B.M.S. COLLEGE OF ENGINEERING
Autonomous Institute, Affiliated to VTU
Estd. 1946

DEPARTMENT OF CSE

CTY Project Work In collaboration with HPE

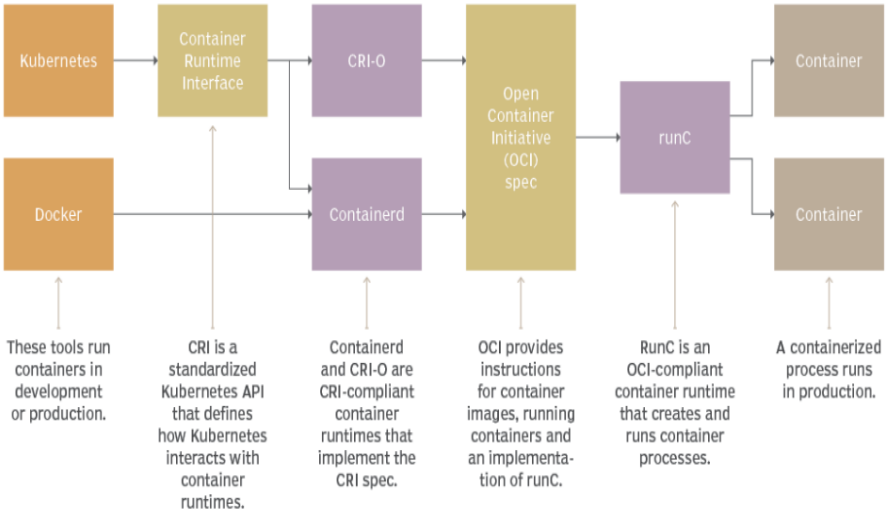
Weekly report

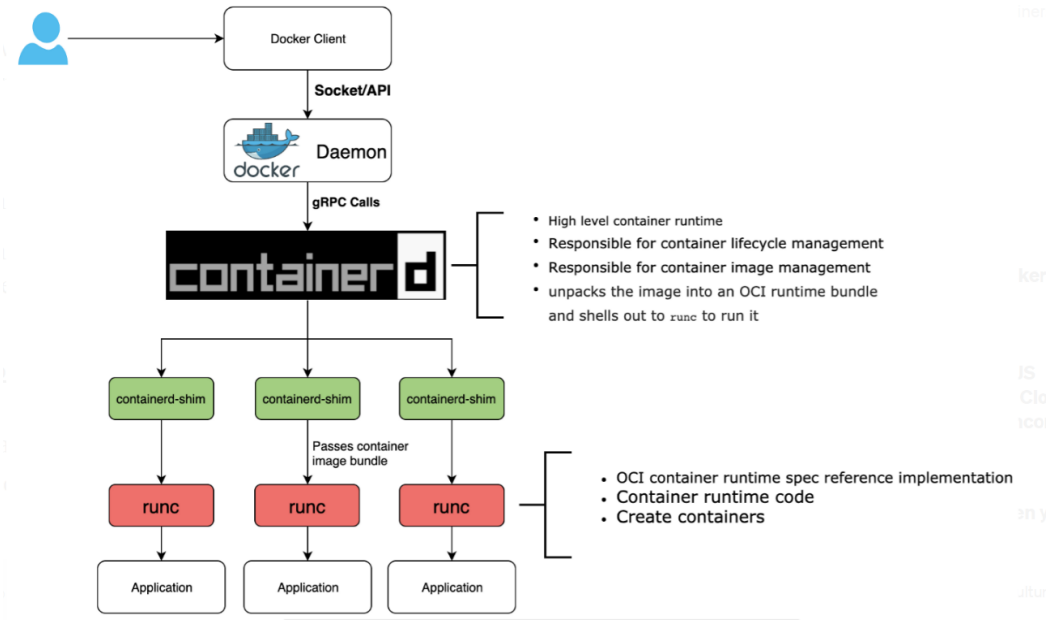
<table>
<tr><td>Project Title</td><td colspan="3">Comparative study of container runtimes</td></tr>
<tr><td rowspan="2">Student Team</td><td>Name: Kizhakel Sharat Prasad
USN: 1BM19CS074
SEM:UG-6th sem</td><td colspan="2">Name: Sharan S Pai
USN: 1BM19CS146
SEM:UG-6th sem</td></tr>
<tr><td>Name: Disha N
USN: 1BM19CS051
SEM:UG-6th sem</td><td colspan="2">Name: Vallisha M
USN: 1BM19CS177
SEM:UG-6th sem</td></tr>
<tr><td>Faculty Mentor</td><td>Dr. Pallavi G B
(Assistant Professor)
Prof. Madhavi R P
(Associate Professor)</td><td>HPE Mentors</td><td>Ravi Mehta</td></tr>
<tr><td>Review for the Period</td><td>30-06-2022</td><td colspan="2">22-07-2022</td></tr>
<tr><td>Github Link</td><td colspan="3">Application tested: https://github.com/sharan-s-pai/dockerized-goals-app
Reports: https://github.com/Sharat-Kizhakel/HPE-CTY</td></tr>
<tr><td>Work done so far</td><td colspan="3">1. Successfully installed five runtimes discussed under problem statement
2. Plotted their cpu usages,memory as well as network usage using prometheus
3. exported data sources from prometheus and plotted relevant dashboards in grafana</td></tr>
</table>

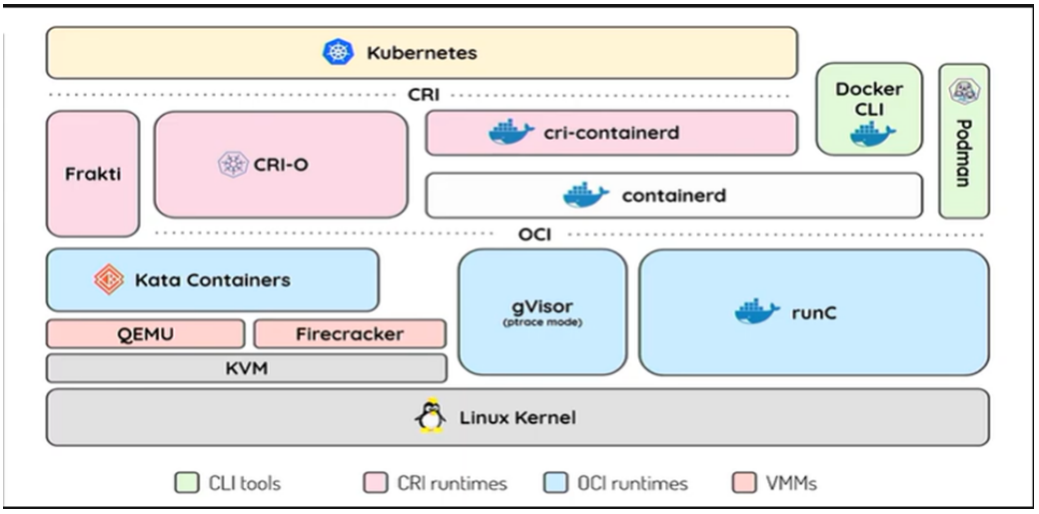
Review	# Introduction **What is a container runtime?** The container runtime is the low-level component that creates and runs containers. Docker currently uses runC, the most popular runtime, which adheres to the OCI standard that defines container image formats and execution. However, because Docker observes OCI-compliance, any OCI-compliant runtime should work. It is software that runs and manages the components required to run containers. They can be broadly classified into two main categories: 1. Open Container Initiative (OCI): The Open Container Initiative (**OCI**) is a Linux Foundation project to design open standards for containers.Established in June 2015 by Docker and other leaders in the container industry. OCI currently contains two specifications: ● the Runtime Specification (runtime-spec) ● Image Specification (image-spec). OCI runtime spec defines how to run the OCI image bundle as a container. OCI image spec defines how to create an OCI Image, which includes an image manifest, a filesystem (layer) serialization, and an image configuration. 2. Container Runtime Interface (CRI): Container runtime interface (CRI) is a plugin interface that lets the kubelet—an agent that runs on every node in a Kubernetes cluster—use more than one type of container runtime. The CRI has additional concerns over an OCI Runtime including image management and distribution, storage, snapshotting, networking (distinct from the CNI), and more. ● Open Container Initiative (OCI) Runtimes ○ Native Runtimes

        - runC
        - Railcar
        - Crun
        - rkt
    - Sandboxed and Virtualized Runtimes
        - gvisor
        - nabla-containers
        - runV
        - clear containers
        - kata-containers
- Container Runtime Interface (CRI)
    - containerd
    - cri-o

How Docker, Kubernetes, OCI, CRI-O, containerd and runC collaborate to run containers




Docker Client
Socket/API
Daemon
gRPC Calls
containerd
High level container runtime
Responsible for container lifecycle management
Responsible for container image management
unpacks the image into an OCI runtime bundle
and shells out to runc to run it
containerd-shim
containerd-shim
containerd-shim
Passes container image bundle
runc
runc
runc
OCI container runtime spec reference implementation
Container runtime code
Create containers
Application
Application
Application


Kubernetes
CRI
Docker CLI
Podman
Frakti
CRI-O
cri-containerd
containerd
OCI
Kata Containers
QEMU
Firecracker
KVM
gVisor
(ptrace mode)
runC
Linux Kernel
CLI tools
CRI runtimes
OCI runtimes
VMMs

Runtimes which we have taken into consideration?

1. **runc:**

   Open Container Initiative runtime runc is a command line client for running applications packaged according to the Open Container Initiative (OCI) format and is a compliant implementation of the Open Container Initiative specification Runc is a so-called “container runtime”.

2. **crun:**

   It is a lightweight fully featured OCI runtime and C library for running containers. is faster than runc and has a much lower memory footprint.

3. **runsc:**

   gVisor is an application kernel, written in Go, that implements a substantial portion of the Linux system surface. It includes an Open Container Initiative (OCI) runtime called runsc that provides an isolation boundary between the application and the host kernel. The runsc runtime integrates with Docker and Kubernetes, making it simple to run sandboxed containers.

4. **sysbox-runc:**

   Sysbox is an open-source and free container runtime (a specialized "runc"), developed by Nestybox, that enhances containers in two key ways:

   - Improves container isolation
   - Enables containers to run same workloads as VMs

5. **kata:**

   kata-runtime, referred to as "the runtime", is the Command-Line Interface (CLI) part of the Kata Containers runtime component. It leverages the virtcontainers package to provide a high-performance standards-compliant runtime that creates hardware-virtualized Linux containers running on Linux hosts.The runtime is **OCI**-compatible, **CRI-O**-compatible, and **Containerd**-compatible, allowing it to work seamlessly with both Docker and Kubernetes respectively.

Application Overview

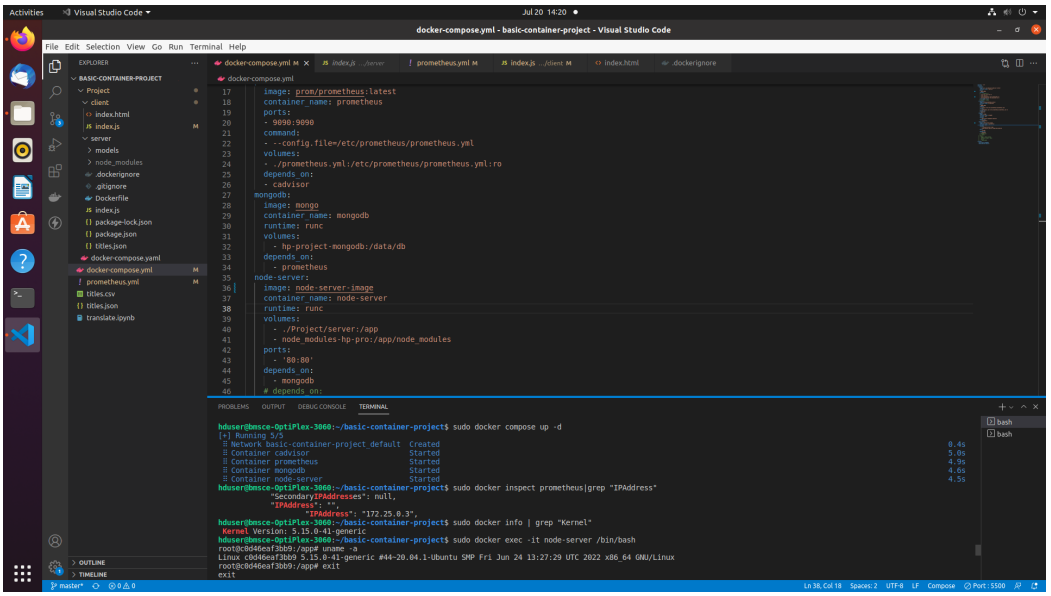
In the course of the Project the same application has been tested with different runtimes. The application in question is a simple react app which comprises of 3 main containers which we monitored namely:

1. node-server
2. docker
3. mongodb

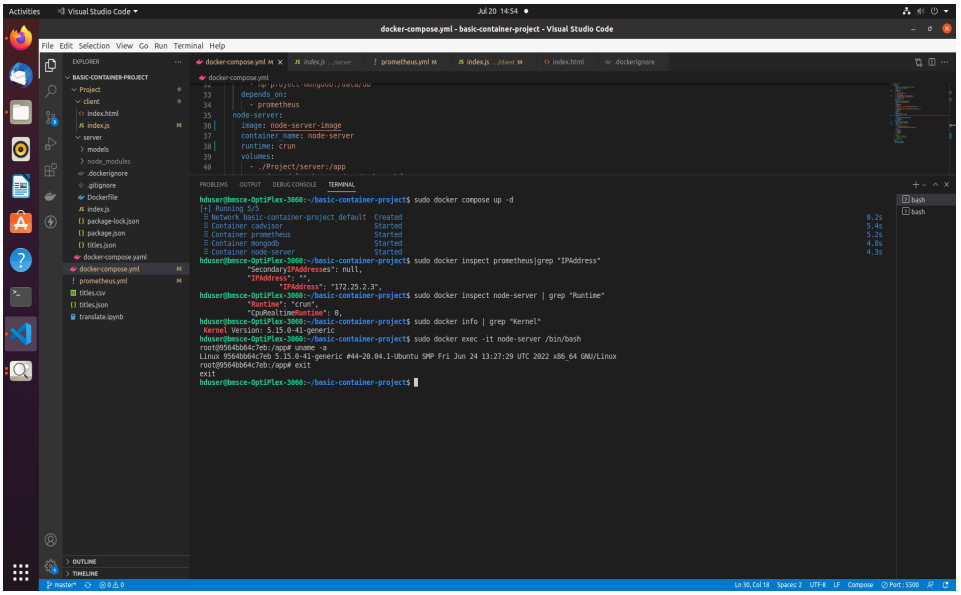
The app contains a docker-compose file with all relevant dependencies including a cadvisor and prometheus port numbers. In addition to this bind mounts have been used to persist data. The performance of a container cannot be tested without sending a significant payload to the server. For this purpose a movies dataset has been used. The runtime was changed everytime we editing the installed runtime in the available runtimes in the daemon.json file which comes installed by default with docker. The sure test for whether we are using the new runtime can be confirmed by checking the kernel version using the uname option in docker.

The following are the screenshots proving successful installation of runtimes:

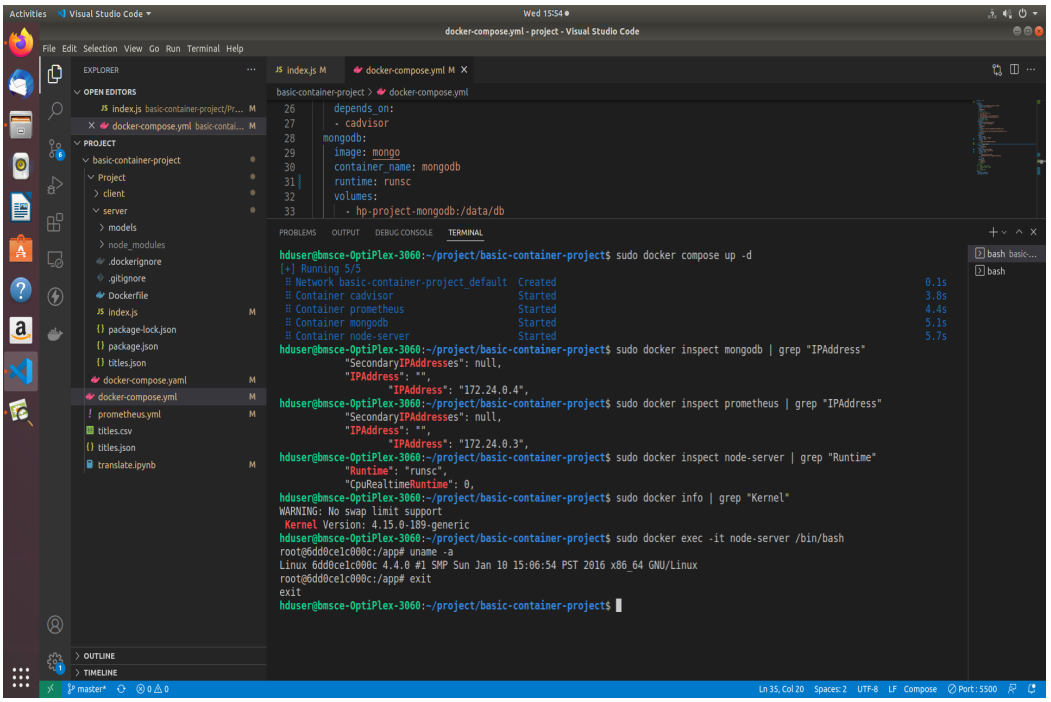
1. runc



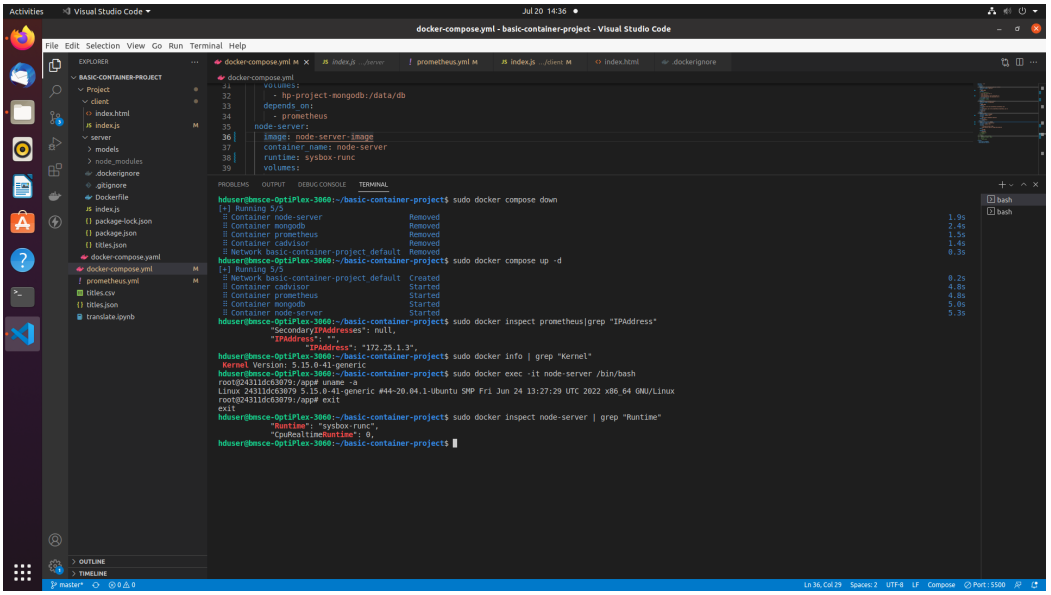
2. crun



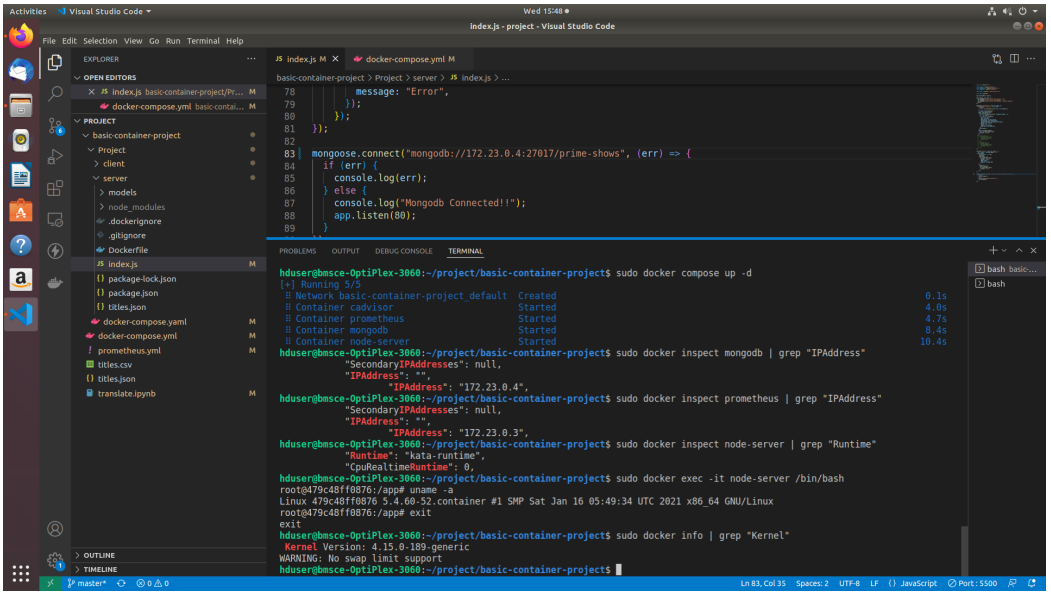
3. runsc



4. sysbox-runc



5. kata



Essential Metrics for Container Monitoring

When considering metrics it will be helpful to pick a concrete metrics platform in order to show examples of collection and alerting. In the world of the Cloud Native technologies, Prometheus is the obvious choice for metrics collection, alerting, and in conjunction with Grafana visualization. The four golden signals of monitoring are latency, traffic, errors, and saturation. If one can measure any four metrics of your user-facing system, emphasis should be laid on the four given below.

Latency — The time it takes to service a request

	Traffic — A measure of how much demand is being placed on your system **Errors —** The rate of requests that fail **Saturation —** How "full" your service is. However another method of abstraction regarding docker container metrics is the USE method which we have adopted for our project. "USE" stands for Utilization, Saturation and Errors when looking at the **resources** in a given system. They have the following definitions: ● **Resource**: all physical server functional components (CPUs, disks, busses, …) ● **Utilization**: the average time that the resource was busy servicing work ● **Saturation**: the degree to which the resource has extra work which it can't service, often queued ● **Errors**: the count of error events The most important resources containers provide are CPU, Memory, Network and Disk. We have queried the relevant metrics in PROMQL. The four metrics given by prometheus are the important ones. Cgroup is a linux feature to limit, police, and account the resource usage for a set of processes. It provides mechanism to limit and monitor system resources like CPU time, system memory, disk bandwidth, network bandwidth, etc. The cgroups works by dividing resources into groups and then assigning tasks to those groups. Docker uses cgroups to limit the system resources.

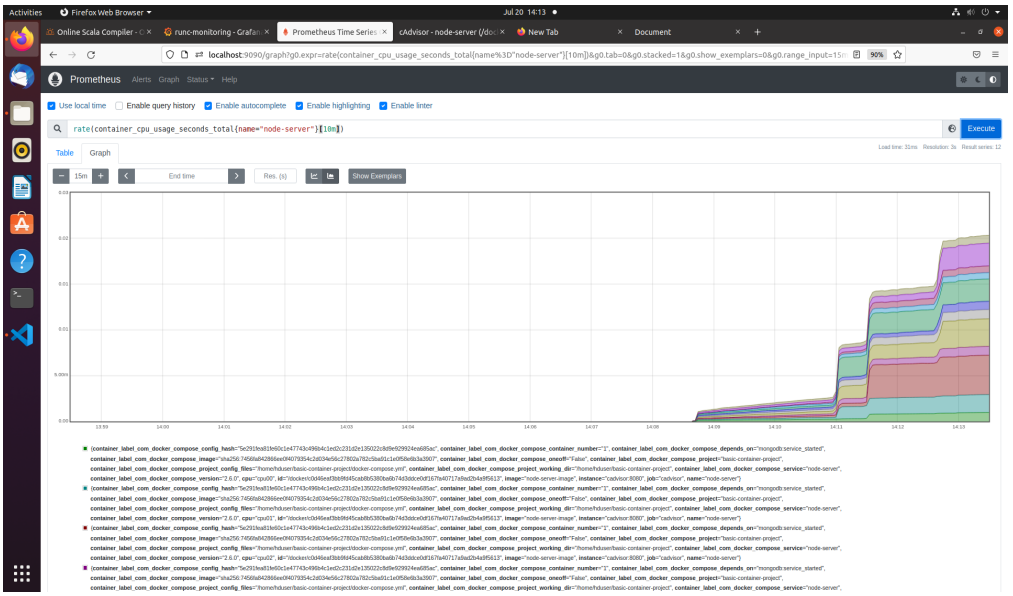
Expression	Description	For
`rate(container_cpu_usage_seconds_total{name="container_name"}[1m])`	The cgroup's CPU usage in the last minute	The given container
`container_memory_usage_bytes{name="container_name"}`	The cgroup's total memory usage (in bytes)	The given container
`rate(container_network_transmit_bytes_total[1m])`	Bytes transmitted over the network by the container per second in the last minute	All containers
`rate(container_network_receive_bytes_total[1m])`	Bytes received over the network by the container per second in the last minute	All containers

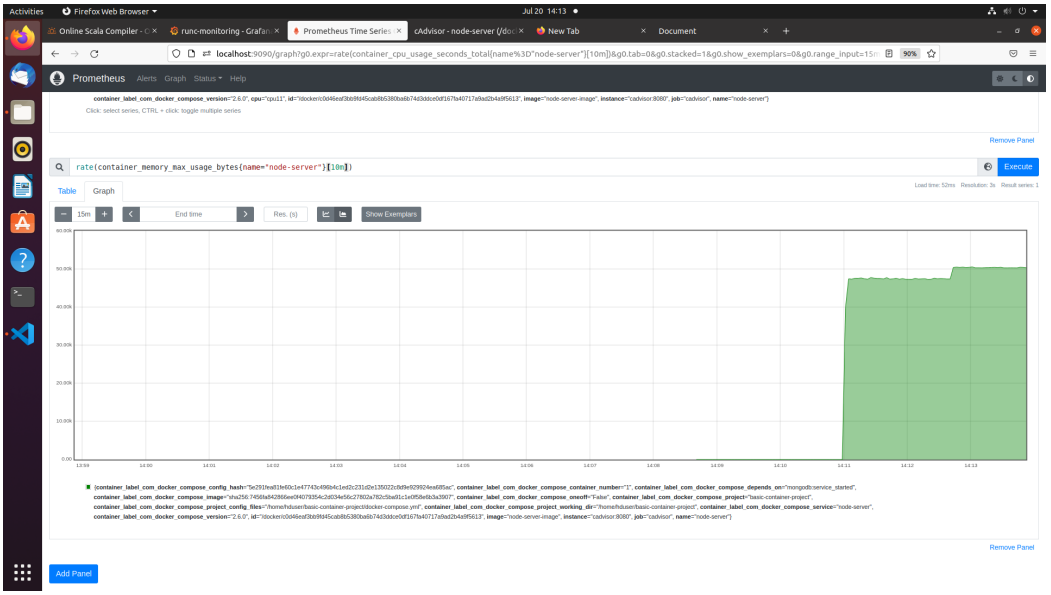
Prometheus scrapes for one request

In every case two containers were taken into consideration namely node-server and mongodb

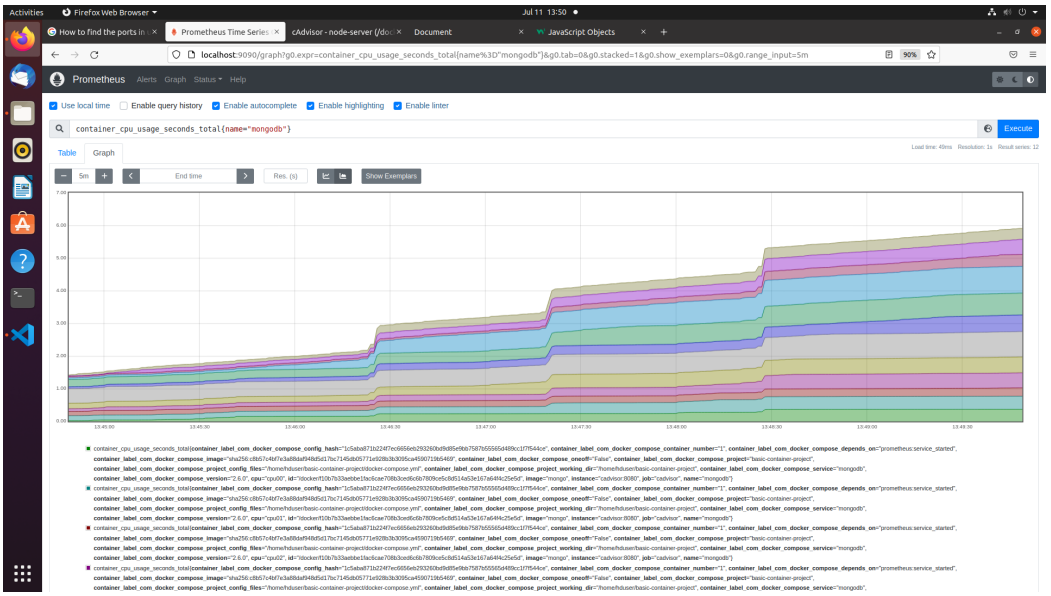
1. **runc:**

    - node-server


Activities
Firefox Web Browser
Jul 20 14:13
Prometheus
Alerts
Graph
Status
Help
Use local time
Enable query history
Enable autocomplete
Enable highlighting
Enable linter
rate(container_cpu_usage_seconds_total{name="node-server"}[10m])
Execute
Table
Graph
15m
End time
Res. (s)
Show Exemplars

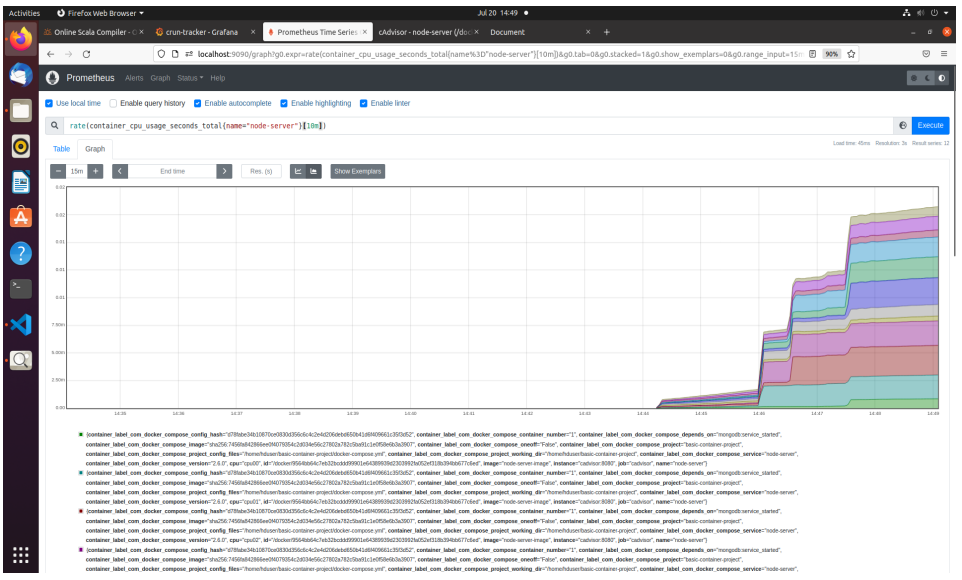

Activities
Firefox Web Browser
Jul 20 14:13
Prometheus
Alerts
Graph
Status
Help
rate(container_memory_max_usage_bytes{name="node-server"}[10m])
Execute
Table
Graph
15m
End time
Res. (s)
Show Exemplars
Remove Panel
Add Panel

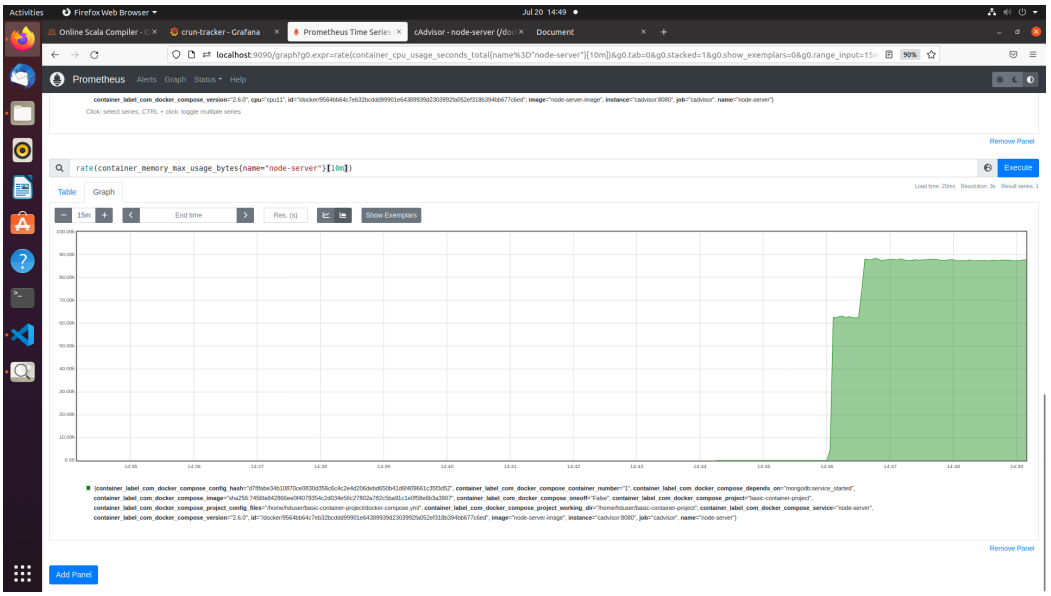
- mongodb



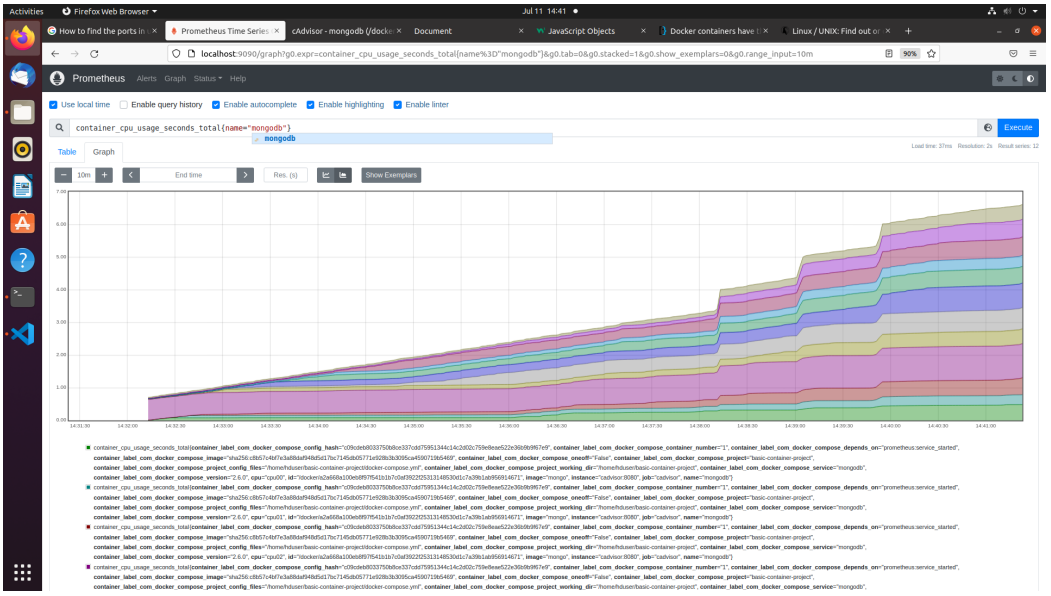
2. crun:

- node-server



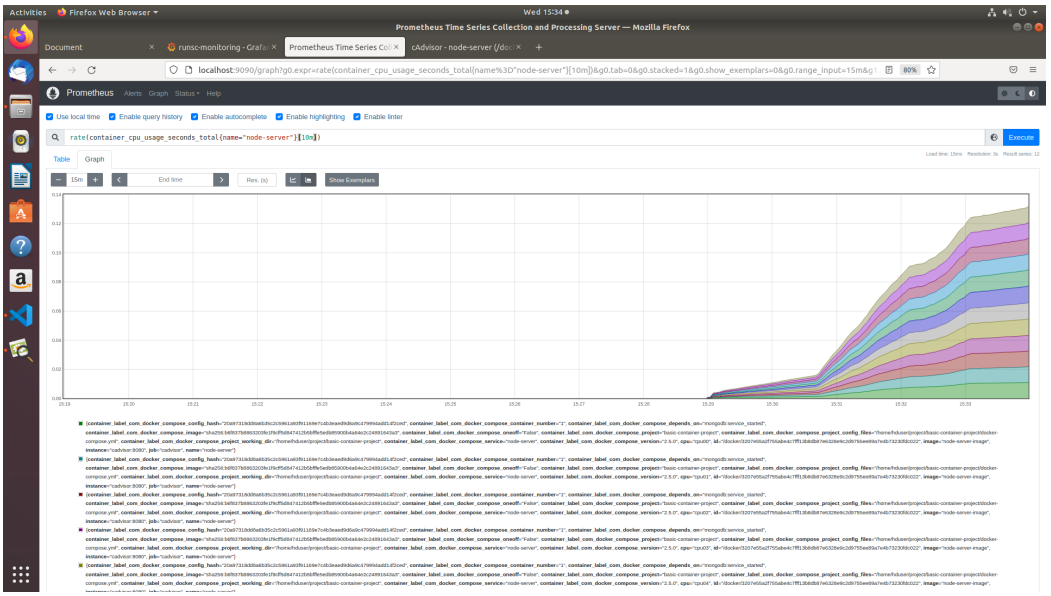

Activities
Firefox Web Browser
Jul 20 14:49
Prometheus Time Series
cAdvisor - node-server (/do
Document
localhost:9090/graph?g0.expr=rate(container_cpu_usage_seconds_total{name%3D"node-server"}[10m])&g0.tab=0&g0.stacked=1&g0.show_exemplars=0&g0.range_input=15
Prometheus
Alerts
Graph
Status
Help
Remove Panel
rate(container_memory_max_usage_bytes{name="node-server"}[10m])
Execute
Table
Graph
15m
End time
Res. (s)
Show Exemplars
Remove Panel
Add Panel

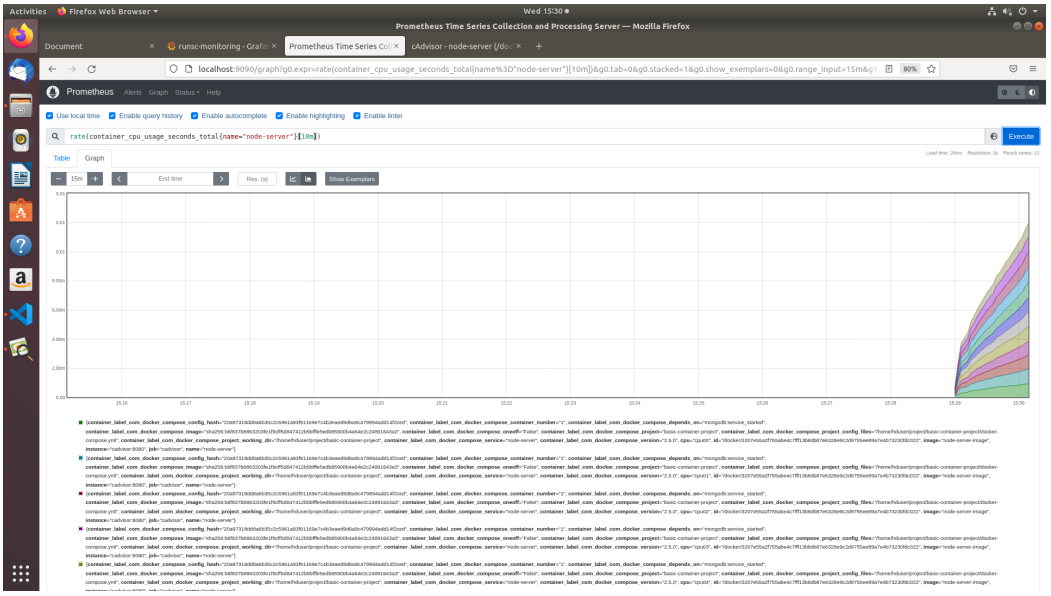
- mongodb


Activities
Firefox Web Browser
Jul 11 14:41
How to find the ports in
Prometheus Time Series
cAdvisor - mongodb (/docke
Document
JavaScript Objects
Docker containers have l
Linux / UNIX: Find out or
localhost:9090/graph?g0.expr=container_cpu_usage_seconds_total{name%3D"mongodb"}&g0.tab=0&g0.stacked=1&g0.show_exemplars=0&g0.range_input=10m
Prometheus
Alerts
Graph
Status
Help
Use local time
Enable query history
Enable autocomplete
Enable highlighting
Enable linter
container_cpu_usage_seconds_total{name="mongodb"}
mongodb
Execute
Table
Graph
10m
End time
Res. (s)
Show Exemplars

3. runsc

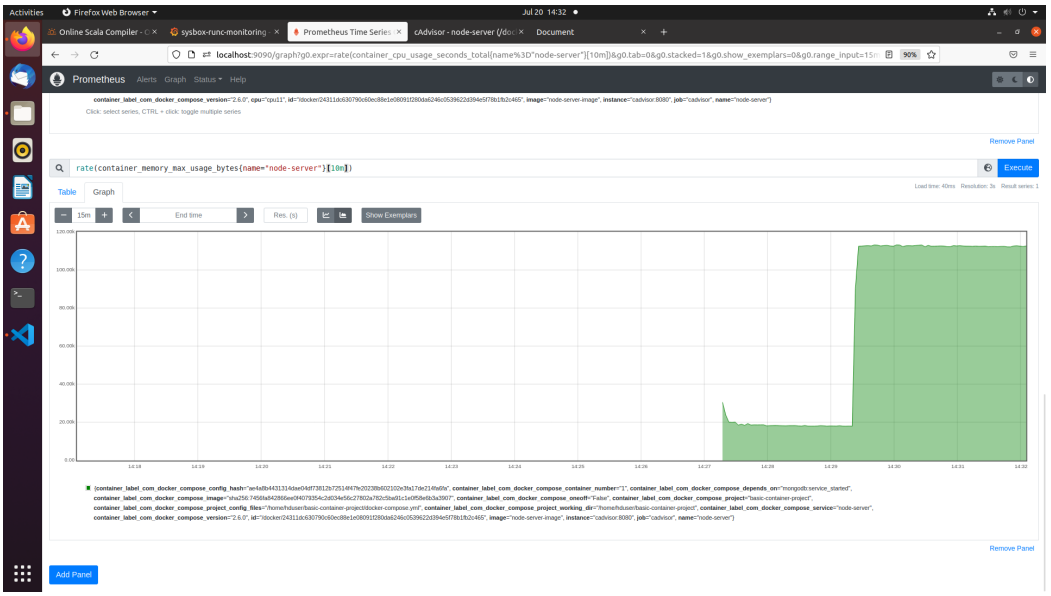
- node-server

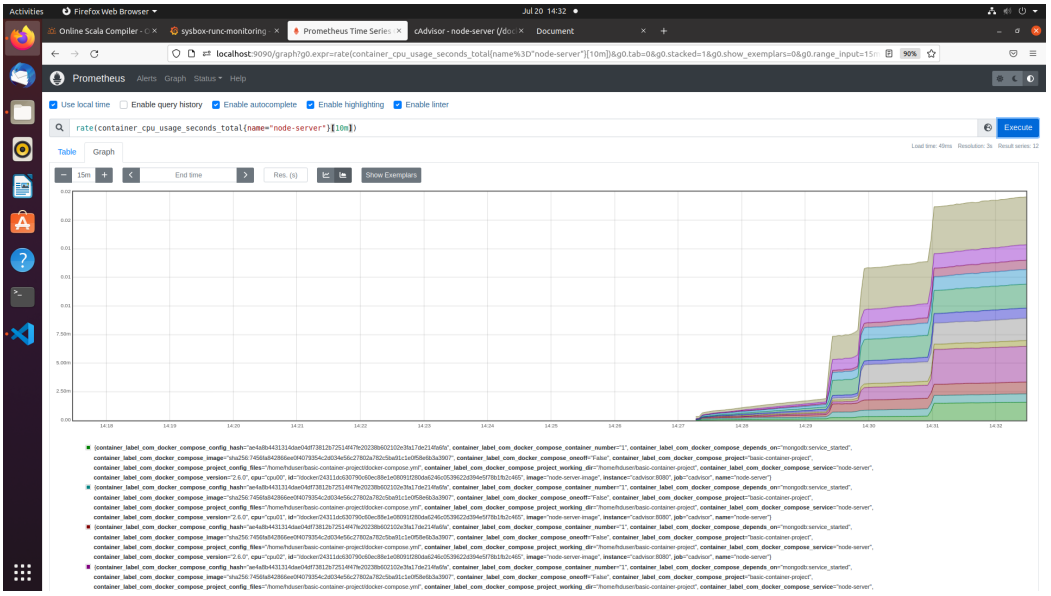




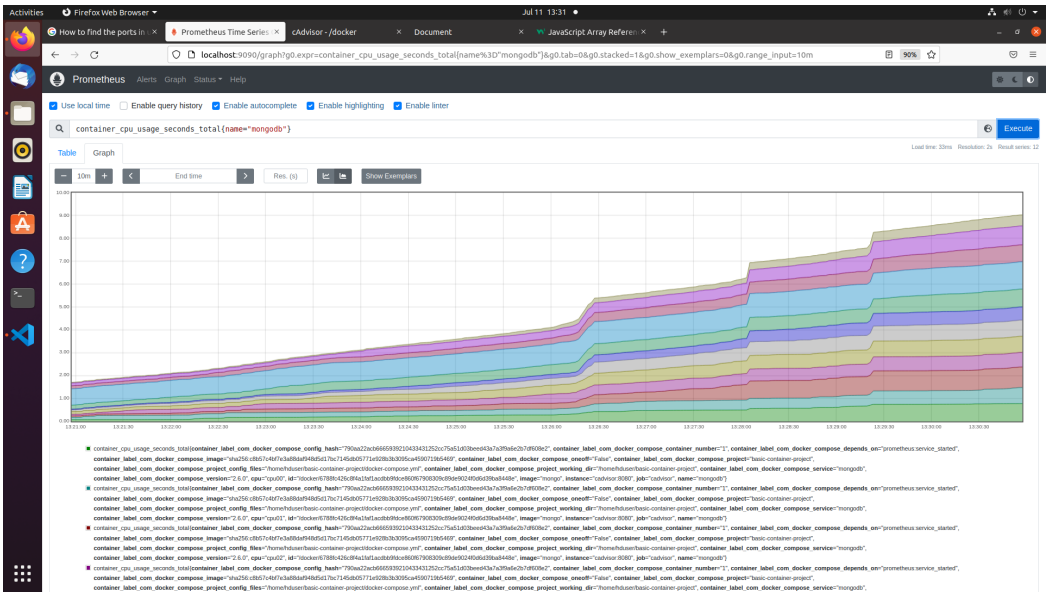
4. sysbox-runc:

- node-server



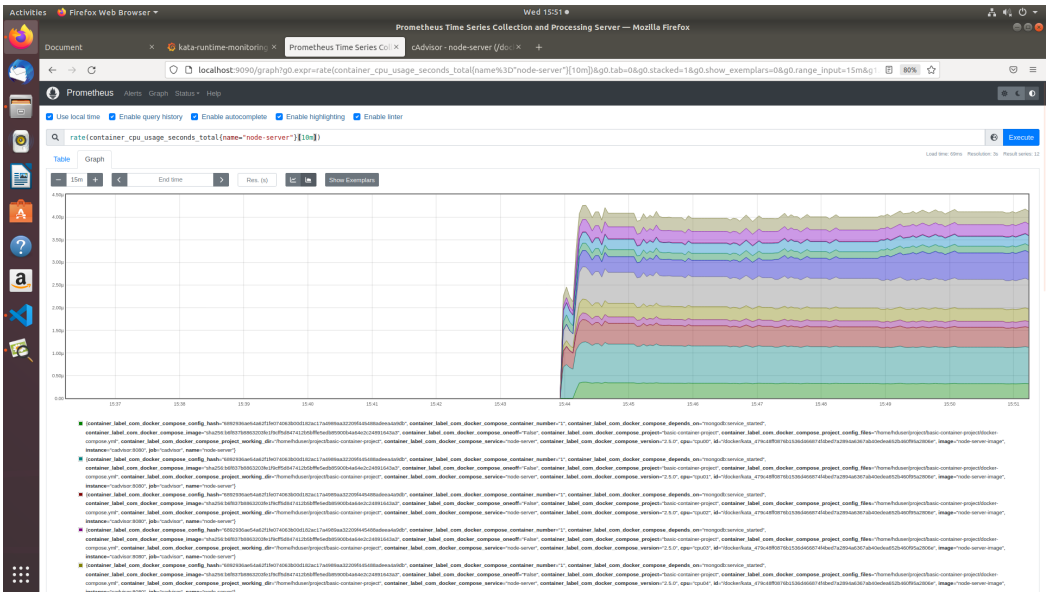


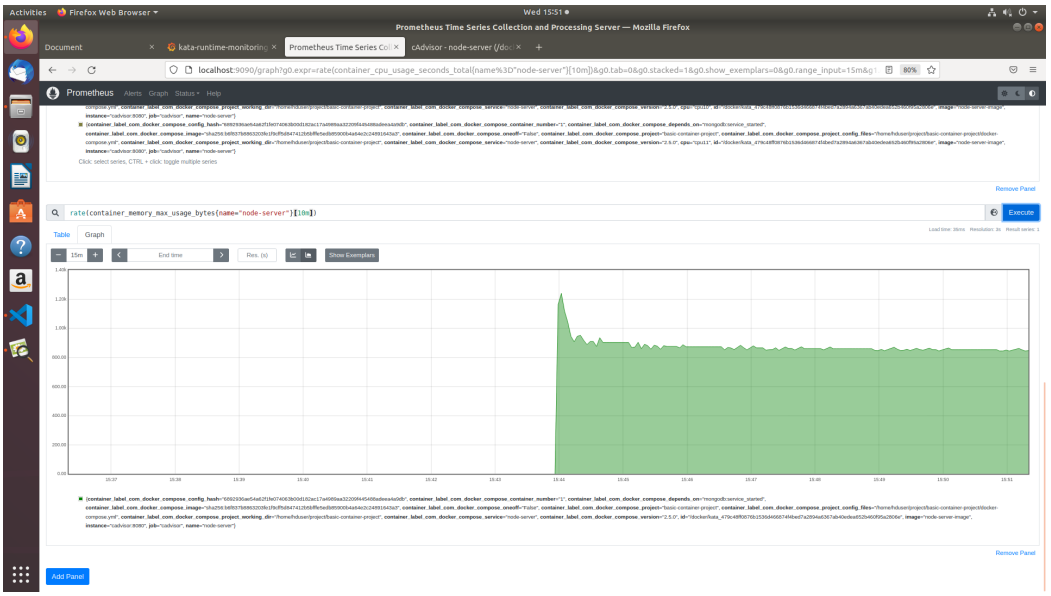
- mongodb



5. kata:

- node-server

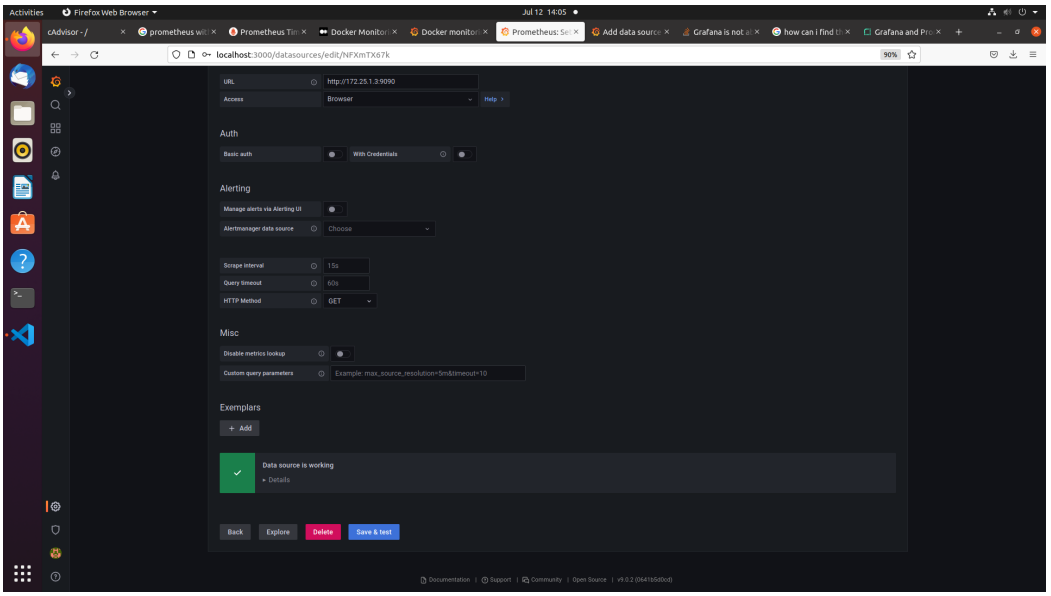




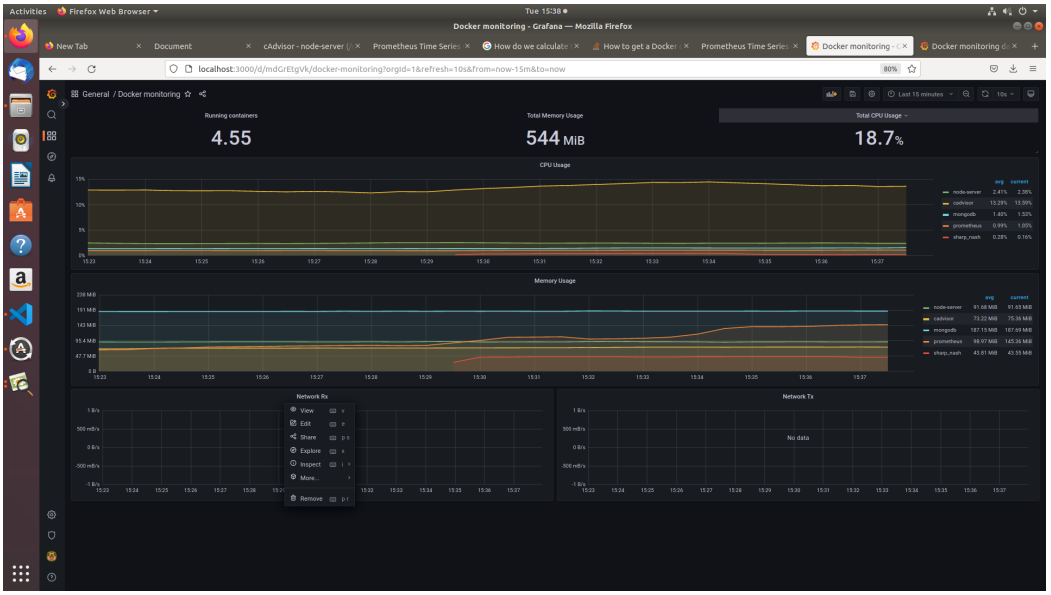
Grafana dashboard for metrics

To get a better visualization with cpu usage breakdown and network utilization and to ensure easier comparison we used the standard grafana dashboard for docker monitoring https://grafana.com/grafana/dashboards/193 . Given below are the screenshots of the dashboards with each runtime.

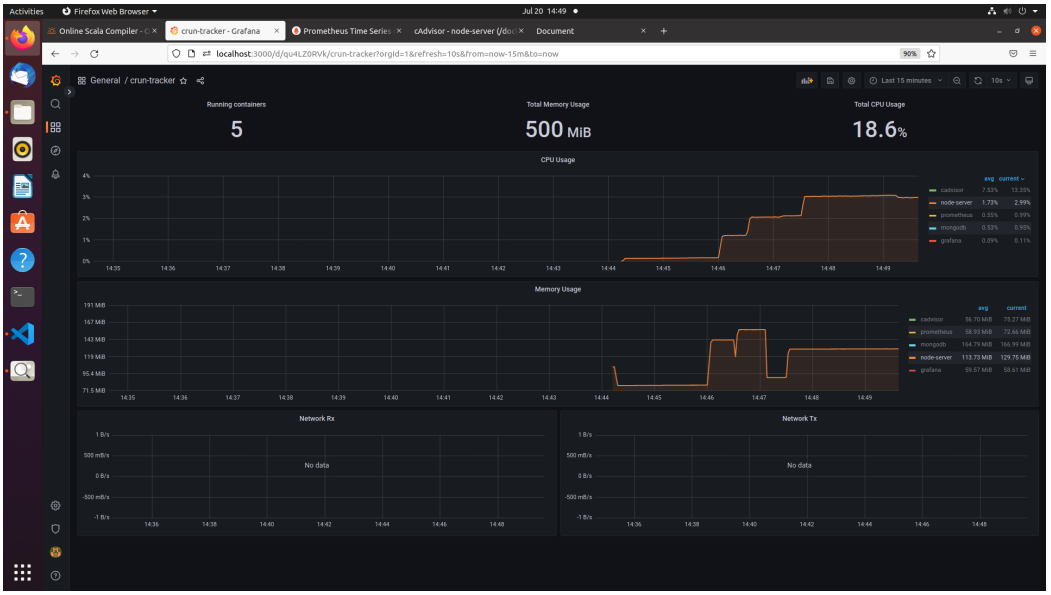
Adding a data source:



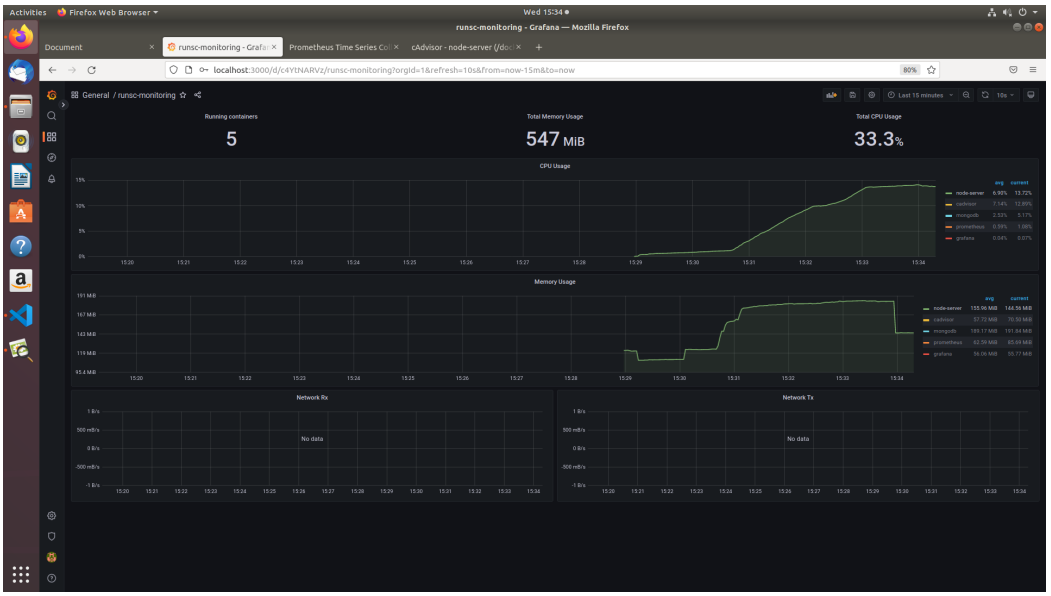
1. **runc:**



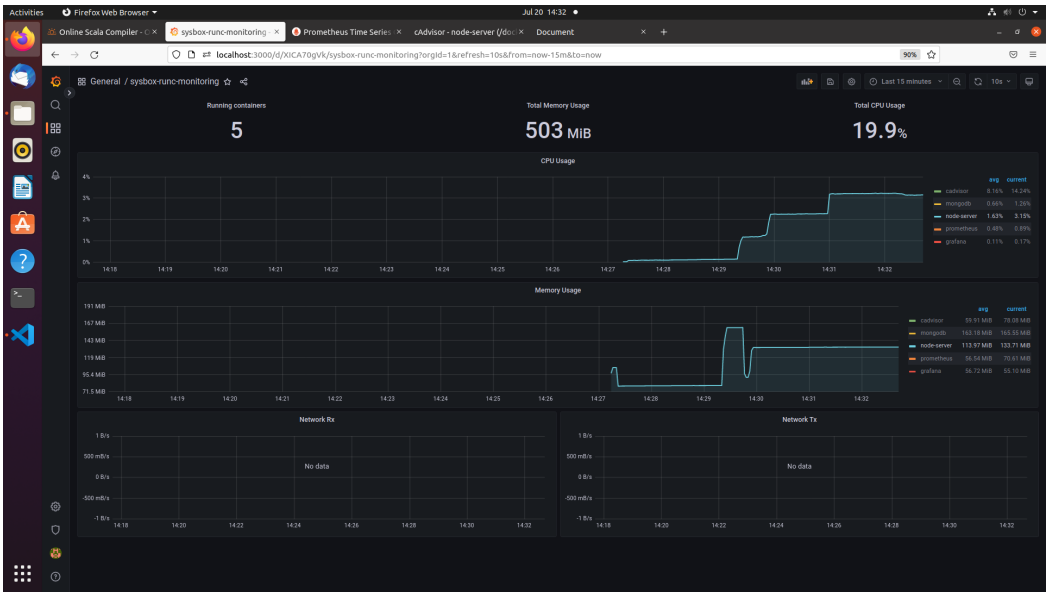
2. **crun:**



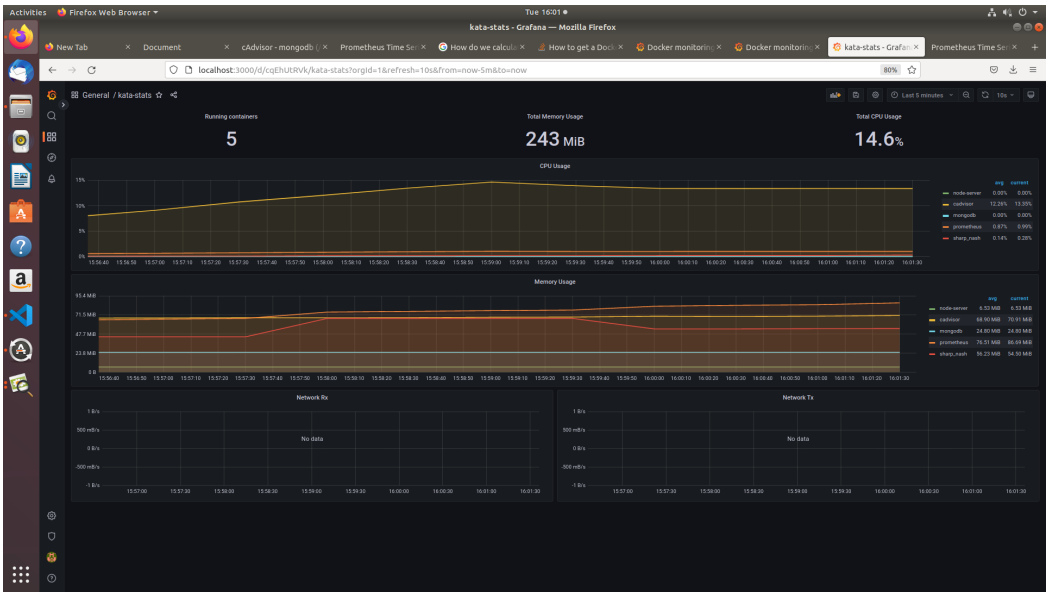
3. **runsc:**



4. sysbox-runc:



5. **kata:**



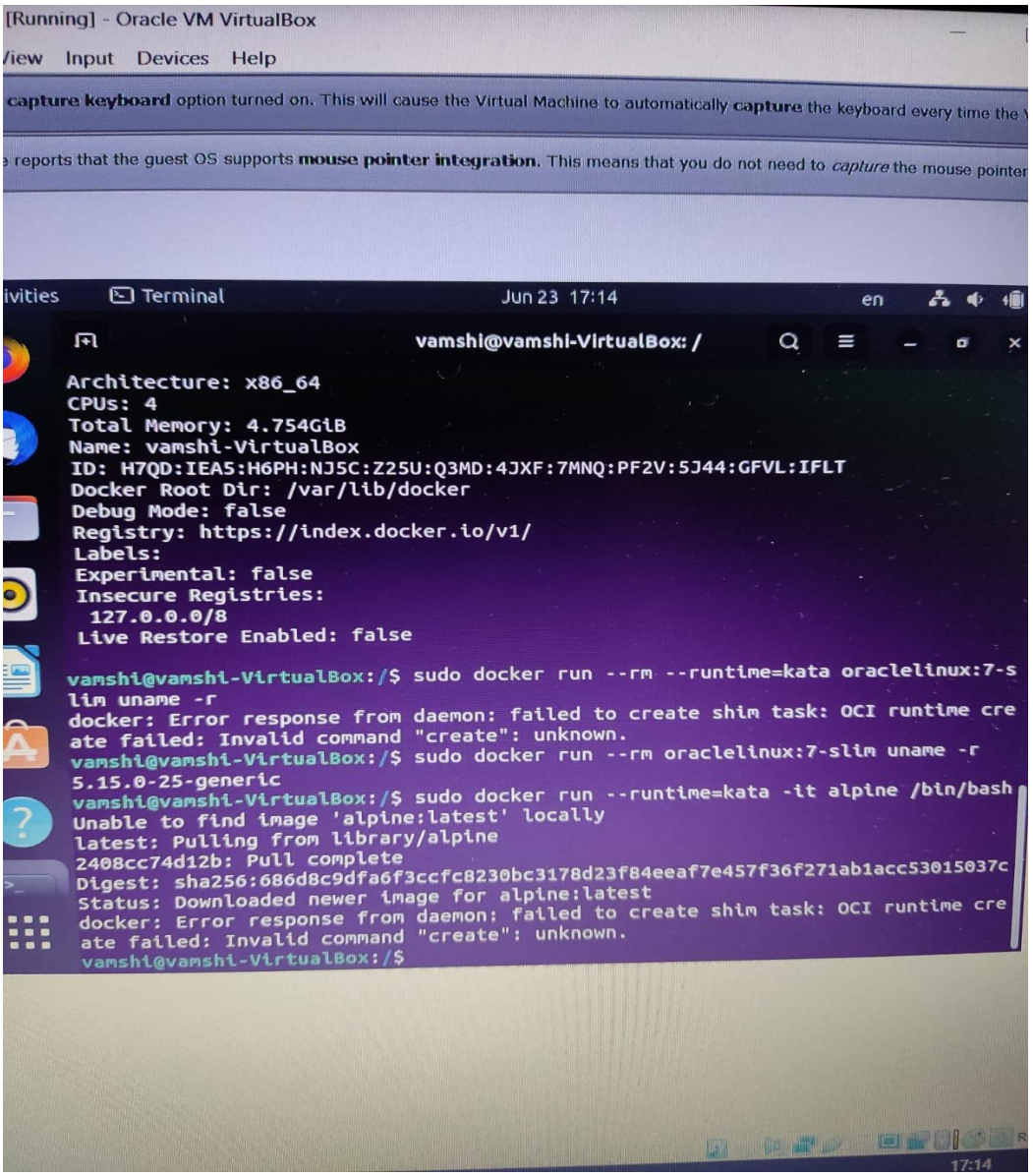
Interpretation of metrics

The metric used here is “cpu usage”. This is a counter metric that counts the number of seconds the CPU has been running in a particular mode. The CPU has several modes such as iowait, idle, user, and system. Because the objective is to count usage, we use a query that excludes idle time:

```
sum by (cpu)(node_cpu_seconds_total{mode!="idle"})
```

The sum function is used to combine all CPU modes. The result shows how many seconds the CPU has run from the start. To tell if the CPU has been busy or idle recently, use the rate function to calculate the growth rate of the counter:

	(sum by (cpu)(rate(node_cpu_seconds_total{mode!="idle"}[5m]))*100 The above query produces the rate of increase over the last five minutes, which lets you see how much computing power the CPU is using By default, a container has no resource constraints and can use as much of a given resource as the host's kernel scheduler allows. memory usage by subtracting cache usage from the total memory usage. The API does not perform such a calculation but rather provides the total memory usage and the amount from the cache so that clients can use the data as needed Out of all the five runtimes kata runtime gives the best cpu and memory utilization without placing any limits on it. Other than kata the default runc runtime gives a good performance along with crun. When looking only at memory utilization for 5 containers over the same time interval of 15 minutes these are the results and Kata could be a suitable open source alternative to runc. However Kata 2.x support for Docker will be removed due to incompatibility of dockershim. as shown below.


[Running] - Oracle VM VirtualBox
View Input Devices Help
capture keyboard option turned on. This will cause the Virtual Machine to automatically capture the keyboard every time the
reports that the guest OS supports mouse pointer integration. This means that you do not need to capture the mouse pointer
Activities Terminal Jun 23 17:14 en
vamshi@vamshi-VirtualBox: /
Architecture: x86_64
CPUs: 4
Total Memory: 4.754GiB
Name: vamshi-VirtualBox
ID: H7QD:IEA5:H6PH:NJ5C:Z25U:Q3MD:4JXF:7MNQ:PF2V:5J44:GFVL:IFLT
Docker Root Dir: /var/lib/docker
Debug Mode: false
Registry: https://index.docker.io/v1/
Labels:
Experimental: false
Insecure Registries:
127.0.0.0/8
Live Restore Enabled: false
vamshi@vamshi-VirtualBox:/$ sudo docker run --rm --runtime=kata oraclelinux:7-slim uname -r
docker: Error response from daemon: failed to create shim task: OCI runtime create failed: Invalid command "create": unknown.
vamshi@vamshi-VirtualBox:/$ sudo docker run --rm oraclelinux:7-slim uname -r
5.15.0-25-generic
vamshi@vamshi-VirtualBox:/$ sudo docker run --runtime=kata -it alpine /bin/bash
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
2408cc74d12b: Pull complete
Digest: sha256:686d8c9dfa6f3ccfc8230bc3178d23f84eeaf7e457f36f271ab1acc53015037c
Status: Downloaded newer image for alpine:latest
docker: Error response from daemon: failed to create shim task: OCI runtime create failed: Invalid command "create": unknown.
vamshi@vamshi-VirtualBox:/$
17:14

Learning Outcomes	• Learnt about containerization pertaining to docker in depth • Explored open-source tools like prometheus and grafana for identifying and analyzing relevant metrics for monitoring containers.
Future Work	• Comparing additional OCI runtimes with Kubernetes as most open source runtime's compatibility with docker is depreciating by the end of 2022. • Investigate other metrics made available by Prometheus.
References	1. https://www.capitalone.com/tech/cloud/container-runtime/ 2. https://devopstales.github.io/home/container-runtimes/ 3. https://github.com/nestybox/sysbox 4. https://github.com/kata-containers/runtime 5. https://github.com/google/gvisor 6. https://blog.freshtracks.io/a-deep-dive-into-kubernetes-metrics-part-3-container-resource-metrics-361c5ee46e66 7. https://opensource.com/article/19/11/introduction-monitoring-prometheus 8. https://tsh.io/blog/grafana-custom-dashboard/ 9. https://medium.com/aeturnuminc/configure-prometheus-and-grafana-in-dockers-ff2a2b51aa1d 10. https://docs.docker.com/config/daemon/prometheus/ 11. https://jpinjpblog.wordpress.com/2020/04/01/monitoring-and-collecting-docker-container-statistics/ 12. https://mahesh-mahadevan.medium.com/monitoring-docker-containers-on-windows-using-prometheus-grafana-c32bbb7ed04