

Report Assignment 1

Disclaimer

I have reused the code and parts of the Report from last Year 2023W to complete the Assignment. Some changes have been made(manifest files + Report).

Implementation

The collector service was implemented the following REST API frame, persons and known-persons. frame will forward a frame from a camera to the image-analysis-service and face-recognition-service. When sending the frame to the image analysis service the face-recognition-service the destination field in the json message contains a destination address pointing to <http://collector-service:80/known-persons> and when sending it to the Image-Analysis-service it provides the destination <http://collector-service:80/persons>. /known-persons relays the message from Face-Recognition-service to the alert-service <http://alert-service:80/alerts>. /persons will relay the message from Image-Analysis-service further to the Section-service <http://section-service:80/persons>. The collector service runs on the following host 0.0.0.0:80 and was developed in python with Flask. All calls are processed synchronous, but asynchronous would probably have increased performance a lot. The project was build on top of the provided docker/python example <https://gitta-lab.cs.univie.ac.at/public-examples/cloud-computing-extra-materials>. Development of the collector service was done on the local machine and used Docker to deploy the services for testing. The REST API was tested with Postman, by telling the camera via http request to send images to the collector-service. During development there were no major issues to talk about. All services were reached as can be seen in the following Image.

```
al-section-service-1 | 172.27.0.6 - - [10/Nov/2023 18:02:25] "POST /persons HTTP/1.1" 200 -
al-collector-service-1 | 172.27.0.2 - - [10/Nov/2023 18:02:25] "POST /persons HTTP/1.1" 200 -
al-image-analysis-service-1 | 172.27.0.6 - - [10/Nov/2023 18:02:25] "POST /frame HTTP/1.1" 200 -
al-face-recognition-service-1 | 172.27.0.6 - - [10/Nov/2023 18:02:25] "POST /frame HTTP/1.1" 204 -
al-collector-service-1 | 172.27.0.7 - - [10/Nov/2023 18:02:25] "POST /frame HTTP/1.1" 200 -
al-camera-service-1 | 172.27.0.1 - - [10/Nov/2023 18:02:25] "POST /stream?toogle=on HTTP/1.1" 200 -
```

```

1  [
2    "code": 200,
3    "type": "STATUS",
4    "message": "Streamed 1 frame(s) in 0.3090975284576416 seconds, with 0 failed requests."
5  ]

```

All Response Codes return 200, except face recognition is 204, which means the face was not recognized. If multiple frames are being processed then the Face-Recognition-service will eventually return a 200 code and the alert service will also be reached.

```
al-section-service-1 | 172.27.0.6 - - [10/Nov/2023 18:09:25] "POST /persons HTTP/1.1" 200 -
al-collector-service-1 | 172.27.0.2 - - [10/Nov/2023 18:09:25] "POST /persons HTTP/1.1" 200 -
al-image-analysis-service-1 | 172.27.0.6 - - [10/Nov/2023 18:09:25] "POST /frame HTTP/1.1" 200 -
al-alert-service-1 | 172.27.0.6 - - [10/Nov/2023 18:09:26] "POST /alerts HTTP/1.1" 200 -
al-collector-service-1 | 172.27.0.3 - - [10/Nov/2023 18:09:26] "POST /known-persons HTTP/1.1" 200 -
al-face-recognition-service-1 | 172.27.0.6 - - [10/Nov/2023 18:09:26] "POST /frame HTTP/1.1" 200 -
al-collector-service-1 | 172.27.0.7 - - [10/Nov/2023 18:09:26] "POST /frame HTTP/1.1" 200 -
al-section-service-1 | 172.27.0.6 - - [10/Nov/2023 18:09:26] "POST /persons HTTP/1.1" 200 -
al-collector-service-1 | 172.27.0.2 - - [10/Nov/2023 18:09:26] "POST /persons HTTP/1.1" 200 -
al-image-analysis-service-1 | 172.27.0.6 - - [10/Nov/2023 18:09:26] "POST /frame HTTP/1.1" 200 -
al-alert-service-1 | 172.27.0.6 - - [10/Nov/2023 18:09:26] "POST /alerts HTTP/1.1" 200 -
al-collector-service-1 | 172.27.0.3 - - [10/Nov/2023 18:09:26] "POST /known-persons HTTP/1.1" 200 -
al-face-recognition-service-1 | 172.27.0.6 - - [10/Nov/2023 18:09:26] "POST /frame HTTP/1.1" 200 -
al-collector-service-1 | 172.27.0.7 - - [10/Nov/2023 18:09:26] "POST /frame HTTP/1.1" 200 -
al-section-service-1 | 172.27.0.6 - - [10/Nov/2023 18:09:26] "POST /persons HTTP/1.1" 200 -
al-collector-service-1 | 172.27.0.2 - - [10/Nov/2023 18:09:26] "POST /persons HTTP/1.1" 200 -
al-image-analysis-service-1 | 172.27.0.6 - - [10/Nov/2023 18:09:26] "POST /frame HTTP/1.1" 200 -
al-face-recognition-service-1 | 172.27.0.6 - - [10/Nov/2023 18:09:26] "POST /frame HTTP/1.1" 204 -
al-collector-service-1 | 172.27.0.7 - - [10/Nov/2023 18:09:26] "POST /frame HTTP/1.1" 200 -
al-section-service-1 | 172.27.0.6 - - [10/Nov/2023 18:09:26] "POST /persons HTTP/1.1" 200 -
al-collector-service-1 | 172.27.0.2 - - [10/Nov/2023 18:09:26] "POST /persons HTTP/1.1" 200 -
al-image-analysis-service-1 | 172.27.0.6 - - [10/Nov/2023 18:09:26] "POST /frame HTTP/1.1" 200 -
```

```

1  [
2    "code": 200,
3    "type": "STATUS",
4    "message": "Streamed 10 frame(s) in 2.439497232437134 seconds, with 0 failed requests."
5  ]

```

Transitioning from Docker based approach to GKE Kubernetes turned out to be quite a difficult task. First all kubernetes-manifest files had to be written as xxx.service.yaml and xxx.deployment.yaml files. The deployment specifies the container and how many Replicas(Pods) are being deployed and from where to pull the docker image. The collector service written in python has been uploaded to Docker and can be pulled via “**st600/collector-service:latest**”. The service manifest file gives the service a name and what port are to be exposed. For testing purposes every service was of the type Loadbalancer. The most difficult problem to overcome was Ingress, because it was unclear how the HTTP request were being resolved on GKE. After Ingress was successfully implemented every service was deployed as type ClusterIp, in order to not expose every service directly and enforce to only get access over Ingress. All services and pods were deployed on the default namespace, except Grafana(the tutorial was followed to the letter).

Communication

NGINX-Ingress:

Nginx ingress was installed on GKE via helm as it was described in the assignment sheet with helm. The my-release-ingress-nginx-controller service (Load Balancer) is basically configured to work as a simple http proxy which forwards all incoming http requests to one of the 4 cameras and Alerts and Section service. An http request can take the form of:

```
curl -v http://<exposed-External-ip-Address>.nip.io/cam<1-4>/stream?toggle=on
```

as concrete examples:

```
curl -v http://34.30.212.46.nip.io/cam1/stream?toggle=on
```

```
curl -v http://34.30.212.46.nip.io/cam2/stream?toggle=on
```

```
curl -v http://34.30.212.46.nip.io/cam3/stream?toggle=on
```

```
curl -v http://34.30.212.46.nip.io/cam4/stream?toggle=on
```

Alert Service:

<http://34.30.212.46.nip.io/alerts-svc/alerts?>

[from](#)={{startOfDay}}&to={{endOfDay}}&aggregate=count

Section Service:

<http://34.30.212.46.nip.io/section-svc/persons?>

[from](#)={{startOfDay}}&to={{endOfDay}}&aggregate=count

Useful Pre-Request Script in Postman:

```
let moment = require('moment');
```

```
pm.variables.set('startOfDay', moment().utc().startOf('day').format('YYYY-MM-DD HH:mm:ss'));
```

```
pm.variables.set('endOfDay', moment().utc().endOf('day').format('YYYY-MM-DD HH:mm:ss'));
```

The implementation of the ingress_v1.yaml is based on the “Simple Fanout” approach (<https://kubernetes.io/docs/concepts/services-networking/ingress/>). This approach routes HTTP traffic to different Services at the same IP address exposed by the ‘**my-release-ingress-nginx-controller**’. After the implementation of Ingress every service had the type of ClusterIp. This means no External Ips were being assigned to all Assignment services.

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
alert-service	ClusterIP	10.8.14.156	<none>	80/TCP	27h
alertmanager-operated	ClusterIP	None	<none>	9093/TCP, 9094/TCP, 9094/UDP	27h
camera-1-service	ClusterIP	10.8.9.241	<none>	8080/TCP	27h
camera-2-service	ClusterIP	10.8.3.42	<none>	8080/TCP	27h
camera-3-service	ClusterIP	10.8.14.201	<none>	8080/TCP	27h
camera-4-service	ClusterIP	10.8.3.24	<none>	8080/TCP	27h
collector-service	ClusterIP	10.8.13.180	<none>	80/TCP	27h
face-recognition-service	ClusterIP	10.8.15.166	<none>	80/TCP	27h
image-analysis-service	ClusterIP	10.8.12.29	<none>	80/TCP	27h
kubernetes	ClusterIP	10.8.0.1	<none>	443/TCP	27h
my-release-ingress-nginx-controller	LoadBalancer	10.8.2.137	34.68.148.0	80:30433/TCP, 443:32731/TCP	27h
my-release-ingress-nginx-controller-admission	ClusterIP	10.8.10.161	<none>	443/TCP	27h
prom-release-kube-promethe-alertmanager	ClusterIP	10.8.3.101	<none>	9093/TCP	27h
prom-release-kube-promethe-blackbox-exporter	ClusterIP	10.8.8.229	<none>	19115/TCP	27h
prom-release-kube-promethe-operator	ClusterIP	10.8.2.166	<none>	8080/TCP	27h
prom-release-kube-promethe-prometheus	ClusterIP	10.8.8.151	<none>	9090/TCP	27h
prom-release-kube-state-metrics	ClusterIP	10.8.7.200	<none>	8080/TCP	27h
prom-release-node-exporter	ClusterIP	10.8.1.250	<none>	9100/TCP	27h
prometheus-operated	ClusterIP	None	<none>	9090/TCP	27h
section-service	ClusterIP	10.8.12.57	<none>	80/TCP	27h

Each service is able to communicate to other services, which have exposed ports, via ClusterIP or DNS resolving, e.g. <http://collector-service:80/known-persons>, but this only works within a GKE cluster. Each service and container has opened the port 80, within a cluster, so each service can be discovered via the service-name:80. The name of a service can be discovered with “**kubectl get svc**”. As described before one of the cameras can be accessed via http request and ordered to send frames to the collector service and the collector service forwards the messages accordingly.

Ingress Configuration:

```
stefan_cabbit@cloudshell:~ (cloud-computing-2024w)$ kubectl describe ingress
Name:          airport-ingress
Labels:        <none>
Namespace:     default
Address:
Ingress Class: nginx-default
Default backend: <default>
Rules:
  Host            Path    Backends
  ----            -
  34.30.212.46.nip.io
    /cam1(/|$)(.*) camera-1-service:8080 (10.96.2.24:80)
    /cam2(/|$)(.*) camera-2-service:8080 (10.96.2.25:80)
    /cam3(/|$)(.*) camera-3-service:8080 (10.96.2.26:80)
    /cam4(/|$)(.*) camera-4-service:8080 (10.96.2.27:80)
    /alert-svc(/|$)(.*) alert-service:80 (10.96.2.23:80)
    /section-svc(/|$)(.*) section-service:80 (10.96.2.30:80)
Annotations:
  nginx.ingress.kubernetes.io/rewrite-target: /$2
  nginx.ingress.kubernetes.io/use-regex: true

Events:
  Type    Reason    Age   From                      Message
  ----    -
  Normal  Sync      8s    nginx-ingress-controller  Scheduled for sync
stefan_cabbit@cloudshell:~ (cloud-computing-2024w)$ curl http://$NGINX_INGRESS_IP.nip.io/cam3/config
{"name": "Random Camera", "section": -1, "event": "entry", "extra-info": ""}stefan_cabbit@cloudshell:~ (cloud-computing-2024w)$
```

This approach Hosts the ingress at <external-ip>.nip.io and

rewrites the address as in the following example:

<http://34.30.212.46.nip.io/cam4/stream?toggle=on>

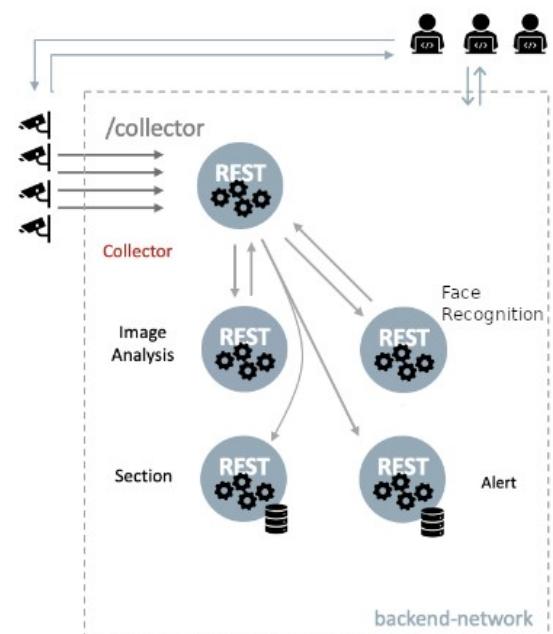
to

<http://34.30.212.46.nip.io/stream?toggle=on>

enabling access to the camera services + section and alert service.

DataFlow

As can be observed in the figure to the right the collector service handles every, that is being sent. This approach was chosen to enable asynchronous calls and to have more control over the messages within the backend-network. This approach would increase the independence of these 5 services, because



the entire communication is being handled via the collector service. If a message changes in some way the Collector can act as a Gatekeeper on what is being transmitted. The other way would have been to send result generated in Image-Analysis to the section-service directly and the Face-Recognition-service sends the result directly to the alert service. Sadly due to time constraints and constant error messages asynchronous was not implemented. Although it has not been implemented successfully asynchronous calls would help to overcome bigger frame-loads.

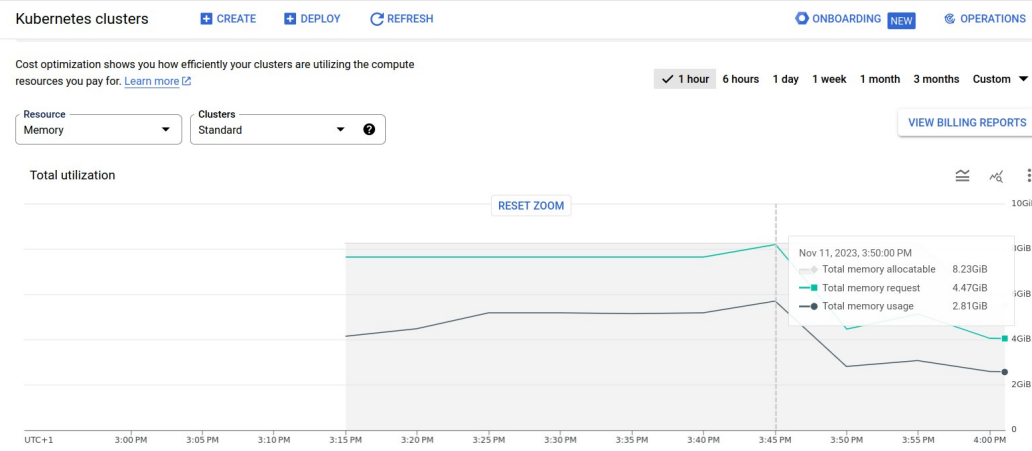
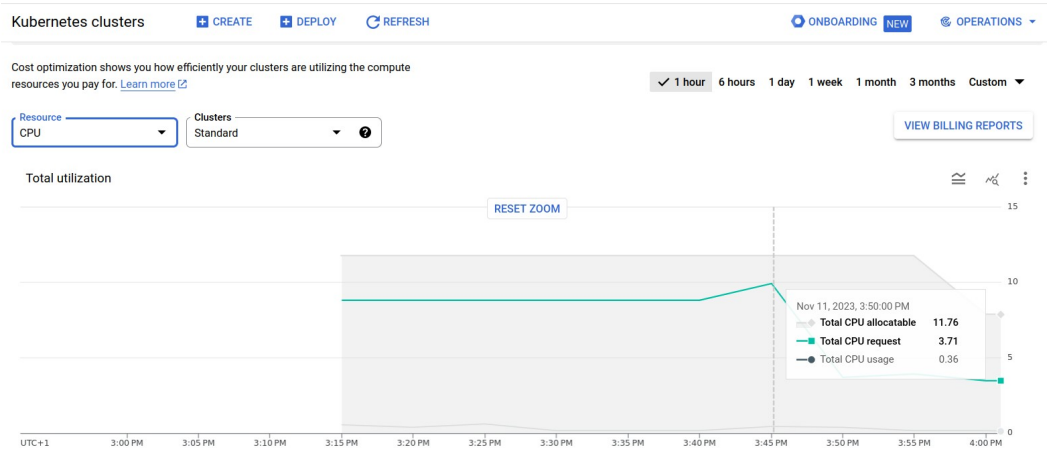
Scaling

The most computationally intensive services were the Face Recognition service and the Image Analysis service. Both represent bottlenecks, thus both have been scaled to 4 pods (or replicas) each. Different combinations like (#Face-Recognition Pods, #Image-Analysis Pods) (1, 7), (2, 6), (3, 5)... have been tried, but (4, 4) has yielded the best results. 4 Pods each for Face Recognition and Image Analysis were the limit what the cluster could support with following limits and requests. Noteworthy are also the self healing properties these replicas offer. If one Pod is being killed either manually or by going over the cpu/memory limit then another pod is being deployed.

	Replicas	CPU – request	CPU – limit	Memory – request	Memory – limit
cameras	4	100m	200m	256Mi	512Mi
collector	1	200m	400m	64Mi	128Mi
image Analysis	4	2400m	8000m	2048Mi	4096Mi
Face Recognition	4	2800m	8000m	2048Mi	4096Mi
alert	1	50m	100m	64Mi	128Mi
section	1	50m	100m	64Mi	128Mi

The first approach to assign limits and requests was assigning the bare minimum so the Pods don't die (used Grafana to determine the request/limit metrics), but this has slowed down performance significantly even though more Pods could be Horizontally scaled. The second approach was to go overboard with assigning cpu limits and request, which has resulted in much better performance, with a constant stream of 10 frames and 0 delay by 4 cameras being handled in about 8 to 10 seconds. This was tested with Postman. A single camera stream of 10 images with 0 delay take about 6 seconds. In comparison if face recognition and image analysis service have only 1 replica then 4 cameras would need around 20-37 seconds to process 10 camera images and a single camera would need 6.7 seconds to process a frame. It stands to reason the scaling works well. The only improvement would be reimplement the collector service with asynchronous calls for further performance increase.

Compared to the calculations in the table above only 12vCPU are actually at disposal, but setting these limits so high has yielded much better performance. Increasing the request metrics is not supported because the scheduler won't allow it, which is something that I have not figured out yet why this metric, while it is far below the limit, is crossing a threshold for the pod scheduler.



Horizontal Auto Scaling for Face Recognition and Image Analysis was implemented with a minimum of 1 Replica and a maximum of 4 Replicas each. The auto scaler uses the cpu utilization metric in order to determine if the service has to be scaled. This might help to utilize the node scaling property of GKE in order to pay for what is used and scale the system if a heavier workload is being processed. The AutoScaler will scale down to 1 if cpu utilization is below the target of 60.

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
cpu-face-recognition-service	Deployment/face-recognition-service	30%/60%	1	4	4	24m
cpu-image-analysis-service	Deployment/image-analysis-service	90%/60%	1	4	3	24m

```
stefan_cabbit@cloudshell:~/A1/kubernetes-manifests (rising-amp-403618)$ kubectl get hpa
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
cpu-face-recognition-service	Deployment/face-recognition-service	0%/60%	1	4	1	26m
cpu-image-analysis-service	Deployment/image-analysis-service	1%/60%	1	4	1	26m

```
stefan_cabbit@cloudshell:~/A1/kubernetes-manifests (rising-amp-403618)$
```

Cluster Configuration, Costs and Discussion

A standard cluster was created in accordance with the Assignment description with the following specs:

- E2 nodes with 4 vCPUs and 4 GB memory.
- 3 nodes and enabled “Cluster Autoscaler”
- Standard Persistent Disk with 100GB
- Cloud Monitoring was enabled for everything(Pods, services...)

GKE Standard Node Pool

3 x



Region: Iowa

2,190 total hours per month

Provisioning model: Regular

Instance type: e2-highcpu-4

USD 216.68

Operating System / Software: Free

Estimated Component Cost: USD 216.68 per 1 month

a) Standard GKE cluster

The full cluster would cost 0,41\$ per hour (price while creating a cluster). 1 day would cost 9.84\$, around 305.05 and a year with 365 days 3591.6\$. Using the calculator <https://cloud.google.com/products/calculator/> would yield 216.68\$ per month, resulting in a yearly price of 2600,16\$

b) Autopilot GKE Clusters



The Autopilot Clusters pricing can not be calculated via the calculator, but on the following website: <https://cloud.google.com/kubernetes-engine/pricing>, a price estimate can be made with the following pricing table. Calculating with 12 vCPU, 12 GB Memory and 100 SSD Storage. The



Price for an autopilot cluster for 1 year would be around 410,24\$.

Taiwan (asia-east1)		Hourly <input checked="" type="radio"/> Monthly <input type="radio"/>		
Item	Regular price	Spot price*	1 year commitment price(USD)	3 year commitment price(USD)
GKE Autopilot vCPU Price (vCPU)	\$37.595	\$11.315	\$30.076	\$20.67725
GKE Autopilot Pod Memory Price (GB)	\$4.160854	\$1.248227	\$3.3286832	\$2.2884697
GKE Autopilot Ephemeral Storage Price (GB)	\$0.046355	\$0.046355	\$0.037084	\$0.0254953
GKE Autopilot Ephemeral SSD Storage Price (GB)**	\$0.117384	\$0.117384	\$0.0939072	\$0.0645612

c) Google Cloud Run

As in IT tradition the assumption is the worst case with 305.05\$ a month for (a), the monthly requests needed to match up with 305.05\$ would be around 300 million per month and around 400 Million for (b)

Cloud Run	
 	
Region: Iowa	
CPU allocation type: CPU is only allocated during request processing	
CPU: 1	
Memory: 0.5 GiB	
CPU allocation time: 7,500,000 vCPU-second	USD 175.68
Memory allocation time: 3,750,000 GiB-second	USD 8.48
Number of requests: 300,000,000	USD 119.20
Minimum number of instances: 1	USD 9.86
USD 313.21	

Cloud Run	
 	
Region: Iowa	
CPU allocation type: CPU is only allocated during request processing	
CPU: 1	
Memory: 0.5 GiB	
CPU allocation time: 10,000,000 vCPU-second	USD 235.68
Memory allocation time: 5,000,000 GiB-second	USD 11.60
Number of requests: 400,000,000	USD 159.20
Minimum number of instances: 1	USD 9.86
USD 416.34	

Buying hardware with similar specs would cost somewhere around 4500 to 6000€ divided by a estimated lifetime of 10 years (depending on components picked), but maintenance and software still has to be added up. Compared to the yearly cost of GKE the cost is manageable compared to owning the hardware, which would need maintenance, security for this assignment and after that there is still the risk that hardware is could break.

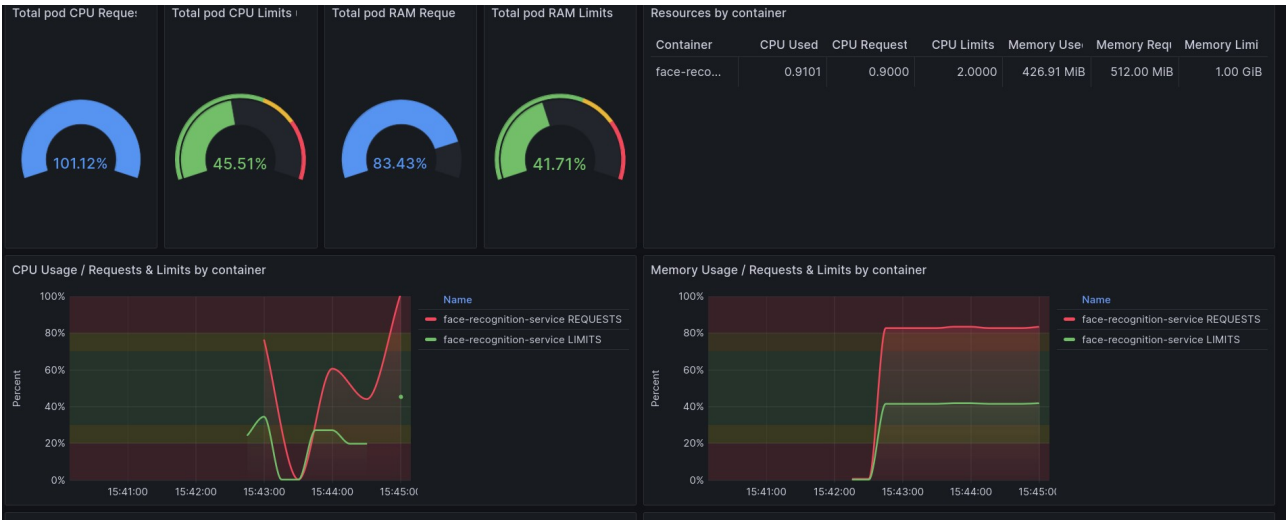
Misc

There have been a few more problems with Prometheus. The assignment description has not provided a complete installation guide, which led me to discover that the following commands work well:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm repo update
helm install prom-release bitnami/kube-prometheus
```

The basic premise of assigning request and limits was at first to assign only as much as is needed. This turned out to be a mistake and then with a few exception the rule that the limit has to be the double of request was implemented which worked pretty well. As in the documentation mentioned it is ok if a pod is over 100% request metrics as can be seen in grafana below.

Workload of face recognition service while 4 cameras stream.



In order to determine if a service is receiving data the following metric in Grafana was used to see if a service could be reached:

