

Cloud Computing Architecture

Semester project

March 29, 2024

Overview

The semester project consists of four parts, two of which are described in detail in this handout. In this project, you will explore how to schedule latency-sensitive and batch applications in a cloud cluster. You will deploy applications inside containers and gain experience using a popular container orchestration platform, [Kubernetes](#). Containers are a convenient and lightweight mechanism for packaging code and all its dependencies so that applications can run quickly and reliably from one computing environment to another.

You will work in groups of three students and submit a single report per group. **Please submit your report in the format of the [project report template](#).** We will be assigning groups for the project, however you will have a chance to optionally let us know your preferences for teammates. If you know one or two other students in the class that you would like to work with on the project, please submit your group preference by March 7th, 2024. To do so, each student in your preferred group should sign up for the same group number in the Project Group Selection page on Moodle. We will notify you about final group assignments on March 11th and then you may redeem your cloud credits and begin working on the project.

Important Dates

March 7th, 2024: Deadline to submit group preferences. **Remember that you must either subscribe to a group or join the general group (Group 1) to be assigned randomly by us.**

March 11th, 2024: Groups are assigned and announced. Start working on project.

March 29th, 2024 at 23:59: Deadline to submit Part 1 and 2 of the project.

May 17th, 2024 at 23:59: Deadline to submit Part 3 and 4 of the project.

We will release Part 3 and 4 of the project mid-April. Parts 3 and 4 are more open-ended and will require more time to complete than Part 1 and 2. Please plan your time accordingly.

Cloud Environment and Credits

To run experiments for the project, you will use Google Cloud. We will provide you with Google Cloud credits for your project. To redeem your cloud credits, please follow the steps in Part 1 (Section [1.1](#)), when your project group assignment is confirmed. Each group member should create

a Google Cloud account at <https://accounts.google.com>. Please use your ETH email address to create the account.

1 Part 1

In Part 1 of this project, you will run a latency-critical application, memcached, inside a container. Memcached is a distributed memory caching system that serves requests from clients over the network. A common performance metric for memcached is the tail latency (e.g., 95th percentile latency) under a desired query rate. You will measure tail latency as a function of queries per second and explore the impact of hardware resource interference. To add different types of hardware resource contention, you will use the iBench microbenchmark suite to apply different sources of interference (e.g., CPU, caches, memory bandwidth).

Follow the setup instructions below to deploy a Google Cloud cluster using the **kops** tool. Your cluster will consist of four virtual machines (VMs). One VM will serve as the Kubernetes cluster master, one VM will be used to run the memcached server application and iBench workloads, and two VMs will be used to run a client program that generates load for the memcached server.

This document contains setup instructions. Answer and submit the questions for Part 1 of the project in the report [template](#).

1.1 Setup Instructions

Installing necessary tools

For the setup of the project, you will need to install [kubernetes](#), [google-tools](#) and **kops**. Instructions based on the operating system on your local machine are provided in the links above. Having installed all the tools successfully, the following three commands should return output in your terminal (for the rest of the document the \$ symbol is there to declare a bash command and you shouldn't type it explicitly):

1. `$ kubectl --help`
2. `$ kops --help`
3. `$./google-cloud-sdk/bin/gcloud --help`

Note that the final command is relative to where you have downloaded the google cloud tools. If you have installed via a package manager or have added the gcloud tools to your `$PATH` you don't need the prefix and you can just type **gcloud**. Note that you have to open a new terminal or refresh your shell using **source** for your `$PATH` to be updated.

All the scripts that you will need for both parts of the project are available [here](#):

```
git clone https://github.com/eth-easl/cloud-comp-arch-project.git
```

Redeeming cloud credits and creating Google Cloud project

Each group member should create a Google Cloud account at <https://accounts.google.com>. Use your ETH email address to create the account. Each group will receive a \$50 Google Cloud coupon code. Select **one** group member to enter their name and ETH email address at the link you will receive when the groups have been assigned. Only redeem one coupon per group. If you need more credits you can get in touch with the TA team.

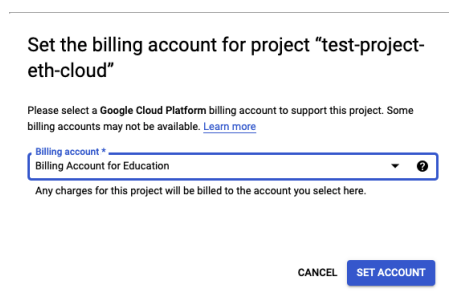
After installing kubernetes tools, connect your local client to your google cloud account using:

gcloud init

A browser window will open and you will have to login in with your ETH address. Afterwards, you will give **google-cloud-sdk** permissions to your account and then in the command line you will pick a name for the project. When creating the project name use **cca-eth-2024-group-XXX** (where XXX is your group number). **Only one group member (who also redeemed the cloud credit coupon) should create the Google Cloud project.** This person will add other group members as Project Owners (see instructions below). After the other group members are added as Project Owners, they will simply select the existing project name when they run the **gcloud init** command. All group members will have access to the project and share the cloud credits.

Do not configure any default computer region and zone. For deploying a cluster on Google Cloud we will modify some of the instructions listed [here](#), which will be given below.

After creating the project you can log into the [google cloud console](#) and will be prompted to select a billing account for the project. In the pop up choose **Billing account for education** as below and click **Set account**:



Afterwards, you can try the command **gcloud compute zones list**. The first time you should get a prompt to enable the compute engine API that looks like this:

```
API [compute.googleapis.com] not enabled on project [project number].
Would you like to enable and retry (this will take a few minutes)? (y/N)?
```

After the API is enabled you can repeat the command which should now yield the following output:

```
$ gcloud compute zones list
```

NAME	REGION	STATUS
us-east1-b	us-east1	UP
us-east1-c	us-east1	UP
us-east1-d	us-east1	UP
us-east4-c	us-east4	UP
us-east4-b	us-east4	UP
us-east4-a	us-east4	UP
us-central1-c	us-central1	UP
us-central1-a	us-central1	UP

us-central1-f	us-central1	UP
us-central1-b	us-central1	UP
us-west1-b	us-west1	UP
us-west1-c	us-west1	UP
us-west1-a	us-west1	UP
europa-west4-a	europa-west4	UP
europa-west4-b	europa-west4	UP
europa-west4-c	europa-west4	UP
europa-west1-b	europa-west1	UP
europa-west1-d	europa-west1	UP
europa-west1-c	europa-west1	UP
europa-west3-c	europa-west3	UP
europa-west3-a	europa-west3	UP
europa-west3-b	europa-west3	UP
europa-west2-c	europa-west2	UP
europa-west2-b	europa-west2	UP
europa-west2-a	europa-west2	UP

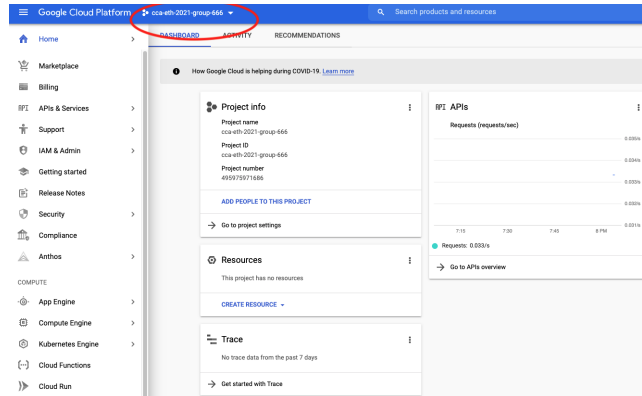
Then you will need to configure your default credentials using:

```
$ gcloud auth application-default login
```

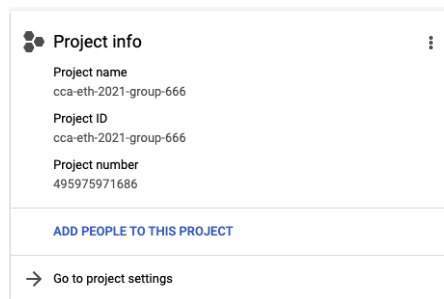
This will redirect you to a browser window where you will login with the same account you used when you setup the `gcloud init` command.

Giving your teammates owner permission to the project

After creating the `cca-eth-2024-group-XXX` project on Google Cloud, give your group members access to the project and cloud credits by navigating to the [Google Cloud console menu](#). Make sure your project is properly displayed on the top left as below:



In the project info click **Add people to this project**.



Type the email addresses of your teammates, select **Owner** as a role and click **Save**. Note that your teammates should have created a google cloud account with their ETH address in advance to put them as project owners.

Add principals and roles for "cca-eth-2022-lcvetkovic" resource

Enter one or more principals below. Then select a role for these principals to grant them access to your resources. Multiple roles allowed. [Learn more](#)

New principals

fstrati@ethz.ch

Role

Owner

Condition

[Add condition](#)

Full access to most Google Cloud resources. See the list of included permissions.

[+ ADD ANOTHER ROLE](#)

SAVE

CANCEL

Deploying a cluster using **kops**

At this point you will deploy a cluster using **kops**. First of all you will need to create an empty bucket to store the configuration for your clusters. Do this by running:

```
$ gsutil mb gs://cca-eth-2024-group-XXX-ethzid/
```

... where **XXX** is your group number and **ethzid** is your ETH username. Then run the following command to have the **KOPS_STATE_STORE** command to your environment for the subsequent steps:

```
$ export KOPS_STATE_STORE=gs://cca-eth-2024-group-XXX-ethzid/
```

If you open another terminal this and other environmental variables will not be preserved. You can preserve it by adding it with an `export` command to your `.bashrc`. You should substitute the number of your group and your ETH username as before.

Small Hint: Since Windows users cannot use the "export" command, you can manually add this environment variable. This tip also applies to the following "export" commands and "PROJECT='gcloud config get-value project'" commands.

For the first part of the exercise you will need a 3 node cluster. Two VMs will have 2 cores. One of these VMs will be the node where **memcached** and **iBench** will be deployed and another will be used for for the **mcperf** memcached client which will measure the round-trip latency of memcached requests. The third VM will have 8 cores and hosts the **mcperf** client which generates the request load for the experiments.

Before you deploy the cluster with **kops** you will need an ssh key to login to your nodes once they are created. Execute the following commands to go to your `.ssh` folder and create a key:

```
$ cd ~/.ssh
$ ssh-keygen -t rsa -b 4096 -f cloud-computing
```

Once you have created the key, go to lines 16 and 43 of the **part1.yaml** file (provided in the github link above) and **substitute the placeholder values with your group number and ethzid**. Then run the following commands to create a kubernetes cluster with 1 master and 2 nodes.

```
$ PROJECT=`gcloud config get-value project`
$ kops create -f part1.yaml
```

We will now add the key as a login key for our nodes. Type the following command:

```
$ kops create secret --name part1.k8s.local sshpublickey admin -i ~/.ssh/cloud-computing.pub
```

We are ready now to deploy the cluster by typing:

```
$ kops update cluster --name part1.k8s.local --yes --admin
```

Your cluster should need around 5-10 minutes to be deployed. You can validate this by typing:

```
$ kops validate cluster --wait 10m
```

The command will terminate when your cluster is ready to use. If you get a **connection refused** or **cluster not yet healthy** messages, wait while the previous command automatically retries. When the command completes, you can type:

```
$ kubectl get nodes -o wide
```

... to get the status and details of your nodes as follows:

NAME	STATUS	ROLES	AGE	VERSION	INTERNAL-IP	EXTERNAL-IP
master-europe-west3-a-2s21	Ready	master	3m2s	v1.19.7	10.156.0.63	34.107.107.152
memcache-server-jrk4	Ready	node	102s	v1.19.7	10.156.0.61	34.107.94.26
client-agent-vg5v	Ready	node	98s	v1.19.7	10.156.0.62	34.89.236.52
client-measure-ngwk	Ready	node	102s	v1.19.7	10.156.0.60	35.246.185.27

You can connect to any of the nodes by using your generated ssh key and the node name. For example to connect to the client-agent node, you can type:

```
$ gcloud compute ssh --ssh-key-file ~/.ssh/cloud-computing ubuntu@client-agent-vg5v \
    --zone europe-west3-a
```


Running memcached and the mcperf load generator

To launch memcached using Kubernetes, run the following:

```
$ kubectl create -f memcache-t1-cpuset.yaml
$ kubectl expose pod some-memcached --name some-memcached-11211 \
    --type LoadBalancer --port 11211 \
    --protocol TCP

$ sleep 60
$ kubectl get service some-memcached-11211
```

Then run the following:

```
$ kubectl get pods -o wide
```

The output should look like:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
some-memcached	1/1	Running	0	42m	100.96.3.3	memcache-server-zns8

Use the IP address above (100.96.3.3 in this example) as the **MEMCACHED_IP** in the remaining instructions. Now ssh into both the **client-agent** and **client-measure** VMs and run the following commands to compile the **mcperf** memcached load generator:

```
$ sudo apt-get update
$ sudo apt-get install libevent-dev libzmq3-dev git make g++ --yes
$ sudo cp /etc/apt/sources.list /etc/apt/sources.list~
$ sudo sed -Ei 's/^# deb-src /deb-src /' /etc/apt/sources.list
$ sudo apt-get update
$ sudo apt-get build-dep memcached --yes
$ cd && git clone https://github.com/shaygalon/memcache-perf.git
$ cd memcache-perf
$ git checkout 0afbe9b
$ make
```

On the **client-agent** VM, you should now run the following command to launch the **mcperf** memcached client load agent with 16 threads:

```
$ ./mcperf -T 16 -A
```

On the **client-measure** VM, run the following command to first load the memcached database with key-value pairs and then query memcached with throughput increasing from 5000 queries per second (QPS) to 55000 QPS in increments of 5000:

```
$ ./mcperf -s MEMCACHED_IP --loadonly
$ ./mcperf -s MEMCACHED_IP -a INTERNAL_AGENT_IP \
    --noload -T 16 -C 4 -D 4 -Q 1000 -c 4 -t 5 -w 2\
    --scan 5000:55000:5000
```

... where **MEMCACHED_IP** is from the output of **kubectl get pods -o wide** above and **INTERNAL_AGENT_IP** is from the Internal IP of the client-agent node from the output of **kubectl get nodes -o wide**. You should look at the output of **./mcperf -h** to understand the different flags in the above commands.

Introducing Resource Interference

Now we are going to introduce different types of resource interference with **iBench** microbenchmarks. Run the following commands:

```
$ kubectl create -f interference/ibench-cpu.yaml
```

This will launch a CPU interference microbenchmark. You can check it is running correctly with:

```
$ kubectl get pods -o wide
```

(wait until READY 1/1 and STATUS Running shows before starting a run).

When you have finished collecting memcached performance measurements with CPU interference, you should kill the job by running:

```
$ kubectl delete pods ibench-cpu
```

You can apply the above three steps for any of the six **ibench-cpu**, **ibench-l1d**, **ibench-l1i**, **ibench-l2**, **ibench-l1c**, and **ibench-membw** interference microbenchmarks. For Part 1 you will perform experiments to investigate the effect of the different types of interference. After now having followed this tutorial, you are able to run those experiments. First, start with reading the information of what to run for Part 1 in the project report template.

Deleting your cluster

IMPORTANT: you must delete your cluster when you are not using it! Otherwise, you will easily use up all of your cloud credits! When you are ready to work on the project, you can easily re-launch the cluster with the instructions above.

To delete your cluster, run on your local machine the command:

```
$ kops delete cluster part1.k8s.local --yes
```

If you encounter an API permissions error, make sure to enable the IAM API by visiting <https://console.cloud.google.com/apis/api/iam.googleapis.com/overview?project=<yourprojecthere>>. Make sure to replace the placeholder with your project name.

1.2 Notes

- Writing a script to automatize the data collection for Parts 1 and 2 is not mandatory and will not affect the grading. However, using automation scripts will be required for Parts 3 and 4, thus we encourage you to practice this approach in order to save time in the future.
- Parts 3 and 4 of the project are more resource-demanding and more costly in comparison to Parts 1 and 2 so make sure to plan your budget (usage of redeemed cloud credits) accordingly.

2 Part 2

In Part 2 of this project, you will run eight different throughput-oriented (“batch”) workloads from the **PARSEC** (and **SPLASH2x**) benchmark suite: **blackscholes**, **canneal**, **dedup**, **ferret**, **freqmine**, **radix** and **vips**. You will first explore each workload’s sensitivity to resource interference using iBench on a small 2 core VM (**e2-standard-2**). This is somewhat similar to what you did in Part 1 for **memcache**. Next, you will investigate how each workload benefits from parallelism by measuring the performance of each job with 1, 2, 4, 8 threads on a large 8 core VM (**e2-standard-8**). In the latter scenario, no interference is used.

Follow the setup instructions below to deploy a Google Cloud cluster and run the batch applications. Answer and submit the questions for Part 2 of the project in the report [template](#).

2.1 Setup

In order to complete this Part of the project, we will have to study the behavior of **PARSEC** in two different contexts. For both, we will require that **kubectl**, **kops** and **gcloud sdk** are set up. This should already be the case if you have completed Part 1.

We have provided you with a set of **yaml** files which are useful towards spawning **kubectl** jobs for workloads and interference. The interference files are the same as in Part 1, **but you must change the nodetype from memcached to parsec**. The workloads are in the **parsec-benchmarks** folder in the github repo. All these files cover the workloads in the **PARSEC** suite, as well as the **iBench** interference sources relevant for this part: **cpu**, **l1d**, **l1i**, **l2**, **l1c**, **memBW**.

2.1.1 PARSEC Behavior with Interference

For the first half of Part 2, you will have to set up a single node cluster consisting of a VM with 2 CPUs. For this, we will employ **kops** and make use of the **part2a.yaml** file (make sure to update the file with values for your GCP project and configBase):

```
$ export KOPS_STATE_STORE=<your-gcp-state-store>
$ PROJECT=`gcloud config get-value project`
$ kops create -f part2a.yaml
$ kops update cluster part2a.k8s.local --yes --admin
$ kops validate cluster --wait 10m
$ kubectl get nodes -o wide
```

If successful, you should see something like this:

NAME	STATUS	ROLES	AGE	VERSION	INTERNAL-IP	EXTERNAL-IP
master-europe-west3-a-9nx1	Ready	master	3m2s	v1.19.7	10.156.0.46	34.107.0.118
parsec-server-s28x	Ready	node	104s	v1.19.7	10.156.0.47	35.234.110.58

Now you should be able to connect to the **parsec-server** VM using either **ssh**:

```
$ ssh -i ~/.ssh/cloud-computing ubuntu@35.234.110.58
```

Or by using **gcloud**:

```
$ gcloud compute ssh --ssh-key-file ~/.ssh/cloud-computing ubuntu@parsec-server-s28x \
--zone europe-west3-a
```

To make sure that the jobs can be scheduled successfully, run the following command in order to assign the appropriate label to the **parsec** node (replace the **<parsec-server-name>** with the name of the parsec server observed in the output of the **kubectl get nodes** command):

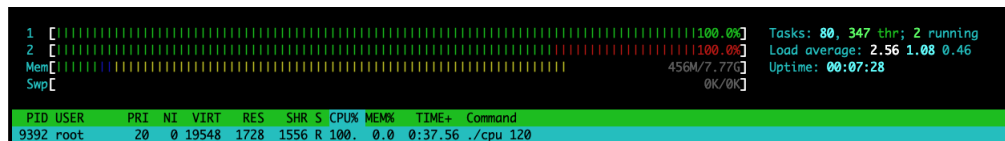
```
$ kubectl label nodes <parsec-server-name> cca-project-nodetype=parsec
```

For this part of the study we will sometimes require to set up some form of interference, and also deploy a job. For this example, we will use the **PARSEC dedup** job together with **iBench** CPU interference. Here is where we will use **kubectl** together with some of the **yaml** files we provide. The following code snippet spins up the interference, and runs the **PARSEC dedup** job:

```
$ kubectl create -f interference/ibench-cpu.yaml # Wait for interference to start
$ kubectl create -f parsec-benchmarks/part2a/parsec-dedup.yaml
```

Please note that, for Part 2a, you should use the job templates contained in the **parsec-benchmarks/part2a** folder. **blackscholes**, **canneal**, **ferret** and **frequemine** use the *simlarge* dataset, while **dedup**, **radix**, and **vips** use the *native* dataset. This is specified in the startup command for the container in the template file.

Make sure that the interference has properly started **before** running the **PARSEC** job. One way to see if the interference and the **PARSEC** job has started refers to **ssh**-ing into the VM and using the **htop** command to inspect running processes. You should see an image like below:



You can get information on submitted jobs using:

```
$ kubectl get jobs
```

In order to get the output of the **PARSEC** job, you will have to collect the logs of its pods. To do so, you will have to run the following commands.

```
$ kubectl logs $(kubectl get pods --selector=job-name=<job_name> \
--output=jsonpath='{.items[*].metadata.name}')
```

Note that the job name needs to match the one you get from **kubectl get jobs**.

Run experiments sequentially and wait for one benchmark to finish before you spin up the next one. Once you are done with running one experiment, make sure to terminate the started jobs. You can terminate them all together using:

```
$ kubectl delete jobs --all
$ kubectl delete pods --all
```

Alternatively, you can do so one-by-one using the following command:

```
$ kubectl delete job <job_name>
```

IMPORTANT: you must delete your cluster when you are not using it! Otherwise, you will easily use up all of your cloud credits! When you are ready to work on the project, you can easily re-launch the cluster with the instructions above. To delete your cluster, use the command:

```
$ kops delete cluster part2a.k8s.local --yes
```

If you encounter an API permissions error, make sure to enable the IAM API by visiting <https://console.cloud.google.com/apis/api/iam.googleapis.com/overview?project=<yourprojecthere>>. Make sure to replace the placeholder with your project name.

2.1.2 PARSEC Parallel Behavior

For the second half of Part 2, you will have to look into the parallel behavior of **PARSEC**, more specifically, how does the performance of various jobs in **PARSEC** change as more threads are added (more specifically 1, 2, 4 and 8 threads). For this part of the study, no interference is used.

You will first have to spawn a cluster as in section 2.1.1, however, this time use the **part2b.yaml** file we provided (make sure to update the file with values for your GCP project and configBase). Once more, this will be a single node cluster with an 8 CPU VM. You will have to vary the number of threads for each **PARSEC** job. To do so, change the value of the **-n** parameter in the relevant **yaml** files. The corresponding **.yaml** files are in **parsec-benchmarks/part2b** folder of the GitHub repo. Note that, for Part 2b, all of the jobs use the *native* dataset.

Other relevant instructions for this task can be found in section 2.1.1.

IMPORTANT: you must delete your cluster when you are not using it! Otherwise, you will easily use up all of your cloud credits! When you are ready to work on the project, you can easily re-launch the cluster with the instructions above. To delete your cluster, use the command:

```
$ kops delete cluster part2b.k8s.local --yes
```

2.2 Notes

- Writing a script to automate the data collection for Parts 1 and 2 is not mandatory and will not affect the grading. However, using automation scripts will be required for Parts 3 and 4, thus we encourage you to practice this approach in order to save time in the future.
- Parts 3 and 4 of the project are more resource-demanding and more costly in comparison to Parts 1 and 2 so make sure to plan your budget (usage of redeemed cloud credits) accordingly.

3 Part 3

In Part 3 of the project, you will combine the input gained from the previous two parts. You will now co-schedule the latency-critical memcached application from Part 1 and all seven batch applications from Part 2 in a heterogeneous cluster, consisting of VMs with a different number of cores. Your cluster will consist of a VM for the Kubernetes master (same as in Part 1), 3 VMs for the mcperv clients (2 agents and 1 measure machine), and 3 heterogeneous VMs (with 2, 4, and 8 cores respectively) which will be labeled as **node-a-2core**, **node-b-4core**, **node-c-8core**, respectively, and used to run memcached and the batch applications. Note that these VMs also have different configurations (as you can see in the `part3.yaml` file): **node-a-2core** is of type **n2d-highcpu-2**, **node-b-4core** is of type **n2d-highmem-4**, and **node-c-8core** is of type **e2-standard-8**. The number of CPUs, the CPU platform, and the amount of memory differ in these VMs, which is something that you should take into account when designing your scheduling policy.

Your goal is to design a scheduling policy that will minimize the time it takes for all seven batch workloads to complete (their makespan), while guaranteeing a tail latency service level objective (SLO) for the long-running memcached service. It might be helpful to take into account the characteristics of the batch applications you noted in Part 2 of the project (e.g. speedup across cores, total runtime, etc.). For this part of the project, the memcached service will receive requests from the client at a steady rate, and you will measure the request tail latency. Your scheduling policy should minimize the makespan of all batch applications, **without violating a strict service level objective** for memcached of **1 ms** 95th percentile latency at **30K QPS**. You also must ensure that all seven batch applications complete successfully, as jobs may abort due to errors (e.g. out of memory). **Use the native dataset size for all batch applications.** At every point in time, you must use as many resources of your cluster as possible.

When designing and implementing your scheduling policy, you will experiment with different collocation and resource management strategies using Kubernetes mechanisms. Utilize the knowledge you gained about the performance characteristics of each application in Parts 1 and 2 of the project. This information will help you decide the degree of parallelism you should run each workload with, and which applications you should colocate on shared resources.

The project report template containing the questions and free space you should use to enter your results can be found here: [template](#).

You may modify the **YAML** files provided, write a script for controlling the batch applications, or apply any other techniques you choose, as long as you describe them clearly in your report. You can choose which jobs to colocate, which degree of parallelism to use, and when to launch particular batch applications. You may use any Kubernetes mechanism you wish to implement your scheduling policy. You may find [node/pod affinity](#) and/or [resource requests/limits](#) particularly useful. You also may want to use **taskset** in the container command arguments to pin containers to certain CPU cores of a node. Keep in mind that a job may fail due to the lack of resources. You can use **kubectl describe jobs** to monitor jobs.

3.1 Setup

Run the following command to create a Kubernetes cluster with 1 master and 6 nodes. Make sure to update the `part3.yaml` file with the name of your project and your ConfigBase.

```
$ export KOPS_STATE_STORE=<your-gcp-state-store>
$ PROJECT='gcloud config get-value project'
$ kops create -f part3.yaml
```

You are now ready to deploy the cluster by executing:

```
$ kops update cluster --name part3.k8s.local --yes --admin
```

Your cluster should need around 5-10 minutes to be deployed. You can validate the cluster with the command:

```
$ kops validate cluster --wait 10m
```

The command will terminate when your cluster is ready to use. Afterwards, you can run:

```
$ kubectl get nodes -o wide
```

to get the status and details of your nodes as follows:

NAME	STATUS	ROLES	AGE	VERSION	INTERNAL-IP	EXTERNAL-IP
client-agent-a-d81z	Ready	node	23m	v1.19.7	10.156.15.222	35.234.120.124
client-agent-b-xpt7	Ready	node	23m	v1.19.7	10.156.15.224	34.107.4.82
client-measure-x1xw	Ready	node	23m	v1.19.7	10.156.15.223	35.242.212.158
master-europe-west3-a-cdp2	Ready	master	24m	v1.19.7	10.156.15.225	34.89.196.131
node-a-2core-qtrb	Ready	node	23m	v1.19.7	10.156.15.221	34.89.217.203
node-b-4core-gq6s	Ready	node	23m	v1.19.7	10.156.15.220	34.107.20.21
node-c-8core-3kz9	Ready	node	23m	v1.19.7	10.156.15.226	34.107.23.202

To connect to any of the machines you can run:

```
$ gcloud compute ssh --ssh-key-file ~/.ssh/cloud-computing ubuntu@<MACHINE_NAME> \
    --zone europe-west3-a
```

Modify the memcached and batch applications YAML files from Parts 1 and 2 of the project and use the **kubectl create** commands to launch the workloads in the cluster. You may want to write automated scripts to launch the jobs. Automated scripts are not a requirement in this part of the project, but we encourage you to use them here as they will be compulsory in Part 4. The memcached job must start first and continue running throughout the whole experiment, while receiving a constant load of 30K QPS from the **mcperf** client. After making sure you have started memcached and the client load, you can start the batch jobs in the desired order. Your goal is to minimize the time from the moment the first batch job was started, to the moment the last batch job completes, while also ensuring that the 95th percentile latency for memcached remains below 1ms.

For Part 3 and Part 4, you must use a modified version of **mcperf**. It provides two features: it adds two columns that contain the start and end time for each measurement, and it allows variable traces (needed for Part 4 of the project). To install the augmented version of **mcperf** on **client-agent-*** and **client-measure**, follow the instructions below:

```
$ sudo sh -c "echo deb-src http://europe-west3.gce.archive.ubuntu.com/ubuntu/ jammy main \
    restricted >> /etc/apt/sources.list"
$ sudo apt-get update
$ sudo apt-get install libevent-dev libzmq3-dev git make g++ --yes
$ sudo apt-get build-dep memcached --yes
$ git clone https://github.com/eth-easl/memcache-perf-dynamic.git
$ cd memcache-perf-dynamic
$ make
```

Instead of sweeping the request throughput, as in Part 1, you now want to generate load at a constant rate of approximately 30K QPS, while periodically reporting latency (e.g. every 10 seconds). To do this, run the following command on the `client-agent-a` machine:

```
$ ./mcperf -T 2 -A
```

and the following command on the `client-agent-b` machine:

```
$ ./mcperf -T 4 -A
```

and the following command on the `client-measure` VM:

```
$ ./mcperf -s MEMCACHED_IP --loadonly
$ ./mcperf -s MEMCACHED_IP -a INTERNAL_AGENT_A_IP -a INTERNAL_AGENT_B_IP \
    --noload -T 6 -C 4 -D 4 -Q 1000 -c 4 -t 10 \
    --scan 30000:30500:5
```

You can get the execution time of each batch job by parsing the JSON output of the `kubectl` command that returns information about the jobs, including their start and completion time. To do this, run the following command after all jobs have been completed:

```
$ kubectl get pods -o json > results.json
$ python3 get_time.py results.json
```

where `get_time.py` is a python script that you can find [here](#).

IMPORTANT: you must delete your cluster when you are not using it! Otherwise, you will easily use up all of your cloud credits! When you are ready to work on the project again, you can easily re-launch the cluster with the instructions from above.

To delete your cluster, use the command:

```
$ kops delete cluster --name part3.k8s.local --yes
```

3.2 Questions

Use the report template to answer the questions and submit your results for Part 3 of the project.

3.3 Submission

For Part 3 of the project, we expect you to submit:

- The PDF file containing the answers to the posed questions, in the form of the filled project report template.
- All YAML files you have modified or newly created.
- All scripts you have used for automation (if you used any).
- All other scripts or files you used, and consider useful for the understanding of your scheduling policy.
- Your measurement output files, **in the format explained below**:
 - Your submission **must** contain the measurements for the results described in your report.

- In the root of your submission archive, place a directory called **part_3_results_group_XXX**, where **XXX** is your group number represented with **3 digits** (e.g. for group 1, **XXX** equals 001).
- In that directory, place 6 files - 3 **.json** and 3 **.txt** files. The **.json** files **must** be named **pods_1.json**, **pods_2.json** and **pods_3.json**. The **.txt** files **must** be named **mcperf_1.txt**, **mcperf_2.txt** and **mcperf_3.txt**.
- Each **.json** file should contain the full output of the **get pods** command of the corresponding run.
- Each **.txt** file should contain the output of the mcperf execution for the corresponding run. You can find an example of the expected mcperf output format [here](#). In the general case, copying from the console should be sufficient to match the required format. But, it is your responsibility to make sure that the format of all your **.txt** files matches the one in the example given above.
Note: Trailing new lines and whitespaces are ignored. You can use either Unix-like line endings (**\n**) or Windows-like line endings (**\r\n**).
- Please follow the instructions stated above. **Divergence from the required format can lead to subtraction of points.**

There are no additional requirements regarding the structure of the other requested files.

4 Part 4

In Part 4 of the project you will co-schedule the batch applications on a single 4-core server running memcached. In contrast to Part 3, the load on the long-running memcached service will now be dynamically varied, such that the number of cores needed by the memcached service to meet the tail latency SLO will range from 1 to 2 cores. Your goal is to design a scheduling policy that grows and shrinks the resource allocation of memcached and opportunistically uses (temporarily) available cores to complete the batch jobs as quickly as possible. Your scheduling policy must guarantee a memcached tail latency SLO of 1ms 95th percentile latency. For this part of the project, you will be using a cluster consisting of 4 nodes: a 2-core VM cluster master, a 4-core high memory VM for the **memcached** server and batch jobs, a 16-core VM for the **mcperf** agent, and a 2-core VM for the **mcperf** measurement machine.

You are required to implement your own controller to launch jobs and dynamically adjust their available resources based on your scheduling policy. In this part of the project, we will not be using Kubernetes because it does not provide an API to change a container's resource allocation during runtime. Instead, you will use Docker to launch containers and run the batch workloads, and to dynamically adjust their resources. For memcached, we provide instructions for installing and running it directly on the VM (rather than in a Docker container) and for using the **taskset** command to dynamically adjust its resources. The reason why we do not use Docker to run memcached in this part of the assignment is that we have observed that memcached's resources are not effectively constrained with **docker --cpuset-cpus**. This occurs due to the fact that most of the processing in memcached is network packet processing, which executes in kernel threads. Your controller should monitor CPU utilization and/or other types of resources and metrics to decide if resources need to be adjusted to meet the SLO. Your controller should make dynamic resource allocation decisions, such that the batch jobs are completed as quickly as possible, while still enforcing memcached's SLO.

For this part of the project you should also use the augmented version of **mcperf**, which is capable of generating random loads on the memcached server, as well as specific load traces. Refer to the instructions provided in Part 3 to install this version.

Implementing the controller and the scheduling policy

We recommend implementing your controller in python and using the [Docker Python SDK](#) to manage containers. Alternatively, you may implement the controller in Go using the [Docker Go SDK](#). You can find [examples of managing containers using the Docker SDK](#), for both Python and Go. If you plan on using such an SDK, you might find it useful to use the shell command **sudo usermod -a -G docker <your-username>**. This will allow you to use the SDK programmatically, without encountering permission errors. You will also be able to run docker commands without using **sudo**.

In addition to *running* containers, you will also need to *update* containers while they are running. Updating a container refers to dynamically adjusting the properties of the container, such as the CPU allocation. You can read more about updating containers in the [Docker update command documentation](#). You can update docker containers using Docker SDK commands. In case you find it helpful, you can also [pause](#) and [unpause](#) containers. This is an option you may explore, but it is not required. Pausing a container has the effect of temporarily stopping the execution of the processes in the container (i.e. releasing CPU resources), while retaining the container's state (i.e. keeping the container's memory resources). Unpausing a container resumes the execution of the

processes in that container.

Your controller should run on the 4-core high memory memcached server and monitor the CPU utilization. The controller should then use the CPU utilization statistics to make dynamic scheduling decisions. You can [monitor CPU utilization on the server](#) by reading and post-processing data from `/proc/stat` files on the VM. There are also language specific options for monitoring metrics, such as [psutil](#) for Python.

In addition to CPU utilization, you can also use other inputs for your scheduling policy if you wish to do so. This is not required, but may let you implement an even better scheduling policy. Make sure that your project report contains explanations of any additional controller inputs you choose to consider in your scheduling policy.

Evaluating the scheduling policy

You will evaluate your scheduling policy with a dynamic **mcperf** load trace we provide (see instructions below). You should use **mcperf** to investigate the performance of your scheduling policy with various load traces (e.g. try different random seeds and time intervals). Experimenting with various load traces will allow you to analyze when and why does your policy perform well and to understand in which scenarios the policy does not adapt appropriately.

Generating the plots

In this part of the project you will be asked to generate some plots which often require you to aggregate data gathered from different VMs. This can be challenging, since you'll need to temporally correlate data across different VMs. A straightforward way to do this is to save the Unix time whenever you log an event, as this time is roughly synchronized across VMs. You can further use other information such as dynamic mcperf's `--qps_interval` or `-t` parameter (see [documentation here](#)). Our dynamic mcperf version should also print the simulation's start and end Unix times in the output logs by default. Another alternative is to use the shell command `date +%s`. These times can then be used when generating the plots to synchronize events that take place on different VMs.

4.1 Setup

4.1.1 Installation

Run the following command to create a kubernetes cluster with 1 master and 3 nodes.

```
$ export KOPS_STATE_STORE=<your-gcp-state-store>
$ PROJECT='gcloud config get-value project'
$ kops create -f part4.yaml
```

You are now ready to deploy the cluster by running:

```
$ kops update cluster --name part4.k8s.local --yes --admin
```

Your cluster should need around 5-10 minutes to be deployed. You can validate the cluster with the command:

```
$ kops validate cluster --wait 10m
```

The command will terminate when your cluster is ready to use. Afterwards you can run:

```
$ kubectl get nodes -o wide
```

to get the status and details of your nodes as follows:

NAME	STATUS	ROLES	AGE	VERSION	INTERNAL-IP	EXTERNAL-IP
client-agent-bf7q	Ready	node	111s	v1.19.7	10.138.0.33	35.230.78.193
client-measure-5v6m	Ready	node	116s	v1.19.7	10.138.0.32	35.227.161.236
master-us-west1-a-kh69	Ready	master	3m23s	v1.19.7	10.138.0.34	35.247.63.197
memcache-server-qmql	Ready	node	111s	v1.19.7	10.138.0.31	34.83.56.78

You will first need to manually install memcached on the **memcache-sever** VM. To do so, you must first use the following commands:

```
$ sudo apt update
$ sudo apt install -y memcached libmemcached-tools
```

To make sure the installation succeeded, run the following command:

```
$ sudo systemctl status memcached
```

You should see an output similar to the one pasted underneath:

```
memcached.service - memcached daemon
  Loaded: loaded (/lib/systemd/system/memcached.service; enabled; vendor preset: enabled)
  Active: active (running) since Thu 2021-04-01 08:21:26 UTC; 10min ago
    Docs: man:memcached(1)
  Main PID: 11796 (memcached)
    Tasks: 10 (limit: 4915)
  CGroup: /system.slice/memcached.service
          -11796 /usr/bin/memcached -m 64 -p 11211 -u memcache -l 127.0.0.1 ...
```

You will need to expose the service to the outside world, and increase its default starting memory. To do so, open memcached's configuration file using the command:

```
$ sudo vim /etc/memcached.conf
```

To update memcached's memory limit, look for the line starting with **-m** and update the value to 1024. Similarly, to expose the memcached server to external requests, locate the line starting with **-l** and replace the localhost address with the internal IP of the **memcache-server** VM. In this file you can also specify the number of memcached threads by introducing a line starting with **-t**, followed by the number of threads. After entering all of the desired changes, save the file, and then execute the next command to restart memcached with the new configuration:

```
$ sudo systemctl restart memcached
```

Running **sudo systemctl status memcached** again should yield an output similar as before, but you should see the updated parameters in the command line. If you completed these steps successfully, memcached should be running and listening for requests on the VMs internal IP on port 11211.

On **client-agent** and **client-measure** machines, install the augmented version of mcperf following the instructions from Part 3.

On the **client-agent** VM, you should then run the following command to launch the **mcperf** memcached client load agent with 16 threads:

```
$ ./mcperf -T 16 -A
```

On the **client-measure** VM, run the following commands to first load the memcached database with key-value pairs and then to query memcached with a dynamic load generator, which will produce a random throughput between 5000 and 100000 queries per second during each interval. The throughput target will change and will be assigned to another QPS value for the next time interval. Note that, in contrast from before, the output appears only at the end of the measurement. In the example below the interval duration is set to 2 seconds, whilst the overall execution time is 10 seconds, this will result in five different QPS intervals:

```
$ ./mcperf -s INTERNAL_MEMCACHED_IP --loadonly
$ ./mcperf -s INTERNAL_MEMCACHED_IP -a INTERNAL_AGENT_IP \
  --noload -T 16 -C 4 -D 4 -Q 1000 -c 4 -t 10 \
  --qps_interval 2 --qps_min 5000 --qps_max 100000
```

The **INTERNAL_MEMCACHED_IP** and **INTERNAL_AGENT_IP** are the internal IPs of the **memcache-sever** and **client-agent** retrieved from the output of `kubectl get nodes -o wide`.

For more information on the dynamic load generator, and the available options it provides, check the guide in the [README.md](#) of the public repository.

Batch jobs can be started using Docker. For instance, one can start the **blackscholes** job on core 0 (`--cpuset-cpus="0"` parameter) and with 2 threads (`-n 2` parameter) using the following command:

```
docker run --cpuset-cpus="0" -d --rm --name parsec \
  anakli/cca:parsec_blackscholes \
  ./run -a run -S parsec -p blackscholes -i native -n 2
```

Feel free to inspect the **YAML** files for the batch jobs, provided in the previous parts of the project, to further understand their command line arguments. You can find the rest of the docker images [here](#).

Make sure to use the native datasets for the jobs and the following image versions:

- blackscholes: `anakli/cca:parsec_blackscholes`
- canneal: `anakli/cca:parsec_canneal`
- dedup: `anakli/cca:parsec_dedup`
- ferret: `anakli/cca:parsec_ferret`
- freqmine: `anakli/cca:parsec_freqmine`
- radix: `anakli/cca:splash2x_radix`
- vips: `anakli/cca:parsec_vips`

IMPORTANT: You must delete your cluster when you are not using it! Otherwise, you will easily use up all of your cloud credits! When you are ready to work on the project again, you can easily re-launch the cluster using the instructions above.

To delete your cluster, use the following command:

```
$ kops delete cluster --name part4.k8s.local --yes
```

4.1.2 Setting resource limits

`taskset` is an essential command used for setting the process CPU affinity. For instance, running `taskset -a -cp 0-2 <pid>` will bind all threads (`-a` switch) of the running process indicated by `<pid>` (`-p` parameter) to the CPUs 0, 1 and 2 (`-c` parameter). One can also use this command when starting up processes. More information on `taskset` can be obtained [here](#).

For Docker, the `--cpuset-cpus` parameter is used to set the cores a container is able to use. This parameter can be set when spinning up a container (e.g. `sudo docker run --cpuset-cpus="0-2" ...`) or updated when a container is already running (e.g. `docker container update --cpuset-cpus="0-2" CONTAINER`).

You are also free to use other methods to dynamically adjust resource allocation for your jobs. This can refer to resources other than CPU cores.

4.2 Questions

Use the report template to answer the questions and submit your results for Part 4 of the project.

4.3 Submission

For part 4 of the project, we expect you to submit:

- The PDF file containing the answers to the posed questions, in the form of the filled project report template.
- The script you used to automate the scheduler.
- All other scripts or files you used, and consider needed/useful for the script above.
- Your measurement output files, **in the format explained below**:
 - Your submission **must** contain the measurements for the results described in your report.
 - In the root of your submission archive, place two directories called `part_4_3_results_group_XXX` and `part_4_4_results_group_XXX`, where `XXX` is your group number represented with **3 digits** (e.g. for group 1, `XXX` equals 001).
 - Each of the directories should have 6 files inside. They **must** be named `jobs_1.txt`, `jobs_2.txt`, `jobs_3.txt` and `mcperf_1.txt`, `mcperf_2.txt`, `mcperf_3.txt`.
 - Each `mcperf_i.txt` file should contain the output of the `mcperf` execution for the corresponding run. You can find an example of the expected `mcperf` output format [here](#). In the general case, copying from the console should be sufficient to match the required format. But, it is your responsibility to make sure that the format of all your `mcperf_i.txt` files matches the one in the example given above.

Note: Trailing new lines and whitespaces are ignored. You can use either Unix-like line endings (`\n`) or Windows-like line endings (`\r\n`).
 - The `jobs_i.txt` files should contain the container execution log for the corresponding run.
 - * Since you are not expected to use Kubernetes for this part, you have to produce a text-based log.

- * We provide a [utility class](#) in Python that does exactly that. Feel free to re-implement this class in any language you decide to use, but **the output must adhere to the format of the provided Python class**.
 - * Each line in the file represents an event. It starts with a date in the ISO format (e.g. `2023-04-12T09:52:37.019688`), followed by the event name (`start`, `end`, `update_cores`, `pause`, `unpause`, or `custom`), and the job name (`memcached`, `blackscholes`, `canneal`, `dedup`, `ferret`, `freqmine`, `radix`, `vips`, `scheduler`).
 - * A `start` event must be followed by two more elements that represent: **1)** the list of CPU cores (`[0, 1, 2, 3]`) the process was assigned at the beginning and **2)** the number of (software) threads it is started with.
 - * An `update_cores` event has an additional argument that represents the new list of assigned cores.
 - * A `custom` event has an arbitrary string (that is *URL-encoded*) as the last parameter. Use this event if you are applying different techniques, that are not supported by the logger, or if you want to add comments to the trace.
 - * Trailing whitespaces and newlines are ignored, you can use either Unix-like line endings (`\n`) or Windows-like line endings (`\r\n`).
 - * The file must start with a `start` event for the `scheduler`, and end with an `end` event for the `scheduler`. These two events should not have a core assignment specified.
 - * Remember that each PARSEC job that you `start` must eventually `end`.
 - * Remember that `memcached` needs a `start` event, but it doesn't necessarily need an `end`. If `memcached` is already running, log the `start memcached` event just after the `start scheduler` event.
 - * Refer to [this file](#) for an example.
- Please follow the instructions stated above. **Divergence from the required format can lead to subtraction of points.**
 - Please make sure your files are complete and that the measurement files match the plots and descriptions used in your project report. **Divergence from these instructions can lead to subtraction of points.**

There are no additional requirements regarding the structure of the other requested files.

5 FAQ

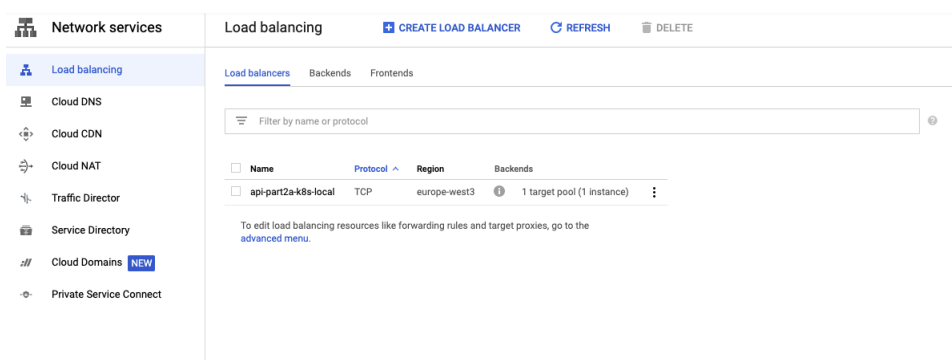
- When running **kops create**:
 - if you get the following error: **failed to create file as already exists: gs://cca-eth-2024-group-XXX-ethzid/part1.k8s.local/config. error: error creating cluster: file already exists**, you need to delete the contents of your Google Cloud storage bucket, the recreate it with the following commands:

```
$ gsutil rm -r gs://cca-eth-2024-group-XXX-ethzid/  
$ gsutil mb gs://cca-eth-2024-group-XXX-ethzid/
```
 - if you get the following error: **Error: error creating cluster: error writing Cluster "part1.k8s.local": error from acl provider "k8s.io/kops/acl/gce": error querying bucket "...": googleapi: Error 404: The requested project was not found., notFound**, make sure you have set the credentials correctly:

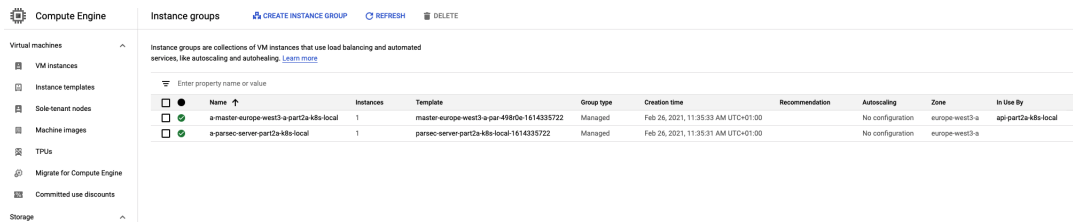
```
$ gcloud auth application-default login
```
- When ssh-ing into a cluster node, if you get an error like **WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!**

```
...  
Offending ED25519 key in /Users/username/.ssh/known_hosts:9  
...  
Host key verification failed
```

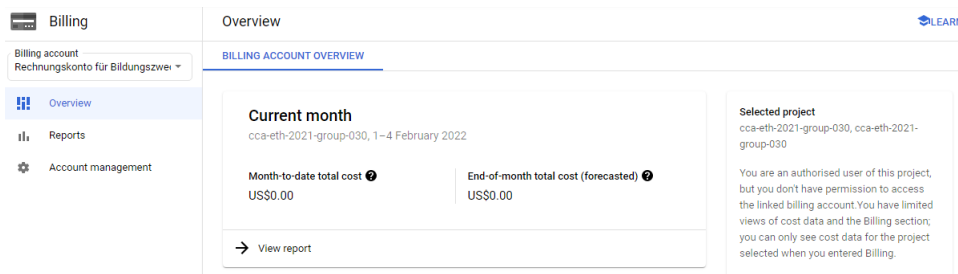
then you need to run **ssh-keygen -R <host>** where **<host>** is the IP address of the server you want to access.
- If **kubectl** commands prompt you for a username and password, or if **kops validate** says **Unauthorized**, first try to re-export the k8s credentials configuration using **kops export kubecfg --admin**. If it still does not work, delete the cluster and recreate it from scratch.
- If for any reason you cannot delete the cluster with the **kops** command do the following:
 - Go to **console.cloud.google.com**
 - Type in the search bar the term “Load balancers”. You should be redirected to a page similar to the one below:



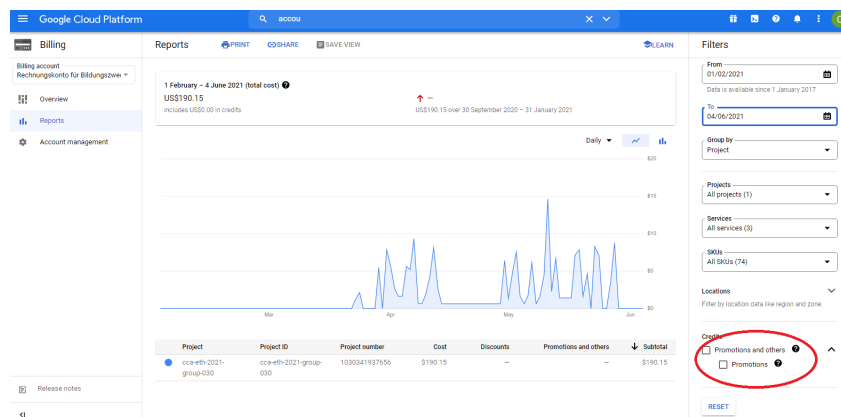
- Select and delete the load balancer.
- Then type in the search bar the term “Instance groups”. You should be redirected to a page similar to the one below:



- Select and delete all the instance groups.
- Delete your Google Cloud storage bucket by typing:
\$ **gsutil rm -r gs://cca-eth-2024-group-XXX-ethzid/**
- Also under ”External IP addresses” check there are no charges for left over static IPs.
- If your Google Cloud Credits are disappearing even though no charges appear on your Billing Overview, make sure you have unselected ”Promotions”
- Go to **console.cloud.google.com**
- Type in the search bar the term “Account Overview”. Select ”Go to linked billing account” if prompted. You should be redirected to a page similar to the one below:



- Click on ”View report”.
- Make sure you unclick ”Promotions and Other” as shown below and select a reasonable To/From time range:



- If you run out of credits for your project, please email cloud-arch-ta@lists.inf.ethz.ch to request additional cloud credits.