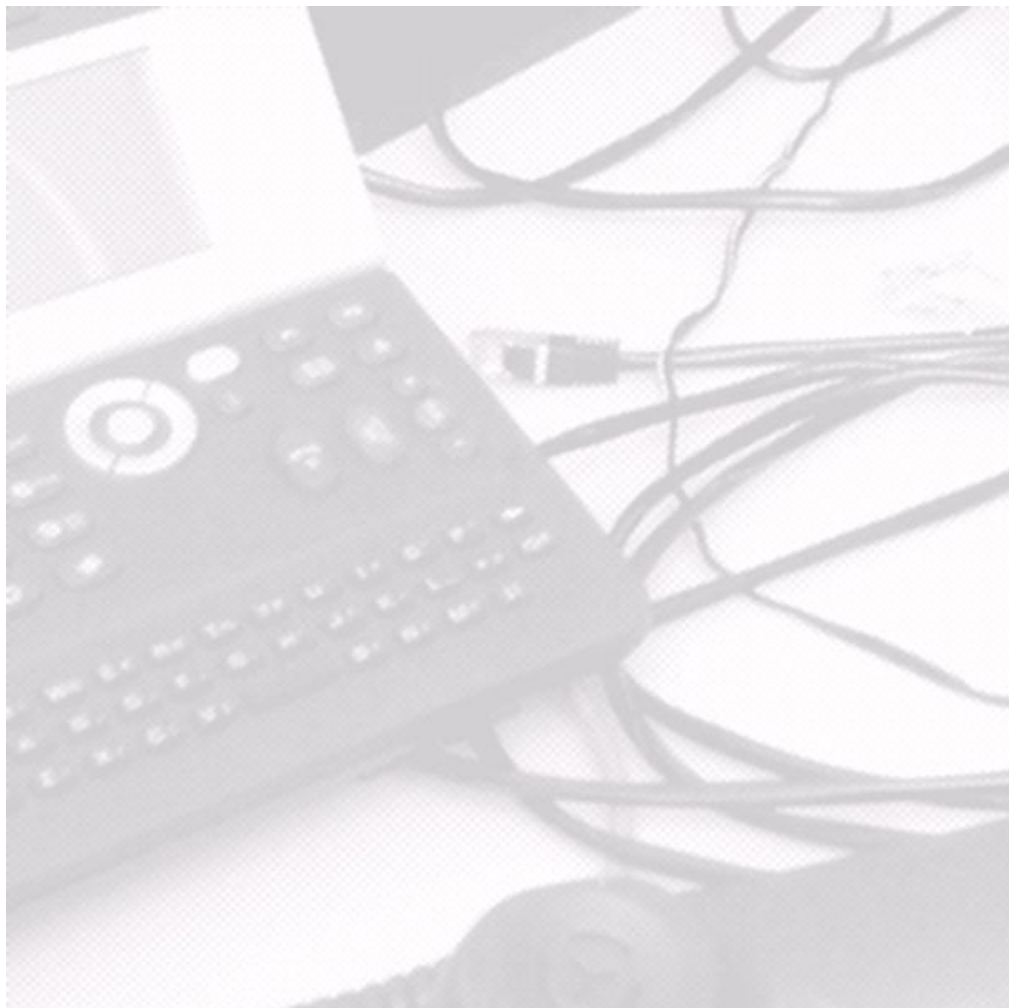


Kubernetes laboratory



Lab part 3: Kubernetes monitoring with Prometheus and Grafana

Prepared by: dr inż. Dariusz Bursztynowski

Acknowledgements: The author is grateful to the following students for their help in preparing the lab (in alphabetical order): Hubert Daniłowicz, Franciszek Dec, Jerzy Jastrzębiec-Jankowski, Maciej Maliszewski, Miłosz Marchewka, Adrian Osędowski, Cezary Osuchowski, Piotr Polnau, Jan Sosulski, Filip Wrześniewski, Marta Zielińska

ZSUT. Zakład Sieci i Usług Teleinformatycznych
Instytut Telekomunikacji
Wydział Elektroniki i Technik Informacyjnych
Politechnika Warszawska

Last update: April 2025

WARNING: Before powering up your devices, please make sure you are using **the correct power adapter**. The power adapters in the boxes have the same DC plug type, but they are **SIGNIFICANTLY DIFFERENT** in terms of output voltage (WiFi routers as Linksys are powered by 12V DC, while the TP-Link switch requires 53V DC). **Powering our WiFi router from the TP-Link switch power adapter will immediately damage the router.**

Table of contents

1.	Installing Prometheus stack.....	3
1.1.	Installation.....	3
1.2.	Main components of the stack	5
1.2.1.	Prometheus operator	5
1.2.2.	Selected service monitors.....	5
1.2.2.1.	Node-exporter service monitor	6
1.2.2.2.	Kube State Metrics.....	6
1.2.2.3.	Kubelet service monitor	6
1.3.	Prometheus server (the main application)	6
1.3.1.	Final configurations of Prometheus server.....	7
1.3.1.1.	Enabling access to all required resources	7
1.3.1.2.	Assigning persistent storage.....	8
1.3.1.3.	Exposing Prometheus dashboard as LoadBalancer service.....	10
1.3.2.	Deploying the updates to a running Prometheus server	11
1.4.	Optional: deploying monitoring extensions in our cluster.....	14
1.4.1.	Traefik service monitor	14
2.	Grafana.....	15
2.1.	Manifest preparation	15
2.1.1.	Update the manifests	15
2.1.2.	Deploy Grafana, check main components and open the first dashboard	17
2.2.	Checking Grafana – first graphs	18
3.	Monitoring at the resource and service levels.....	20
3.1.	Resource metric monitoring: loading the cluster using congestion pod.....	20
3.2.	Service metric monitoring: monitoring the number of user sessions in 5G network	20
4.	Useful readings and videos.....	21

1. Installing Prometheus stack

We will use kube-prometheus project to install Prometheus stack. Kube-prometheus tutorial is available [here](#). Official documentation of the project is available [here](#).

Prometheus is the collector of metrics popular in Kubernetes deployments to monitor clusters including server infrastructure and the services that are run in Kubernetes clusters. It uses so-called service monitors that provide information Prometheus can scrape and present to analytical applications. Prometheus will use persistent storage, and we will specify for how long it will keep the data. You can have more than one instance of Prometheus, for example to securely isolate different types of monitoring data (e.g., one instance of Prometheus per customer). But to save our resources we will use one for everything we want to monitor.

Our Prometheus instance will be controlled by Prometheus Operator – a dedicated Kubernetes controller run in the cluster. It is responsible for the deployment and life cycle management of all components related to the Prometheus instance. After Prometheus components are installed, Prometheus Operator will monitor the Kubernetes API server for changes to Prometheus-related objects and ensure that the current Prometheus deployments (there can be multiple Prometheus deployments in one cluster) match these objects.

In principle, we will install Prometheus stack according to kube-prometheus project following the instructions available [here](#). This project provides example configurations for a complete cluster monitoring stack based on Prometheus and the Prometheus Operator. This includes deployment of multiple Prometheus and Alertmanager instances, metrics exporters such as the node_exporter for gathering node metrics, scrape target configuration linking Prometheus to various metrics endpoints, and example alerting rules for notification of potential issues in the cluster. It also provides Grafana and Grafana dashboards. For more info on kube-prometheus, please refer to their web page.

One can save a little bit of resources deploying only those modules of the full kube-prometheus stack that will be mandatory for her/his use case (e.g., in our case we will not need Alertmanager, PrometheusRule, PrometheusAdapter and BlackboxExporter). Such missing CRDs can always be included at a later time if needed. However, to make our work easier we'll deploy the full stack leaving experimenting with possible adjustments to your own decision.

In the remainder of this section, we first run full installation procedure of the whole stack (subsection 1.1) and then describe selected components of the platform and fine-tune a couple of settings according to our needs (subsection 1.2).

1.1. Installation

As explained above, we will use kube-prometheus project to install Prometheus stack in our cluster: <https://github.com/prometheus-operator/kube-prometheus>.

Note 1: If you happen to check the manifests of kube-prometheus in more detail you will often see the term RBAC (role based control) declaration for various platform components. RBAC is the key mechanism to control access to the resources of particular types in Kubernetes cluster to list them, execute CRUD operations on them, and receive notifications about their state change. In this context, RBAC should be contrasted against so called Network Policies which in Kubernetes are used to control the communication among pods and services in the cluster, and between the cluster and the external world. More on RBAC in Kubernetes can be found, e.g., in [this document](#). Also worth noting is the [link here](#).

Note 2: If you are working with a fresh k3s cluster then by default it contains the right RBAC settings to accommodate kube-prometheus installation. In case of doubts, check the **Prerequisites** for installing kube-prometheus described [here](#); run the command `kubeadm config print init-defaults --component-configs KubeletConfiguration` and inspect it for actual values of `authentication.webhook.enabled` (should be true) and `authorization.mode` (should be Webhook) (defaults are listed [here](#)). In such a case you may need to install kubeadm package on your management node, though.

Note 3: As mentioned in our Kubernetes Lab Part 1 guide, the [compatibility matrix](#) between Kubernetes and kube-prometheus limits the scope of tested releases of Kubernetes. However, newer versions of kubernetes will probably work. Currently we prefer version 0.14 of kube-prometheus formally compatible with Kubernetes 1.31 (as of 28th March 2025). It uses Prometheus v2.54.x and works well with Open5GS monitoring targets¹.

We install kube-prometheus according to the [quickstart](#) procedure. To this end follow the steps listed below.

1. It is recommended to create a directory named, e.g., `monitoring`² on the management host. We will use it to store only kube-prometheus manifests, and we will modify some of them according to our needs. Effectively, if you `git clone`³ kube-prometheus you will need to move the contents of its directory `manifests` to directory `monitoring`, and then delete locally cloned (already unnecessary) kube-prometheus repo (below, commands to be executed are printed in **bold**).

```
xubuntu@xubulab:~/k3s-taskforce/manifests$ mkdir monitoring
xubuntu@xubulab:~/k3s-taskforce/manifests$ cd monitoring

# clone complete kube-prometheus, copy relevant release manifests and delete unnecessary stuff
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ git clone -b release-0.14 --single-branch https://github.com/prometheus-operator/kube-prometheus.git
Cloning into 'kube-prometheus'...
remote: Enumerating objects: 16665, done.
remote: Counting objects: 100% (21/21), done.
remote: Compressing objects: 100% (17/17), done.
remote: Total 16665 (delta 12), reused 6 (delta 4), pack-reused 16644
Receiving objects: 100% (16665/16665), 8.64 MiB | 2.27 MiB/s, done.
Resolving deltas: 100% (11078/11078), done.
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ cp -r kube-prometheus/manifests/* ./
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ rm -r kube-prometheus
```

2. The best way is to follow the installation steps according to [this](#). You can copy-paste from there (recommended), only **changing manifests for monitoring** in the path name (below, the commands to be executed are printed in **bold**).

```
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ cd ..
xubuntu@xubulab:~/k3s-taskforce/manifests$ kubectl apply --server-side -f monitoring/setup
customresourcedefinition.apiextensions.k8s.io/alertmanagerconfigs.monitoring.coreos.com
serverside-applied
...
namespace/monitoring serverside-applied
xubuntu@xubulab:~/k3s-taskforce/manifests$ kubectl wait \
--for condition=Established \
--all CustomResourceDefinition \
--namespace=monitoring
customresourcedefinition.apiextensions.k8s.io/addons.k3s.cattle.io condition met
```

¹ Newer versions rely on Prometheus >= 3.0.0 which does not work well for Open5GS and a [workaround to this is needed](#). One seemingly straightforward approach relies on using [Prometheus scrape classes](#).

² You can assign whatever name you like to this directory.

³ Methods as `wget -r` that are often used for bulk file downloads from webpages do not work for github.

```

customresourcedefinition.apiextensions.k8s.io/helmcharts.helm.cattle.io condition met
. . .
xubuntu@xubulab:~/k3s-taskforce/manifests$ kubectl apply -f monitoring/
alertmanager.monitoring.coreos.com/main created
networkpolicy.networking.k8s.io/alertmanager-main created
. . .
# after a couple of minutes, check all pods from monitoring namespace are running
xubuntu@xubulab:~/k3s-taskforce/manifests$ kubectl get pods -n monitoring
NAME                                READY   STATUS    RESTARTS   AGE
node-exporter-crk6b                 2/2     Running   0           4m21s
prometheus-operator-776c6c6b87-w89ml 2/2     Running   0           4m18s
alertmanager-main-0                 2/2     Running   0           3m17s
node-exporter-79fp9                 2/2     Running   0           4m21s
node-exporter-shjjs                 2/2     Running   0           4m21s
node-exporter-sd2j7                 2/2     Running   0           4m21s
blackbox-exporter-6fd586b445-fmdlx   3/3     Running   0           4m24s
kube-state-metrics-66659c89c-m9d6f   3/3     Running   0           4m21s
alertmanager-main-1                 2/2     Running   1 (105s ago) 3m17s
alertmanager-main-2                 2/2     Running   0           3m17s
grafana-6849bbf859-vjt25            1/1     Running   0           4m22s
prometheus-k8s-1                    2/2     Running   0           3m13s
prometheus-k8s-0                    2/2     Running   0           3m13s
prometheus-adapter-757f9b4cf9-2rbz9  1/1     Running   0           4m19s
prometheus-adapter-757f9b4cf9-zb7x5  1/1     Running   0           4m19s
xubuntu@xubulab:~/k3s-taskforce/manifests$

```

1.2. Main components of the stack

In this subsection, selected components from the monitoring stack installed in the previous section are briefly described and slightly configured to meet our needs.

1.2.1. Prometheus operator

Beginner's guide to using Prometheus Operator: <https://blog.container-solutions.com/prometheus-operator-beginners-guide>. Prometheus operator is a kind of Kubernetes controller that manages the lifecycle of Prometheus application (all the microservices Prometheus is composed of) using the models of all needed components. Those models are provided in the form of custom resource descriptors (CRD). We will create all the CRDs that define the Prometheus, Alertmanager, and ServiceMonitor abstractions used to configure the monitoring stack, as well as the Prometheus Operator controller and Service.

1.2.2. Selected service monitors

General architecture of Prometheus in Kubernetes including the role of service monitor is described, e.g., at <https://github.com/prometheus-operator/prometheus-operator/blob/main/Documentation/user-guides/getting-started.md>. Prometheus operator uses ServiceMonitors to define a set of targets to be monitored by Prometheus. It uses `matchLabel` selectors in ServiceMonitor definition to define which Services to monitor, the namespaces to look for, and the port on which the metrics are exposed. ServiceMonitor CRD are used by the operator to configure Prometheus server to scrap desired services (those indicated in ServiceMonitor). More explanations regarding Prometheus operator and the role of service monitor custom resource can be found in several other places, e.g., in <https://blog.container-solutions.com/prometheus-operator-beginners-guide>.

The kube-prometheus project we use here provides service monitors for all control plane components of Kubernetes platform, i.e. (see the link above the links given in the intro to section 1):

- kubernetesControlPlane-serviceMonitorApiserver
- kubernetesControlPlane-serviceMonitorCoreDNS
- kubernetesControlPlane-serviceMonitorKubeControllerManager
- kubernetesControlPlane-serviceMonitorKubeScheduler
- kubernetesControlPlane-serviceMonitorKubelet

Actually, all components of the kube-prometheus stack have their service monitor that is scraped by Prometheus (so even more monitors than those listed above are present). In the following, we describe only a couple of selected monitors and for more information the reader is referred to the original documentation of the kube-prometheus project.

1.2.2.1. Node-exporter service monitor

Node-exporter is a service monitor for Prometheus that is designed to provide only hardware and OS metrics exposed by the cluster nodes running Linux kernel. It is deployed as a Kubernetes Daemon set. Other metrics as service level metrics for microservices or for Kubernetes level components/resources (API server, pods, deployments, etc.) have to be gathered by Prometheus using other exporters than node-exporter.

Note: node-exporter deployed in our stack is similar to this <https://github.com/carlosedp/cluster-monitoring/blob/master/manifests/node-exporter-daemonset.yaml> (link daemonset only, the rest is straightforward to locate). There are small differences between them (e.g., for the daemonset there's updateStrategy included in "our" version contrary to the linked one, different container repos are referenced, etc.), but the overall shape is the same including additional flags setup for docker container according to this https://github.com/prometheus/node_exporter#docker. We will use the RedHat repo.

1.2.2.2. Kube State Metrics

According to the original documentation, *kube-state-metrics (KSM)* is a simple service that listens to the Kubernetes API server and generates metrics about the state of the objects. It is not focused on the health of the individual Kubernetes components, but rather on the health of the various objects inside, such as deployments, nodes and pods. *kube-state-metrics* is about generating metrics from Kubernetes API objects **without modification**. This ensures that features provided by *kube-state-metrics* have the same grade of stability as the Kubernetes API objects themselves. In turn, this means that *kube-state-metrics* in certain situations may not show the exact same values as *kubectl*, as *kubectl* applies certain heuristics to display comprehensible messages. *kube-state-metrics* **exposes raw data unmodified from the Kubernetes API**, this way users have all the data they require and perform heuristics as they see fit.

1.2.2.3. Kubelet service monitor

Kubelet is an important part of Kubernetes control plane, and it also exposes Prometheus metrics by default on port 10255. Actually, *kube-state-metrics* available in kube-prometheus (see above) collects lots of data, and some of them overlap with Kubelet provided metrics, nevertheless some information can be collected only from Kubelet.

1.3. Prometheus server (the main application)

Prometheus is the core application of the stack responsible for scraping, storing and providing access to the monitoring information (see, e.g., [here](#) for a short intro to Prometheus server).

1.3.1. Final configurations of Prometheus server

This subsection describes how to adjust Prometheus CRD to: 1) enable Prometheus access to all required resources, 2) assign persistent storage to Prometheus server, and 3) enable external access to Prometheus dashboard using service type LoadBalancer (similarly as we did for MatalLB). To this end, updates to kube-prometheus manifests stored as described in section 1.1 will be needed.

1.3.1.1. Enabling access to all required resources

As mentioned previously, the access to resources in Kubernetes means the ability to execute CRUD operations on them, obtaining lists of resources, and being notified about state changes of resources. As a monitoring component, Prometheus at least needs to read the state of resources of certain types as well as list and watch them (watching means being notified about changes of resource state). To allow Prometheus this sort of access to resources update the rules attribute in manifest file `prometheus-clusterRole.yaml` according to [this cluster role](#). It defines which apiGroups of Kubernetes API can be accesses by the Prometheus server in our cluster. Take care to preserve the values of all labels in `prometheus-clusterRole.yaml` as in the original version of this file. A short intro to Kubernetes RBAC API can be found [here](#). **Read it to understand what we do.** Technically, you have to complete the **yellow** rule part of the manifest as shown in the following table.

```
$ nano prometheus-clusterRole.yaml
# edit your file/save; don't blindly copy-paste everything from here
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  labels:
    app.kubernetes.io/component: prometheus
    app.kubernetes.io/instance: k8s
    app.kubernetes.io/name: prometheus
    app.kubernetes.io/part-of: kube-prometheus
    app.kubernetes.io/version: 2.42.0 # 2.42.0 is an old version and it will probably differ from
yours - preserve the version from the original kube-prometheus manifest you downloaded
  name: prometheus-k8s
# the following should be filled in according to
# https://github.com/prometheus-operator/prometheus-
operator/blob/main/Documentation/rbac.md#prometheus-rbac
rules:
- apiGroups: [""]
  resources:
    - nodes
    - nodes/metrics
    - services
    - endpoints
    - pods
  verbs: ["get", "list", "watch"]
- apiGroups: [""]
  resources:
    - configmaps
  verbs: ["get"]
- apiGroups:
  - discovery.k8s.io
  resources:
    - endpointslices
  verbs: ["get", "list", "watch"]
- apiGroups:
  - networking.k8s.io
  resources:
    - ingresses
  verbs: ["get", "list", "watch"]
- nonResourceURLs:
  - /metrics
  - /metrics/slis
  verbs: ["get"]
```


1.3.1.2. Assigning persistent storage

Assigning persistent storage to Prometheus server will need updating resource kind *Prometheus* to configure *affinity rules*, *replica number*, and *persistent volume claim* (PVC) for Prometheus server storage. Short explanation of these terms is given below.

Persistent storage is needed to preserve historical monitoring data even across system shutdowns. One would want to declare PVC (Persistent Volume Claim) for Prometheus server directly following Rancher recommendations: <https://github.com/rancher/local-path-provisioner#usage>. However, the guidelines for using *Prometheus operator* provided [here](#) recommend embedding the PVC in Prometheus CRD. Additionally, in case of using *kube-prometheus* (our case) the recommended form of such a PVC specification is given [here](#) (we do not need a long retention time). We will stick to the guidelines outlined in both of those documents.

- **Persistent volume claim explanation:** When deploying an application that needs to retain its data, you'll need to create persistent storage. Persistent storage allows you to store application data external from the pod running your application and maintain application data even if the application's pod fails. **Persistent volume (PV)** is a piece of storage allocated in the Kubernetes cluster, while **persistent volume claim (PVC)** is a request for storage. For details on how PVs and PVCs work and interrelate, refer to the official Kubernetes documentation on storage. Below we use the simplest way of setting up *dynamic* persistent storage with a local-storage provider for Prometheus. More advanced methods using explicit definition of persistent volume claim and persistent volume (also *static*) can be found in:
 - <https://docs.k3s.io/storage>
 - Two recommended interesting discussions:
 - <https://blog.differentpla.net/blog/2021/12/17/k3s-dynamic-pv/>
 - <https://blog.differentpla.net/blog/2021/12/17/k3s-static-pv/>
 - <https://github.com/prometheus-operator/prometheus-operator/blob/main/Documentation/platform/storage.md>
- **Affinity rules explanation:** We want Prometheus not to run on control (master) node(s) to limit the load put onto the control plane in the cluster. This can be achieved in many ways. We already limited the access to the control node(s) for non-critical workloads by setting appropriate flag in k3s install script that resulted in setting taint `CriticalAddonsOnly=true:NoExecute` on the master node. We saw this flag in operation when we discussed why MetalLB speakers run only on worker nodes. Basically, this *taint* should be sufficient to exclude the master from running Prometheus. You can check using `kubectl` where Prometheus pod runs in your cluster. If this taint were not set then other options could be used to achieve the same effect and they are shown/commented in the frame given below. You can try using your preferred option (one is sufficient), carefully commenting/uncommenting respective parts of the manifest if needed. A quick guide on nodeSelectors and affinity rules can be found [here](#).
- **Assigning storage to Prometheus:** Open the file `prometheus-prometheus.yaml` and change its contents according to the frame presented below. Notice we set `replicas=1` (originally it is "2", but one replica is sufficient for us and it also saves resources in our cluster). Watch blank identents when copy-pasting, and DO NOT copy-paste everything, but only yellow parts. Actually, we change only `replicas`, `retention` and `storage` setting. Remaining parts are only examples showing how certain rules could be enforced.

```
# edit file prometheus-prometheus.yaml
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
```



```

metadata:
  labels:
    app.kubernetes.io/component: prometheus
    app.kubernetes.io/instance: k8s
    app.kubernetes.io/name: prometheus
    app.kubernetes.io/part-of: kube-prometheus
    app.kubernetes.io/version: 2.42.0 # keep version the same as in your original file
  name: k8s
  namespace: monitoring
spec:
  alerting:
    alertmanagers:
      - apiVersion: v2
        name: alertmanager-main
        namespace: monitoring
        port: web
  enableFeatures: []
  externalLabels: {}
  image: quay.io/prometheus/prometheus:v2.42.0 # keep version the same as in your original file
# We would like Prometheus to run on worker nodes only, not on the control node. In the following, we
# describe two options that could be used if we had not set CriticalAddonsOnly=true:NoExecute during
# cluster installation.
# Originally, only node selector "kubernetes.io/os: linux" is set for Prometheus (see 3 lines below)
# and to force Prometheus run on workers one could add "node-role.kubernetes.io/worker: true" as shown
# in the commented line that follows "kubernetes.io/os: linux"
  nodeSelector:
    kubernetes.io/os: linux
# node-role.kubernetes.io/worker: true
# As mentioned before, we set taint CriticalAddonOnly for the master in RBPi4 clusters so selector
# worker: true effectively is not needed in this case. On RBpi clusters, I set worker everywhere so
# in my case this selector is also not needed. But the example can demonstrate how workload
# placement can be enforced in Kubernetes.
# disabled Prometheus from running on the master and basically we achieve our goal with the default
# nodeSelector settings.
#
# Alternatively, we can use the affinity mechanism instead of nodeSelector.
# Affinity mechanism in general is more expressive than nodeSelector and offers more flexibility in
# defining placement constraints for Kubernetes workloads. In the following we configure affinity
# rules for Prometheus that give the same effect as the ones based on nodeSelector discussed above.
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: kubernetes.io/os
                operator: In
                values:
                  - linux
          - key: node-role.kubernetes.io/worker
            operator: In
            value:
              - true
  podMetadata:
    labels:
      app.kubernetes.io/component: prometheus
      app.kubernetes.io/instance: k8s
      app.kubernetes.io/name: prometheus
      app.kubernetes.io/part-of: kube-prometheus
      app.kubernetes.io/version: 2.42.0 # keep version the same as in your original file
  podMonitorNamespaceSelector: {}
  podMonitorSelector: {}
  probeNamespaceSelector: {}
  probeSelector: {}
# one replica and retention to 2 days is sufficient for us
  replicas: 1 # one replica will be sufficient in our case
  retention: 2d # we probably do not need longer retention time
  resources:
    requests:
      memory: 400Mi
  ruleNamespaceSelector: {}
  ruleSelector: {}
  securityContext:
    fsGroup: 2000
    runAsNonRoot: true
    runAsUser: 1000
  serviceAccountName: prometheus-k8s
# (blank braces {} mean "all") All service monitors from all namespaces can be scraped by Prometheus
  serviceMonitorNamespaceSelector: {}
  serviceMonitorSelector: {}
  version: 2.42.0

```

```
# SPIW addition: define storage which suits our basic needs and allows to control the amount of disk
# space used; unfortunately, at the time of this writing (March 2025) the declaration of 2Gi will be
# ignored in k3s and as a workaround we use a soft control in the form of retention property set to
# two days (see above)
storage:
  volumeClaimTemplate:
    spec:
      storageClassName: local-path
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 2Gi
```

In the example we use k3s [local path provisioner](#) for creating the volume. It dynamically creates volumes on the nodes. However, this is not network storage. Once created, the volume will forever reside on the k3s node the pod which uses the volume is executed on. During node's downtime, the pods will not be able to start on other nodes, because they won't find their volumes. There's no support for resizing a volume created with local path provisioner. To change the size of the volume, you'll have to destroy it and recreate it (it's your responsibility to backup the data). If you need more flexibility in managing volumes in k3s consider using [Longhorn](#) (also [here](#) and [here](#)). For any case, check the requirements as described [here](#).

In Prometheus CR (the frame presented above) we can indicate the durability of the monitoring data stored (default duration is 24h) thus preserving a level of control over the disk space used by Prometheus. In the example, we have set it to 2 days using the `retention` attribute. In case of facing storage shortages the retention time can be decreased.

1.3.1.3. Exposing Prometheus dashboard as LoadBalancer service

To expose Prometheus dashboard as a service of type LoadBalancer requires some updates to service manifests stored in our `monitoring/` directory. One option is to create a new⁴ manifest for a Service of type LoadBalancer as shown below. In the example we make MetalLB assign a specific external IP address (chosen from the MetalLB pool) to the Prometheus service (`loadBalancerIP: 192.168.1.201`). In your case this address can be different.

```
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ cp prometheus-service.yaml prometheus-service-ext.yaml
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ nano prometheus-service-ext.yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/component: prometheus
    app.kubernetes.io/instance: k8s
    app.kubernetes.io/name: prometheus
    app.kubernetes.io/part-of: kube-prometheus
    app.kubernetes.io/version: 2.42.0 # keep version the same as in the original file
  name: prometheus-k8s-ext
  namespace: monitoring
spec:
  ports:
    - name: web
      port: 9090 # could be changed, eg., with 80, but 9090 will not be in conflict with prometheus-k8s anyway
      targetPort: web
    - name: reloader-web
      port: 8080
      targetPort: reloader-web
  selector:
    app.kubernetes.io/component: prometheus
    app.kubernetes.io/instance: k8s
    app.kubernetes.io/name: prometheus
    app.kubernetes.io/part-of: kube-prometheus
  type: LoadBalancer
  loadBalancerIP: 192.168.1.201 # predefined IP address; adjust to your environment
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$
```

⁴ Creating new service is optional. You can well declare LoadBalancer by modifying the existing manifest `prometheus-service.yaml`.

1.3.2. Deploying the updates to a running Prometheus server

- a) Run the `apply` command, check the location of the Prometheus pod and the PV (actual volume) (in our case it is worker kpi094).

```
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ cd ..
xubuntu@xubulab:~/k3s-taskforce/manifests$ kubectl apply -f monitoring/
# ... skipping the output from command execution

# checking the created Pod, PVC and PV and their placement

xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ kubectl get pods -n monitoring -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP            NODE
prometheus-k8s-0                    2/2     Running   0           63m   10.42.1.58    kpi094
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ kubectl get pvc -n monitoring
NAME                                STATUS    VOLUME                                     CAPACITY
ACCESS MODES   STORAGECLASS   AGE
prometheus-k8s-db-prometheus-k8s-0 Bound        pvc-b40a14ef-5a6b-49ce-bb6e-53bc56a489ca    2Gi
RWO            local-path     68m
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ kubectl describe -n monitoring pvc prometheus-
k8s-db-prometheus-k8s-0
Name:          prometheus-k8s-db-prometheus-k8s-0
Namespace:     monitoring
StorageClass:  local-path
Status:        Bound
Volume:        pvc-b40a14ef-5a6b-49ce-bb6e-53bc56a489ca
Labels:        app.kubernetes.io/instance=k8s
                app.kubernetes.io/managed-by=prometheus-operator
                app.kubernetes.io/name=prometheus
                operator.prometheus.io/name=k8s
                operator.prometheus.io/shard=0
                prometheus=k8s
Annotations:   pv.kubernetes.io/bind-completed: yes
                pv.kubernetes.io/bound-by-controller: yes
                volume.beta.kubernetes.io/storage-provisioner: rancher.io/local-path
                volume.kubernetes.io/selected-node: kpi094
                volume.kubernetes.io/storage-provisioner: rancher.io/local-path
Finalizers:    [kubernetes.io/pvc-protection]
Capacity:      2Gi
Access Modes:  RWO
VolumeMode:    Filesystem
Used By:       prometheus-k8s-0
Events:        <none>
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ kubectl describe -n monitoring pv pvc-
b40a14ef-5a6b-49ce-bb6e-53bc56a489ca
Name:          pvc-b40a14ef-5a6b-49ce-bb6e-53bc56a489ca
Labels:        <none>
Annotations:   pv.kubernetes.io/provisioned-by: rancher.io/local-path
Finalizers:    [kubernetes.io/pv-protection]
StorageClass:  local-path
Status:        Bound
Claim:         monitoring/prometheus-k8s-db-prometheus-k8s-0
Reclaim Policy: Delete
Access Modes:  RWO
VolumeMode:    Filesystem
Capacity:      2Gi
Node Affinity:
  Required Terms:
    Term 0:      kubernetes.io/hostname in [kpi094]
Message:
Source:
  Type:          HostPath (bare host directory volume)
  Path:          /var/lib/rancher/k3s/storage/pvc-b40a14ef-5a6b-49ce-bb6e-
53bc56a489ca_monitoring_prometheus-k8s-db-prometheus-k8s-0
  HostPathType:  DirectoryOrCreate
Events:         <none>
```

It can be seen that the pod and the PV (the volume) are placed in the same node – kpi094. Interesting fact is that the PV has the property Node-Affinity set to `kubernetes.io/hostname in [kpi094]`; the PVC also has annotation `volume.kubernetes.io/selected-node: kpi094`. That means that if node kpi094 fails than

even if the Prometheus pod will be rescheduled to another node, the volume will not due to being bound to node kpi094.

b) Check the access to the Prometheus dashboard

External (i.e., from your local network) access to Prometheus dashboard can be checked via browser or curl trying to reach <prometheus-service-ip-add>:9090. With all installation details as outlined above, you will get HTTP 404 error. This is because the default NetworkPolicy for Prometheus disallows such traffic and the policy needs to be updated. In the following frame the steps needed to check the reachability and update NetworkPolicy to enable external access are documented. Notice that Kubernetes NetworkPolicy mechanism is in essence similar to Security Groups known from OpenStack, although functionally richer.

NOTE: NetworkPolicy mechanism seems to be a bit loose in regard to address blocks (*ipBlock*) in the sense that it is not unequivocally defined (known for us as users) which exactly address will be handled by the rule. The reason is that cluster ingress and egress mechanisms often require rewriting the source or destination IP of packets. In effect, at the moment of writing the manifest it is unknown whether the rule will be applied to the address set by the application (as we would like it to be the case) or it will be applied to address substituted by Kubernetes (by ingress controller/kube-proxy, ...). This results in that NetworkPolicy mechanism should be used with great care for *ipBlocks*. A relevant note on that can be found [here](#), and some useful NetworkPolicy recipes can be found [here](#). That is also the reason why certain CNI plugins (e.g. Cilium) offer their own and rich network policy frameworks.

```
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ kubectl get services -n monitoring
NAME                                TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S)
prometheus-operator                ClusterIP      None            <none>           8080/TCP
node-exporter                      ClusterIP      None            <none>           9100/TCP
kube-state-metrics                 ClusterIP      None            <none>           8443/TCP,9443/TCP
prometheus-k8s-ext                 LoadBalancer  10.43.253.155   192.168.1.201    9090:30922/TCP
prometheus-k8s                     ClusterIP      10.43.177.245   <none>           9090/TCP,8080/TCP
prometheus-k8s-nodeport            NodePort       10.43.56.246    <none>           9090:30090/TCP
prometheus-operated                ClusterIP      None            <none>           9090/TCP

xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ curl 192.168.1.201:9090
curl: (7) Failed to connect to 192.168.1.202 port 9090: No route to host

xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ nano prometheus-networkPolicy.yaml
# update adding a rule to allow traffic from the local net to the pod address space of the cluster
# ipBlock cidr: 10.42.0.0/16 has been added as an addition to cope with NetworkPolicy uncertainties
# explained above and observed in practice.
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  labels:
    app.kubernetes.io/component: prometheus
    app.kubernetes.io/instance: k8s
    app.kubernetes.io/name: prometheus
    app.kubernetes.io/part-of: kube-prometheus
    app.kubernetes.io/version: 2.42.0
  name: prometheus-k8s
  namespace: monitoring
spec:
  egress:
  - {}
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app.kubernetes.io/name: prometheus
      ports:
      - port: 9090
        protocol: TCP
      - port: 8080
        protocol: TCP
    - from:
    - podSelector:
        matchLabels:
          app.kubernetes.io/name: grafana
      ports:
      - port: 9090
        protocol: TCP
# with cidr: 10.42.0.0/16 as below ipBlock cidr: 192.168.1.0/24 may not be necessary - to be tested
# keeping both blocks is safe (for simplicity, one could use one block 0.0.0.0/0 to allow ALL ingress)
  - from:
    - ipBlock:
        cidr: 192.168.1.0/24
    - ipBlock:
        cidr: 10.42.0.0/16
  podSelector:
    matchLabels:
      app.kubernetes.io/component: prometheus
      app.kubernetes.io/instance: k8s
      app.kubernetes.io/name: prometheus
      app.kubernetes.io/part-of: kube-prometheus
  policyTypes:
  - Egress
  - Ingress

# check the reachability of Prometheus dashboard again
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ kubectl apply -f prometheus-networkPolicy.yaml
networkpolicy.networking.k8s.io/prometheus-k8s configured
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ curl 192.168.1.201:9090
<a href="/graph">Found</a>.
```

c) Check if Prometheus is monitoring selected services

After achieving reachability you can access Prometheus dashboard with browser to see its main panels and check the health of installed monitors. To this end enter 192.168.1.201:9090 in your browser.

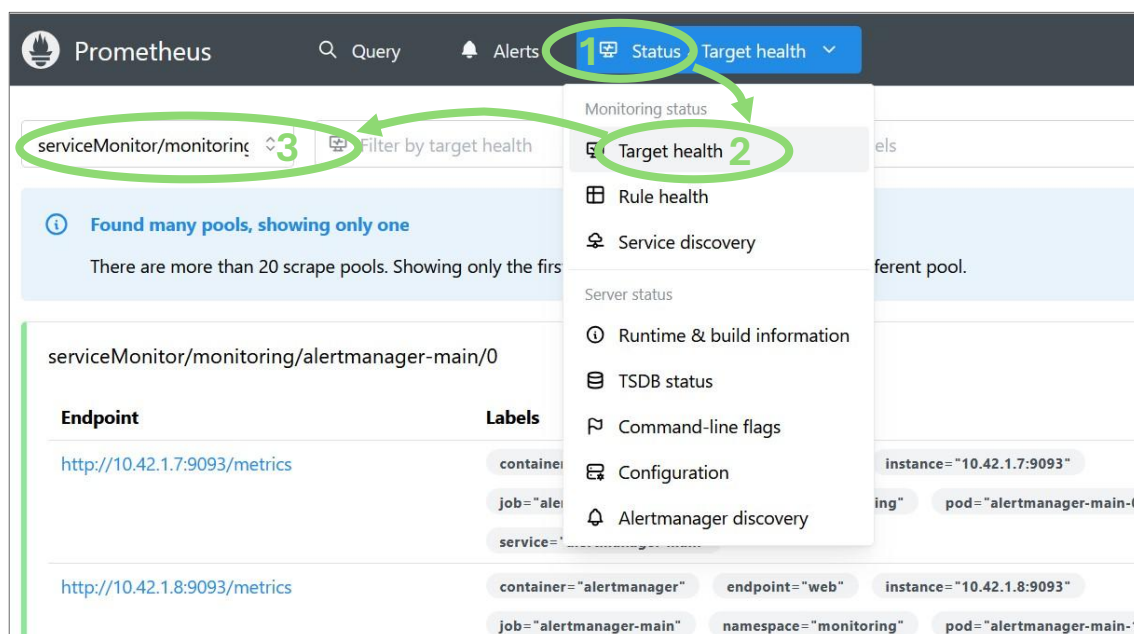


Figure 1 First touch of Prometheus dashboard – just checking how it looks like.

How to query with PromQL: <https://www.opsramp.com/guides/prometheus-monitoring/promql/>

1.4. Optional: deploying monitoring extensions in our cluster

Other monitors for our platform components⁵ can be added, for example one may want to monitor Traefik or MetalLB. In the following, instructions are provided to include such extensions. This step is optional, but it may be valuable for newcomers by presenting how simple it is to start monitoring new application component once appropriate service monitor is available (i.e., when one does not have to write one on her/his own).

1.4.1. Traefik service monitor

Traefik comes with its Prometheus service monitor. If so, it can be good to also have Traefik metrics covered in our monitoring stack. Below is a list of reference links for Traefik service monitor we will use:

<https://github.com/cablespaghetti/k3s-monitoring>

<https://github.com/cablespaghetti/k3s-monitoring/blob/master/traefik-servicemonitor.yaml>

<https://raw.githubusercontent.com/cablespaghetti/k3s-monitoring/master/traefik-servicemonitor.yaml>

Follow the steps from the table below to deploy Traefik service monitor for Prometheus.

⁵ Of course, Prometheus monitors for „regular“ application services can also be installed but such applications are not covered in this part of the lab and can be included in the future as an extension of our lab series.


```
# make directory and download the manifests
xubuntu@xubulab:~/k3s-taskforce/manifests$ mkdir traefik-service-monitor
xubuntu@xubulab:~$ cd traefik-service-monitor
xubuntu@xubulab:~/k3s-taskforce/manifests/traefik-service-monitor$ wget
https://raw.githubusercontent.com/cablespaghetti/k3s-monitoring/master/traefik-
servicemonitor.yaml

#update the manifest to add namespace: monitoring if it is missing:
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    app: traefik
    release: prometheus
    name: traefik
    namespace: monitoring
spec:
  endpoints:
    - port: metrics
  namespaceSelector:
    matchNames:
      - kube-system
  selector:
    matchLabels:
      app: traefik

# deploy traefik service monitor
xubuntu@xubulab:~/k3s-taskforce/manifests/traefik-service-monitor$ kubectl apply -f traefik-
servicemonitor.yaml

# check service monitors deployed so far
xubuntu@xubulab:~/k3s-taskforce/manifests$ kubectl get servicemonitors -A
NAMESPACE      NAME                      AGE
monitoring     traefik                   16s
monitoring     kube-state-metrics       9m49s
monitoring     kubelet                   6m41s
monitoring     node-exporter             12m
xubuntu@xubulab:~/k3s-taskforce/manifests/traefik-service-monitor$
```

2. Grafana

Grafana provides a rich set of dashboards to visualize the metrics scraped by Prometheus from the services monitored.

2.1. Manifest preparation

2.1.1. Update the manifests

- a) Create⁶ LoadBalancer service for Grafana and update NetworkPolicy to enable external traffic to Grafana. These steps are similar to the ones from the previous section for Prometheus. Remember to set an unused IP address from MetalLB pool if you want to choose the address yourself (otherwise MetalLB will assign one on its own). Finally, update grafana-deployment.yaml. All details are shown in the table below.

⁶ Creating new service is optional. You can well declare LoadBalancer by modifying the existing manifest grafana-service.yaml.

```

# create a new file, say grafana-service-ext.yaml, defining LoadBalancer service for Grafana
dashboard
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/component: grafana
    app.kubernetes.io/name: grafana
    app.kubernetes.io/part-of: kube-prometheus
    app.kubernetes.io/version: 9.1.7
  name: grafana-ext
  namespace: monitoring
spec:
  type: LoadBalancer
  loadBalancerIP: 192.168.1.202
  ports:
    - name: http
      port: 3000
      targetPort: http
  selector:
    app.kubernetes.io/component: grafana
    app.kubernetes.io/name: grafana
    app.kubernetes.io/part-of: kube-prometheus

# update file grafana-networkPolicy.yaml adding the following in the "ingress" part the same way
# as for Prometheus:
- from:
  - ipBlock:
      cidr: 192.168.1.0/24
  - ipBlock:
      cidr: 10.42.0.0/16

# NOTE: the part below can be skipped only when installing the ORIGINAL version of kube-prometheus
# (as exactly described in kube-prometheus github). Otherwise, manifest grafana-deployment.yaml
# has to be updated. In our case, WE DO NOT FOLLOW the original install as we have changed some
# details. Therefore, we need to update the description of the readiness probe in Grafana deployment
# as indicated below.
# You may want to first check what happens without updating the readiness probe settings; to this
# end, you will need to run the command:
#   kubectl describe pod -n monitoring <grafana-pod-current-name>
# and check the Events section at the very end. What do you find?
# All in all, the readiness probe test needs more time to pass, and the yellow part below should
# be added in the manifest grafana-deployment.yaml as shown below (only uncommented lines matter).

    - containerPort: 3000
      name: http
    readinessProbe:
      httpGet:
        path: /api/health
        port: http

# SPIW added - beginning >>>>
      failureThreshold: 3
      initialDelaySeconds: 10
      periodSeconds: 30
      successThreshold: 1
      timeoutSeconds: 2

#-----
#       livenessProbe:
#         failureThreshold: 3
#         initialDelaySeconds: 30
#         periodSeconds: 10
#         successThreshold: 1
#         tcpSocket:
#           port: 3000
#         timeoutSeconds: 1
# SPIW added - end <<<<

```

b) Configure persistent storage for Grafana

Grafana does not store monitoring data from Prometheus, but it uses dashboards and their JSON templates must be available to Grafana in run time. If you do not designate a location for information storage, then all your Grafana data including dashboards disappears as soon as you stop your container. Therefore, to save our data, we need to set up persistent storage for Grafana. You have to create grafana-pvc.yaml file and update grafana-deployment.yaml. Follow the steps in the frame below. Noticeably, this is another way of providing persistent volume, different from the one we used for Prometheus.

```
# Create and edit file grafana-pvc.yaml to define a volume claim for grafana
```

```
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring/grafana$ nano grafana-pvc.yaml
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: grafana-local-path-pvc
  namespace: monitoring
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: local-path
  resources:
    requests:
      storage: 500Mi
```

```
# Update grafana deployment file to create a volume from the pvc defined above.
```

```
# To this end search for the right location in the deployment file of the phrase volumes of the
```

```
# grafana container - it follows the attribute serviceAccountName: Grafana.
```

```
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring/grafana$ nano grafana-deployment.yaml
```

```
. . .
```

```
# serviceAccountName: grafana # this part has to be found
```

```
# volumes:
```

```
# Grafana-storage will be persisting to store dashboard templates; remaining volumes
```

```
# should remain ephemeral (secret {}, configMap {}, etc.).
```

```
# Volume emptyDir {} name: grafana-storage should be commented out as being substituted by our
# persistent volume. The yellow part to be added, emptyDir commented out/deleted, and the beginning
# of section serviceAccountName: grafana can look like this:
```

```
serviceAccountName: grafana
```

```
volumes:
```

```
- name: grafana-storage
```

```
  persistentVolumeClaim:
```

```
    claimName: grafana-local-path-pvc
```

```
# - emptyDir: {} # to be commented out/deleted
```

```
# name: grafana-storage # to be commented out/deleted
```

2.1.2. Deploy Grafana, check main components and open the first dashboard

a) Edit updated elements (service, policy, pvc, deployment) to make sure all is fine. Deploy Grafana stack.

```
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ nano grafana-service-ext.yaml
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ nano grafana-networkPolicy.yaml
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ nano grafana-pvc.yaml
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ nano grafana-deployment.yaml
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ cd ..
xubuntu@xubulab:~/k3s-taskforce/manifests$ kubectl apply -f monitoring/
alertmanager.monitoring.coreos.com/main unchanged

. . .

xubuntu@xubulab:~/k3s-taskforce/manifests$ kubectl get pods -n monitoring
NAME                                READY   STATUS    RESTARTS   AGE
node-exporter-r454m                 2/2     Running   2 (22h ago) 27h
node-exporter-xlhbs                 2/2     Running   2 (22h ago) 27h
prometheus-operator-7b4d465f47-785hb 1/1     Running   1 (22h ago) 27h
node-exporter-th6k7                 2/2     Running   2 (22h ago) 27h
node-exporter-ggxr5                 2/2     Running   2 (22h ago) 27h
kube-state-metrics-8658546b69-c6wnk 3/3     Running   3 (22h ago) 27h
prometheus-k8s-0                    2/2     Running   2 (22h ago) 23h
grafana-96d64cd76-jt82p             1/1     Running   0           88s
xubuntu@xubulab:~/k3s-taskforce/manifests$ kubectl get pvc -n monitoring
NAME                                STATUS    VOLUME                                     CAPACITY   ACCESS MODES
STORAGECLASS  AGE
prometheus-k8s-db-prometheus-k8s-0  Bound     pvc-a4da82ff-b549-48a8-8e62-6e9a35e864d9  2Gi        RWO
local-path    23h
grafana-local-path-pvc               Bound     pvc-ad9d7a13-75cb-4522-a8d0-fe9627cb0cce  500Mi      RWO
local-path    2m42s
xubuntu@xubulab:~/k3s-taskforce/manifests$ kubectl describe -n monitoring pvc grafana-local-path-pvc
Name:          grafana-local-path-pvc
Namespace:     monitoring
StorageClass:  local-path
Status:        Bound
Volume:        pvc-ad9d7a13-75cb-4522-a8d0-fe9627cb0cce
Labels:        <none>
Annotations:   pv.kubernetes.io/bind-completed: yes
               pv.kubernetes.io/bound-by-controller: yes
               volume.beta.kubernetes.io/storage-provisioner: rancher.io/local-path
               volume.kubernetes.io/selected-node: kpi093
               volume.kubernetes.io/storage-provisioner: rancher.io/local-path
Finalizers:    [kubernetes.io/pvc-protection]
Capacity:      500Mi
Access Modes:  RWO
VolumeMode:    Filesystem
Used By:       grafana-96d64cd76-jt82p
Events:
  Type     Reason          Age          From          Message
  --     -
xubuntu@xubulab:~/k3s-taskforce/manifests$ kubectl get services -n monitoring
NAME                                TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
prometheus-operator                 ClusterIP       None             <none>            8080/TCP         28h
node-exporter                       ClusterIP       None             <none>            9100/TCP         27h
kube-state-metrics                  ClusterIP       None             <none>            8443/TCP,9443/TCP 27h
prometheus-k8s-external              LoadBalancer   10.43.253.155    192.168.1.201    9090:30922/TCP   24h
prometheus-k8s                      ClusterIP       10.43.177.245    <none>            9090/TCP,8080/TCP 24h
prometheus-k8s-nodeport              NodePort        10.43.56.246     <none>            9090:30090/TCP   24h
prometheus-operated                  ClusterIP       None             <none>            9090/TCP         24h
grafana-external                     LoadBalancer   10.43.125.169    192.168.1.202    3000:31921/TCP   5m39s
grafana-local                       ClusterIP       10.43.56.206     <none>            3000/TCP         5m39s
xubuntu@xubulab:~/k3s-taskforce/manifests$
```

2.2. Checking Grafana – first graphs

a) Checking the access to Grafana dashboard

Disclaimer: this section is not intended to be a tutorial on using Grafana and Prometheus, but rather an entry point to start checking our cluster metrics in the most straightforward way. To acquire deeper knowledge and skills the reader is referred to other sources.

Enter **192.168.1.202:3000** in your browser and you will be presented the invitation panel of Grafana.

On the first logging use the credentials **admin:admin**, and you will be prompted to set new ones (reentering admin:admin does not work, at least the pwd has to be changed). Then follow the steps⁷ from Figure 2 for setting up a simple dashboard for the cluster: <https://grafana.com/grafana/dashboards/8171-kubernetes-nodes/>. This dashboard is very simple and describes only servers. However, other dashboards with more information about the cluster (particular workloads, etc.) exist and can be used. Useful dashboards that should work out of the box are the following (you will have to figure out the meaning of all their graphs on your own):

- Currently (February 2025), kube-prometheus comes with built-in dashboard named **Default**. It provides quite rich set of statistics from the cluster and it is sufficient to select it from the *Dashboards* panel. In reference to Figure 2, it will be sufficient to make step 2 and then select *Default* in the *Dashboards* panel.
- Kubernetes Cluster (cluster level overview of workloads deployed, based on Prometheus metrics exposed by kubelet, node-exporter, nginx ingress controller) <https://grafana.com/grafana/dashboards/7249-kubernetes-cluster/>.
- Kubernetes Cluster (Prometheus) (summary metrics for containers running on Kubernetes nodes) <https://grafana.com/grafana/dashboards/6417-kubernetes-cluster-prometheus/>.

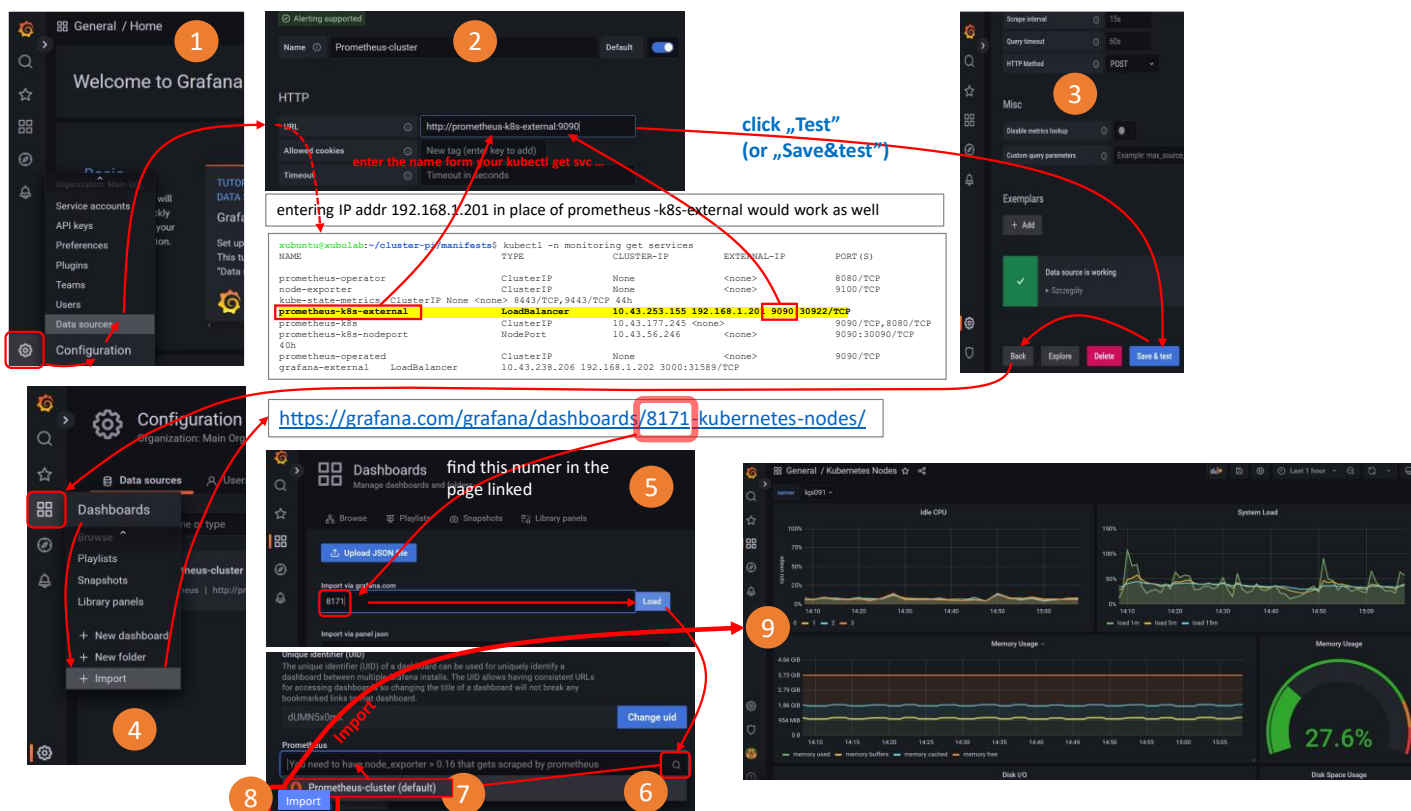


Figure 2 Configuring exemplary Grafana dashboard for the cluster.

⁷ Actually, Grafana panel has changed since the time of making this drawing so you have to interpret the figure creatively. Nevertheless, there are always two main steps to use a dashboard: you have to indicate the **Data source** of monitoring information (i.e., from where to read the data: *Connections->Data sources*) and then select (or import for the first time) Grafana **Dashboard** you want to use for presenting the graphs (*Dashboards->select or import dashboard*). As Data source, our Prometheus instance is reachable at <http://<IPaddress-of-ext-service>:9090> or <http://prometheus-k8s-ext:9090>.

3. Monitoring at the resource and service levels

This part of the lab aims to provide insight into the differences between resource and service level monitoring in the context of resource and service orchestration. For obvious reasons, the demonstration is very simple, but we hope it will be enough to gain a first understanding of the nature of management at both levels.

3.1. Resource metric monitoring: loading the cluster using congestion pod

The goal of this part of the lab is to provide basic intuition on resource monitoring that can be used by resource management/orchestration, unaware of the service aspects of management. The scenario involves monitoring artificial load created with the use of custom pods.

The resource monitoring experiment is described [here](#). Refer to that page to complete this part of the lab.

3.2. Service metric monitoring: monitoring the number of user sessions in 5G network

The main objective of this part of the laboratory is to show how service-level monitoring can be used for resource orchestration where decision-making is based on service level indicators. This should be clearly distinguished from resource monitoring and resource-level management/orchestration based on resource-level metrics (as CPU or RAM usage).

The scenario involves monitoring an emulated 5G network. The RAN part of the network will be emulated using UERANSIM, 5G core will be built upon containerized Open5GS platform, and the deployment of both components will be based on slightly customized [Gradiant Helm charts](#).

The topic of 5G network configuration and deployment is discussed briefly to understand the structure of the deployment in terms of 5G network architecture. We will only cover the absolute minimum necessary to run the 5G environment and manage the number of active terminals in the network. Then we will monitor a selected service-level metric using Prometheus. This will underpin our understanding of service monitoring in preparation to Project 2.

In the project, we will design and deploy a simple resource level orchestrator operating based on service-level metrics. It will monitor the number of user sessions in the network and scale vertically the UPF⁸ (User Plane Function) container. Scaling operation will be triggered by the number of user sessions crossing predefined thresholds. Since our primary focus is on service metric monitoring and scaling resources, we won't go into too much detail about how the 5G core works.

The service monitoring experiment and mandatory details related to Open5GS are described [here](#). Refer to that page to complete the lab. **Important: use the branch `upf-scaling-spiw`.**

⁸ In 5G core network architecture, UPF is the function responsible for terminating GTP tunnels that carry user traffic to/from user terminals. It handles all data plane traffic (in volume, dominant traffic in the network). In the basic scenario, those tunnels span between UPF nodes and gNB nodes (the base stations).

4. Useful readings and videos

(Note: new valuable stuff is appearing all the time so one can find other interesting sources.)

Kubernetes troubleshooting

<https://lumigo.io/kubernetes-troubleshooting/>

<https://www.groundcover.com/kubernetes-troubleshooting>

<https://komodor.com/learn/kubernetes-troubleshooting-the-complete-guide/>

Node/pod Affinity/anti-affinity explained: <https://medium.com/kubernetes-tutorials/learn-how-to-assign-pods-to-nodes-in-kubernetes-using-nodeselector-and-affinity-features-e62c437f3cf8>

topologyKey: `kubernetes.io/hostname` – scope of single host

topologyKey: `kubernetes.io/zone` – scope of single zone

`node-role.kubernetes.io/worker=true`

`node-role.kubernetes.io/controlplane=true`

`kubectl label node <node name> node-role.kubernetes.io/<role name>=<value>`

`kubectl get nodes --show-labels`

- kubectl cheat-sheet

<https://kubernetes.io/docs/reference/kubectl/cheatsheet/>

- delete all resources from namespace

<https://stackoverflow.com/questions/47128586/how-to-delete-all-resources-from-kubernetes-one-time>

- How to Monitor a Kubernetes Cluster in 2022 with Prometheus & Grafana

<https://www.youtube.com/watch?v=YDtuwINTzRc>

- Modern Grafana dashboards for kubernetes

<https://medium.com/@dotdc/a-set-of-modern-grafana-dashboards-for-kubernetes-4b989c72a4b2>

- Adding Grafana dashboard using configMap

<https://fabianlee.org/2022/07/06/prometheus-adding-a-grafana-dashboard-using-a-configmap/>

- Simple Prometheus & Grafana setup with PVC

<https://medium.com/@bmbvfx/kubernetes-persistent-volume-claim-prometheus-grafana-4e821e283edc>

- Accessing Kubernetes services

OK: <https://blog.alexellis.io/primer-accessing-kubernetes-services/>

<https://kubernetes.io/docs/tasks/access-application-cluster/access-cluster-services/>

Prometheus/AlertManager/Grafana/Loki - ciekawe, do pooglądania, wykorzystanie AlertManager

<https://www.youtube.com/watch?v=NABZqKq1McE>

przy okazji: strona z opisami helm-chartów dla aplikacji - B. CIEKAWA

<https://artifacthub.io/>

CNCF landscape

<https://landscape.cncf.io/>

Kubernetes API Basics - Resources, Kinds, and Objects

<https://iximiuz.com/en/posts/kubernetes-api-structure-and-terminology/>