

## **Develop Kubernetes cluster to perform ML training and inference on IBM Cloud**

### **KUBERNETES – A DEEP DIVE**

Containers help in bundling up and running applications. In a production environment, containers need to be managed to run applications in a way that ensures minimal downtime. This is where Kubernetes comes in handy. Kubernetes is an open-source platform for managing containerized workloads and services.[1] It provides a framework to run distributed systems resiliently.

Kubernetes provides the building blocks for building developer platforms. Certain solutions like deployment, scaling, load balancing, logging, monitoring, and alerting that may seem very common to PaaS offerings. However, Kubernetes is not monolithic, and these solutions are optional and pluggable.

#### ***Kubernetes components***

When you deploy Kubernetes, you get a cluster.[2] A Kubernetes cluster consists of a set of worker machines to run containerized applications. These worker machines are called nodes. A cluster has at least one worker node.

Pods that are the components of the application workload that are hosted within a worker node. A cluster usually runs multiple nodes.

The control plane manages the worker nodes and the Pods in the cluster. The control plane usually runs across multiple computers providing fault-tolerance and high availability. The control plane is responsible for making global decisions about any machines on the cluster, detecting and responding to cluster events.

YAML files help in the configuration of the Kubernetes cluster to spin up resources.[3] YAML is a superset of JSON. YAML files help in defining a Kubernetes manifest. They are very convenient and flexible.

#### ***Making Kubernetes cluster on IBM Cloud***

The Kubernetes cluster I deployed on IBM Cloud has just one node (for cost purposes) with one container. I first set up a Virtual Private Cloud to get assigned a subnet where I could run my Kubernetes cluster. To access this subnet, I created a public gateway and attached the subnet to it. Then I went to Kubernetes and created a cluster through UI. The cluster creation took around 10 minutes.

I connected to the cluster via CLI. The commands used for the same were:

```
ibmcloud login -a cloud.ibm.com -r eu-de -g 2022-spring-student-ns4451
```

```
ibmcloud plugin install container-service
```

```
ibmcloud ks cluster config --cluster c9ji30qf00phnf50a4jg
```

```
kubectl config current-context
```

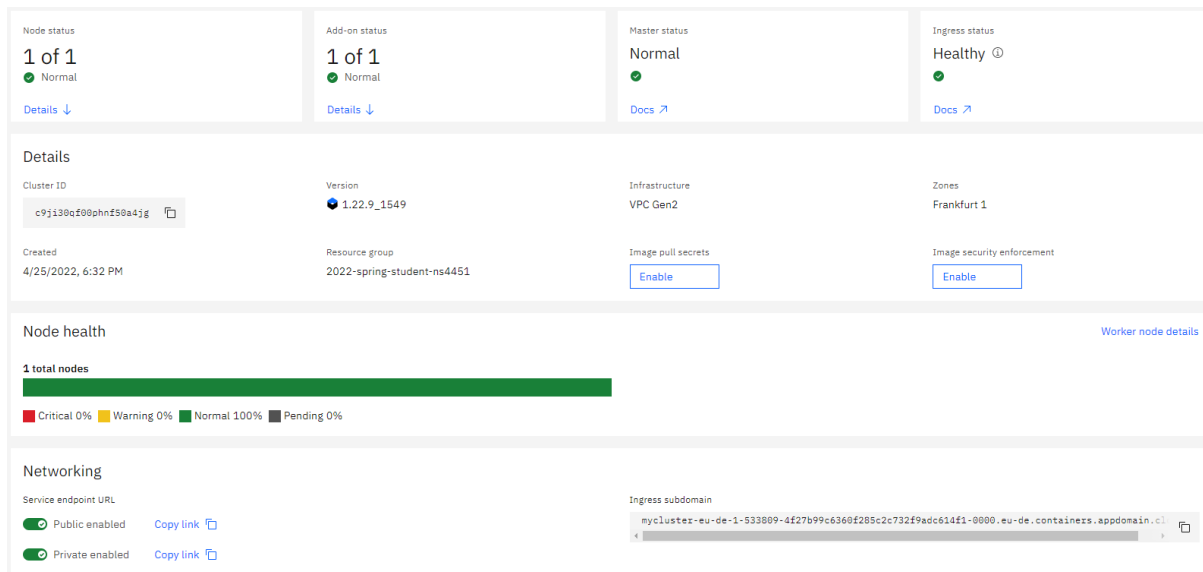


Figure 1: 1-node Kubernetes cluster deployed on IBM Cloud

```
C:\WINDOWS\system32>ibmcloud login -a cloud.ibm.com -r eu-de -g 2022-spring-student-ns4451
API endpoint: https://cloud.ibm.com

Email> ns4451@nyu.edu

Password>
Authenticating...
OK

Targeted account NYU-ML Cloud (7d0717b570cb4f35bdef5deb642380b8) <-> 2238606

Targeted resource group 2022-spring-student-ns4451

Targeted region eu-de

API endpoint:      https://cloud.ibm.com
Region:           eu-de
User:             ns4451@nyu.edu
Account:          NYU-ML Cloud (7d0717b570cb4f35bdef5deb642380b8) <-> 2238606
Resource group:   2022-spring-student-ns4451
CF API endpoint:
Org:
Space:
```

Figure 2: Logging into IBM Cloud via CLI

```
C:\WINDOWS\system32>ibmcloud plugin install container-service
Looking up 'container-service' from repository 'IBM Cloud'...
Plug-in 'container-service[kubernetes-service] 1.0.403' found in repository 'IBM Cloud'
Attempting to download the binary file...
 27.02 MiB / 27.02 MiB [=====] 100.00% 3s
28337664 bytes downloaded
Installing binary...
OK
Plug-in 'container-service 1.0.403' was successfully installed into C:\Users\Admin\bluemix\plugins\container-service. Use 'ib

C:\WINDOWS\system32>ibmcloud ks cluster config --cluster c9ji30qf00phnf50a4jg
OK
The configuration for c9ji30qf00phnf50a4jg was downloaded successfully.

Added context for c9ji30qf00phnf50a4jg to the current kubeconfig file.
You can now execute 'kubectl' commands against your cluster. For example, run 'kubectl get nodes'.
If you are accessing the cluster for the first time, 'kubectl' commands might fail for a few seconds while RBAC synchronizes.

C:\WINDOWS\system32>kubectl config current-context
mycluster-eu-de-1-cx2.2x4/c9ji30qf00phnf50a4jg
```

Figure 3: ks cluster config through CLI

## ML MODEL – MNIST DIGIT RECOGNITION

I ran the MNIST digit recognition model that I have used before in the previous assignments. I cloned the git repository [4] to get main.py file. I modified it to run it for 3 epochs and only keep the commands required for model training. Also, I named my persistent volume and stored the model (.pth extension) in the same to load it later.

```
def main():
    # Hard coding the batch-size=256, epochs=3, seed
    parser = argparse.ArgumentParser(description='PyTorch MNIST Example')
    parser.add_argument('--lr', type=float, default=0.001, metavar='LR',
                        help='learning rate (default: 0.001)')
    parser.add_argument('--gamma', type=float, default=0.7, metavar='M',
                        help='Learning rate step gamma (default: 0.7)')
    parser.add_argument('--no-cuda', action='store_true', default=False,
                        help='disables CUDA training')
    parser.add_argument('--log-interval', type=int, default=10, metavar='N',
                        help='how many batches to wait before logging training status')
    parser.add_argument('--save-model', action='store_true', default=True,
                        help='For Saving the current Model')
    args = parser.parse_args()
    use_cuda = not args.no_cuda and torch.cuda.is_available()

    seed = 1
    batch_size = 256
    epochs = 3

    torch.manual_seed(seed)

    device = torch.device("cuda" if use_cuda else "cpu")

    train_kwargs = {'batch_size': batch_size}
    if use_cuda:
        cuda_kwargs = {'num_workers': 1,
                       'pin_memory': True,
                       'shuffle': True}
        train_kwargs.update(cuda_kwargs)

    transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])
    dataset1 = datasets.MNIST('../data', train=True, download=True,
                             transform=transform)
    train_loader = torch.utils.data.DataLoader(dataset1,**train_kwargs)

    model = Net().to(device)
    optimizer = optim.Adadelta(model.parameters(), lr=args.lr)

    scheduler = StepLR(optimizer, step_size=1, gamma=args.gamma)
    for epoch in range(1, epochs + 1):
        train(args, model, device, train_loader, optimizer, epoch)
        scheduler.step()

    if args.save_model:
        torch.save(model.state_dict(), "/trainedmnist/mnist_cnn.pth")
```

Figure 4: The modified main.py file to train the model. Hardcoded the epochs and gave the path to the persistent volume.

For testing the model, I created another python file infer.py. This file loads the trained model from the persistent volume (using the same path as the .pth file), sets up a Flask server, accepts a test image and gives the result of the image.

```
def inference(img):
    model = Net().to('cpu')
    model.load_state_dict(torch.load('/trainedmnist/mnist_cnn.pth'))
    model.eval()
    img = img.to('cpu')
    output = model(img)
    index = output.data.numpy().argmax()
    return str(index)

api = Flask(__name__)

@api.route('/inference', methods=['POST'])
def get_result():
    res = {}
    file = request.files['image']
    if not file:
        res['result'] = 'Image 404'
    else:
        res['result'] = 'Inference completed.'
        image = Image.open(file.stream).convert('L')
        transform_obj = transform()
        image = transform_obj(image)
        image = image.unsqueeze(0)
        ans = inference(image)
        res['Predicted Digit'] = ans

    return json.dumps(res)

api.run(host='0.0.0.0', port=5000)

def main():
    img = Image.open("testimage.png")

    img = transform(img)
    img = img.unsqueeze(0)
    ans = inference(img)
    print(ans)
```

Figure 5: Code for testing an image on the trained MNIST model from the same persistent volume path. I am running a Flask server at the given port number.

## DOCKER IMAGES FOR MODEL TRAINING AND INFERENCE

To create an environment where our model training could run, I created the following dockerfile. I imported the base image of pytorch and installed all its dependencies in the environment.

```
1 FROM pytorch/pytorch:latest
2
3 MAINTAINER Niharika Sinha "ns4451@nyu.edu"
4
5 RUN mkdir /home/user
6 USER root
7
8 COPY main.py .
9
10 CMD python3 main.py
```

Figure 6: The docker file for training the MNIST model

I built the docker file and pushed it to docker hub. This makes the docker image public and available for the Kubernetes worker nodes to fetch it through YAML files.

```
docker build -t dockerfile .
```

```
docker login -u ns4451docker
```

```
docker tag dockerfile:latest ns4451docker/cmlhw5
```

```
docker push ns4451docker/cmlhw5
```

Similarly, I made a docker file for testing the model and pushed this too to docker hub. I also installed flask via pip in this docker image so that I could send the test image to the flask server.

```
1 FROM pytorch/pytorch:latest
2 RUN pip install flask
3
4 COPY infer.py .
5
6 CMD python3 infer.py
7
8 USER root
```

Figure 7: The dockerfile to test the MNIST model

Commands to push the test model environment to docker hub:

```
docker build -t dockerfile .
```

```
docker login -u ns4451docker
```

```
docker tag dockerfile:latest ns4451docker/cmlhw5inf
```

```
docker push ns4451docker/cmlhw5inf
```

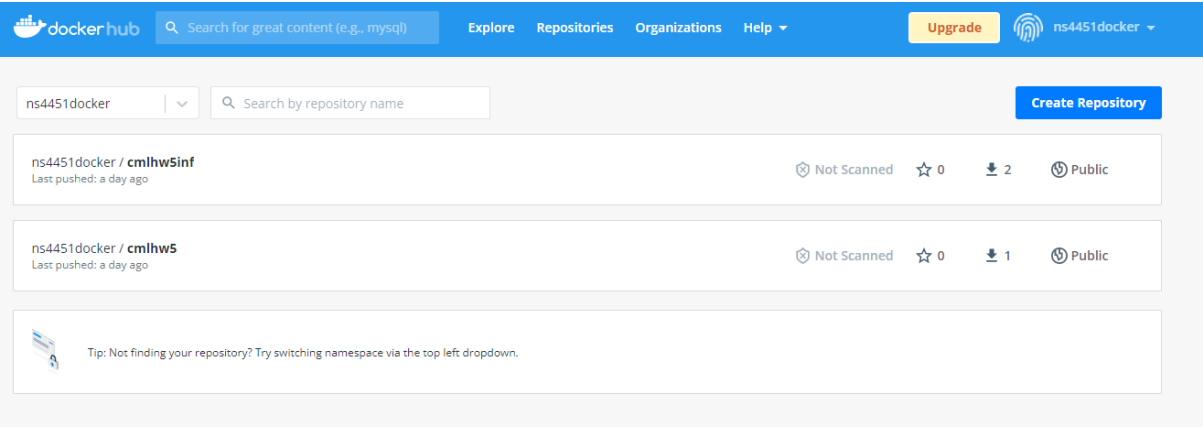


Figure 8: Docker hub showing the 2 repositories containing the train and test docker images

IMAGE LAYERS ⓘ

1	ADD file ... in /	25.47 MB
2	CMD ["bash"]	0 B
3	ARG PYTORCH_VERSION	0 B
4	LABEL com.nvidia.volumes.needed=nvidia_driver	0 B
5	RUN [1 PYTORCH_VERSION=v1.11.0 /bin/sh -c	9.47 MB
6	COPY /opt/conda /opt/conda # buildkit	2.55 GB
7	ENV PATH=/opt/conda/bin:/usr/local/sbin:/usr/local/bin:/...	0 B
8	ENV NVIDIA_VISIBLE_DEVICES=all	0 B
9	ENV NVIDIA_DRIVER_CAPABILITIES=compute,utility	0 B
10	ENV LD_LIBRARY_PATH=/usr/local/nvidia/lib:/usr/local/nvi...	0 B
11	ENV PYTORCH_VERSION=v1.11.0	0 B
12	WORKDIR /workspace	98 B
13	MAINTAINER Niharika Sinha "ns4451@nyu.edu"	0 B
14	RUN /bin/sh -c mkdir /home/user	148 B
15	USER root	0 B
16	COPY main.py . # buildkit	1.66 KB
17	CMD ["/bin/sh" "-c" "python3 main.py"]	0 B

Figure 9: Docker image layers for model training

IMAGE LAYERS ⓘ

1	ADD file ... in /	25.47 MB
2	CMD ["bash"]	0 B
3	ARG PYTORCH_VERSION	0 B
4	LABEL com.nvidia.volumes.needed=nvidia_driver	0 B
5	RUN [1 PYTORCH_VERSION=v1.11.0 /bin/sh -c	9.47 MB
6	COPY /opt/conda /opt/conda # buildkit	2.55 GB
7	ENV PATH=/opt/conda/bin:/usr/local/sbin:/usr/local/bin:/...	0 B
8	ENV NVIDIA_VISIBLE_DEVICES=all	0 B
9	ENV NVIDIA_DRIVER_CAPABILITIES=compute,utility	0 B
10	ENV LD_LIBRARY_PATH=/usr/local/nvidia/lib:/usr/local/nvi...	0 B
11	ENV PYTORCH_VERSION=v1.11.0	0 B
12	WORKDIR /workspace	98 B
13	RUN /bin/sh -c pip install	1.8 MB
14	COPY infer.py . # buildkit	1.01 KB
15	CMD ["/bin/sh" "-c" "python3 infer.py"]	0 B
16	USER root	0 B

Figure 10: Docker image layers for model testing

## YAML FILES – PVC, MODEL TRAINING AND DEPLOYMENT

*PersistentVolume* (PV) is a piece of persistent storage in the Kubernetes cluster. It is just like any other resource in a cluster. Its lifecycle is independent of the lifecycle of any pod using the storage. A *PersistentVolumeClaim* (PVC) is a request for storage by a user. It is similar to a Pod. PVCs consume PV resources.[5]

Before executing any operations on the cluster, I had to claim persistent volume on it. I did so through executing the following mypvc.yaml file, allotting 10 GB of storage to the MNIST model workspace.

```

1  apiVersion: v1
2  kind: PersistentVolumeClaim
3  metadata:
4    name: my-pvc
5  spec:
6    storageClassName: ibmc-vpc-block-5iops-tier
7    accessModes:
8      - ReadWriteOnce
9    resources:
10     requests:
11       storage: 10Gi

```

Figure 11: mypvc.yaml file

I ran the following commands through cli using kubectl to run this file on Kubernetes cluster.

```
kubectl apply -f mypvc.yaml
```

```

C:\Users\Niharika Sinha\OneDrive\NYU\courses\CML\hw5>kubectl apply -f mypvc.yaml
persistentvolumeclaim/my-pvc created

```

Figure 12: Running mypvc.yaml on Kubernetes cluster


Persistent Volume Claims						
Name	Namespace	Labels	Status	Volume	Capacity	Access Modes
 my-pvc	default	-	Bound	<a href="#">pvc-cab3002e-2ab3-4bf7-b302-b4d6a3e446d2</a>	10Gi	<a href="#">Show all</a>

Figure 13: PVC created on Kubernetes cluster as seen on the Kubernetes dashboard

Next, I prepared train.yaml file to pull the docker image for training the MNIST model from docker hub, where I had pushed the docker image earlier. This yaml file takes in the details of my docker hub repository which has the docker image for training the MNIST model.

```

1  apiVersion: batch/v1
2  kind: Job
3  metadata:
4    name: trainingjob
5  spec:
6    template:
7      spec:
8        containers:
9        - name: trainingjob
10          image: ns4451docker/cmlhw5
11          command: ["python3", "main.py"]
12          volumeMounts:
13            - name: myvolume
14              mountPath: /trainedmnist
15        restartPolicy: Never
16        volumes:
17          - name: myvolume
18            persistentVolumeClaim:
19              claimName: my-pvc

```

Figure 14: training.yaml file

Executing the following command pulls the specified docker image and runs the training job for 3 epochs.

kubectl apply -f training.yaml

```

C:\Users\Niharika Sinha\OneDrive\NYU\courses\CML\hw5>kubectl apply -f training.yaml
job.batch/trainingjob created

```

Figure 15: Running training.yaml on Kubernetes cluster

A training job is created which gets executed in the Kubernetes pod and saves the model in the persistent storage that I had created through mypvc.yaml.

Jobs			
Name	Namespace	Images	Labels
<span style="color: green;">●</span> trainingjob	default	<a href="#">Show all</a>	<a href="#">Show all</a>

Figure 16: The training job completed in the Kubernetes pod

Executing the following commands helps check the status of the training job on Kubernetes pod.

```

C:\Users\Niharika Sinha\OneDrive\NYU\courses\CML\hw5>kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
trainingjob--1-4qs7 0/1     Completed 0           9m43s

```

Figure 17: Checking the status of the training job on the pod

Once the trained model is saved to the persistent storage, I moved on to testing a sample image. Through infer.py, I had set up a flask server which will listen for requests and return the response. Using the curl command, I could ping sample images to the flask server and get the result through trained model as a response. I created



deploymentservice.yaml to pull the testing docker image from docker hub and deploy it in the Kubernetes pod.

```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: mydeployment
5    labels:
6      app: api
7  spec:
8    replicas: 1
9    selector:
10     matchLabels:
11       app: api
12    template:
13     metadata:
14       labels:
15         app: api
16     spec:
17       containers:
18         - name: mydeployment
19           image: ns4451docker/cmlhw5in:
20           ports:
21             - containerPort: 5000
22           volumeMounts:
23             - name: myvolume
24               mountPath: /trainedmnist
25       volumes:
26         - name: myvolume
27           persistentVolumeClaim:
28             claimName: my-pvc
29  ---

```

Figure 18: Deployment of deploymentservice.yaml

```

31  apiVersion: v1
32  kind: Service
33  metadata:
34    name: myservice
35  spec:
36    type: LoadBalancer
37    selector:
38      app: api
39    ports:
40      - protocol: TCP
41        port: 5000
42        targetPort: 5000

```

Figure 19: Load balancer service of deploymentservice.yaml

Executing the following command pulls the specified docker image and runs the service on Kubernetes pod. It starts a service and deployment on the Kubernetes pod.

```
kubectl apply -f deploymentservice.yaml
```

```
C:\Users\Niharika Sinha\OneDrive\NYU\courses\CML\hw5>kubectl apply -f deployment-service.yaml
service/myservice created

C:\Users\Niharika Sinha\OneDrive\NYU\courses\CML\hw5>kubectl apply -f deployment-service.yaml
deployment.apps/mydeployment created
service/myservice unchanged
```

Figure 20: Executing the deployment-service.yaml file


Services				
Name	Namespace	Labels	Type	Cluster IP
 <a href="#">myservice</a>	default	-	LoadBalancer	172.21.145.113

Figure 21: The load balancer service status through Kubernetes dashboard


Deployments				
Name	Namespace	Images	Labels	Pods
 <a href="#">mydeployment</a>	default	<a href="#">Show all</a>	<a href="#">Show all</a>	1 / 1

Figure 22: Deployment status through Kubernetes dashboard

## TESTING THROUGH PINGING FLASK SERVER

With the load balancer service and the deployment now executed on the Kubernetes pod, the flask server is up and running at port 5000. From my local cmd, I pinged the test image to flask server using the following command:

```
curl --request POST -F "image=@testimage.jpg" http://localhost:5000/inference
```



Figure 23: The test image

```
Niharika Sinha@Niharika MINGW64 ~/OneDrive/NYU/courses/CML/examples/mnist (main)
$ curl --request POST -F "image=@testing.jpg" http://7c862fc9-eu-de.1b.appdomain.c
loud:5000/inference
{"Predicted Digit": "6", "result": "Inference completed."}
```

Figure 24: Pinging the test image to the flask server and getting the result from the trained MNIST model

## DISCUSSION

Being a deep dive into the Kubernetes components, this has truly been a very enriching assignment. I enjoyed working on this a lot since I got to know the setup and working of Kubernetes cluster. Deploying a Kubernetes cluster, and training and testing ML models on the same is something I had not worked on before and learnt through this assignment. I got to learn in-depth about Kubernetes components like nodes, pods and PVC, which was very fascinating to me. I also learnt how to deploy and ping a flask server which was a fun experience.

### *Kubernetes controllers*

Kubernetes controllers are control loops that watch the state of your cluster. They make or request changes when required. Each controller tries to move the current cluster state closer to the desired state. [6]

Since training the model is a one-time job, I used the Kubernetes job controller.[7] Inference however is a service that should continuously keep running. Hence, we implemented a load balancer and a Kubernetes deployment controller.[8]

### *Challenges*

Each individual step is very crucial, including writing of the yaml files and the python scripts. Even a small bug at an early stage has a cascading effect and it becomes quite cumbersome to debug later on. Initially I had created my public gateway in a different region as the subnet, hence I faced issues while pulling the docker image. My deployment through the yaml file was not going through, and I later realised that there was a mismatch in the python file names in the yaml file. Overall, it was a smooth experience.

**REFERENCES**

- [1] <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>
- [2] <https://kubernetes.io/docs/concepts/overview/components/>
- [3] <https://www.mirantis.com/blog/introduction-to-yaml-creating-a-kubernetes-deployment/>
- [4] <https://github.com/pytorch/examples/blob/main/mnist/main.py>
- [5] <https://kubernetes.io/docs/concepts/storage/persistent-volumes/>
- [6] <https://kubernetes.io/docs/concepts/architecture/controller/>
- [7] <https://kubernetes.io/docs/concepts/workloads/controllers/job/>
- [8] <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>
- [9] <https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes>