml
```

✅ 5. Grafana (with Thanos Query as datasource)

`values/grafana.yaml`

```yaml
datasources:
  datasources.yaml:
    apiVersion: 1
    datasources:
      - name: Thanos
        type: prometheus
        access: proxy
        url: http://thanos-query.monitoring.svc.cluster.local:9090
        isDefault: true
        jsonData:
          timeInterval: "15s"
adminPassword: "admin"
```

Install:

```
helm install grafana bitnami/grafana -n monitoring -f values/grafana.yaml
```

One-liner install script (optional)

You can create a small script `install.sh`:

```
##!/bin/bash
NAMESPACE="monitoring"
helm install minio bitnami/minio -n $NAMESPACE --create-namespace -f values/minio.yaml
helm install thanos-receive bitnami/thanos -n $NAMESPACE -f values/thanos-receive.yaml
helm install thanos-store bitnami/thanos -n $NAMESPACE -f values/thanos-store.yaml
helm install thanos-query bitnami/thanos -n $NAMESPACE -f values/thanos-query.yaml
helm install grafana bitnami/grafana -n $NAMESPACE -f values/grafana.yaml
```

Would you like me to package this into a downloadable ZIP for you?

Variant 2: Full prometheus in each cluster with thanos sidecar

Great! Here's a basic **Kubernetes manifest example** and also a **Helm-based setup** outline for deploying **Thanos components** alongside **Prometheus**, including long-term storage with **MinIO (S3-compatible)**. This setup is **lab-ready**, simple, and suitable for small-scale clusters or training.

Thanos Setup with Helm (Prometheus + Thanos + MinIO)

Option A: Using Helm

1. Add Repos

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo add bitnami https://charts.bitnami.com/bitnami
helm repo update
```

---

2. Deploy MinIO (S3-compatible storage)

```
helm install minio bitnami/minio \
  --set accessKey.password=minioadmin \
  --set secretKey.password=minioadmin \
  --set defaultBuckets=thanos \
  --namespace monitoring --create-namespace
```

---

3. Deploy Prometheus with Thanos Sidecar

Create `values-prometheus.yaml`:

```
extraContainers:
  - name: thanos-sidecar
    image: quay.io/thanos/thanos:v0.34.0
    args:
      - sidecar
      - --tsdb.path=/data
      - --prometheus.url=http://localhost:9090
      - --objstore.config-file=/etc/thanos/objstore.yaml
    volumeMounts:
      - name: prometheus-data
        mountPath: /data
      - name: thanos-objstore
        mountPath: /etc/thanos

extraVolumes:
  - name: thanos-objstore
    configMap:
      name: thanos-objstore

service:
  enabled: true
  type: ClusterIP
```

Create a ConfigMap for `objstore.yaml`:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: thanos-objstore
  namespace: monitoring
```

```
data:
  objstore.yaml: |
    type: s3
    config:
      bucket: "thanos"
      endpoint: "minio.monitoring.svc.cluster.local:9000"
      access_key: "minioadmin"
      secret_key: "minioadmin"
      insecure: true
```

Install Prometheus:

```
helm install prometheus prometheus-community/prometheus \
  -f values-prometheus.yaml \
  --namespace monitoring
```

Prometheus Responsibilities

- **Scrapes metrics** from your applications and Kubernetes targets.
- Stores those metrics locally (TSDB).
- **Has all the** `scrape_configs` (e.g., pods, services, kubelets, etc.).
- Runs as usual with no changes to scraping behavior.
- Now runs **with a Thanos Sidecar** container.

Thanos Sidecar Responsibilities

- Reads data from the local TSDB (no scraping itself).
- Uploads blocks to object storage (e.g., MinIO, S3).
- Exposes a gRPC endpoint to Thanos Query or Store Gateway.
- Acts as a **bridge** between Prometheus and Thanos.

Where Are Scrape Configs Defined?

- They are defined in:
  - `prometheus.yaml` if you're managing raw config.
  - Or via `values.yaml` in Helm under `serverFiles.prometheus.yml.scrape_configs`.

Example in Helm:

```
serverFiles:
  prometheus.yml:
    scrape_configs:
      - job_name: 'kubernetes-nodes'
        kubernetes_sd_configs:
          - role: node
        relabel_configs:
          - action: labelmap
            regex: __meta_kubernetes_node_label_(.+)
```

Grafana Loki

Installation von Grafana Loki - Single Instance - für Testing

Voraussetzung:

- Prometheus / Grafana Monitoring - Stack läuft bereits im namespace "monitoring"
- Prometheus ist als release "prometheus" mit helm installiert

Schritt 0: csi ausrollen (nfs-server muss eingerichtet sein)

```
helm repo add csi-driver-nfs https://raw.githubusercontent.com/kubernetes-csi/csi-driver-nfs/master/charts
helm install csi-driver-nfs csi-driver-nfs/csi-driver-nfs --namespace kube-system --version v4.11.0
```

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
  name: nfs-csi
provisioner: nfs.csi.k8s.io
parameters:
  server: 10.135.0.34
  share: /var/nfs
reclaimPolicy: Retain
volumeBindingMode: Immediate
```

```
kubectl apply -f .
```

Schritt 1: Projektordner anlegen

```
cd ~
mkdir loki-single
cd loki-single
```

Damit wird dein Projekt im Home-Verzeichnis (`~/loki-single`) angelegt.

Schritt 2: `values.yaml` erstellen

```
nano values.yaml
```

```
## Disabled for testing, otherwice cluster node needs way more than 8 GB of Memory
chunksCache:
  enabled: false


loki:
  auth_enabled: false
  commonConfig:
    replication_factor: 1
  schemaConfig:
    configs:
      - from: "2024-04-01"
        store: tsdb
        object_store: s3
        schema: v13
        index:
          prefix: loki_index_
          period: 24h
  pattern_ingester:
      enabled: true
  limits_config:
    allow_structured_metadata: true
    volume_enabled: true
  ruler:
```

```
    enable_api: true

minio:
  enabled: true

deploymentMode: SingleBinary

singleBinary:
  replicas: 1

## Zero out replica counts of other deployment modes
backend:
  replicas: 0
read:
  replicas: 0
write:
  replicas: 0

ingester:
  replicas: 0
querier:
  replicas: 0
queryFrontend:
  replicas: 0
querryScheduler:
  replicas: 0
distributor:
  replicas: 0
compactor:
  replicas: 0
indexGateway:
  replicas: 0
bloomCompactor:
  replicas: 0
bloomGateway:
  replicas: 0
```

Schritt 3: Installieren

```
helm repo add grafana https://grafana.github.io/helm-charts
helm repo update

helm upgrade --install loki grafana/loki \
  --namespace loki --create-namespace --version 6.29.0 \
  -f values.yaml
```

Schritt 4: promtail

```
nano promtail-values.yaml
```

```
config:
  clients:
    - url: http://loki-gateway.loki.svc.cluster.local/loki/api/v1/push
```

```
helm install promtail grafana/promtail --namespace loki -f promtail-values.yaml --create-namespace
```

Ref:

- https://grafana.com/docs/loki/latest/setup/install/helm/install-monolithic/

Datasource in Grafana bereitstellen per helm

Voraussetzung:

- Prometheus / Grafana Monitoring - Stack läuft bereits im namespace "monitoring"
- Prometheus ist als release "prometheus" mit helm installiert

Schritt 0: csi ausrollen (nfs-server muss eingerichtet sein)

```
helm repo add csi-driver-nfs https://raw.githubusercontent.com/kubernetes-csi/csi-driver-nfs/master/charts
helm install csi-driver-nfs csi-driver-nfs/csi-driver-nfs --namespace kube-system --version v4.11.0
```

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
  name: nfs-csi
provisioner: nfs.csi.k8s.io
parameters:
  server: 10.135.0.34
  share: /var/nfs
reclaimPolicy: Retain
volumeBindingMode: Immediate
```

```
kubectl apply -f .
```

Schritt 1: Projektordner anlegen

```
cd ~
mkdir loki-single
cd loki-single
```

Damit wird dein Projekt im Home-Verzeichnis (`~/loki-single`) angelegt.

Schritt 2: `values.yaml` erstellen

```
nano values.yaml
```

```
## Disabled for testing, otherwice cluster node needs way more than 8 GB of Memory
chunksCache:
  enabled: false


loki:
  auth_enabled: false
  commonConfig:
    replication_factor: 1
  schemaConfig:
    configs:
      - from: "2024-04-01"
        store: tsdb
```

```
        object_store: s3
        schema: v13
        index:
          prefix: loki_index_
          period: 24h
  pattern_ingester:
      enabled: true
  limits_config:
    allow_structured_metadata: true
    volume_enabled: true
  ruler:
    enable_api: true

minio:
  enabled: true

deploymentMode: SingleBinary

singleBinary:
  replicas: 1

## Zero out replica counts of other deployment modes
backend:
  replicas: 0
read:
  replicas: 0
write:
  replicas: 0

ingester:
  replicas: 0
querier:
  replicas: 0
queryFrontend:
  replicas: 0
queryScheduler:
  replicas: 0
distributor:
  replicas: 0
compactor:
  replicas: 0
indexGateway:
  replicas: 0
bloomCompactor:
  replicas: 0
bloomGateway:
  replicas: 0
```

Schritt 3: Installieren

```
helm repo add grafana https://grafana.github.io/helm-charts
helm repo update

helm upgrade --install loki grafana/loki \
  --namespace loki --create-namespace --version 6.29.0 \
  -f values.yaml
```

Schritt 4: promtail

```
nano promtail-values.yaml
```

```
config:
  clients:
    - url: http://loki-gateway.loki.svc.cluster.local/loki/api/v1/push
```

```
helm install promtail grafana/promtail --namespace loki -f promtail-values.yaml --create-namespace
```

Ref:

- https://grafana.com/docs/loki/latest/setup/install/helm/install-monolithic/

Wo finde ich Loki in Grafana ?

---

1. Explore → Logs (Ad-hoc-Logsuche)

Schritte:

1. Links im Menü auf **"Explore"** klicken
2. Oben links im Dropdown die **Loki-Data Source** auswählen (z. B. `Loki`)
3. Du kannst jetzt:
   - Nach **Labels** filtern (`{job="my-app"}`)
   - Per LogQL Abfragen wie `|= "error"` verwenden
   - Live-Logs anzeigen (unten rechts: **Live** aktivieren)

Prometheus

Prometheus-Metriktypen (engl. metric types)

Welche gibt es ?

- Counter
- Gauge
- Histogram
- Summary

In der Prometheus-Dokumentation werden sie auch explizit als metric types bezeichnet. Wenn du also über „eine Gauge“ sprichst, meinst du korrekt: „Eine Metrik vom Typ Gauge“

1. Counter

Ein Counter ist ein **ständig wachsender Wert**. Er beginnt bei 0 und kann **nur steigen** (außer bei einem Reset, z. B. Pod-Restart).

Beispiel:

- `http_requests_total` – zählt, wie viele HTTP-Anfragen es gab
- `errors_total` – zählt Fehlermeldungen

> ***+ Nur hochzählend!*** *Für „Rate“-Abfragen ideal.*

Typischer PromQL-Ausdruck:

```
rate(http_requests_total[5m])
```

Zeigt, wie viele Anfragen pro Sekunde in den letzten 5 Minuten kamen.

---

2. Gauge

Ein Gauge ist ein **aktueller Messwert**, der **steigen und sinken** kann – wie ein Thermometer.

Beispiel:

- `memory_usage_bytes` – aktueller Speicherverbrauch
- `cpu_temperature` – aktuelle CPU-Temperatur

> *Ideal für Zustände wie Auslastung, offene Verbindungen etc.*

PromQL:

```
memory_usage_bytes
```

zeigt den letzten bekannten Wert.

---

3. Histogram

Ein Histogram misst **Verteilungen von Werten**, z. B. Antwortzeiten. Es zählt, **wie viele Ereignisse in bestimmte Wertebereiche ("Buckets")** fallen.

Beispiel:

- `http_request_duration_seconds_bucket`

Diese Metrik ist gekoppelt mit:

- `_count` (Gesamtanzahl)
- `_sum` (Summe aller Werte)

> *Sehr nützlich für Latenzen und Antwortzeitverteilungen.*

PromQL-Beispiel (90. Perzentil über 5 Minuten):

```
histogram_quantile(0.9, rate(http_request_duration_seconds_bucket[5m]))
```

---

4. Summary (ähnlich wie Histogram, aber clientseitig berechnet)

Ein Summary enthält direkt **Perzentile**, allerdings:

- weniger aggregierbar über mehrere Instanzen
- erzeugt mehr Metriken
- eher selten verwendet in modernen Setups

Beispiel:

- `http_request_duration_seconds{quantile="0.9"}`

> ⚠ *Für verteilte Systeme nicht gut skalierbar → lieber Histogram verwenden!*

Zusammenfassung als Tabelle:

Typ	Eigenschaften	Beispiel	Ideal für...
Counter	nur steigend	`http_requests_total`	Events, Fehler, Anfragen
Gauge	auf- und absteigend	`memory_usage_bytes`	Zustände, Nutzung
Histogram	Buckets + Summe/Count	`*_bucket`, `*_sum`, `*_count`	Latenzverteilung, SLA-Analyse
Summary	Clientseitige Perzentile	`*_quantile`	Einfache Latenzmetriken

Kubernetes Multi-Cluster (using Thanos)

Prerequisites: What is Thanos

What is Thanos ?

Thanos is an open-source **highly available, long-term storage solution for Prometheus**. It extends Prometheus by adding **global querying, deduplication, downsampling, and retention capabilities** across multiple Prometheus instances.

Why use Thanos (or: What are the problems with Prometheus)

In simple terms:

Prometheus is great for monitoring, but it has **limitations**:

- **No built-in high availability** (HA)
- **No long-term storage** (TSDB is local and limited)
- **Difficult to query across multiple Prometheus servers**

Thanos solves all that by sitting on top of Prometheus.

What are the key components of Thanos ?

Key Components of Thanos:

1. **Sidecar** – Sits next to each Prometheus, uploads data to object storage (e.g. S3, GCS, MinIO).
2. **Store Gateway** – Reads historical data from the object storage.
3. **Query** – Global query engine that federates multiple Prometheus instances.
4. **Compactor** – Compacts and down-samples metrics data to reduce storage usage.
5. **Ruler** – Allows global alerting & recording rules.
6. **Receiver (optional)** – Receives remote writes directly, useful in cloud-native setups.

Benefits

What You Get with Thanos:

Feature	Benefit
Global Query View	One place to query multiple Prometheus
HA via Sidecars	Prometheus replicas + deduplication
Object Storage	S3/GCS/MinIO for infinite retention
Downsampling	Better performance for old data
Alerting Rules	Centralized alerting with Thanos Ruler

Components

Explanation

In a **Kubernetes context**, **Thanos** is typically deployed as a set of components (usually as Deployments, StatefulSets, and Services), each responsible for extending **Prometheus** to enable **long-term storage, high availability, and global querying**. Here's a breakdown of the **main Thanos components** and how they work **in Kubernetes**:

---

1. Thanos Sidecar

- **Runs alongside each Prometheus instance** as a sidecar container.
- **Functions**:
  - Uploads Prometheus TSDB blocks to object storage (e.g., S3, GCS, MinIO).
  - Serves the Prometheus data to Thanos Query.
- **Deployment**: Typically as a container inside the same **Pod** as Prometheus.

---

2. Thanos Query

- A **central query layer** that aggregates data from multiple Prometheus + Sidecar instances or other Thanos components (e.g., Store, Ruler).
- **Functions**:

- Provides a single PromQL query interface across multiple Prometheus data sources.
- Used by **Grafana** as the data source for querying global metrics.

- **Deployment**: Separate Deployment or StatefulSet with a Service.

---

3. Thanos Store Gateway

- Reads **historical data** directly from the object store (S3, GCS, etc.).
- **Functions**:
  - Makes historical data available to Thanos Query.
  - Doesn't collect or scrape; it's for **read-only** access to blocks in object storage.
- **Deployment**: Typically a separate Deployment, often with a PersistentVolume for caching.

---

4. Thanos Compactor

- Periodically **compacts, deduplicates, and downsamples** blocks in object storage.
- **Functions**:
  - Reduces storage costs and speeds up queries.
  - Only one active instance should run at a time to avoid conflicts.
- **Deployment**: CronJob or Deployment with single replica.

---

5. Thanos Ruler

- Equivalent to Prometheus's **ruler**, but works across **Thanos data sources**.
- **Functions**:
  - Runs alerting and recording rules on **global data**.
  - Can write rule results to object storage or remote write targets.
- **Deployment**: Standalone component, often with object storage access.

---

6. Thanos Bucket Web (optional)

- UI to **inspect the contents of the object storage bucket** used by Thanos.
- **Functions**:
  - Helps debug or verify the TSDB blocks and compaction.
- **Deployment**: Optional Deployment or sidecar for inspection.

Bonus: Common Setup Practices in Kubernetes

- **Object Storage**: S3 / GCS / MinIO is needed for Sidecar, Store, Compactor, Ruler.
- **Service Discovery**: Thanos components use gRPC and DNS-based discovery (Kubernetes Services).
- **Monitoring**: Often monitored by a separate Prometheus instance.
- **HA**: Thanos Query and Store components can be scaled horizontally for HA.

Would you like a sample Helm values file or a manifest example for deploying Thanos components in Kubernetes?

Thanos Compactor

Explanation

What does the Thanos Compactor do?

The **Compactor** is a component of Thanos that:

1. **Compacts smaller blocks into larger ones**
   - Prometheus writes data in 2-hour blocks.
   - The sidecar uploads these raw blocks to object storage.
   - Over time, this results in lots of small blocks.
   - The Compactor merges them (e.g., 2h blocks → 10h → 48h → 2-week blocks), improving query performance.
2. **Downsamples old data** (optional)

- Creates lower-resolution versions (e.g., 5m, 1h step sizes) to speed up long-range queries.
- Useful for dashboards or queries covering weeks/months of data.

3. **Applies retention policies**

- Deletes old blocks that exceed the configured retention period.

So when do you need the Compactor?

You need the **Compactor** if:

- You're using **object storage** (S3, GCS, etc.) for long-term data
- You want efficient storage and faster queries
- You need **downsampling** or **retention control**

On what does then compactor act

- The compactor act directly on the data uploaded to s3-storage

How many times do I need to install compactor ?

- I will only need to install it once in the cluster (and i needs read/write access to the s3 data)

Kubernetes Multi-Cluster (using Cortex - multi-tenant tsdb's)