Cloud & ML : Assignment 5

Shiv Ratan Sinha – `srs9969, N16386999`

April 30, 2022

Objective

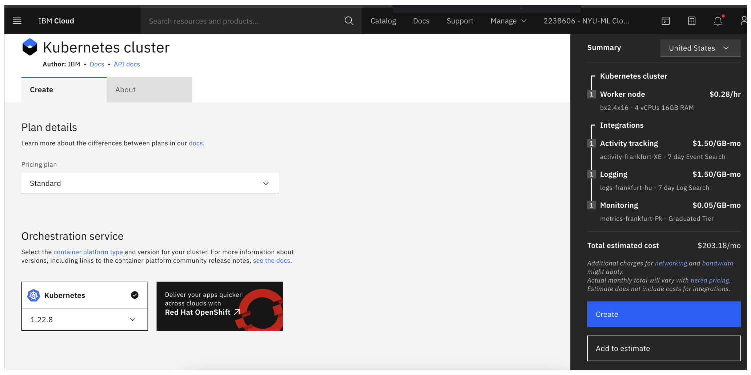
This report summarizes the development of container and Kubernetes artifacts done to perform DL training and DL inference by hosting in IBM Kubernetes cluster.

Cluster creation

A Kubernetes (K8s) cluster is a grouping of nodes. It is used to run containerized apps in an efficient, automated, distributed, and scalable manner. Kubernetes clusters allow software developers like us to orchestrate and monitor containers across multiple physical, virtual, and cloud servers. The nodes in the cluster can either be virtual machines or bare metal machines.

Steps

1. I went to kubernetes service in IBM dashboard and chose create cluster.



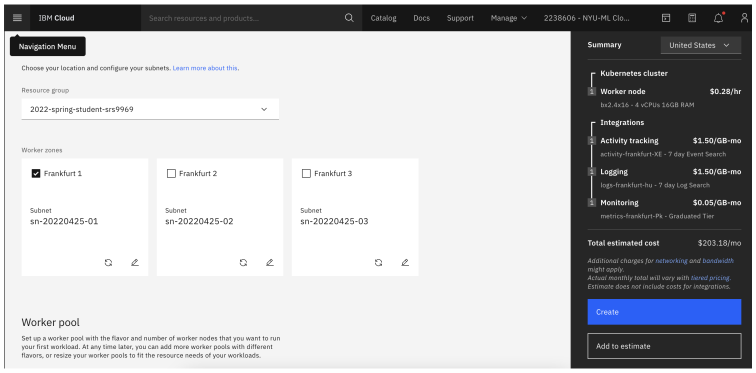
Kubernetes cluster
Author: IBM • Docs • API docs

Plan details
Learn more about the differences between plans in our docs.
Pricing plan: Standard

Orchestration service
Select the container platform type and version for your cluster. For more information about versions, including links to the container platform community release notes, see the docs.
Kubernetes ✓ 1.22.8
Deliver your apps quicker across clouds with Red Hat OpenShift

Summary — United States
Kubernetes cluster
1 Worker node $0.28/hr
bx2.4x16 - 4 vCPUs 16GB RAM
Integrations
1 Activity tracking $1.50/GB-mo
activity-frankfurt-XE - 7 day Event Search
1 Logging $1.50/GB-mo
logs-frankfurt-hu - 7 day Log Search
1 Monitoring $0.05/GB-mo
metrics-frankfurt-Pk - Graduated Tier
Total estimated cost $203.18/mo
Additional charges for networking and bandwidth might apply.
Actual monthly total will vary with tiered pricing.
Estimate does not include costs for integrations.

Figure 1: Cluster creation begin



Choose your location and configure your subnets. Learn more about this.

Resource group: 2022-spring-student-srs9969

Worker zones

[✓] Frankfurt 1	Frankfurt 2	Frankfurt 3
Subnet sn-20220425-01	Subnet sn-20220425-02	Subnet sn-20220425-03

Worker pool

Set up a worker pool with the flavor and number of worker nodes that you want to run your first workload. At any time later, you can add more worker pools with different flavors, or resize your worker pools to fit the resource needs of your workloads.

Summary — United States

Kubernetes cluster
1 Worker node $0.28/hr
bx2.4x16 - 4 vCPUs 16GB RAM

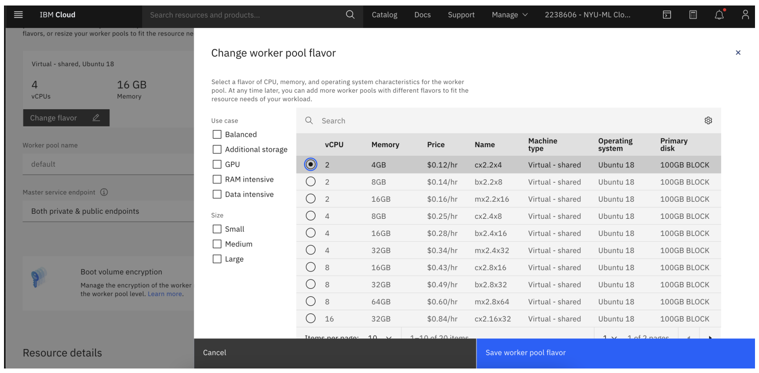
Integrations
1 Activity tracking $1.50/GB-mo
activity-frankfurt-XE - 7 day Event Search
1 Logging $1.50/GB-mo
logs-frankfurt-hu - 7 day Log Search
1 Monitoring $0.05/GB-mo
metrics-frankfurt-Pk - Graduated Tier

Total estimated cost $203.18/mo

Additional charges for networking and bandwidth might apply.
Actual monthly total will vary with tiered pricing.
Estimate does not include costs for integrations.

Figure 2: Region selection

2. I chose the Frankfurt area, faced lot of issues during creation with us-east.
3. In there I chose standard plan, kubernetes or orchestration.
4. In there I chose worker pool of 2 vCPUs and 4gb memory.
5. I switched off the integrations as those were not needed.



flavors, or resize your worker pools to fit the resource ne

Virtual - shared, Ubuntu 18
4 vCPUs 16 GB Memory

Worker pool name: default

Master service endpoint: Both private & public endpoints

Boot volume encryption
Manage the encryption of the worker the worker pool level. Learn more.

Resource details

Change worker pool flavor

Select a flavor of CPU, memory, and operating system characteristics for the worker pool. At any time later, you can add more worker pools with different flavors to fit the resource needs of your workload.

Use case: Balanced, Additional storage, GPU, RAM intensive, Data intensive

Size: Small, Medium, Large

	vCPU	Memory	Price	Name	Machine type	Operating system	Primary disk
●	2	4GB	$0.12/hr	cx2.2x4	Virtual - shared	Ubuntu 18	100GB BLOCK
○	2	8GB	$0.14/hr	bx2.2x8	Virtual - shared	Ubuntu 18	100GB BLOCK
○	2	16GB	$0.16/hr	mx2.2x16	Virtual - shared	Ubuntu 18	100GB BLOCK
○	4	8GB	$0.25/hr	cx2.4x8	Virtual - shared	Ubuntu 18	100GB BLOCK
○	4	16GB	$0.28/hr	bx2.4x16	Virtual - shared	Ubuntu 18	100GB BLOCK
○	4	32GB	$0.34/hr	mx2.4x32	Virtual - shared	Ubuntu 18	100GB BLOCK
○	8	16GB	$0.43/hr	cx2.8x16	Virtual - shared	Ubuntu 18	100GB BLOCK
○	8	32GB	$0.49/hr	bx2.8x32	Virtual - shared	Ubuntu 18	100GB BLOCK
○	8	64GB	$0.60/hr	mx2.8x64	Virtual - shared	Ubuntu 18	100GB BLOCK
○	16	32GB	$0.84/hr	cx2.16x32	Virtual - shared	Ubuntu 18	100GB BLOCK

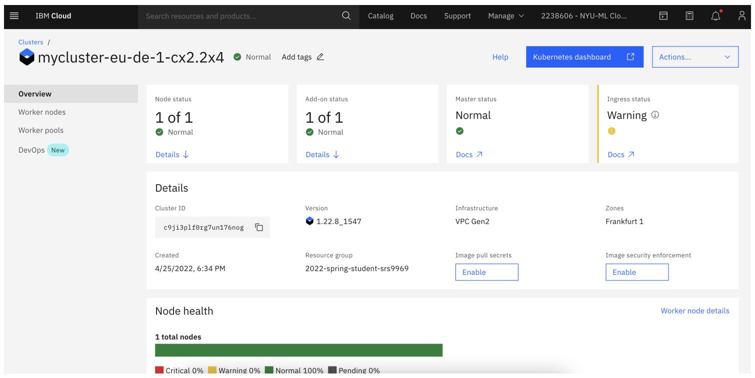
Figure 3: Type of worker pool

Public gateway

To be able to access the containers from outside, I created Public gateway. It enables a subnet and all its attached virtual server instances to connect to the outside world. By default they are private.

Steps

1. For this I went to public gateway in IBM dashboard and created a new one.



mycluster-eu-de-1-cx2.2x4 Normal Add tags

Node status: 1 of 1 Normal
Add-on status: 1 of 1 Normal
Master status: Normal
Ingress status: Warning

Details

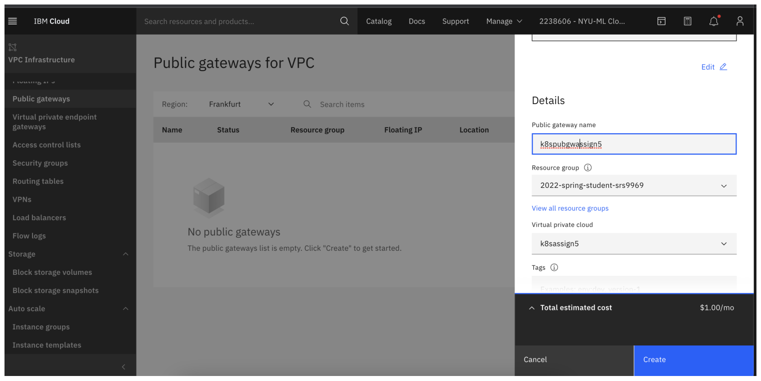
Cluster ID	Version	Infrastructure	Zones
c9ji3plf0rg7un176nog	1.22.8_1547	VPC Gen2	Frankfurt 1
Created	Resource group	Image pull secrets	Image security enforcement
4/25/2022, 6:34 PM	2022-spring-student-srs9969	Enable	Enable

Node health

1 total nodes

🟥 Critical 0% 🟨 Warning 0% 🟩 Normal 100% ⬛ Pending 0%

Figure 4: Final cluster creation



Public gateways for VPC

Region: Frankfurt

No public gateways

The public gateways list is empty. Click "Create" to get started.

Details

Public gateway name: k8spubgwassign5

Resource group: 2022-spring-student-srs9969

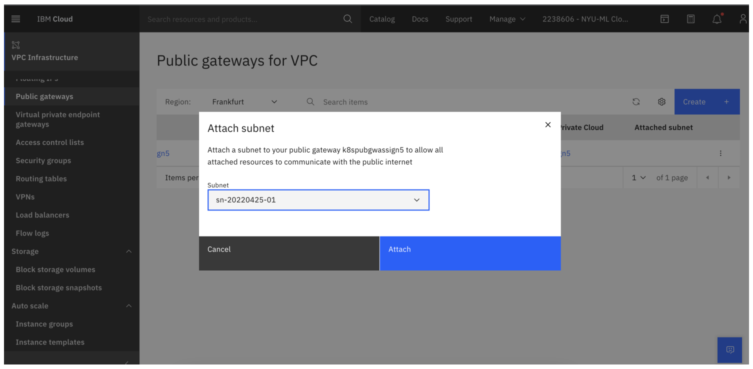
Virtual private cloud: k8sassign5

Tags

Total estimated cost $1.00/mo

Figure 5: Gateway creation 1

2. Here I attached the gateway to my existing subnet which was in the kubernetes cluster that I had created above.



Public gateways for VPC

Attach subnet

Attach a subnet to your public gateway k8spubgwassign5 to allow all attached resources to communicate with the public internet

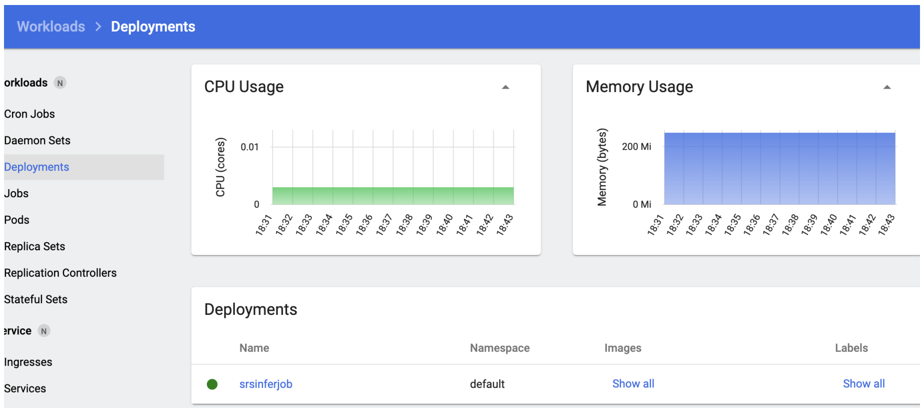
Subnet: sn-20220425-01

Figure 6: Gateway creation 2

Important concepts and exploration

Kubernetes Deployment

A deployment consists of dozens of identical Pods that share no unique identifiers. Deployments run multiple copies of your application and replace any instances that fail or become unresponsive automatically. Deployments facilitate the availability of one or more instances of your application to service user requests in this way. Kubernetes Deployment Controller is responsible for managing deployments. Every deployment makes use of a Pod template that specifies its Pods. As defined by the Pod specification, every Pod needs to have specific properties such as how it operates, what applications are running inside it, what volumes it mounts, its labels, and more. **For training purposes and above reasons I used this controller.**



Deployments

	Name	Namespace	Images	Labels
●	srsinferjob	default	Show all	Show all

Kubernetes Pods

Kubernetes creates and manages pods, which are the smallest deployable units of computing. The pod consists of a group of one or more containers, each with shared storage and network resources, the containers themselves having a specification for the manner in which they should run. There is always a shared context for the Pod's contents, including co-location and co-scheduling. In addition to representing an application-specific "logical host," a pod contains applications which are relatively tightly coupled. An application that runs on the same physical or virtual machine as a cloud application is analogous to one that runs on the same logical server in an on-premises environment.

Kubernetes Service

A Kubernetes Service is an abstraction that describes a logical collection of Pods along with the methods by which they can be accessed. This is useful in microservices architecture as well. For a set of Pods, Kubernetes provides a single DNS name and individual IP addresses, and can load-balance among them.

Kubernetes Service LoadBalancer

This type of service in k8S is used to expose the service externally using a cloud provider's load balancer.

Kubernetes Job

When we create a Job, we create one or more Pods, and we continue to retry the execution of the Pods until a specified number of them successfully complete. Once a pod has completed successfully, it is recorded in the Job. A task (a job) is completed when a specified number of successful completions is reached. Suspended jobs will have their active Pods deleted until they are resumed. **For inference purposes and above reasons this controller was used** to keep it running and balance load using LB.

Persistent Volume

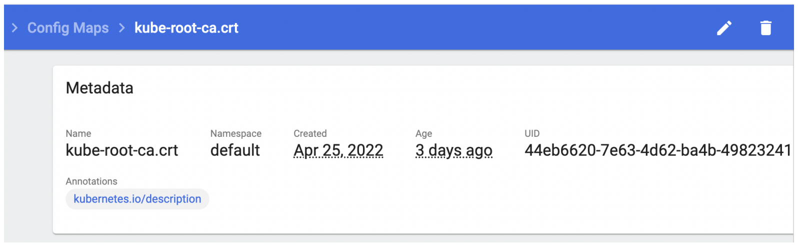
PVs are virtual storage instances added as volumes to a cluster. The Persistent volume points to a physical storage device in your IBM Cloud infrastructure account and abstracts the API that is used to communicate with the storage device.

Persistent Volume Claim

It is the request to provision persistent storage with a specific type and configuration. Kubernetes storage classes are used to specify the persistent storage type that one needs. Based on the configuration that is defined in the storage class, the physical storage device is reserved and provisioned into your IBM Cloud infrastructure account.

Kubernetes root CA certificate

In a deployed Kubernetes cluster, the root CA certficate signs all the other serving and client certificates used by various components for various purposes. This root CA certificate may need to be updated for security or administrative reasons while the cluster is still running.



Metadata

Name	Namespace	Created	Age	UID
kube-root-ca.crt	default	Apr 25, 2022	3 days ago	44eb6620-7e63-4d62-ba4b-49823241

Annotations

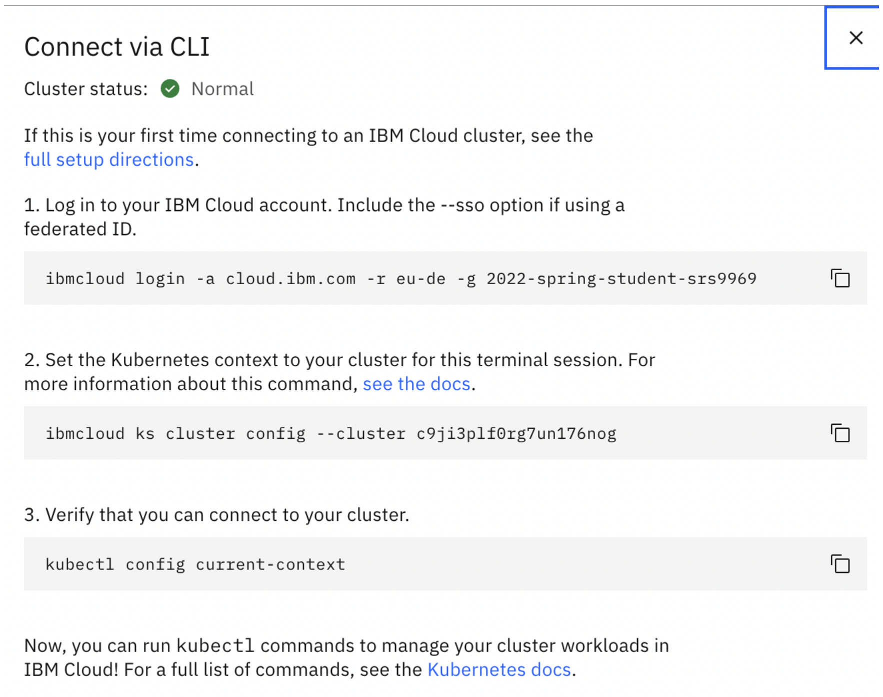
kubernetes.io/description

Figure 7: root ca.crt file

The real Hands-on

Kubectl

This is a commandline tool to run commands and control the kubernetes cluster. I installed it based on the steps taught in lecture. Then I used the below shown steps to hit commands to the k8s cluster using kubectl.



Connect via CLI

Cluster status: Normal

If this is your first time connecting to an IBM Cloud cluster, see the full setup directions.

1. Log in to your IBM Cloud account. Include the --sso option if using a federated ID.

```
ibmcloud login -a cloud.ibm.com -r eu-de -g 2022-spring-student-srs9969
```

2. Set the Kubernetes context to your cluster for this terminal session. For more information about this command, see the docs.

```
ibmcloud ks cluster config --cluster c9ji3plf0rg7un176nog
```

3. Verify that you can connect to your cluster.

```
kubectl config current-context
```

Now, you can run `kubectl` commands to manage your cluster workloads in IBM Cloud! For a full list of commands, see the Kubernetes docs.

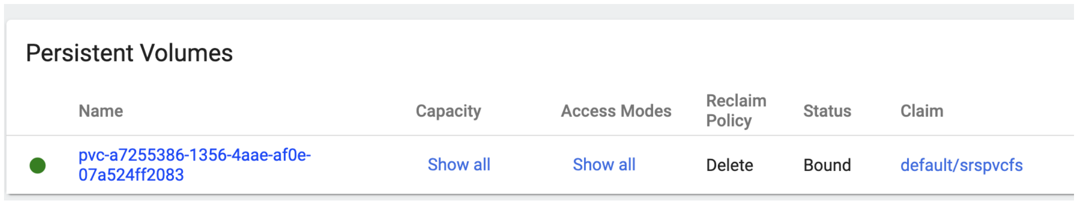
Figure 8: Steps to connect to cluster using IBM cloud

PVC

Persistent volume is needed to store the trained model during the training phase and make use of the model in the inference phase. The model can be used if it has been persisted through the training job.
Therefore, below I have created a PVC which helps me get a PV.

I used the *preNpost/pvc.yaml* for the purpose.

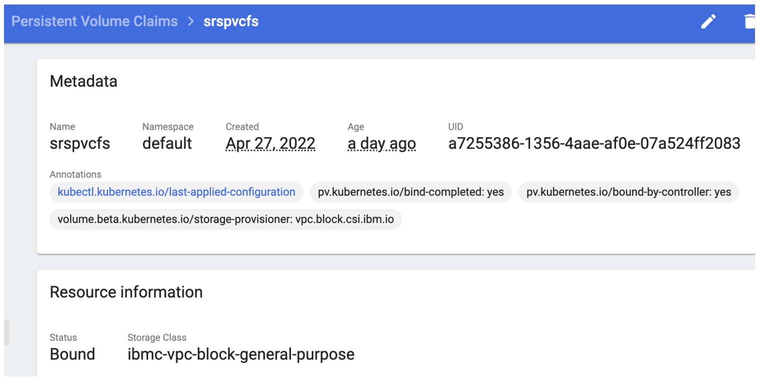
Then the command **kubectl apply -f pvc.yaml** was used to provision the PV.



Persistent Volumes

	Name	Capacity	Access Modes	Reclaim Policy	Status	Claim
●	pvc-a7255386-1356-4aae-af0e-07a524ff2083	Show all	Show all	Delete	Bound	default/srspvcfs

Figure 9: PV



Persistent Volume Claims > srspvcfs

Metadata

Name	Namespace	Created	Age	UID
srspvcfs	default	Apr 27, 2022	a day ago	a7255386-1356-4aae-af0e-07a524ff2083

Annotations

kubectl.kubernetes.io/last-applied-configuration
pv.kubernetes.io/bind-completed: yes
pv.kubernetes.io/bound-by-controller: yes
volume.beta.kubernetes.io/storage-provisioner: vpc.block.csi.ibm.io

Resource information

Status	Storage Class
Bound	ibmc-vpc-block-general-purpose

Figure 10: PVC

Training

Here the objective was to create a dockerfile which downloads our custom image from the docker hub repository and then then runs the code for training on the IBM k8S cluster. I have used the MNIST dataset to train and reused our previous assignment works based on the Professor's suggestion.

1. I created a dockerfile (*training/Dockerfile*)

2. I used the MNIST training/mnistscript.py file

3. The dockerfile runs the training on the MNIST dataset for 1 epoch.

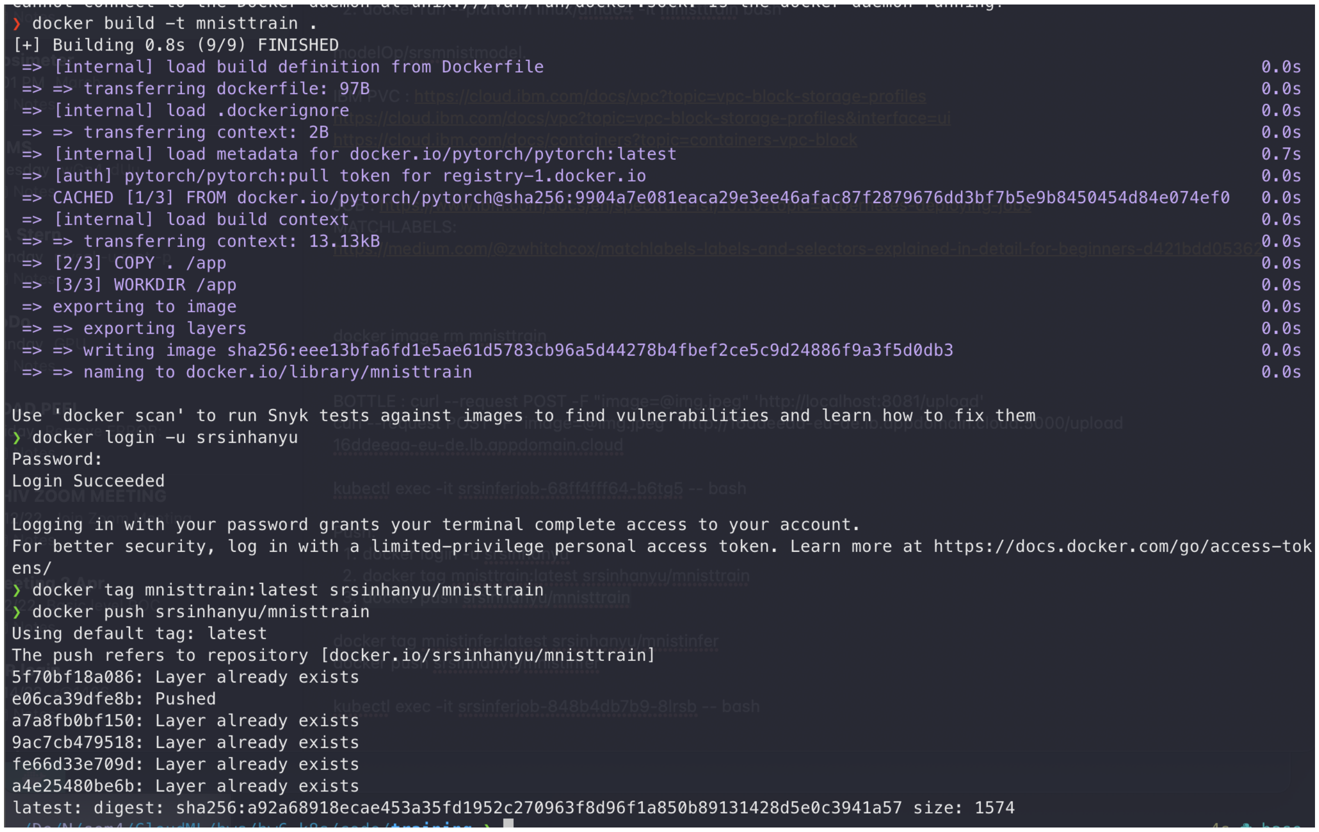
4. The image is built locally and then pushed to the docker hub registry under *srsinhanyu*.

Train Job

1. I used the *preNpost/train.yaml* for the purpose.

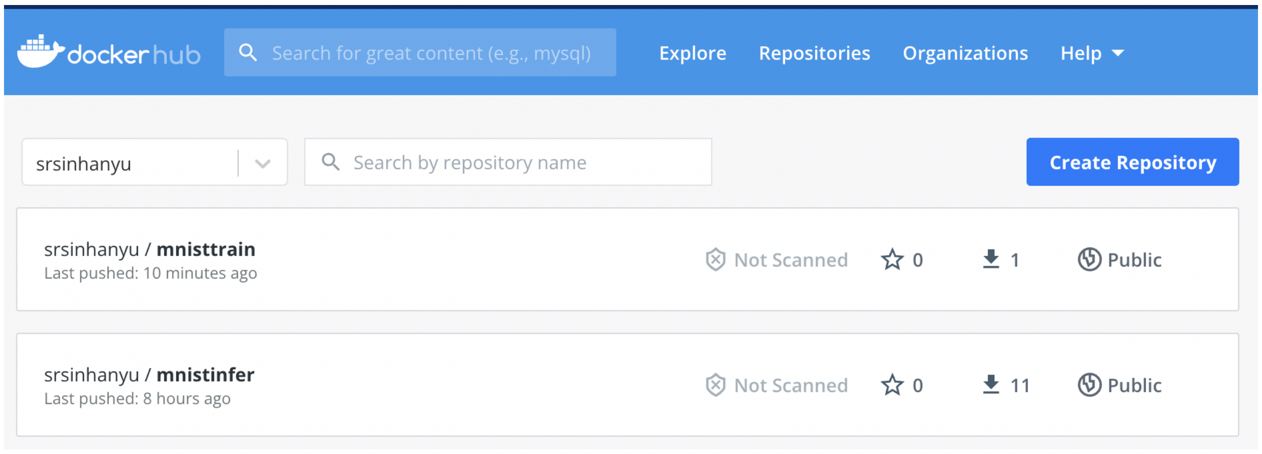
2. Then the command **kubectl apply -f train.yaml** was used to provision the PV.

3. This created a Job in the k8S cluster.



```
❯ docker build -t mnisttrain .
[+] Building 0.8s (9/9) FINISHED
 => [internal] load build definition from Dockerfile                                                          0.0s
 => => transferring dockerfile: 97B                                                                           0.0s
 => [internal] load .dockerignore                                                                             0.0s
 => => transferring context: 2B                                                                               0.0s
 => [internal] load metadata for docker.io/pytorch/pytorch:latest                                             0.7s
 => [auth] pytorch/pytorch:pull token for registry-1.docker.io                                                0.0s
 => CACHED [1/3] FROM docker.io/pytorch/pytorch@sha256:9904a7e081eaca29e3ee46afac87f2879676dd3bf7b5e9b8450454d84e074ef0  0.0s
 => [internal] load build context                                                                             0.0s
 => => transferring context: 13.13kB                                                                          0.0s
 => [2/3] COPY . /app                                                                                         0.0s
 => [3/3] WORKDIR /app                                                                                        0.0s
 => exporting to image                                                                                        0.0s
 => => exporting layers                                                                                       0.0s
 => => writing image sha256:eee13bfa6fd1e5ae61d5783cb96a5d44278b4fbef2ce5c9d24886f9a3f5d0db3                  0.0s
 => => naming to docker.io/library/mnisttrain                                                                 0.0s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
❯ docker login -u srsinhanyu
Password:
Login Succeeded

Logging in with your password grants your terminal complete access to your account.
For better security, log in with a limited-privilege personal access token. Learn more at https://docs.docker.com/go/access-tok
ens/
❯ docker tag mnisttrain:latest srsinhanyu/mnisttrain
❯ docker push srsinhanyu/mnisttrain
Using default tag: latest
The push refers to repository [docker.io/srsinhanyu/mnisttrain]
5f70bf18a086: Layer already exists
e06ca39dfe8b: Pushed
a7a8fb0bf150: Layer already exists
9ac7cb479518: Layer already exists
fe66d33e709d: Layer already exists
a4e25480be6b: Layer already exists
latest: digest: sha256:a92a68918ecae453a35fd1952c270963f8d96f1a850b89131428d5e0c3941a57 size: 1574
```

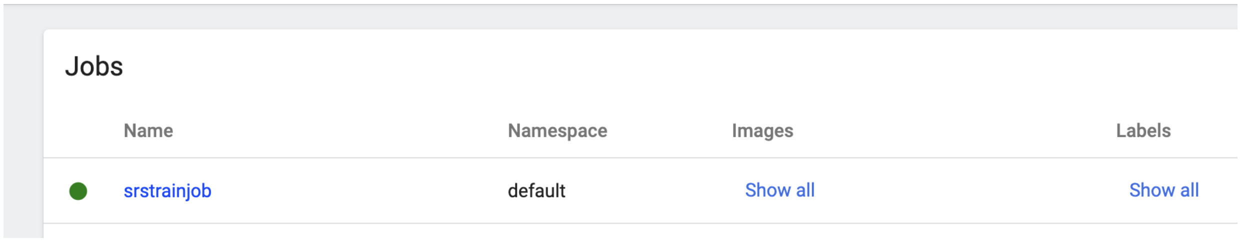
Figure 11: Steps to push the train image



srsinhanyu / **mnisttrain**
Last pushed: 10 minutes ago   Not Scanned   0   1   Public

srsinhanyu / **mnistinfer**
Last pushed: 8 hours ago   Not Scanned   0   11   Public

Figure 12: Train and Inference image on docker hub

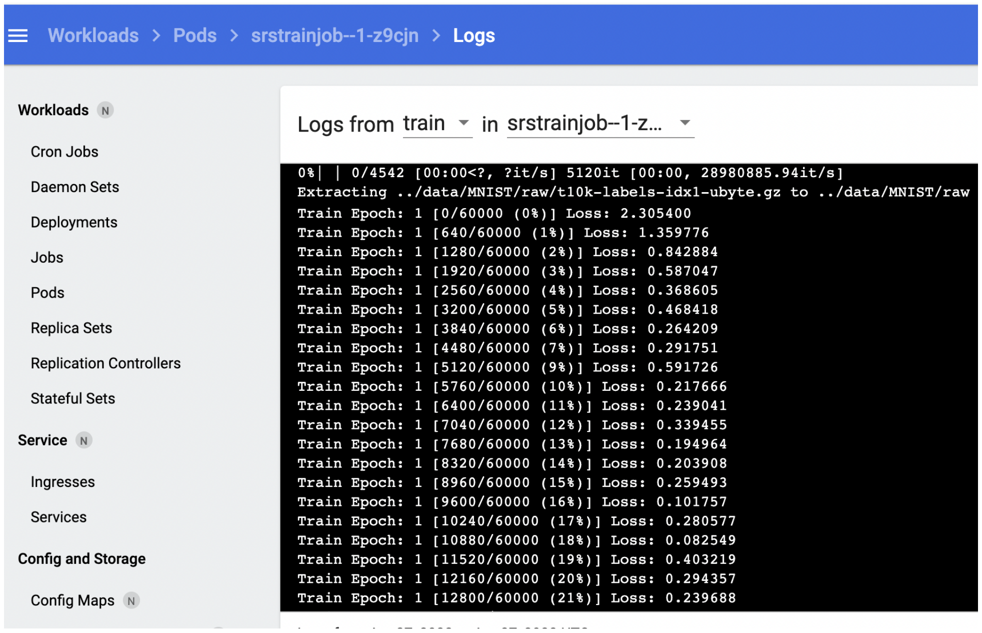


Jobs

	Name	Namespace	Images	Labels
●	srstrainjob	default	Show all	Show all

Figure 13: Job in k8s

4. A single container is created which pulls the *srsinhanyu/mnisttrain* image from the docker hub and trains the model for 1 epoch.



```
0%|  | 0/4542 [00:00<?, ?it/s] 5120it [00:00, 28980885.94it/s]
Extracting ../data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ../data/MNIST/raw
Train Epoch: 1 [0/60000 (0%)] Loss: 2.305400
Train Epoch: 1 [640/60000 (1%)] Loss: 1.359776
Train Epoch: 1 [1280/60000 (2%)] Loss: 0.842884
Train Epoch: 1 [1920/60000 (3%)] Loss: 0.587047
Train Epoch: 1 [2560/60000 (4%)] Loss: 0.368605
Train Epoch: 1 [3200/60000 (5%)] Loss: 0.468418
Train Epoch: 1 [3840/60000 (6%)] Loss: 0.264209
Train Epoch: 1 [4480/60000 (7%)] Loss: 0.291751
Train Epoch: 1 [5120/60000 (9%)] Loss: 0.591726
Train Epoch: 1 [5760/60000 (10%)] Loss: 0.217666
Train Epoch: 1 [6400/60000 (11%)] Loss: 0.239041
Train Epoch: 1 [7040/60000 (12%)] Loss: 0.339455
Train Epoch: 1 [7680/60000 (13%)] Loss: 0.194964
Train Epoch: 1 [8320/60000 (14%)] Loss: 0.203908
Train Epoch: 1 [8960/60000 (15%)] Loss: 0.259493
Train Epoch: 1 [9600/60000 (16%)] Loss: 0.101757
Train Epoch: 1 [10240/60000 (17%)] Loss: 0.280577
Train Epoch: 1 [10880/60000 (18%)] Loss: 0.082549
Train Epoch: 1 [11520/60000 (19%)] Loss: 0.403219
Train Epoch: 1 [12160/60000 (20%)] Loss: 0.294357
Train Epoch: 1 [12800/60000 (21%)] Loss: 0.239688
```

Figure 14: Training logs from container

5. once the job is done, the container dies.

6. Now the trained model was saved in the '/mnt' folder of the container, which had the PV mounted in it.

7. Therefore, the model sustains the death of the container.

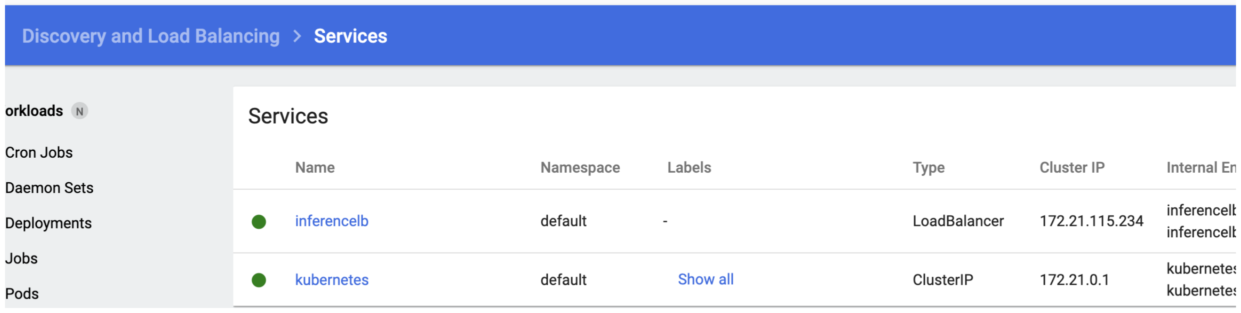
Inference

There are 2 aspects in this, one is creating a service, which listens the a pod based on the selector, and another is creating a container with the label attached to it, which is basically the selector for the service.

Service

As described above the service is used to provide a way to access the network service running on containers.

- I first created a Loadbalancer type of service using the *preNpost/servicelb.yaml* file.
- To request it, I fired the command **kubectl apply -f servicelb.yaml**
- This created the service with name inferencelb which could be seen in k8s IBM dashboard.



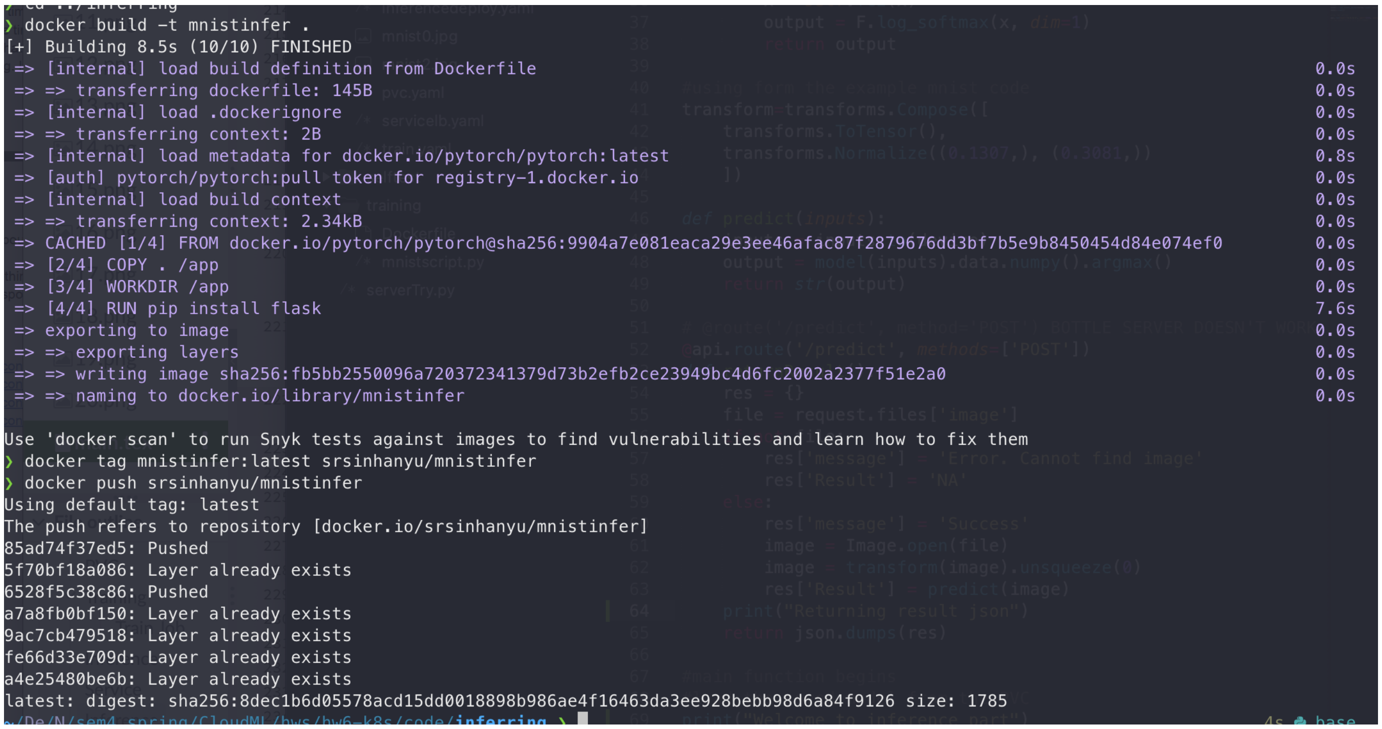
	Name	Namespace	Labels	Type	Cluster IP	Internal En
●	inferencelb	default	-	LoadBalancer	172.21.115.234	inferencelt inferencelt
●	kubernetes	default	Show all	ClusterIP	172.21.0.1	kubernetes kubernetes

Figure 15: Train image on docker hub

Inference-Image creation

Here the image to be used for hosting the server was created.

1. I created a dockerfile (*inferring/Dockerfile*)
2. I used the inferring/inferring.py file which has the code for creating and hosting the Flask server.
3. The dockerfile is simple with just a base image with Flask installed in it and python command to run the server.
4. The image is built locally and then pushed to the docker hub registry under *srsinhanyu*.



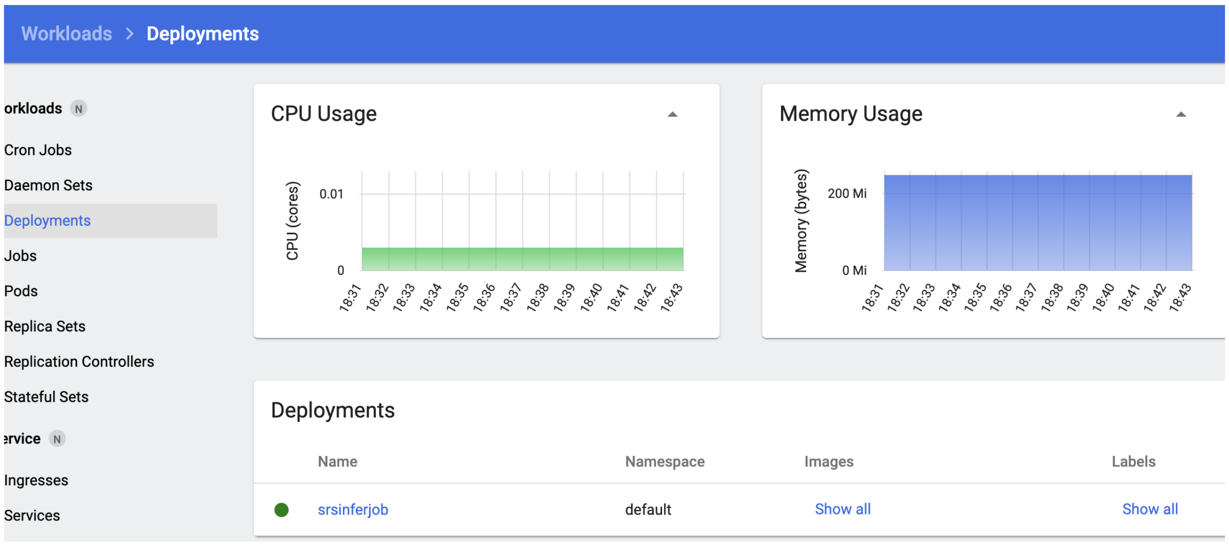
```
❯ docker build -t mnistinfer .
[+] Building 8.5s (10/10) FINISHED
 => [internal] load build definition from Dockerfile                                                                    0.0s
 => => transferring dockerfile: 145B                                                                                    0.0s
 => [internal] load .dockerignore                                                                                       0.0s
 => => transferring context: 2B                                                                                         0.0s
 => [internal] load metadata for docker.io/pytorch/pytorch:latest                                                       0.8s
 => [auth] pytorch/pytorch:pull token for registry-1.docker.io                                                          0.0s
 => [internal] load build context                                                                                       0.0s
 => => transferring context: 2.34kB                                                                                     0.0s
 => CACHED [1/4] FROM docker.io/pytorch/pytorch@sha256:9904a7e081eaca29e3ee46afac87f2879676dd3bf7b5e9b8450454d84e074ef0 0.0s
 => [2/4] COPY . /app                                                                                                   0.0s
 => [3/4] WORKDIR /app                                                                                                  0.0s
 => [4/4] RUN pip install flask                                                                                         7.6s
 => exporting to image                                                                                                  0.0s
 => => exporting layers                                                                                                 0.0s
 => => writing image sha256:fb5bb2550096a720372341379d73b2efb2ce23949bc4d6fc2002a2377f51e2a0                            0.0s
 => => naming to docker.io/library/mnistinfer                                                                           0.0s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
❯ docker tag mnistinfer:latest srsinhanyu/mnistinfer
❯ docker push srsinhanyu/mnistinfer
Using default tag: latest
The push refers to repository [docker.io/srsinhanyu/mnistinfer]
85ad74f37ed5: Pushed
5f70bf18a086: Layer already exists
6528f5c38c86: Pushed
a7a8fb0bf150: Layer already exists
9ac7cb479518: Layer already exists
fe66d33e709d: Layer already exists
a4e25480be6b: Layer already exists
latest: digest: sha256:8dec1b6d05578acd15dd0018898b986ae4f16463da3ee928bebb98d6a84f9126 size: 1785
```

Figure 16: Inference commands

Inference-Deployment

The last and final stage of this task was to create a **deployment** with a pod(here just having 1 container) hosting a local server which was exposed to the outer world using the service created above.

1. I used the *preNpost/inferencedeploy.yaml* for the purpose.
2. Then the command **kubectl apply -f inferencedeploy.yaml** was used to provision the PV.
3. This created a Deployment in the k8S cluster.
4. A single container is created which pulls the *srsinhanyu/mnisttrain* image from the docker hub and starts the Flask server.
5. Now the trained model that was saved in the PV was reused here by mounting the PV in the '/mnt' folder of the deployment container using selectors.
6. The port 8000 was used to connect the container's port to the outside world.



Deployments

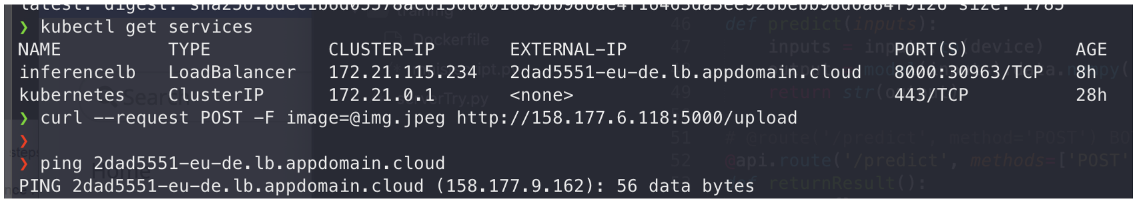
	Name	Namespace	Images	Labels
●	srsinferjob	default	Show all	Show all

Figure 17: Deployments on k8s

Testing

1. For this the IP of the deployment was fetched using: kubectl get services to get external-IP.

2. Then 'ping external-IP' was done to get the IP address to communicate.



```
❯ kubectl get services
NAME          TYPE           CLUSTER-IP      EXTERNAL-IP                        PORT(S)          AGE
inferencelb   LoadBalancer   172.21.115.234  2dad5551-eu-de.lb.appdomain.cloud  8000:30963/TCP   8h
kubernetes    ClusterIP      172.21.0.1      <none>                             443/TCP          28h
❯ curl --request POST -F image=@img.jpeg http://158.177.6.118:5000/upload
❯
❯ ping 2dad5551-eu-de.lb.appdomain.cloud
PING 2dad5551-eu-de.lb.appdomain.cloud (158.177.9.162): 56 data bytes
```

Figure 18: Getting IP

3. From commandline I used curl command to hit the inference Flask server with an image to predict.

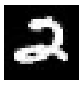
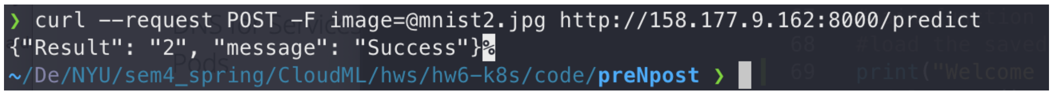


Figure 19: Test image

4. curl –request POST -F image=@mnist2.jpg http://158.177.9.162:8000/predict and output is json with message is request succeeded and the predicted value of the digit image.



```
❯ curl --request POST -F image=@mnist2.jpg http://158.177.9.162:8000/predict
{"Result": "2", "message": "Success"}%
~/De/NYU/sem4_spring/CloudML/hws/hw6-k8s/code/preNpost ❯
```

Figure 20: Prediction in action

Experience & Conclusion

- Overall, it was a challenging and learning experience.
- Few challenges faced were:
  1. Image not being able to downloaded - public gateway not attached.
  2. *CrashLoopBackOff* error - container was crashing due to missing dependencies
  3. Bottle server not working - Here I tried too hard, in the end, after 2-3 hours of debugging, I had no choice but to use Flask server.
- This assignment helped me gain more insights into the Kubernetes world and learn about deployments, services, replicas etc.
- I learnt about Bottle and Flask python servers and how easy (*pun not intended*) they are to deploy.
- You can use *curl –request POST -F image=@mnist2.jpg http://158.177.9.162:8000/predict* with image in preNpost/mnist2.jpg or preNpost/mnist0.jpg to get the predictions.
- **Certificate and Pem** : Certificate and .pem files are submitted too in folder *pemcert* by going to folder cd */Users/shiv/.bluemix/plugins/container-service/clusters/ mycluster-eu-de-1-cx2.2x4-c9ji3plf0rg7un176nog*

Kubernetes controllers

As mentioned in detail above too, training jobs are handled by job controllers since they only need to be executed once until and unless we have another dataset to train the model on, or if the model changes. For inference, a deployment controller is used. As an end-user interacts with the server and receives a response, it needs to keep running to balance the load.

References

1. IBM k8S service
2. IBM public gateway
3. k8s Jobs
4. k8s Pods
5. k8s k8s Deployments
6. k8S Storage basics
7. k8s Service
8. Docker Hub
9. Medium article selectors
10. Flask server python
11. Pytorch MNIST