ACIT 3495 - Project 2 Technical Report

Group 19

Jaguar Perlas

Roy Ortega

Felix Ruttan

Contents

Overview

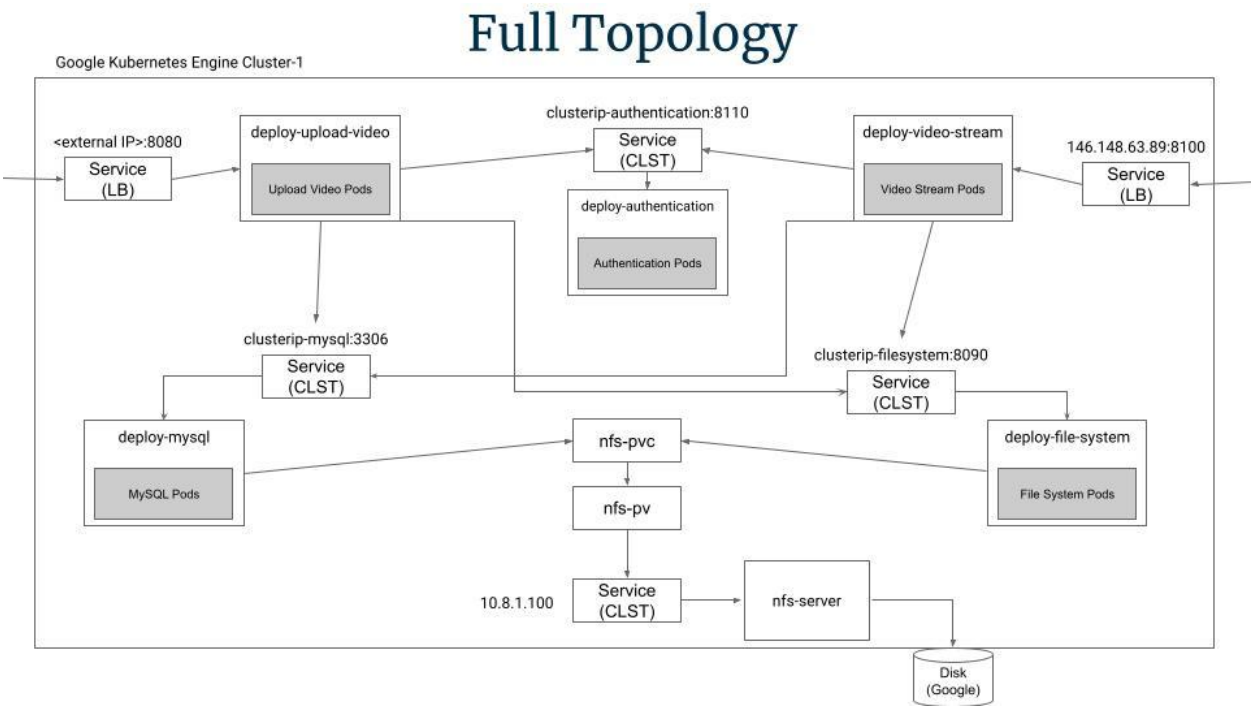
For this Kubernetes project, we are tasked to deploy our system from project 1 and deploy it on Kubernetes and the Cloud. We are also tasked to configure our system so that it can scale horizontally automatically. We have decided to choose Google Cloud as our cloud provider to host our video streaming system, and Google Kubernetes Engine to run our system.

The following steps were taken to accomplish this task:

1. Connect Google Kubernetes Engine (GKE) with our local Kubernetes
2. Upload images to Google Cloud to test connection
3. Write .yml files to build components
4. Test, debug, and reconfigure
5. Implement automatic scaling

Each image (Upload Video, Authentication, Video Stream, MySQL, File System) will be in their own individual deployment to enable our architecture to be decoupled, independent and scalable. There will be no two images in one deployment, such as two pods in one deployment. Each deployment will have their own service (ClusterIP or Load Balancer) to enable communication between deployments.

The following image is the final topology of our system:



Deploying to the Cloud and Kubernetes

Connect Google Kubernetes Engine (GKE) With Our Local Kubernetes

The first step to deploying our system to the Cloud and Kubernetes was to connect GKE with our local Kubernetes. **We used our previously existing empty project and cluster from previous ACIT 3495 labs.**

Thanks to Google Cloud's official documentation, this process was relatively easy. The steps taken to establish connection are:

1. Install the Google Cloud CLI
    a. "Add the gcloud CLI distribution URI as a package source"
        i. *echo "deb [signed-by=/usr/share/keyrings/cloud.google.gpg] https://packages.cloud.google.com/apt cloud-sdk main" | sudo tee -a /etc/apt/sources.list.d/google-cloud-sdk.list*

    b. "Import the Google Cloud public key"
        i. *curl https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key --keyring /usr/share/keyrings/cloud.google.gpg add -*

    c. "Update and install the gcloud CLI"
        i. *sudo apt-get update && sudo apt-get install google-cloud-cli*

2. Initialize gcloud CLI
    a. "Initialize the gcloud CLI"
        i. *gcloud init*

    b. "Accept the option to log in using your Google user account"
        i. *To continue, you must log in. Would you like to log in (Y/n)? Y*

    c. "In your browser, log in to your Google user account and click **Allow** to grant permission to access Google Cloud resources"

    d. "At the command prompt, select a Google Cloud project"
        i. *ACIT3495Ass2*

    e. "If you have the Compute Engine API enabled, *gcloud init* allows you to choose a Compute Engine Zone"
        i. *us-central1-c*

By running the command *gcloud info*, we were able to prove that we have successfully connected GKE with our local Kubernetes shown below:

```
Account: [jagperlas@gmail.com]
Project: [acit3495ass2]

Current Properties:
  [compute]
    zone: [us-central1-c] (property file)
    region: [us-central1] (property file)
```

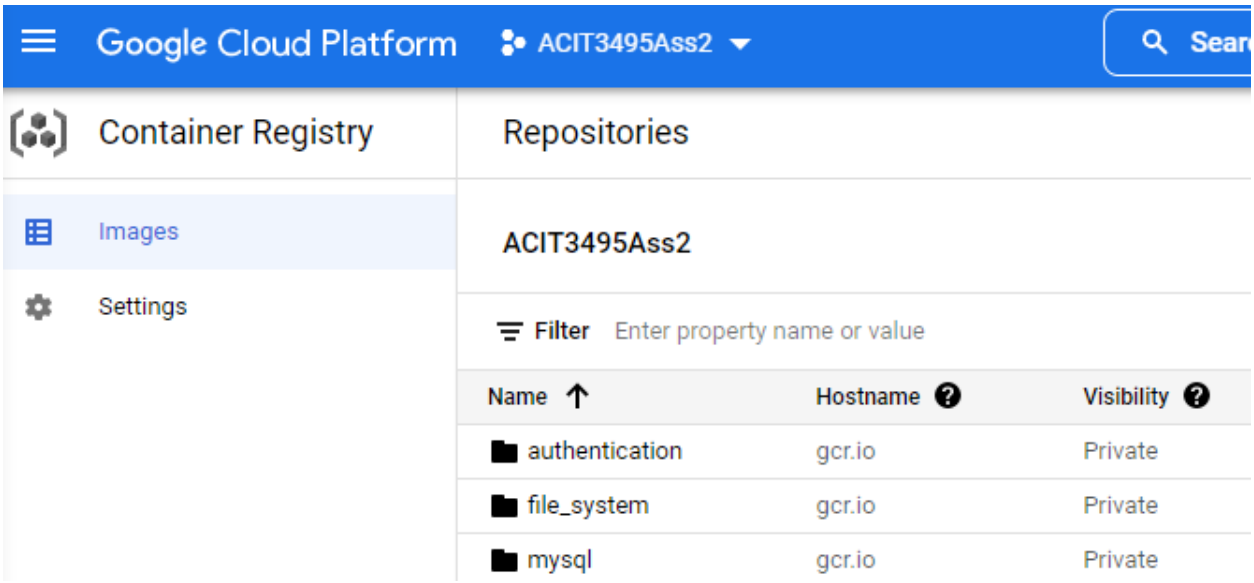
Upload Images to GKE to Test Connection

To further test our connection, we wanted to upload our images as well as test deploying a simple application.

Again, we have followed Google Cloud's official documentation to help us get started. The steps taken to upload an image to Google's Container Registry (GCR) are as follows:

1. Build/pull an image via Docker
    a. *docker build -t mysql:latest.*
2. Tag the Docker image
    a. *docker tag mysql:latest gcr.io/ACIT3495Ass2/mysql*
        i. "gcr.io" is the host name to store images in data centers in the United States.
3. Push the image to Google
    a. *docker push gcr.io/ACIT3495Ass2/mysql*

The results are the following image:



We also wanted to test deploying an image from the GCR. This was easily done with the following command:

kubectl create deployment test --image=gcr.io/acit3495ass2/mysql

With the above command executed, we have successfully deployed an image from GCR as shown below:

```
jperlas@DESKTOP-9QD86NS:~$ kubectl create deployment test --image=gcr.io/acit3495ass2/mysql
deployment.apps/test created
jperlas@DESKTOP-9QD86NS:~$ kubectl get deployments
NAME   READY   UP-TO-DATE   AVAILABLE   AGE
test   1/1     1            1           3s
jperlas@DESKTOP-9QD86NS:~$ kubectl get pods
NAME                    READY   STATUS    RESTARTS   AGE
test-645744f6cc-fscbj   1/1     Running   0          6s
```

Write .yml Files to Build Components

Now that we know we have an established connection to GKE and that we can deploy images from the GCR, we were ready to start building Kubernetes components for our video streaming system.

Planning out our architecture, we have decided to have a deployment .yml file for each image to maintain decoupling, independency, and scalability. We have also decided to have two Load Balancer services, one for upload_video and the other for video_stream, and 3 ClusterIP services, one for Authentication, one for MySQL, and one for file_system.

The reason of having Load Balancer services for upload_video and video_stream is due to those images being client-facing services. Load Balancers in Kubernetes allows us to have an external IP to communicate with the image. To have external traffic into our video steaming system, we need a Load Balancer to accept external traffic.

The reason of having ClusterIPs for Authentication, MySQL, and file_system is due to those images being only internal. Those images have no need of communicating with external traffic, therefore only needing an internal IP and port.

The following are our deployment and service files of our images ***(taken after project completion):***

upload_video

deploy_upload_video.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-upload-video
  labels:
    app: ass1-upload-video
spec:
  selector:
    matchLabels:
      app: ass1-upload-video
  template:
    metadata:
      labels:
        app: ass1-upload-video
    spec:
      containers:
      - name: upload-video
        image: gcr.io/acit3495ass2/upload_video
        ports:
        - containerPort: 8080
        env:
        - name: POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
        resources:
          requests:
            cpu: "250m"
```

svc_upload_video.yml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: lb-upload-video
spec:
  type: LoadBalancer
  ports:
  - port: 8080
    targetPort: 8080
  selector:
    app: ass1-upload-video
```

authentication

deploy_authentication.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-authentication
  labels:
    app: ass1-authentication
spec:
  selector:
    matchLabels:
      app: ass1-authentication
  template:
    metadata:
      labels:
        app: ass1-authentication
    spec:
      containers:
      - name: authentication
        image: gcr.io/acit3495ass2/authentication
        ports:
        - containerPort: 8110
        env:
        - name: POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
        resources:
          requests:
            cpu: "250m"
```

svc_authentication.yml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: clusterip-authentication
spec:
  type: ClusterIP
  ports:
  - port: 8110
  selector:
    app: ass1-authentication
```

MySQL

deploy_mysql.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-mysql
  labels:
    app: ass1-mysql
spec:
  selector:
    matchLabels:
      app: ass1-mysql
  template:
    metadata:
      labels:
        app: ass1-mysql
    spec:
      containers:
      - name: mysql
        image: gcr.io/acit3495ass2/mysql
        ports:
        - containerPort: 3306
        env:
        - name: POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
```

svc_mysql.yml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: clusterip-mysql
  labels:
    app: ass1-mysql
spec:
  type: ClusterIP
  ports:
  - port: 3306
  selector:
    app: ass1-mysql
```

file_system

deploy_file_system.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-file-system
  labels:
    app: ass1-file-system
spec:
  selector:
    matchLabels:
      app: ass1-file-system
  template:
    metadata:
      labels:
        app: ass1-file-system
    spec:
      containers:
      - name: file-system
        image: gcr.io/acit3495ass2/file_system
        ports:
        - containerPort: 8090
        env:
        - name: POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
        volumeMounts:
        - name: nfs
          mountPath: /var/src/app/videos
        resources:
          requests:
            cpu: "250m"
      volumes:
      - name: nfs
        persistentVolumeClaim:
          claimName: nfs-pvc
```

svc_file_system.yml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: clusterip-file-system
spec:
  type: ClusterIP
  ports:
  - port: 8090
  selector:
    app: ass1-file-system
```

video_stream

deploy_video_streaming.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-video-streaming
  labels:
    app: ass1-video-streaming
spec:
  selector:
    matchLabels:
      app: ass1-video-streaming
  template:
    metadata:
      labels:
        app: ass1-video-streaming
    spec:
      containers:
      - name: video-streaming
        image: gcr.io/acit3495ass2/video_streaming
        ports:
        - containerPort: 8100
        env:
        - name: POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
        resources:
          requests:
            cpu: "250m"
```

svc_video_streaming.yml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: lb-video-streaming
spec:
  type: LoadBalancer
  loadBalancerIP: "146.148.63.89"
  ports:
  - port: 8100
    targetPort: 8100
  selector:
    app: ass1-video-streaming
```

Test, Debug, Reconfigure

After writing our .yml files, we've executed each one with the following command:

kubectl apply -f <filename>

Although we've deployed our video streaming system, it was not a fully functional application. There were a few issues we needed to solve regarding our code base. These issues were:

- With scaling in mind, we can not have internal hard coded IP addresses
  - Using hard coded IPs means that replicas are not included in the data flow
- Replicas of user facing pods have no way of telling if the user already logged in from another replica
  - If a user logs in from Replica1 and is load balanced to Replica2, the user will need to log in again.
- Needed a way for replicated video storage and MySQL pods to share one volume
  - Otherwise, replicated video storage pods will have an empty storage
- We needed a way to automatically scale up/down our pods
  - With the above issues solved, we were able to manually scale our pods while maintaining full functionality. Now we must automatically scale our system.
- Long process of pushing a change
  - It was time consuming to individually build, tag, push, and deploy every change made to our application

Addressing Issues (Hard Coded IPs)

In project 1, our system uses static IP addresses for each image and uses hardcoded IPs within API requests. With scaling in mind, we can not have hardcoded IPs. When scaling, each pod has a unique IP address. Our system must send traffic to the appropriate service (ClusterIP) so that the service can load balance traffic evenly between replicated images/pods.

Below is an example of what we've changed:

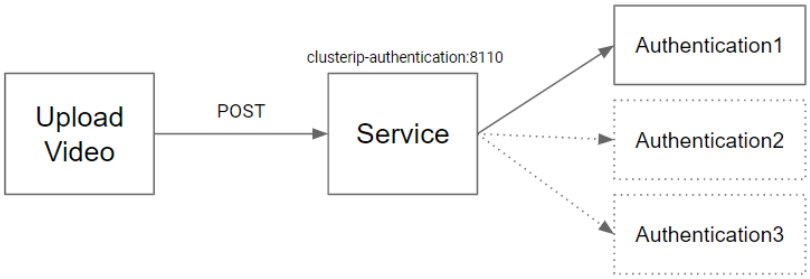
Before :

```
axios.post('http://192.168.100.40:8110/'
```

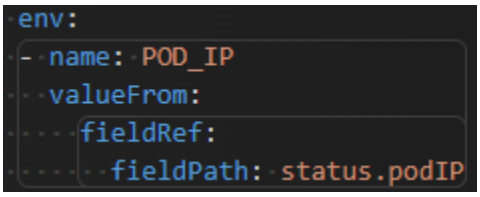
After:

```
axios.post('http://clusterip-authentication:8110/'
```

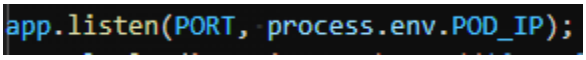
With this change, the POST request will now be sent to *clusterip-authentication:8110*, which is the domain name for the Authentication service. Below is a diagram showing how this works:

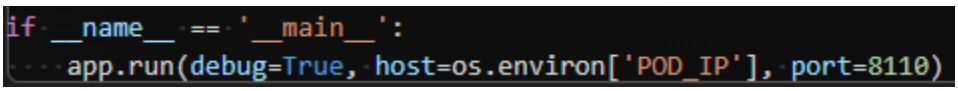


In addition to changing hardcoded IP addresses to domain names, we've added an environmental variable to each image called *POD_IP* and mapped it to the pod's own IP address:



This allows all pods to easily call an environment variable to grab it's own IP address. This is used for pods to listen on the correct IP as shown below for both Node.JS and Flask:





Addressing Issues (Login)

In project 1, our login system uses sessions to store a Boolean value to determine whether the user is authenticated or not. This session would only persist if the connection between the user and the image/pod has not been terminated. In project 1 there were not any replicas of any service, so this was not an issue before.

For this project, project 2, it is an issue since we now have replicas. When the user visits the upload_video service, gets load balanced to upload_video replica 1 and authenticates, the user and upload_video replica 1 establish a session and the Boolean value "True" is stored in that session. The problem is when the user gets redirected back to the Load Balancer and gets load balanced to upload_video replica 2. The session with upload_video replica 1 gets terminated and the Boolean value is deleted, requiring the user to authenticate again.

Our solution is to use cookies to store data in the users browser and not rely on the session. This allows the user to stay logged in even if being load balanced to another replica. We've also changed our code to check for authentication within the cookie and not in the session.

Below is an example of our change:

Before:

```
app.get('/', (req, res) => {
  let sessionData = req.session
  if (sessionData.authenticated != true) {
    res.redirect('/login')
  } else {
    res.sendFile(PATH + "index.html")
  }
});
```

After:

```
app.get('/', (req, res) => {
  let cookieData = req.cookies
  if (cookieData.authenticated != 'true') {
    res.redirect('/login')
  } else {
    res.sendFile(PATH + "index.html")
  }
});
```

Addressing Issues (Shared Volumes)

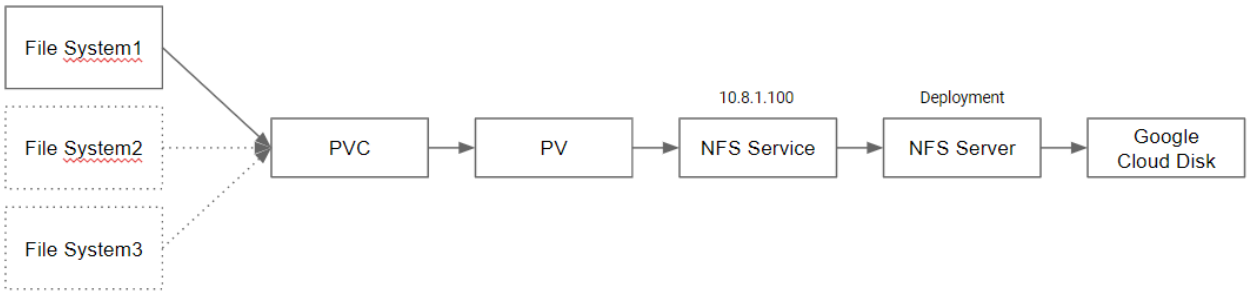
Now that we have replicas of our images, there is an issue when replicating images that use storage to store files, such as MySQL and file_system images.

Replicated MySQL and file_system images will be created with empty storages. This becomes a problem when traffic is load balanced to one of these empty storage images. When traffic is load balanced to one of these empty storage images, an error will be raised.

For example, when a user on video_streaming clicks on a link to watch a video, the GET request to get the video data may be load balanced to a file_system replica that does **not** contain the video, resulting in the user not being able to watch the requested video and raising a 404 error.

The solution to this is to create a shared volume so that both MySQL and file_system replicas can access the same data storage. From research, the best way of doing this is to use a Network File System (NFS) to support ReadWriteMany. The default storage class "standard" does not support ReadWriteMany.

The image below shows a topology of our solution:



In our solution, we have created a PVC, PV, NFS Service, NFS Server, and a disk on Google Cloud.

The screenshots below show the .yml configuration for the new components:

pvc_nfs.yml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: ""
  resources:
    requests:
      storage: 1Gi
  volumeName: nfs
```

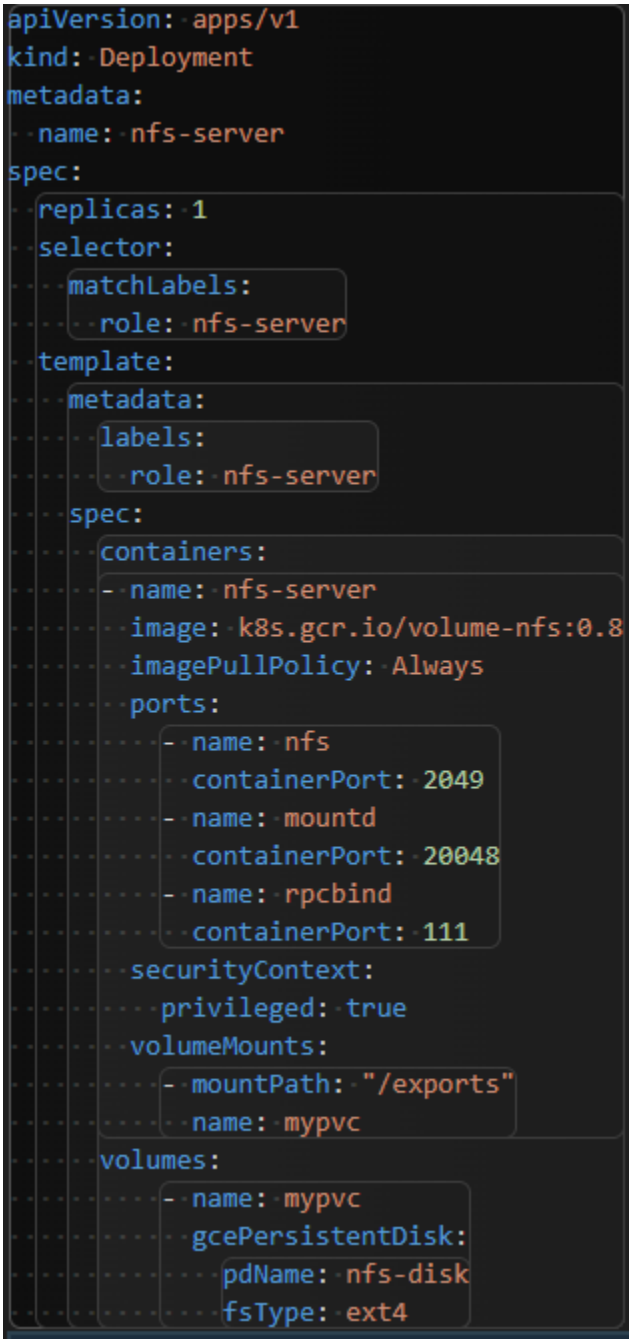
pv_nfs.yml

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: 10.8.1.100
    path: "/"
```

svc_nfs_server.yml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nfs-server
spec:
  clusterIP: 10.8.1.100
  ports:
    - name: nfs
      port: 2049
    - name: mountd
      port: 20048
    - name: rpcbind
      port: 111
  selector:
    role: nfs-server
```

deploy_nfs_server.yml



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nfs-server
spec:
  replicas: 1
  selector:
    matchLabels:
      role: nfs-server
  template:
    metadata:
      labels:
        role: nfs-server
    spec:
      containers:
      - name: nfs-server
        image: k8s.gcr.io/volume-nfs:0.8
        imagePullPolicy: Always
        ports:
          - name: nfs
            containerPort: 2049
          - name: mountd
            containerPort: 20048
          - name: rpcbind
            containerPort: 111
        securityContext:
          privileged: true
        volumeMounts:
          - mountPath: "/exports"
            name: mypvc
      volumes:
          - name: mypvc
            gcePersistentDisk:
                pdName: nfs-disk
                fsType: ext4
```

The information on how to set up an NFS server was referenced here.

Addressing Issues (Long Process of Pushing Change)

With the addition to many .yml files and many image changes, it proved to be time consuming to manually push changes. The manual process goes as follows for an update of the upload_video image:

1. Edit the image
2. *docker build* -t upload_video:latest .
3. *docker tag upload_video*:latest gcr.io/acit3495ass2/upload_video
4. *docker push gcr*.io/acit3495ass2/upload_video
5. *kubectl delete* -f deploy_upload_video.yml
6. *kubectl apply* -f deploy_uplooad_video.yml

On average, this process takes about 1-2 minutes including time to install dependencies. With all the files considered, the process is usually 10 minutes. To be more efficient, we have produced a Shell script to automate the process. This script sets up the infrastructure from a clean state:

deploy.sh

```
kubectl delete --all deployments
kubectl delete --all pv
kubectl delete --all pvc
kubectl delete --all hpa
kubectl delete -f scale_deployments.yml
kubectl delete service clusterip-authentication
kubectl delete service clusterip-file-system
kubectl delete service clusterip-mysql
kubectl delete service lb-upload-video
kubectl delete service lb-video-streaming

docker build -t jperlas/upload_video:latest ./upload_video
docker build -t jperlas/mysql:latest ./mysql_db
docker build -t jperlas/authentication:latest ./authentication
docker build -t jperlas/video_streaming:latest ./video_streaming
docker build -t jperlas/file_system:latest ./file_system

docker tag jperlas/upload_video:latest gcr.io/acit3495ass2/upload_video
docker tag jperlas/mysql:latest gcr.io/acit3495ass2/mysql
docker tag jperlas/authentication:latest gcr.io/acit3495ass2/authentication
docker tag jperlas/video_streaming:latest gcr.io/acit3495ass2/video_streaming
docker tag jperlas/file_system:latest gcr.io/acit3495ass2/file_system

docker push gcr.io/acit3495ass2/upload_video
docker push gcr.io/acit3495ass2/mysql
docker push gcr.io/acit3495ass2/authentication
docker push gcr.io/acit3495ass2/video_streaming
docker push gcr.io/acit3495ass2/file_system

kubectl apply -f deploy_nfs_server.yml
kubectl apply -f svc_nfs_server.yml
kubectl apply -f pv_nfs.yml
kubectl apply -f pvc_nfs.yml

kubectl apply -f svc_mysql.yml
kubectl apply -f svc_upload_video.yml
kubectl apply -f svc_authentication.yml
kubectl apply -f svc_video_streaming.yml
kubectl apply -f svc_file_system.yml

kubectl apply -f deploy_mysql.yml
kubectl apply -f deploy_upload_video.yml
kubectl apply -f deploy_authentication.yml
kubectl apply -f deploy_video_streaming.yml
kubectl apply -f deploy_file_system.yml

kubectl apply -f scale_deployments.yml
```

Implement Automated Scaling

With the issues fixed above, we were able to scale our application successfully manually by using the following command (example for upload_video):

kubectl scale deployment deploy_upload_video --replicas 2

```
deploy-upload-video      2/2     2            2
```

```
deploy-upload-video-799c5589c7-kzxwt      1/1     Running
deploy-upload-video-799c5589c7-sj7sd      1/1     Running
```

The task given for this project is to scale up/down automatically horizontally our system. With a bit of research, this is done possible by HorizontalPodAutoScaler (HPA).

From the official Kubernetes website: "*a HorizontalPodAutoscaler automatically updates a workload resource (such as a Deployment or StatefulSet), with the aim of automatically scaling the workload to match demand".*

Following the GKE documentation as guide to implement HPA, we were able to write a single .yml file with multiple components:

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: scale-file-system
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: deploy-file-system
  minReplicas: 2
  maxReplicas: 10
  targetCPUUtilizationPercentage: 10
```

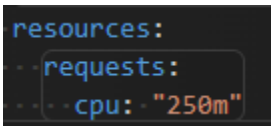
```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: scale-upload-video
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: deploy-upload-video
  minReplicas: 2
  maxReplicas: 10
  targetCPUUtilizationPercentage: 10
```

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: scale-authentication
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: deploy-authentication
  minReplicas: 2
  maxReplicas: 10
  targetCPUUtilizationPercentage: 10
```

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: scale-video-stream
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: deploy-video-streaming
  minReplicas: 2
  maxReplicas: 10
  targetCPUUtilizationPercentage: 10
```

Each image starts with a minimum of 2 replicas. Once reaching a CPU utilization metric of 10%, more replicas will be created up to a max of 10. It Is also stated that if only using CPU as a metric, no other installation is needed for the infrastructure.

In addition to writing the .yml file, we've also had to specify requests for CPU to autoscale based on CPU utilization. An example is shown below, added to the end of the *deploy_upload_video.yml* file:



Closing / Further Issues

Overall, this project was a challenge. Yet, it encouraged us to learn with intent, read documentation, and get a learning experience by troubleshooting issues with our build. There were a few issues that were left unsolved, unfortunately.

These issues are the following:

- HorizontalPodAutoScaling is buggy
    - Sometimes doesn't grab CPU metrics from pods
    - Slow to scale down
- Deleting components via "kubectl delete" sometimes gets stuck
    - Had to manually delete via GUI in Google
- Difficulties in trying to scale MySQL
    - Issues with MySQL requiring the folder in the NFS to be empty
    - Issues with locks when multiple MySQL instances are pointing towards the same folder
    - Not able to finish

Other than MySQL not being able to scale, this project is complete.

To run our system, execute the command in the folder:

./deploy.sh