

# Documentation Assignment 2 - Container Orchestration, Kubernetes

## VU Cloud Computing, Faculty of Computer Science, University of Vienna

Written by: Jovana Krtolica

Matrikel Number: a11946909

For the second assignment, the task was to use Kubernetes to run the system on the Google Kubernetes Engine and use Google Cloud Run to make serverless containers.

### 1. Kubernetes:

The first step towards solving the assignment would be to enable the Kubernetes Engine API. After enabling the API, I have created my Kubernetes cluster. To create the cluster, I went to the option Kubernetes Engine in the Google Cloud Platform. Then I have clicked on the create button and from there I have generated my first cluster. The name of the cluster is **assignment2**. From the options, I have chosen the zone **europa-central-2-a** and the **default-3-node cluster**. Then I have entered in the cluster and pulled the newest version of my Git repository.

The first step towards deployment was to create .yaml files for each pod which represents the image of the container. There are eight files like this where each file represents a pod of one service image. Pods represent group of one or more containers with shares storage and network resources as well as a specification for how to run the containers. I have decided to use "one-container-per-Pod" model because it is the most common Kubernetes use-case and the Pod is actually wrapped around a single container. In the POD files, I specify the name of the pod, the name of the container, the image location from where it needs to be pulled and the container port. For the five services implemented already, the pulling of the images is realized in an easy way as the images are existing on Docker Hub. For the services implemented by myself, I have followed steps from the tutorial specified in the link in order to build the images.

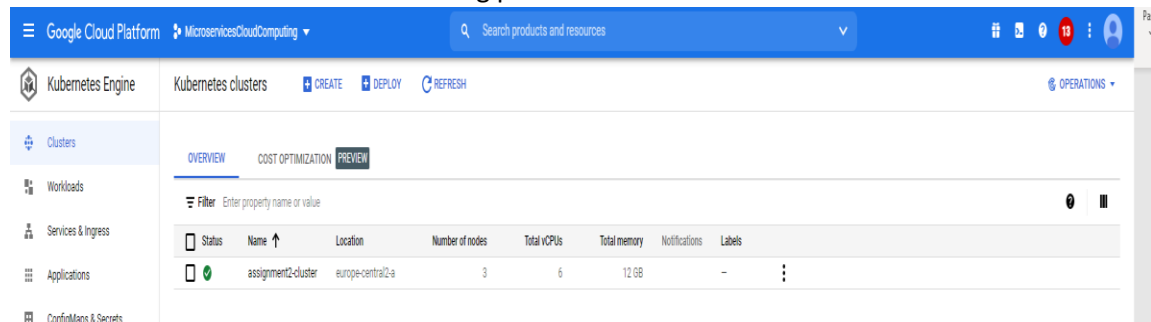
<https://cloud.google.com/kubernetes-engine/docs/tutorials/hello-app>

The first thing I have done is that I have created a repository, then built the images and pushed the images to the Artifact Registry. When I have created the cluster, I have taken the image URL of each image built and added that image in the pod files. The URLs for the three images are the following:

For each service I have a POD .yaml file. To execute any command within the cluster, the user must be connected with the cluster. As I named my project **microservicecloudcomputing** and my cluster **assignment2-cluster**, I execute the following command in the Google Shell to connect to the cluster:

```
gcloud container clusters get-credentials assignment2-cluster --zone Europe-central2-a --project microservicecloudcomputing
```

The cluster can be seen on the following picture:



After this command is executed, the user needs to pull the repository from Git and go to destination **a11946909->Assignment2->Services** to start executing the commands. It is advised just to be sure for the user to create the .jar files. The jar files can be executed in the following way:

- Navigate to **a11946909->Assignment2->Services->cpanel-service**
- Execute first **mvn clean**, then **mvn package**
- Navigate to **target** and execute **ls** to be sure that the .jar was created
- Exit from this service and navigate to **human-detection** or **collector** and repeat the steps

A **Pod** represents the smallest deployable unit of computing that can be created and managed in Kubernetes. The Pods I have created have only one container in themselves, they contain the image required for that pod and create a container on a certain predefined port. The Pods themselves have ClusterIP meaning that Kubernetes will create a stable IP address that is accessible from nodes inside the cluster and this was a requirement in the assignment. When arriving at the destination **a11946909->Assignment2->Services**, the **Pod** files are applied in GKE through the following command:

```
kubectl apply -f {name_of_pod_file}
```

To check all the pods and their lifecycle, I have used the following command:

```
kubectl get pods
```

After all the **Pods** were running, I have created a **Service** object for each service. Service represents an abstraction which is defining logical set of Pods and a policy by which accessing should be made. The set of Pods targeted by Service is usually determined by a selector. For each Pod I have created a Service object which listens on TCP and on some exposed port which in the **service.yml** file can be seen by the variable targetPort. The service knows for which Pod it is made based on the app label in the file. I have chosen the type of the Service to be **LoadBalancers**

**because** the Ingress file exposes and uses services of type NodePort or LoadBalancer. The following picture shows all the services in GKE with their name, status, type, endpoints, pods, namespaces and related clusters.

Name	Status	Type	Endpoints	Pods	Namespace	Clusters
alert	OK	External load balancer	34.116.234.214.80	1/1	default	assignment2-cluster
camera	OK	External load balancer	34.116.232.240.80	1/1	default	assignment2-cluster
collector	OK	External load balancer	34.118.86.39.31700	1/1	default	assignment2-cluster
cpanel	OK	External load balancer	34.116.152.145.3800	1/1	default	assignment2-cluster
face-recognition	OK	External load balancer	34.118.104.75.80	1/1	default	assignment2-cluster
image-analysis	OK	External load balancer	34.118.92.142.80	1/1	default	assignment2-cluster
ingress-nginx-controller	OK	External load balancer	34.116.169.124.80	1/1	ingress-nginx	assignment2-cluster
ingress-nginx-controller-admission	OK	Cluster IP	10.64.4.130	1/1	ingress-nginx	assignment2-cluster
section	OK	External load balancer	34.118.124.190.80	1/1	default	assignment2-cluster

After the services, I have written the **Deployment** objects for the deploying of each service. A Deployment allows you to describe an application's life cycle, such as which images to use for the app, the number of pods there should be, and the way in which they should be updated. Additionally, it ensures the desired number of pods are running and they are always available. The Kubernetes deployment object lets you to deploy a replica set or a pod, to update them, to rollback to previous deployment versions, scale a deployment, pause it or continue it. I have written deployment files for each service. Some of the services include two replicas, whereas some of the services include only one replica. Replicas are important because we need to put the pod in it as if we deploy the pod directly, we get a single vulnerable pod with no self-healing and no scalability. That is why the deployment object itself contains the pods. On the following picture, the deployment objects can be seen under Workloads in the GKE.

Name	Status	Type	Pods	Namespace	Cluster
alert	OK	Deployment	1/1	default	assignment2-cluster
camera	OK	Deployment	1/1	default	assignment2-cluster
collector	OK	Deployment	1/1	default	assignment2-cluster
cpanel	OK	Deployment	1/1	default	assignment2-cluster
face-recognition	OK	Deployment	1/1	default	assignment2-cluster
human-detection	OK	Deployment	1/1	default	assignment2-cluster
image-analysis	OK	Deployment	1/1	default	assignment2-cluster
ingress-nginx-admission-create	OK	Job	0/1	ingress-nginx	assignment2-cluster
ingress-nginx-admission-patch	OK	Job	0/1	ingress-nginx	assignment2-cluster
ingress-nginx-controller	OK	Deployment	1/1	ingress-nginx	assignment2-cluster
section	OK	Deployment	1/1	default	assignment2-cluster

The service object as well as the deployment object for each microservice is included in a manifest .yaml file for the service. The pod file DOES NOT need to be executed when there is a manifest file because the manifest file contains the pods. When we are going to deploy to Kubernetes or create a Kubernetes resource like pod, replica, service or deployment, the manifest file is written which describes those objects and its attributes in yaml and json. It is recommended to have this file because you can version control it and use it in a repeatable way.

**kubectl apply -f {name\_of\_manifest\_file}**

The manifest file will then create the objects which are specified in the file with the value on the word Kind. All the files which are created can be seen either in GKE under workloads for deployment objects and under services for service objects. Additionally, you can see them by entering the following commands.

**Kubectl get services**

**kubectl get deployments**

All the manifest files with the Kubernetes objects inside them are located inside the Git repository on the following location **a11946909->Assignment2-Services** and are named with regards to the microservice they refer to.

## 2. Ingress file:

The Ingress file represents an API object which manages external access to the services in a cluster typically using HTTP requests. It exposes the HTTP routes from outside the cluster to services within the cluster. In order to create an Ingress there must be an Ingress controller because only creating an Ingress resource has no effect. To install the controller, I have followed the following tutorial.

<https://kubernetes.github.io/ingress-nginx/deploy/>

Execute the commands in the following order:

```
helm upgrade --install ingress-nginx ingress-nginx \
  --repo https://kubernetes.github.io/ingress-nginx \
  --namespace ingress-nginx --create-namespace
```

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.0.4/deploy/static/provider/cloud/deploy.yaml
```

```
kubectl get pods --namespace=ingress-nginx
```

After the installation was successful, I have started writing the Ingress file. In the file, I specify the paths and the services for which these paths are meant as well as their ports. The Ingress file is located inside the Git repository and when pulled from the Google Shell to the a11946909->Assignment2->Services, it can be executed with the following command:

**kubectl apply -f ingress-file.yml**

The file can then be found inside GKE in the Services & Ingress part, under the Ingress tab. I have tested the paths whether they work and got results back which are going to be shown in the following pictures. In the following three pictures, the Ingress file can with its functionalities can be seen:

Kubernetes Engine

Ingress Details
REFRESH
EDIT
DELETE
KUBECTL

**airport-system-ingress-file**

DETAILS
TELEMETRY
LOGS
EVENTS
YAML

Cluster

assignment2-cluster

Namespace

default

Created

Nov 11, 2021, 1:21:09 AM

Labels

No labels set

Annotations

kubernetes.io/ingress.class: nginx  
nginx.ingress.kubernetes.io/rewrite-target: \$2

SHOW ALL ANNOTATIONS

Type

Ingress

IP address

34.116.169.124

Routes

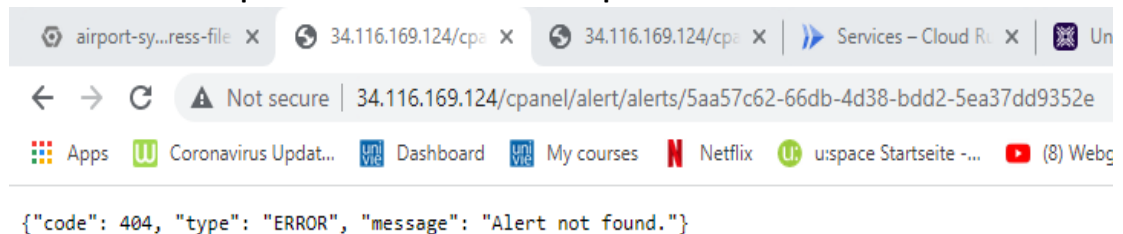
Route	Service	Pod selector	Clusters	Pods
34.116.169.124/camera(/ \$)(.*)	camera	app = camera	assignment2-cluster	1/1
34.116.169.124/alert(/ \$)(.*)	alert	app = alert	assignment2-cluster	1/1
34.116.169.124/section(/ \$)(.*)	section	app = section	assignment2-cluster	1/1
34.116.169.124/face-recognition(/ \$)(.*)	face-recognition	app = face-recognition	assignment2-cluster	1/1
34.116.169.124/image-analysis(/ \$)(.*)	image-analysis	app = image-analysis	assignment2-cluster	1/1
34.116.169.124/collector(/ \$)(.*)	collector	app = collector	assignment2-cluster	1/1
34.116.169.124/human-detection(/ \$)(.*)	human-detection	No pod selectors set	assignment2-cluster	0/0
34.116.169.124/cpanel/collector(/ \$)(.*)	collector	app = collector	assignment2-cluster	1/1
34.116.169.124/cpanel/alert(/ \$)(.*)	alert	app = alert	assignment2-cluster	1/1
34.116.169.124/cpanel/section(/ \$)(.*)	section	app = section	assignment2-cluster	1/1
34.116.169.124/*	cpanel	app = cpanel	assignment2-cluster	1/1

## Serving pods

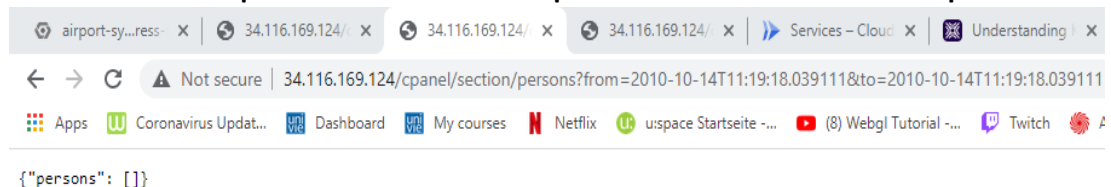
Service	Name	Status	Restarts	Created on ↑
alert	alert-66ff48c556-ts7kj	✓ Running	0	Nov 11, 2021, 1:07:19 AM
section	section-85556cf5d4-6zdf5	✓ Running	0	Nov 11, 2021, 1:07:26 AM
camera	camera-9dd6f879-lwwsw	✓ Running	0	Nov 11, 2021, 1:07:34 AM
face-recognition	face-recognition-8748659cc-shn88	✓ Running	0	Nov 11, 2021, 1:07:44 AM
image-analysis	image-analysis-7ff99c7b59-zj6k8	✓ Running	0	Nov 11, 2021, 1:08:08 AM
cpanel	cpanel-5c67797799-2qfjp	✓ Running	0	Nov 11, 2021, 1:08:18 AM
collector	collector-56467cfb77-8bxf7	✓ Running	0	Nov 11, 2021, 1:08:43 AM

Now in the following pictures, the successful routes from the Ingress will be shown with having a response returned in the Browser.

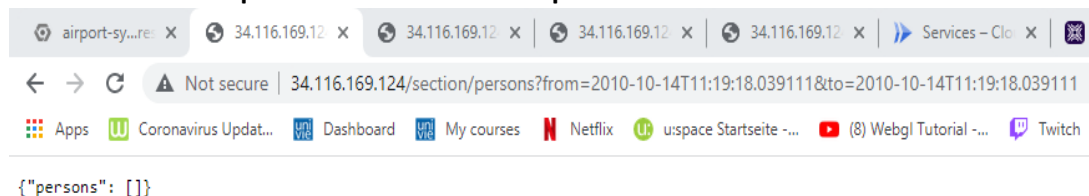
### a) HTTP call to the request URL for GET Alert from cpanel with a certain UUID



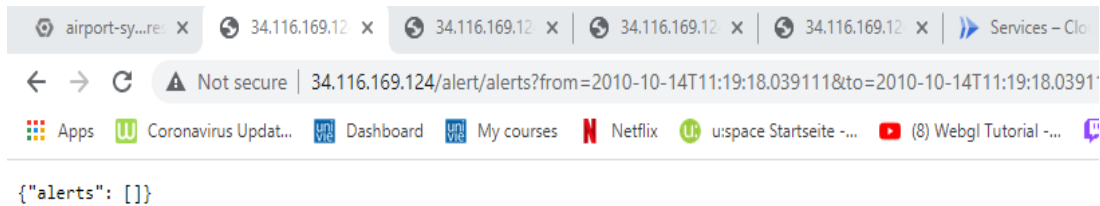
### b) HTTP call to the request URL for GET Section persons with from and to from cpanel



### c) HTTP call to the request URL for GET section persons from section

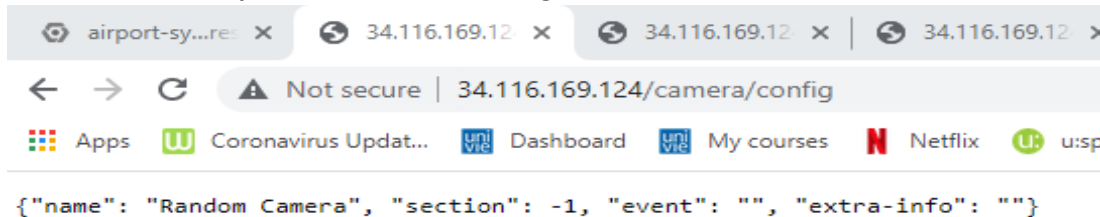


### d) HTTP call to the request URL from GET alerts from alert



```
{"alerts": []}
```

e) HTTP call to the request URL from GET configuration from camera



```
{"name": "Random Camera", "section": -1, "event": "", "extra-info": ""}
```

The other calls for the services with shown examples and for services like human-detection, face-analysis, collector and image-recognition, should be realized through sending a request body as they are mainly POST requests which require to have a request body in order to be successful.

### 3. GCP Pricing Calculator:

For this part of the assignment, I have used the Google Cloud Pricing calculator which can be found on the following link:

<https://cloud.google.com/products/calculator/#id=d2cb9edb-c256-4aa6-a218-74a936e1da68>

I have used this calculator to calculate the cost estimation for using certain features of the Google Cloud Platform and do analysis for which option fits best.

a) **Standard GKE cluster** - For the standard GKE Standard Cluster, I have done estimations both for monthly usage only and for commitment usage. Regarding the GKE Standard Node Pool, I have selected the total number of nodes in the Node Pool to be three. For the operating system, I have selected the free version which offers the container-optimized option. I have chosen a standard VM machine for general purpose from the series E2. The machine type is e2-standard-2 with 2vCPUs and 8GB RAM. For the boot disk I have chosen the standard persistent disk with boot size of 100 GiB. The datacenter location is europe-central2 in Warsaw and for the GKE Clusters I have chosen 1 zonal cluster. When I added the commitment usage to be for one year, the total estimated cost per month was 114.92 EUR. For the second case scenario, I have not added a commitment usage and I added an average hours per day of each node running to be 24 hours and the days to be 7 days per week. With the new usage time, the estimated cost per month was 211.98 EURO per month, which is almost 100 EUR more per month. Here we can argue two things, that if the cluster is needed for one year it is better to get a commitment usage packet because it would be cheap. On the other hand, if

the cluster is only needed from one until 6 months it would be better to not add the commitment usage packet. Paying 6 months per 211.98 EUR will already almost get to the amount you would pay for a cluster with commitment usage for a year. The Standard cluster manages the cluster's underlying infrastructure and gives the user more node configuration flexibility.

- b) **Autopilot GKE cluster** - When configuring the same configuration used for the Standard Cluster to the cost estimation page for a GKE Autopilot Cluster, the cost drastically changes. I get the total estimated cost to be 387.81 EUR per one month usage. The Autopilot cluster is more expensive since it manages the cluster's underlying infrastructure, including nodes and node pools, while giving you an optimized cluster with a hands-off experience. This type of cluster does not offer the committed usage option which is indeed making the Standard Cluster much cheaper both on a monthly and yearly level.
- c) **Google Cloud Run** - For this part of the calculations, I have reached the most interesting conclusion so far. In my opinion, getting Google Cloud Run would be the best fit because it is cheaper and it is almost impossible to do so much requests in one month period. Based on the calculations I have made, each service will have the location europe-central2 Warsaw, always allocate the CPU, have 1 CPU with 512GB RAM memory, execution time would be 500 requests per ms, 1 instance. The number of the requests per month varies depending on whether there would be a committed usage feature activated. If the committed usage feature is activated for one year, then each service would have 20,900,000 requests for execution per month. This sums up to the point that each service will cost 14.22 EURO per month and if there are 8 services multiplied by 14.22 EUR, then the amount would be 113.76 EUR per month, almost the same as a Standard Cluster price with commitment on a one year. A second thing happens when there is no commitment packet added. Then, each service could execute around 40,500,000 requests. Each service would then cost around 26.35 EUR per month which if there are 8 services multiplied by 26.35 would be 210.8 EUR per month, which is almost the same price as the Standard price per month with a non-committed package.

#### 4. Serverless GCR:

For this part of the assignment, I have used the option on Google Cloud Platform called Google Cloud Run. I have generated a new service through the create server option. In the container image URL, I have added the location of the human-detection image based in the artifact registry. The URL of the image is the following:

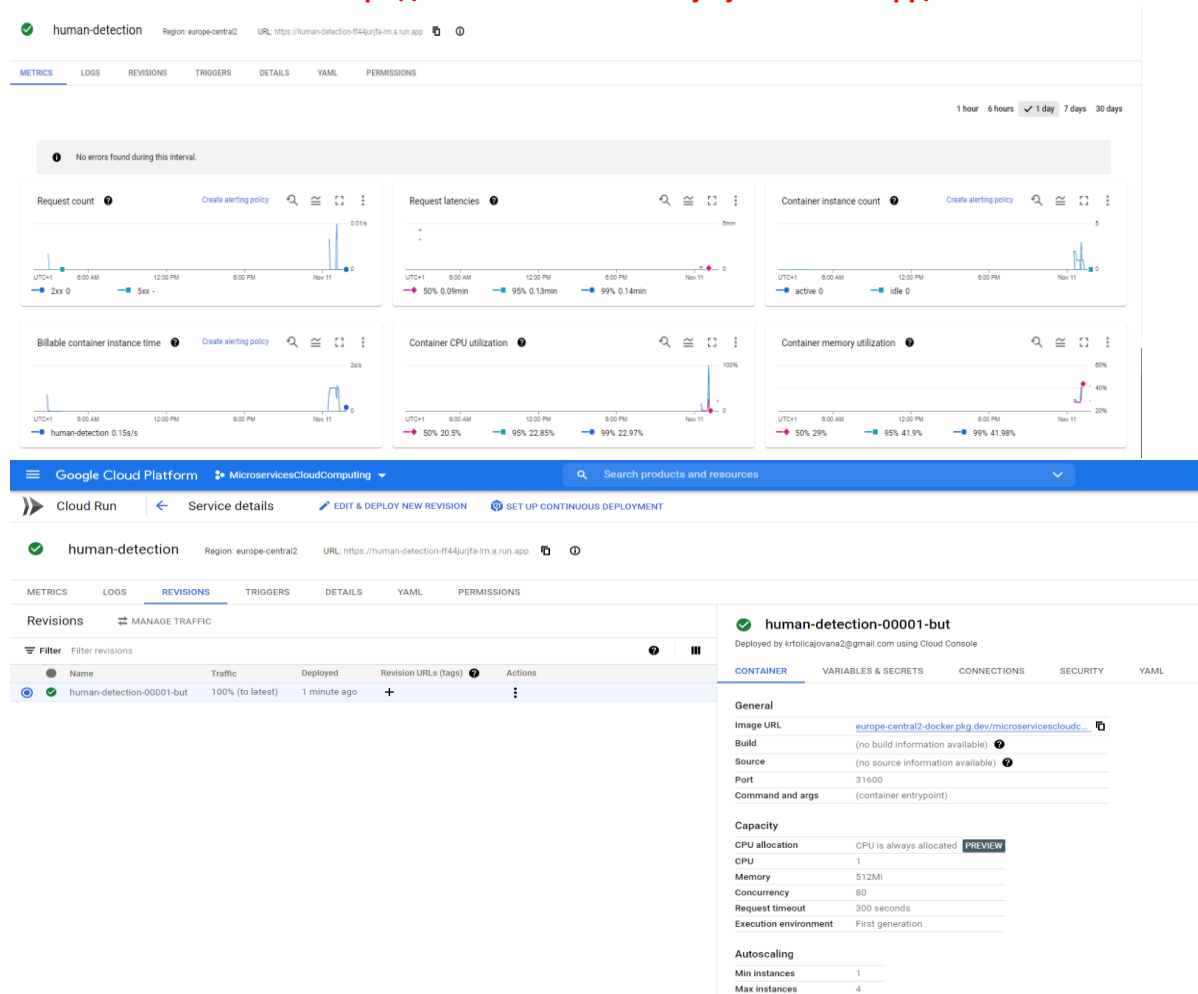
[europe-central2-docker.pkg.dev/microservicescloudcomputing/assignment2-repo/human-detection@sha256:82e6d58c785ce676a75059bf8a439ec5e4bf70f615873005dfe6d18c968cb08d](https://europe-central2-docker.pkg.dev/microservicescloudcomputing/assignment2-repo/human-detection@sha256:82e6d58c785ce676a75059bf8a439ec5e4bf70f615873005dfe6d18c968cb08d)

Then I added the name of the service and as a region I have selected europe-central2. I allocate the CPU always. Under minimum instances I add 1 and maximum 5. The container port is 31600, the capacity is 512MiB and there is only one CPU. Request timeout is 300 seconds and maximum requests per container is 80. I allow all traffic. Following below there is an URL for the service as



well as a screen shot from Google Cloud Run screen where it could be seen that the service is running on GCR.

**URL of Human-Detection: <https://human-detection-ff44jurfa-lm.a.run.app/>**



## 5. Problems:

The first problem which I have encountered was that when I started working on the assignment and just experimenting, I forgot to turn off the clusters which spent me a lot of credits. The spent amount would be less if the clusters did not contain any images inside them, but because I was trying to add the images in the cluster, there appeared to be a pull error which appeared many times and subtracted money credits in my account. In two days, my credits have gone from 20.45 until 6.47 meaning that the clusters have spent 14 credits.

A second problem I have encountered was with the building of the images for the services I have developed. My first idea was to build the image inside the service folder through building the Dockerfile. I did not manage to make this scenario work, so I followed the tutorial for building images inside the Artifact Registry. Then I used the URL from the images and added this URL for

each service in its own Pod. Regarding this problem, another problem emerged, where I had to change the Dockerfile for each of the three services regarding the location of the .jar location. Since I was not using docker-compose.yml file for this assignment, that is why I needed to make the change.

Another problem I had was that I could not create an Ingress file with services of type ClusterIP. I have read online that Ingress files are only possible with services with types LoadBalancer or NodePort.