eBPF Exporter

Getting timer counters into Prometheus

```
code: |
  BPF_HASH(counts, u64);

  // Generates tracepoint__timer__timer_start
  TRACEPOINT_PROBE(timer, timer_start) {
      counts.increment((u64) args->function);
      return 0;
  }
```

Code to run in the kernel and populate the map

```
metrics:
  counters:
    - name: timer_start_total
      help: Timers fired in the kernel
      table: counts
      labels:
        - name: function
          size: 8
          decoders:
            - name: ksym
tracepoints:
  timer:timer_start: tracepoint__timer__timer_start
```

How to turn map into metrics readable by Prometheus

CLOUDFLARE

Pros

- **Lightweight and efficient metric collection** using eBPF technology
- **Highly customizable** with support for various eBPF programs and metrics
- I**ntegration with Prometheus** ecosystem for easy monitoring and alerting

Cons

- **Steep learning curve** for users unfamiliar with eBPF programming
- **Limited documentation for advanced use cases** and custom program development

NOTE :

- **ebpf_exporter does NOT run bpftrace or BCC programs directly.**

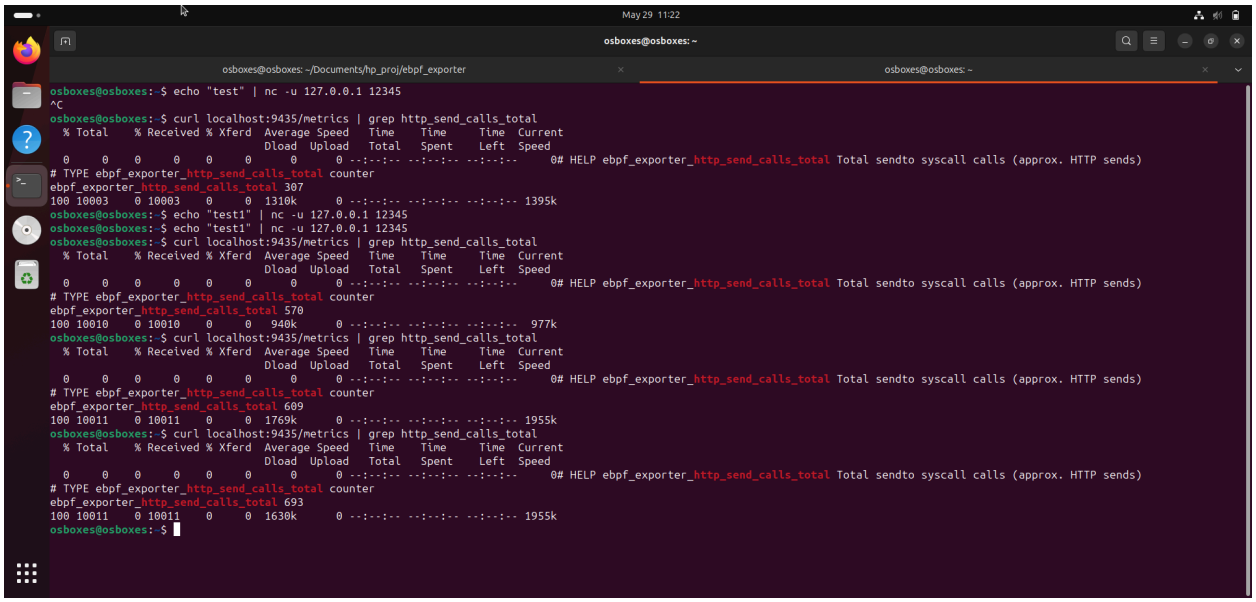
- ebpf_exporter *only* loads precompiled `.bpf.o` eBPF ELF objects.
- To "use bpftrace or BCC" with ebpf_exporter, you'd need to recompile their code to `.bpf.o` (non-trivial, requires writing eBPF C code).
- Otherwise, use BCC or bpftrace *alone* and create your own exporter layer for Prometheus metrics.

Good link with comparison : https://best-of-web.builder.io/library/cloudflare/ebpf_exporter

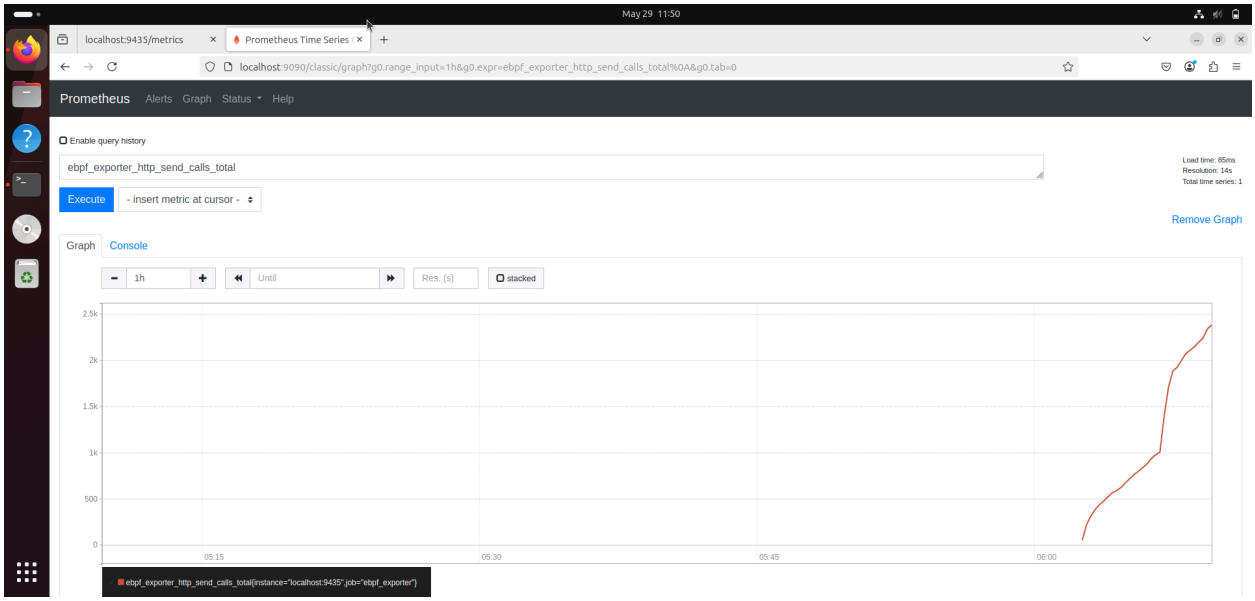
---

Running the exporter for a ebpf program tracking - http_send_calls_total

Output of curl from the metrics page where ebpf exporter stores metrics



Prometheus integration-



Trial eBPF exporter setup monitors the number of `sendto()` syscalls on your machine in real-time and exposes that count as a Prometheus metric.

INTEGRATION WITH iSCSI

Step	How it fits in the setup
Write eBPF code	Attach to block or network tracepoints/kprobes
Compile	Compile eBPF C to `.bpf.o`
Configure	Add metrics to `config.yaml` mapping BPF maps to Prometheus metrics
Run exporter	Start `ebpf_exporter` pointing to your config
Scrape	Prometheus scrapes metrics endpoint
Visualize	Grafana dashboard for iSCSI metrics

Textfile Collector

- The **`textfile-collector`** is a feature of **Prometheus Node Exporter** that lets you **manually write custom metrics to a file(iscsi_metrics.prom)**, and Node Exporter will **automatically expose those metrics** to Prometheus.
- In this case, use **iostat** to scrape read/write IOPs. The metrics are called `iscsi_lun_write_kbps` and `iscsi_lun_read_kbps` but this is a misnomer. Metrics printed are from device=sda i.e hardware level.
- Using helm, setup chart for prometheus and grafana in the namespace monitoring. Wait for all pods to get into running state.
- Use command `kubectl edit daemonset prometheus-prometheus-node-exporter -n monitoring` to edit the deployment for node exporter to add under the section args `--collector.textfile.directory=/textfile-collector`

  Under the section volumeMounts:

```
  - name: textfile-metrics
    mountPath: /textfile-collector
    readOnly: false
```

  Under section volumes:

```
- name: textfile-metrics
  hostPath:
    path: /var/lib/node_exporter/textfile_collector
```

- Edit and save deployment. It will restart pods after this edit.
- In this case, we have used minikube. So, ssh into minikube shell and execute these commands:

```
sudo mkdir -p /var/lib/node_exporter/textfile_collector
iscsi_dummy_metric{lun="test"} 1' | sudo tee
/var/lib/node_exporter/textfile_collector/iscsi.prom
sudo tee /usr/local/bin/iscsi_metrics.sh > /dev/null << 'EOF'
#!/bin/bash

OUTPUT="/var/lib/node_exporter/textfile_collector/iscsi_metrics.prom
"
exec > "$OUTPUT"
exec 2>&1

echo "# HELP iscsi_lun_read_kbps Read KB/s from iSCSI LUN"
echo "# TYPE iscsi_lun_read_kbps gauge"

read_kbps=$(iostat -d /dev/sda | awk 'NR==4 {print $3}')
echo "iscsi_lun_read_kbps{device=\"sda\"} $read_kbps"
```

```
echo "# HELP iscsi_lun_write_kbps Write KB/s from iSCSI LUN"
echo "# TYPE iscsi_lun_write_kbps gauge"

write_kbps=$(iostat -d /dev/sda | awk 'NR==4 {print $4}')
echo "iscsi_lun_write_kbps{device=\"sda\"} $write_kbps"
EOF
sudo apt install -y sysstat' 'sudo apt-get install -y cron
sudo service cron start
sudo systemctl enable cron
sudo crontab -e
```

```
[04/17/25]seed@VM:~/Downloads$ minikube ssh
docker@minikube:~$ sudo tee /usr/local/bin/iscsi_metrics.sh > /dev/null << 'EOF'
#!/bin/bash

OUTPUT="/var/lib/node_exporter/textfile_collector/iscsi_metrics.prom"
exec > "$OUTPUT"
exec 2>&1

echo "# HELP iscsi_lun_read_kbps Read KB/s from iSCSI LUN"
echo "# TYPE iscsi_lun_read_kbps gauge"

read_kbps=$(iostat -d /dev/sdb | awk 'NR==4 {print $3}')
echo "iscsi_lun_read_kbps{device=\"sdb\"} $read_kbps"

echo "# HELP iscsi_lun_write_kbps Write KB/s from iSCSI LUN"
EOFo "iscsi_lun_write_kbps{device=\"sdb\"} $write_kbps"')
docker@minikube:~$
docker@minikube:~$ sudo chmod +x /usr/local/bin/iscsi_metrics.sh
docker@minikube:~$ sudo /usr/local/bin/iscsi_metrics.sh
cat /var/lib/node_exporter/textfile_collector/iscsi_metrics.prom
```

- In the prompt that comes up, add this line in the end of file and save:

```
* * * * * /usr/local/bin/iscsi_metrics.sh
```

- To check these changes:

```
cat /var/lib/node_exporter/textfile_collector/iscsi_metrics.prom
```

Also use kubectl exec to check in the pod as well

```
# TYPE iscsi_lun_read_kbps gauge
iscsi_lun_read_kbps{device="sda"} 258.34
# HELP iscsi_lun_write_kbps Write KB/s from iSCSI LUN
# TYPE iscsi_lun_write_kbps gauge
iscsi_lun_write_kbps{device="sda"} 408.67
docker@minikube:~$ cat /var/lib/node_exporter/textfile_collector/is
prommetrics.p
# HELP iscsi_lun_read_kbps Read KB/s from iSCSI LUN
# TYPE iscsi_lun_read_kbps gauge
iscsi_lun_read_kbps{device="sda"} 249.04
# HELP iscsi_lun_write_kbps Write KB/s from iSCSI LUN
# TYPE iscsi_lun_write_kbps gauge
iscsi_lun_write_kbps{device="sda"} 396.17
docker@minikube:~$
```

```
echo 'iscsi_dummy_metric{lun="test"} 1' | sudo tee /var/lib/node_exporter/textf
ile_collector/iscsi.prom
iscsi_dummy_metric{lun="test"} 1
docker@minikube:~$ ls /var/lib/node_exporter/textfile_collector
iscsi.prom
docker@minikube:~$ exit
logout
[04/17/25]seed@VM:~/Downloads$ kubectl exec -it prometheus-prometheus-node-expo
rter-gcq58 -n monitoring -- ls /textfile-collector
iscsi.prom
[04/17/25]seed@VM:~/Downloads$ curl -s http://localhost:9100/metrics | grep isc
si_dummy_metric
# HELP iscsi_dummy_metric Metric read from /textfile-collector/iscsi.prom
# TYPE iscsi_dummy_metric untyped
iscsi_dummy_metric{lun="test"} 1
[04/17/25]seed@VM:~/Downloads$
```

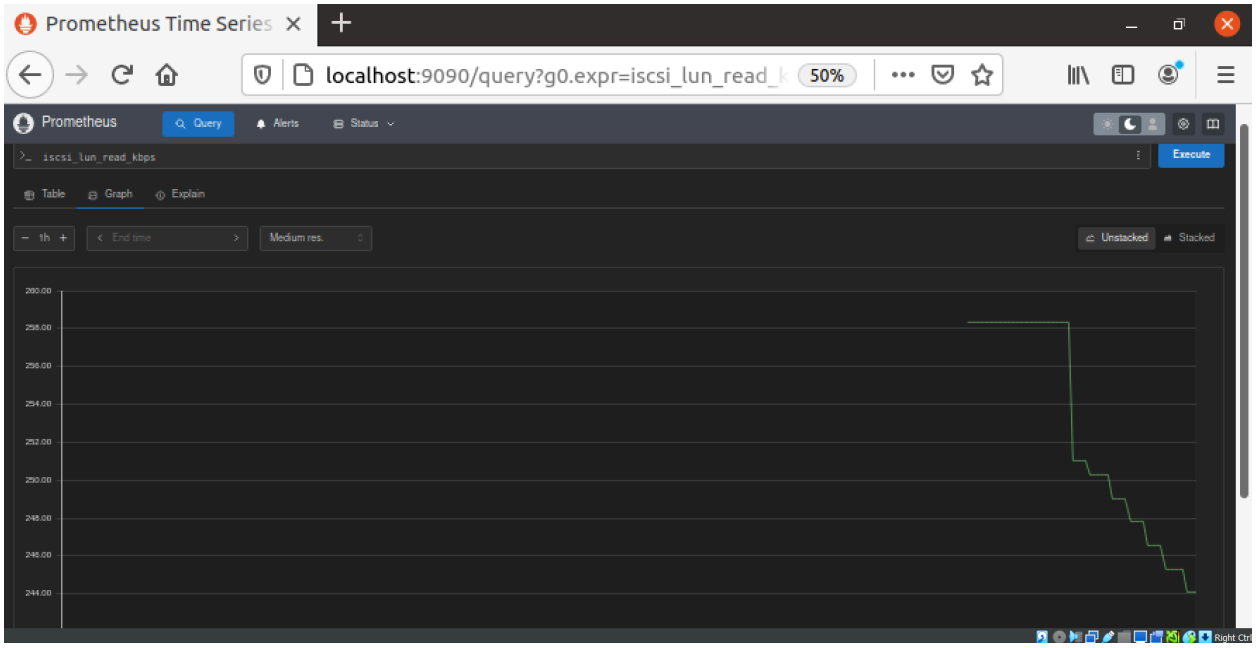
```
[04/17/25]seed@VM:~/Downloads$ curl -s http://localhost:9100/metrics | grep isc
si
# HELP iscsi_dummy_metric Metric read from /textfile-collector/iscsi.prom
# TYPE iscsi_dummy_metric untyped
iscsi_dummy_metric{lun="test"} 1
# HELP iscsi_lun_read_kbps Read KB/s from iSCSI LUN
# TYPE iscsi_lun_read_kbps gauge
iscsi_lun_read_kbps{device="sda"} 247.78
# HELP iscsi_lun_write_kbps Write KB/s from iSCSI LUN
# TYPE iscsi_lun_write_kbps gauge
iscsi_lun_write_kbps{device="sda"} 394.62
node_textfile_mtime_seconds{file="/textfile-collector/iscsi.prom"} 1.744883093e
+09
node_textfile_mtime_seconds{file="/textfile-collector/iscsi_metrics.prom"} 1.74
4885202e+09
[04/17/25]seed@VM:~/Downloads$
```

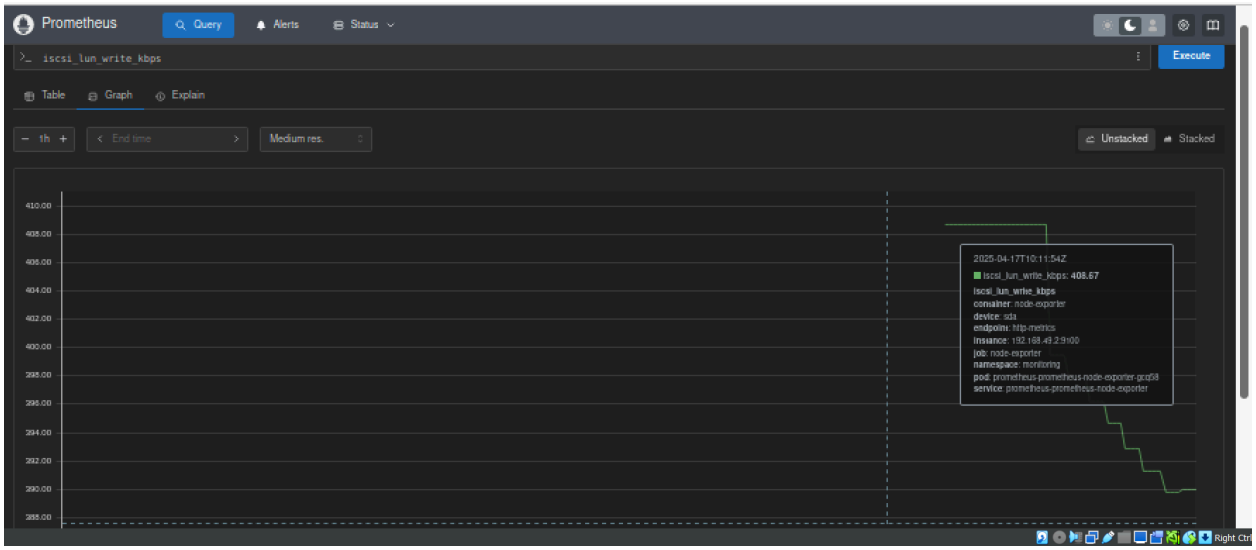
- Finally, port-forward prometheus service to use its UI:

```
kubectl port-forward svc/prometheus-kube-prometheus-n
monitoring 9090
```

```
[04/17/25]seed@VM:~/Downloads$ kubectl port-forward prometheus-prom
etheus-node-exporter-gcq58 9100:9100 -n monitoring
Forwarding from 127.0.0.1:9100 -> 9100
Forwarding from [::1]:9100 -> 9100
Handling connection for 9100
Handling connection for 9100
Handling connection for 9100
Handling connection for 9100
Handling connection for 9100
Handling connection for 9100
Handling connection for 9100
Handling connection for 9100
Handling connection for 9100
```

- In the browser, enter `iscsi_lun_read_kbps` or `iscsi_lun_write_kbps` in the expression field. Go to the Graph tab





- Now, to use the same in grafana, check that only one replica runs because if not, it gives an error as the SQLite database lock cannot be acquired by any replica. Delete any other replicas other than the most recent one. Monitor the pods until they stabilize

```
kubectl get rs -n monitoring | grep grafana
kubectl delete rs <name> -n monitoring
kubectl get pods -n monitoring | grep grafana
```

- Another patch used is to remove the environment variables GF_SERVER_ROOT_URL and GF_SERVER_SERVE_FROM_SUB_PATH and restart the pods. Now delete the replicas created with the above commands and wait till pods stabilize and use below commands to check that the variables are deleted in the new pod:

```
kubectl patch deployment prometheus-grafana -n monitoring \
  --type=json \
  -p='[
    {
      "op": "remove",
      "path": "/spec/template/spec/containers/2/env/8"
    },
    {
      "op": "remove",
      "path": "/spec/template/spec/containers/2/env/7"
    }
  ]'
kubectl rollout restart deployment prometheus-grafana -n monitoring
kubectl get rs -n monitoring | grep grafana
kubectl delete rs <name> -n monitoring
kubectl get pods -n monitoring | grep grafana
kubectl exec -n monitoring prometheus-grafana-<pod-name> -c grafana
-- printenv | grep GF_SERVER
```

- Another error could be due to browser compatibility. In SeedUbuntu20.04, which has an older version of Firefox, it is better to use

```
snap run firefox
```

as otherwise the frontend gives javascript error when checked in developer tools console on the browser.

- Use the below commands to port forward grafana and check whether it is properly connected to database:

```
kubectl port-forward pod/prometheus-grafana-<pod-name> 3000:3000 -n
monitoring


curl http://localhost:3000/api/health
```

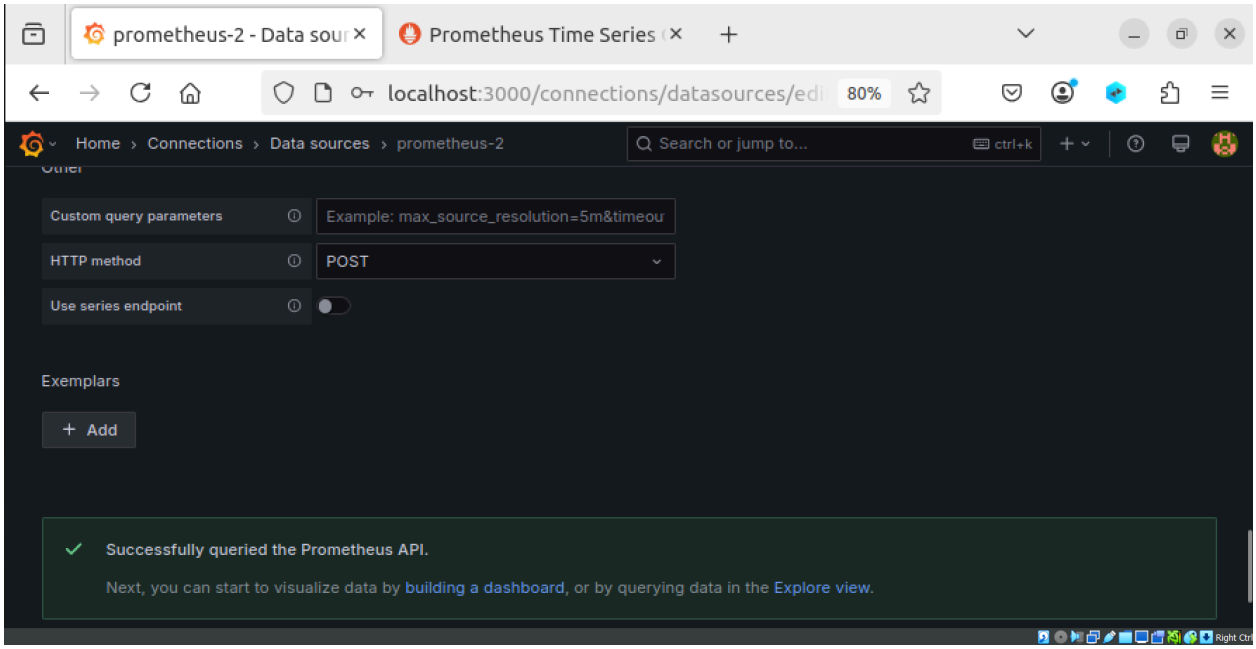
- Login to grafana (for first time) using the credentials shown in the pod in this case (admin and prom-operator):

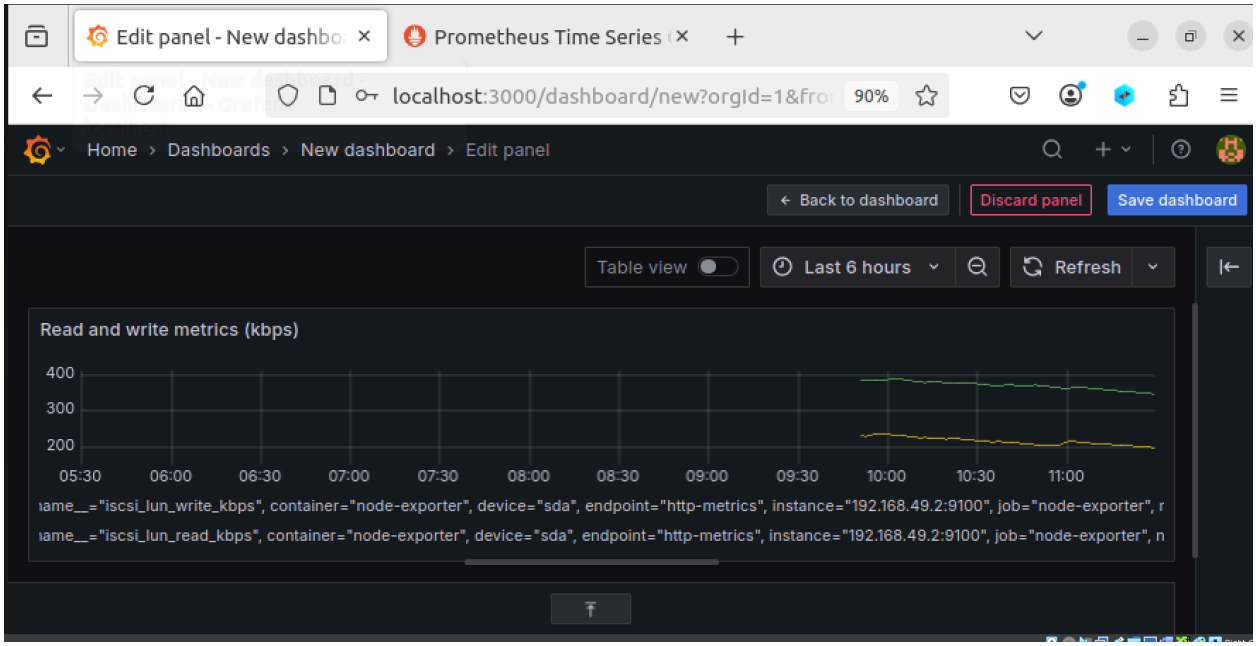
```
kubectl get secret prometheus-grafana -n monitoring -o
jsonpath="{.data.admin-user}" | base64 -d && echo

kubectl get secret prometheus-grafana -n monitoring -o
jsonpath="{.data.admin-password}" | base64 -d && echo
```

- Add prometheus as a data source in grafana (use this for the field kubernetes service URL:

```
http://prometheus-server.monitoring.svc.cluster.local:9090
```

- Click Test and Save





- Add both metrics and run queries. Save dashboard

What are the actual storage level metrics needed to be scraped by prometheus?

- The read/write IOPs from the pod that hosts the application running on Kubernetes?
- The node level metrics with a daemonset of the underlying hardware, not OS of the pod?

Another approach: Mount Kubernetes Persistent Volume on LUN and Scrape Metrics with Prometheus + Grafana

STEP-1: Setup Basic iSCSI Target on the node(VM)

- Enter the interactive `targetcli` shell to manage iSCSI configuration.
- Create a 1 GB file (`lun1.img`) and use it as a virtual disk (`disk1`).
- Define a new iSCSI target with a unique **IQN** (iSCSI Qualified Name).
- Attach the virtual disk (`disk1`) to the iSCSI target as LUN 0.
- Allow access to the target from an initiator (the client).
- Listen for iSCSI connections on all interfaces on port **3260** (standard iSCSI port).
- Set target attributes to automatically create ACLs for new initiators and enable write on LUN
- Exit targetcli
- Sign into the LUN, check with lsblk to see how it is exposed (in this case it is exposed as /dev/sdb). Format the new LUN with ext4

Code:

$ LUN.sh

```
#!/bin/bash

sudo targetcli
# Create a fileio backstore using the image file
/backstores/fileio> create disk1 /var/iscsi_disks/lun1.img 1G

# Create a new iSCSI target
/iscsi> create iqn.2025-06.com.example:uml

# Create a LUN for the target using the fileio disk
/iscsi/iqn.2025-06.com.example:uml/tpg1/luns> create /backstores/fileio/disk1

# Allow any initiator (for testing; restrict in production)
/iscsi/iqn.2025-06.com.example:uml/tpg1/acls> create iqn.1994-05.com.initiator:client

# Create a portal that listens on all interfaces (or your specific IP)
/iscsi/iqn.2025-06.com.example:uml/tpg1/portals> create 0.0.0.0 3260

cd /iscsi/iqn.2025-06.com.example:uml/tpg1
set attribute generate_node_acls=1
set attribute authentication=0
exit

sudo mkfs.ext4 /dev/sdb
```

<Screenshot>

STEP-2: Bring up k8s cluster, Apply k8s persistent volume with iSCSI plugin

- First we have to remove all previous traces of k8s.
- Change the working directory to where the deployment files reside.
- Uninstall previous helm releases
- Forcefully reset the Kubernetes cluster on the node. Remove cluster config, certificates, etc.
- Disable core dumps by setting the core pattern to a pipe
- Delete previous network configuration files used by Kubernetes' CNI (Container Network Interface).
- Delete virtual network interfaces created by Kubernetes networking plugins (e.g., Flannel, CNI).
- Stop the Kubernetes node agent (`kubelet`), and container runtimes (`containerd` or `docker`).
- Forcefully kill any lingering Kubernetes-related processes
- Clean up configuration and data directories for Kubernetes and etcd.
- Turn off swap memory, which Kubernetes recommends/needs to be disabled for stability.
- Logs out of the iSCSI target and unmounts the LUN (`/dev/sdb`) if previously mounted

<screenshot>

- Restart Docker and kubelet services to prepare for re-initialization.
- Initialize the cluster with `--pod-network-cidr=10.244.0.0/16` that is required for Flannel (it reserves this CIDR for pod networking).
- Ensure your local config directory exists and copy the Kubernetes admin config to the local user config **without overwriting** if it already exists.
- Ensure the user owns the config files and use export KUBECONFIG so that `kubectl` can use the admin config.
- Verify that the cluster is reachable and deploy the Flannel networking plugin for pod communication.
- Remove the taint that normally prevents pods from being scheduled on the control-plane node (for single-node clusters).

<screenshot>

- Create a dedicated Kubernetes namespace for resource isolation.
- Discover iSCSI targets (on localhost here), log into available LUN, mount the iSCSI LUN temporarily and wipe all contents clean.
- Create Directories for PV HostPaths, deploy custom StorageClass definitions, and deploy PVs pointing to the hostPaths or iSCSI volumes.