A Comparative Study: Cloud Providers Leveraging K8S for ML

12.05.2023

Ritika Motwani (rm6383)
Sameer Ahmed (sa6142)
Caraline Bruzinski (cb4904)

Abstract

Ease of application ML model deployment is being worked on by a lot of major industries and developers. Containers were used for many years to bundle and run various applications. But in a production environment container management and ensuring zero application downtime became a hassle for developers. This increased the need for behavior and tools handled by a complete system. This is where Kubernetes came to rescue. It provided frameworks to train and deploy ML models resiliently. It is an orchestration tool which focuses on automating the creation and deployment of containers. Autoscaling is an important feature to allocate computing resources on demand. It allows users to automatically scale resources provisioned to applications without a lot of human interference. Kubernetes, the most prevalent container orchestration, provides configurations to set autoscaling configurations. In this project we will see how on increasing or decreasing application load the number of nodes associated with the application kubernetes cluster change. .

1. Introduction

In today's rapidly evolving technological landscape, deploying applications on Kubernetes in the cloud has become increasingly crucial for organizations seeking scalability, reliability, and efficiency. Kubernetes, an open-source container orchestration platform, enables businesses to manage and automate the deployment, scaling, and management of containerized applications seamlessly. However, with numerous cloud providers offering

Kubernetes services, such as IBM, Google Cloud Platform (GCP), and Amazon Web Services (AWS), it becomes essential to compare and evaluate these providers to determine the most suitable option for specific business requirements. This comparison helps organizations make informed decisions, considering factors like performance, cost, ease of use, security, and integration capabilities provided by each Kubernetes cloud provider. By conducting such a comparison, businesses can ensure optimal utilization of resources and leverage the best-in-class offerings to meet their application deployment needs.

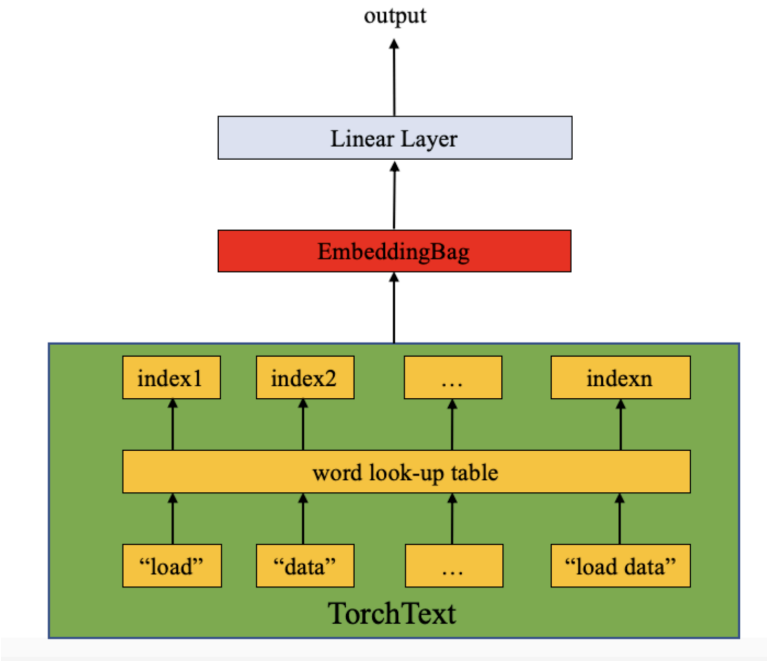
2. Goals

In this project we studied different types of models that are being used for text based sentiment analysis like sentiment analysis with the flair, bert and n-gram. We selected the n-gram based model for classification. We studied ML model deployment via Kubernetes, on each of the three major cloud providers: Google, AWS and IBM. We used Kustomize, which is a Kubernetes native configuration management tool, to ease the deployment process. We reviewed key performance metrics and graphs for monitoring the deployment and CPU utilization, and we studied how resource-demanding loads are handled by the system in tandem with Kubernetes via autoscaling. In this project we chose CPU utilization as a metric set for horizontal autoscaling to change the number of nodes associated with the cluster. We compare the experience of managing our Kubernetes deployment across the providers.

3. Background Information

3.1 Model Used

In this project, we used a text classification model to build our news article classification web app. The model is built using Torchtext which is a Python library built on top of the PyTorch machine learning framework, designed to simplify and streamline the process of text data preprocessing for natural language processing (NLP) tasks[10]. The model is composed of the EmbeddingBag layer plus a linear layer for the classification purpose. EmbeddingBag with the default mode of mean computes the mean value of a bag of embeddings. Additionally, since EmbeddingBag accumulates the average across the embeddings on the fly, EmbeddingBag can enhance the performance and memory efficiency to process a sequence of tensors[11].



3.2 Autoscaling

Scalability is an important feature in kubernetes. The cluster autoscaler helps with adjusting the number of nodes in the cluster when they are over or under utilized. Horizontal pod autoscaler helps with adjusting the number of replicas of the application. Vertical pod autoscalers help with resource requests and limits of a container.

Autoscaling helps applications to handle an increase in traffic to their domain or reduce costs when the resource usage is less. Developers can define the autoscaling policy and the autoscaler performs automatic scaling based on the measured load and the options you configure. Autoscaling works by adding more instances/nodes to your cluster when there is more load (scaling out), and deleting them when the need is lowered (scaling in).

In this project we implemented and studied the benefits of horizontal autoscaling. In horizontal autoscaling, we supply additional servers to meet the end users needs, by often splitting the workload between servers to limit the number of requests any individual server is getting. It means adding additional instances instead of increasing the instance size.

Horizontal auto scaling helps in adjusting the replicas based on requirements, making the application more agile and meeting the changing needs of the end users. It

ensures that applications run at an optimal level of resource utility, and applications are not over provisioned or under provisioned.

Other benefits include improving the performance, availability and resilience. If demand for a request increases in the real world, the systems can scale up by adding more replicas to handle the load, improving the user experience. Applications can meet demand without performance degradation. Automatically increasing the number of replicas if failures happen, helps to ensure that applications remain available and responsive. If the case of demand decreases, autoscaling helps to prevent over-provisioning and reduces resource wastage. It improves the resilience of the application and service. It increases flexibility of the application and helps it adapt to a dynamic environment.

4. Implementation

4.1 IBM

Steps for deployment

1. Create a docker image of the application and push the image to the docker registry.
    a. docker build -t cml-project-2 .
    b. docker tag cml-project-2:latest sameer7483/cml-project-2:latest
    c. docker push sameer7483/cml-project-2:latest
2. Create the IBM Cloud kubernetes cluster using the UI with the VPC configuration.
3. Create yaml files for training (kind: Job), inference (kind: Deployment) and exposing the inference app(kind: Service, type: LoadBalancer).
4. In order to ease the deployment process, instead of applying each of the yaml files individually. We used kustomization.yaml so as to apply all the yaml files using one command.
5. Deploy using a single kustomize command(kubectl apply --kustomize=${PWD}/CICD/ --record=true)
6. The deployed application is: http://1dadaf53-us-south.lb.appdomain.cloud:5000/

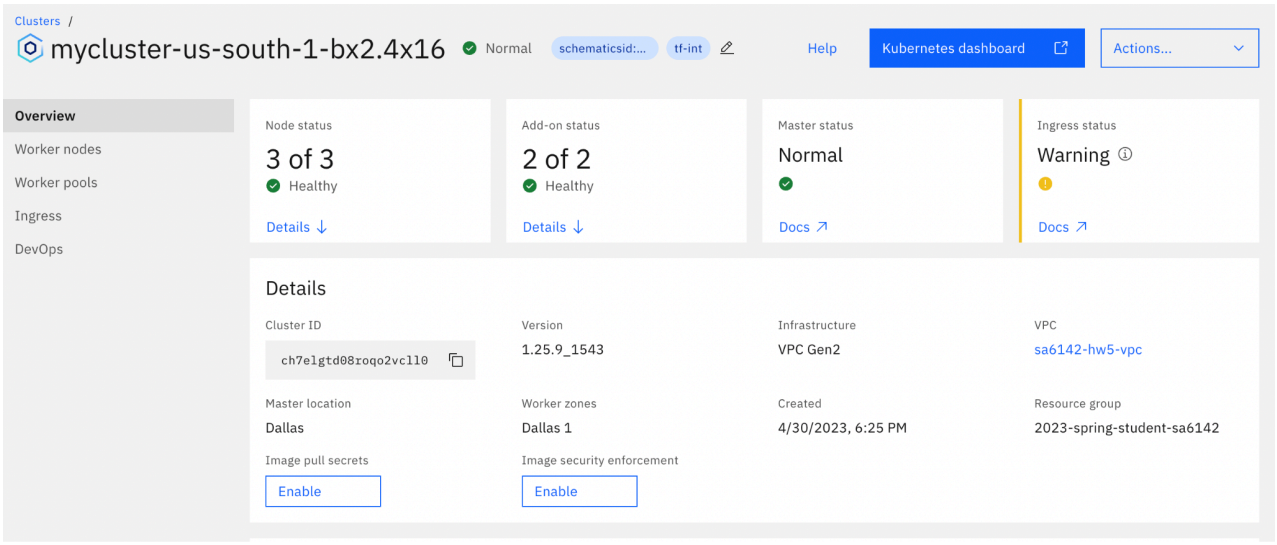


Fig: IBM cloud kubernetes cluster

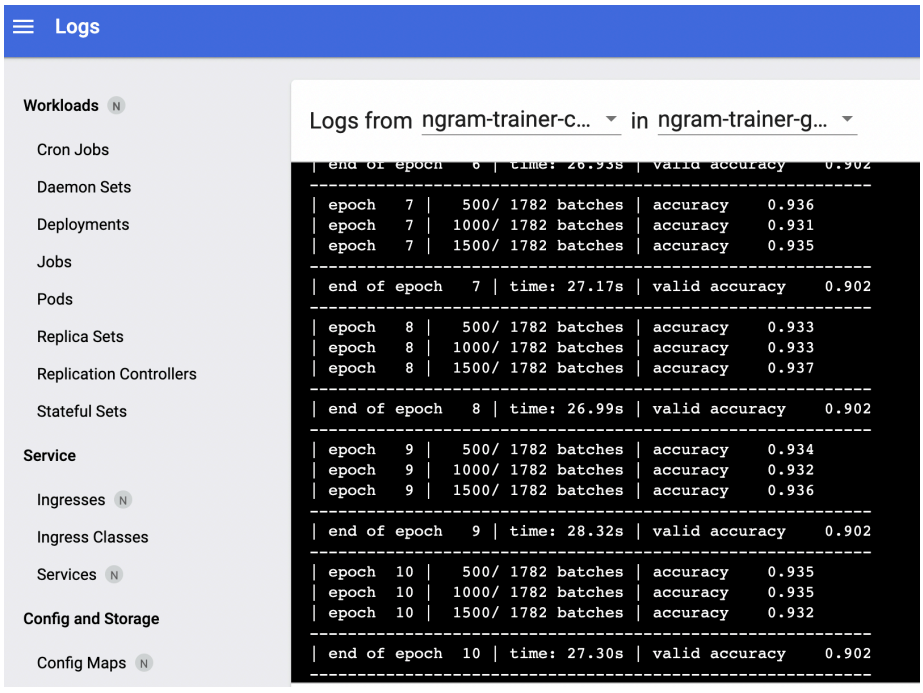


Fig: Training the Torchtext ngram model

```
(base) sameer@Sameers-Air IBM % kubectl get deployments
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
flair-inference     0/1     1            0           2d13h
lstm-inference      0/1     1            0           35h
mnist-inference     1/1     1            1           3d15h
ngram-inference     1/1     1            1           17h
(base) sameer@Sameers-Air IBM %
```

Fig: Status of deployed n-gram inference application on console

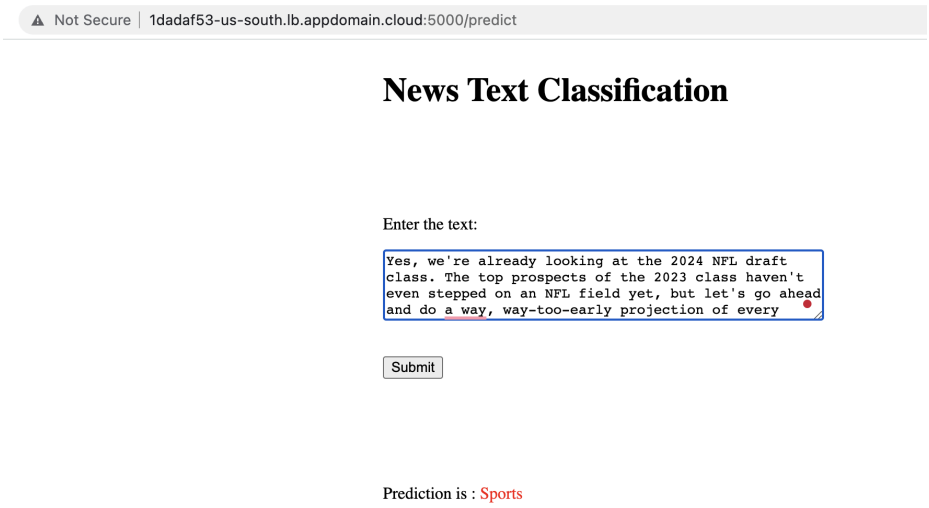


Fig: Deployed news text classification web app powered by IBM cloud

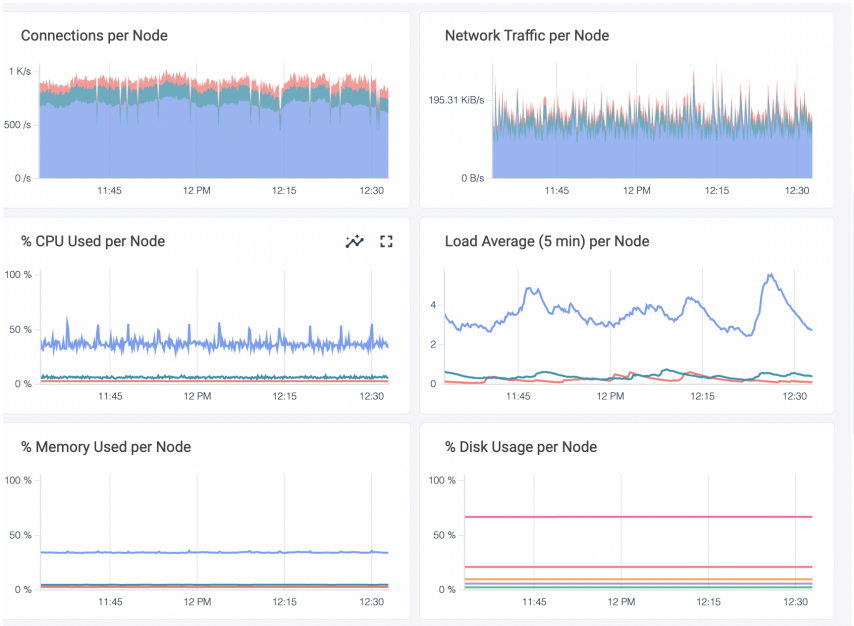


Fig: IBM Cloud kubernetes cluster observability and monitoring tools

Steps to test Autoscaling in IBM

1. Initially we had one node in our ibm cloud cluster with autoscaling disabled.
2. To enable autoscaling, we Installed cluster autoscale add-on using ibmcloud ks cluster addon enable cluster-autoscaler --cluster ch7elgtd08roqo2vcll0
3. Edited the configuration of the config map to enable the autoscaling when the compute load exceeds 10% with maximum number of nodes as 10. kubectl edit cm iks-ca-configmap -n kube-system -o yaml
4. We wrote a python script to send in multiple requests to the endpoint created after the application was deployed. The script sent in multiple parallel requests to the endpoint. There were 3600 simultaneous requests (concurrent requests) sent to replicate how in real world multiple users are using the application at the same time. We used the ThreadPoolExecutorService, and the max workers in python to achieve that. *The script will be attached to the submitted code.*
5. As can be seen in the Fig, as the load increases the worker node count increases with it.

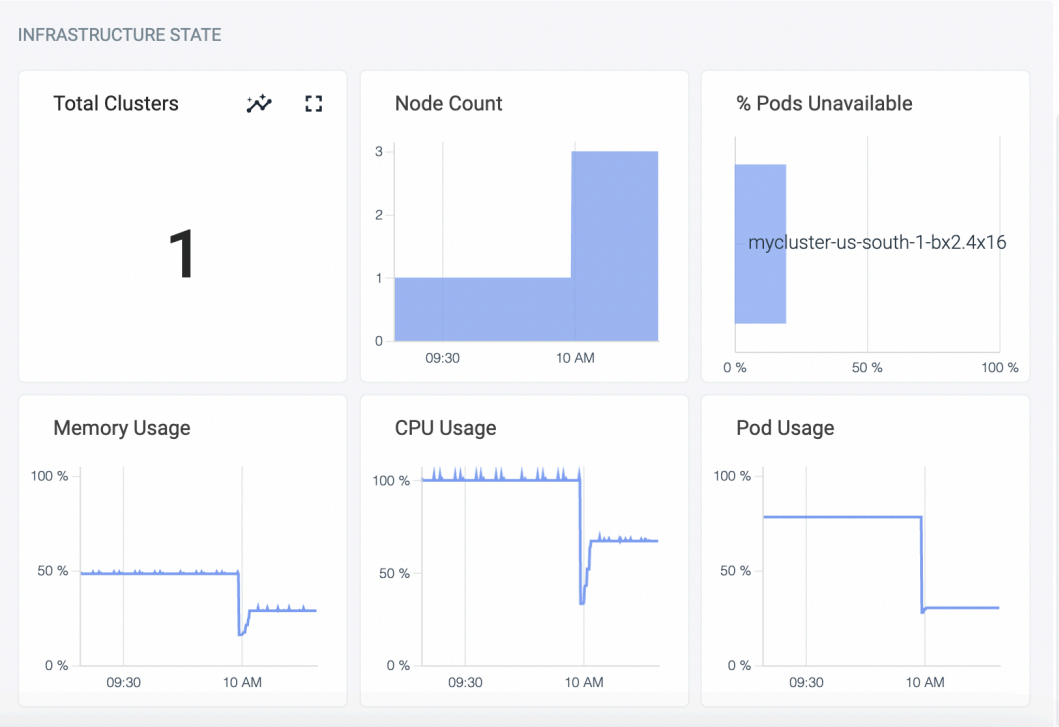


Fig. Showing autoscaling of worker nodes in Ibm cloud

4.2 AWS

With Kubernetes on AWS we are able to reuse the same Deployment Set and load balancer Service definition YAML files, but AWS does not provide sensible defaults on the cluster, so these must be installed separately. Our specific setup follows the instructions below:

Steps for deployment

1. Write the dockerfiles for train and inference, tag and then push it either to docker hub.
    a. *docker tag tag-name [HOSTNAME]/[PROJECT-ID]/image-name*
    b. *docker push [HOSTNAME]/[PROJECT-ID]/image-name*
2. Install `eksctl` via homebrew or another package manager.
3. Create an AWS configuration file `~/.aws/config` with your specific AWS credentials, so that `eksctl` can access your account. These credentials are located in your account page from the UI.
4. Determine the desired cluster properties such as node type and size, deployment region, and the number of nodes for your cluster. Also identify which version of Kubernetes you will be using for the project.
5. Execute the `eksctl` command with desired properties as below:

```
eksctl create cluster \
--name cloud-ml-caraline-eks-cluster \
--version 1.26 \
--region us-east-1 \
--nodegroup-name linux-nodes-eks-cloud-ml \
--node-type t2.xlarge \
--nodes 2
```

6. The command line tool provides status updates to STDOUT as objects are being created. Within roughly 15 minutes the resources will be provisioned on AWS. When the cluster is ready to be used the nodes and the cluster will be visible from the UI and marked in **ready** state.
7. From the command line, check that kubectl has access to your nodes deployed on AWS via `kubectl get nodes`. If your nodes are listed then you may proceed.

One of the key differences with AWS EKS is that AWS does not provide basic defaults on the cluster. The following section pertains to setting up relevant defaults on the cluster:

8. Enable your cluster to access and manage Persistent Volumes and Persistent Volume Claims. Install the AWS Elastic Block Storage (EBS) Container Storage Interface (CSI) driver on the cluster, as persistent volume drivers are not provided from AWS by default when the cluster is provisioned. EBS-CSI Driver is an open source project, developed and provided by the Kubernetes community. Installing the driver allows your containers to automatically create EBS volumes and associated PV from PVC specification files.
9. Add the repository to local:

```
git clone git@github.com:kubernetes-sigs/aws-ebs-csi-driver.git
```

10. Apply the driver to your cluster with Kustomize:

```
kubectl apply -k
"github.com/kubernetes-sigs/aws-ebs-csi-driver/deploy/kubernetes/
overlays/stable/?ref=release-1.18"
```

11. Once applied, STDOUT will print that the drivers have been created on your cluster. In my case, I also had to manually edit and update the version of 1 out of the 8 driver files to use the previous release version.
12. Install the metrics server on your cluster, as this is not provided by default from AWS. Metrics Server is provided by the Kubernetes community. This enables your cluster to collect metrics from your containers:

```
kubectl apply -f
https://github.com/kubernetes-sigs/metrics-server/releases/latest
/download/components.yaml
```

13. Wait about 10 minutes for the metrics server to be up and running. Check that it is running by `kubectl get pods`. Once the metrics server is up you may proceed.

AWS is very sensitive to access roles and permissions. Earlier, we discussed the steps for installing defaults onto the cluster such as metrics server and persistent volume management drivers. The following section pertains to setting up the correct AWS Roles and Policies to use the defaults we have installed onto the cluster.

14. Give permission for your cluster to send collected metrics from the metrics server to AWS CloudWatch. CloudWatch is an AWS UI tool for analyzing and visualizing your objects log files.
    a. IAM Roles ⇒ Find your Node Group Service Role ⇒ Add Policy: AmazonEBSCSIDriverPolicy
15. Give permission for your cluster to access AWS EBS so that kubernetes may create and manage persistent volumes and persistent volume claims.
    a. IAM Roles ⇒ Find your Node Group Service Role ⇒ Add Policy: CloudWatchLogsFullAccess

Finally, we have created the desired defaults and allocated permissioning for these defaults to access AWS objects. We are ready to deploy the Deployment Set, Service, and PVC to the cluster and launch the application. The steps we apply at this point are similar to the ones applied with IBM:

16. Apply the PVC: `kubectl apply -f aws-pvc.yaml`
17. Apply Deployment Set and Service: `kubectl apply -f cml-deploy.yaml`
18. Once deployed, the AWS UI will show the pods have been created and recent log messages. Once both containers have been built, you will be able to access the fully deployed application via the load balancer service URL: (http://ac3090eb44e1d4cfcbd15baddbee603d-2078934410.us-east-1.elb.amazonaws.com/inference)

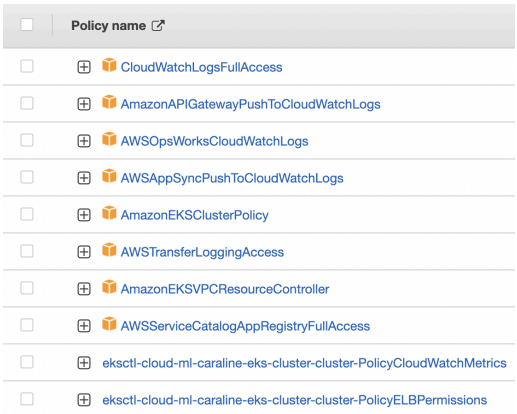


Figure 1: AWS policies on Service Role for NodeGroup

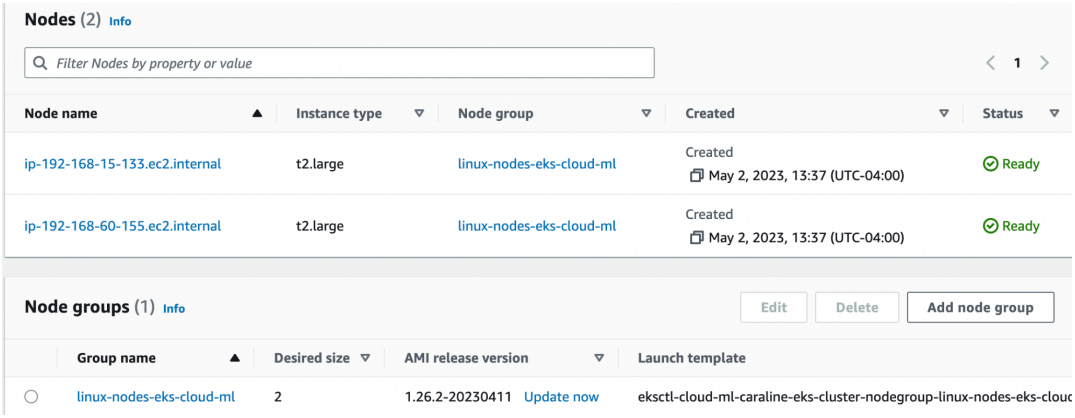


Figure 2: Cluster objects created

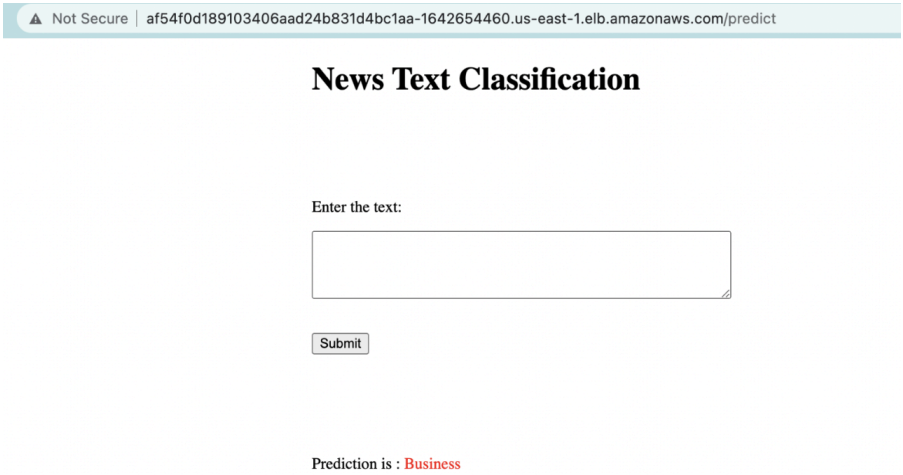


Figure 3: Deployed application on AWS

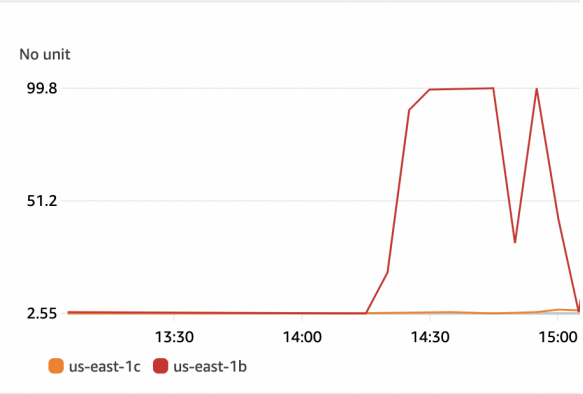


Figure 4: AWS CloudWatch logs for CPU load

Autoscaling in AWS: Setup

Initially for our cluster we had created two nodes, and by default auto scaling is disabled in AWS. Surprisingly, autoscaling can be enabled from the UI.

1. From the console, to go EKS > [*select your cluster*] > NodeGroups > [*select your node group*] > Edit.
2. From the NodeGroup editor panel, set the minimum number of nodes and the maximum number of nodes. For testing, and to keep our costs low, we have chosen 2 as the minimum and 10 nodes as the maximum.
3. From the NodeGroup editor panel, set the threshold for the maximum CPU, the target CPU utilization, for horizontal scaling as `100m` cpu units (10%)
4. We wrote a python script to send in multiple requests to the endpoint created after the application was deployed. The script sent in multiple parallel requests to the endpoint. There were 3600 simultaneous requests (concurrent requests) sent to replicate how in real world multiple users are using the application at the same time. We used the ThreadPoolExecutorService, and the max workers in python to achieve that. *The script will be attached to the submitted code.*

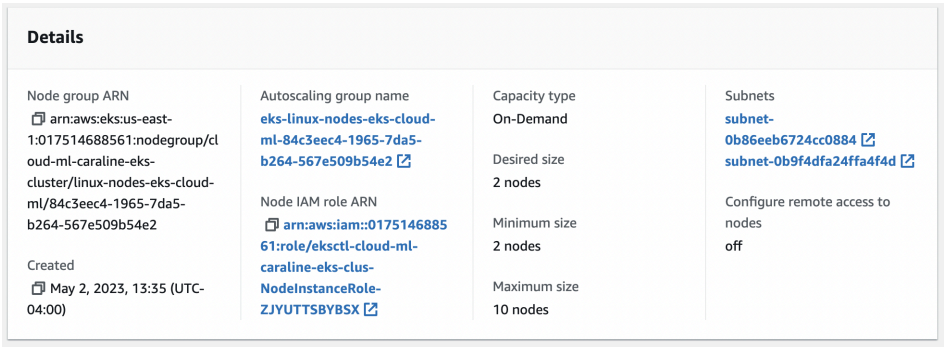


Figure 1: Number of max nodes on the NodeGroup

deployment-controller	Scaled up replica set aws-deploy-cml-7b7fdffb6c to 4 from 1
deployment-controller	Scaled up replica set aws-deploy-cml-7b7fdffb6c to 8 from 4
deployment-controller	Scaled up replica set aws-deploy-cml-7b7fdffb6c to 10 from 8

Figure 2: Scaling of deployments with increasing workload

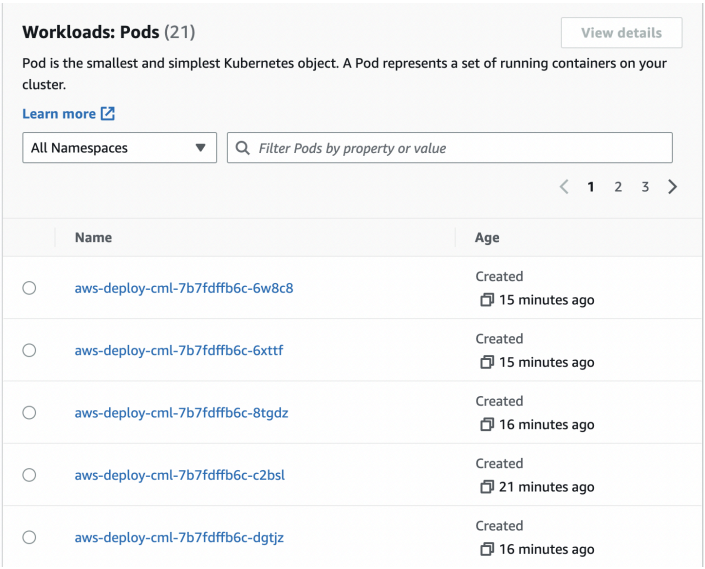


Figure 2: Corresponding scaling of pods across deployments

4.3 GCP

Steps for deployment

7. The first step is to create a separate project in gcloud for managing different containers and the kube cluster. The project can be created through the UI. The project we created was named - news-classification
8. Add in files to gcloud (through the cloud shell). There'll be a train directory with training files. An inference directory with infer files, docker files and a base directory with yaml files.
9. Write the dockerfiles for train and inference, tag and then push it either to docker hub or gcloud container registry.
    a. *docker tag tag-name [HOSTNAME]/[PROJECT-ID]/image-name*
    b. *docker push [HOSTNAME]/[PROJECT-ID]/image-name*
10. Use the pushed images in the yaml files.
11. Create the gcloud kubernetes cluster (Remember to enable Google Kubernetes Engine for the project).

a. *gcloud container clusters create cluster-1 --num-nodes 3 --machine-type g1-small --zone us-central1-c*
b. *gcloud container clusters get-credentials cluster-1 --zone us-central1-c --project [PROJECT_ID]*

12. Update the yaml files to use the docker image that has been pushed. The docker files for training (kind: Job) and inference (kind: Deployment) will be updated.
13. We will use Kustomize here. When there are multiple files, you have to execute them one by one. Kustomize utility helps in customizing raw, template-free YAML files for multiple purposes, leaving the original YAML untouched.
14. Deploy using a single kustomize command
15. The deployed application is: http://34.170.211.169:5000/

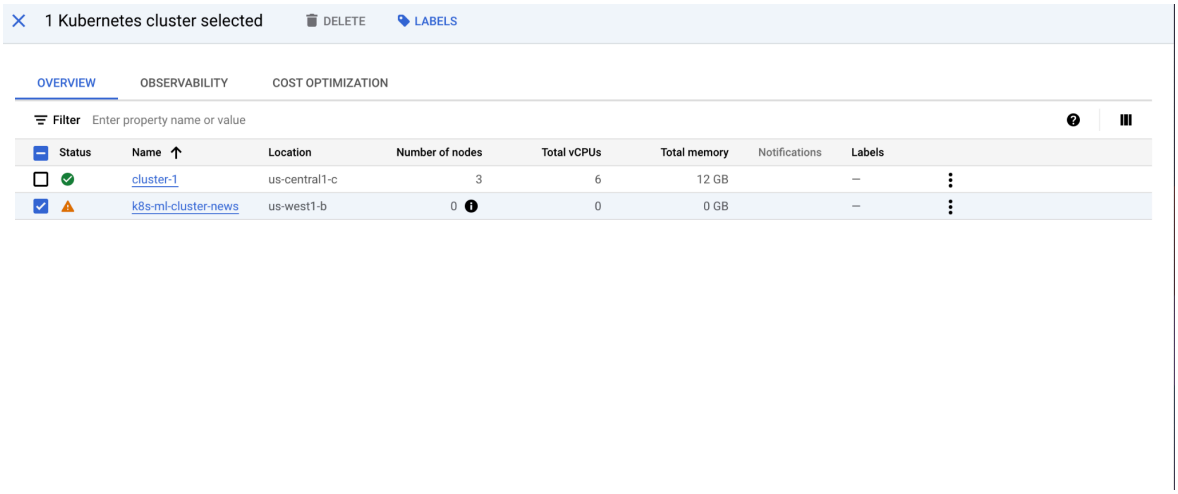


Figure 1: The created Kube cluster in GCloud

```
rm6383@cloudshell:~ (news-classification-385619)$ kubectl get deployment
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
infer   1/1     1            1           16h
rm6383@cloudshell:~ (news-classification-385619)$ kubectl get service
NAME            TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)          AGE
infer-service   LoadBalancer   10.8.147.239   34.170.211.169   5000:31271/TCP   16h
kubernetes      ClusterIP      10.8.144.1     <none>           443/TCP          17h
rm6383@cloudshell:~ (news-classification-385619)$
```

Figure 2: The status of deployment and service on google cloud shell

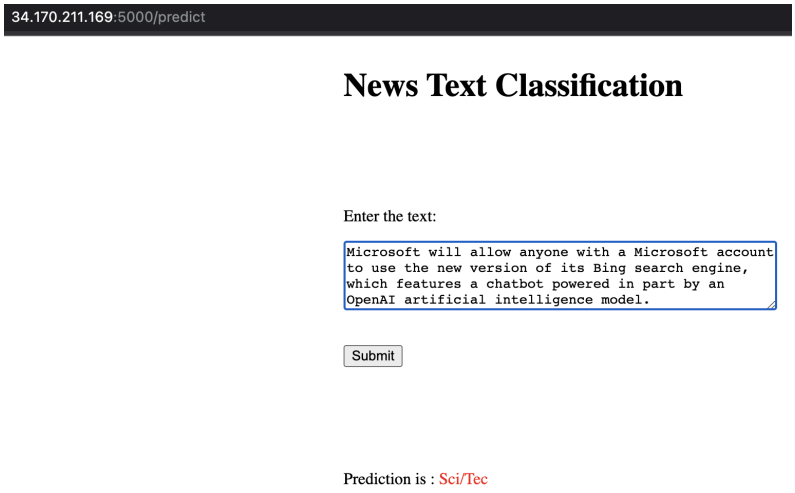


Figure 3: The deployed application and the URL.

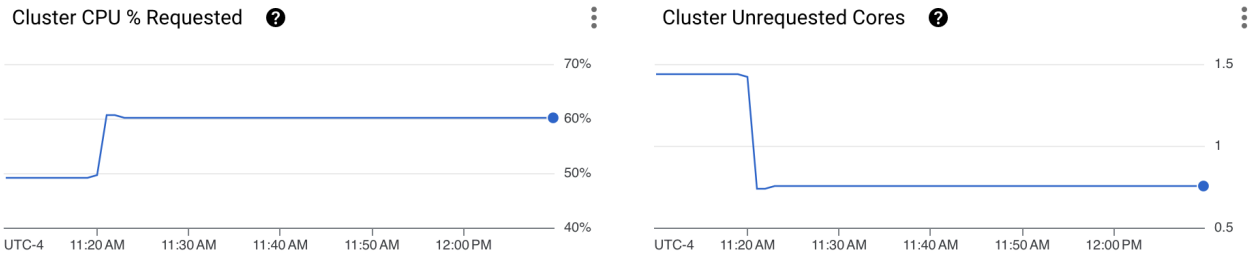


Figure 4: GCP Observability tools

Steps to test Autoscaling in GCP

6. Initially for our cluster we had created three nodes, and by default auto scaling is disabled in GCloud.
7. We wrote a script to send in multiple requests to the endpoint created after the application was deployed. The script sent in multiple parallel requests to the endpoint. There were 3600 simultaneous requests (concurrent requests) sent to replicate how in real world multiple users are using the application at the same time. We used the ThreadPoolExecutorService, and the max workers in python to achieve that. *The script will be attached to the submitted code.*
8. Enabled autoscaling for the cloud kube cluster and added the CPU utilization limit as 30% for the purpose of getting results of this experiment. (The default one is >75%). It tells the autoscaler to collect the CPU utilization of the instances in the group and determine whether it needs to scale. You set the target CPU utilization the autoscaler should maintain and the autoscaler works to maintain that level. (There's a prediction based approach too)

9. After autoscaling was enabled, the script was executed again and the results showed creation of a new node automatically when the set CPU threshold limit was met. One extra node was created.
10. Autoscaling helped with scaling down the number of nodes and instances that were used. After leaving the endpoint unused for 4+ hours, the number of nodes decreased to 2 and the CPU utilization decreased. When the load goes down, the autoscaler does not delete VMs immediately. The autoscaler keeps monitoring capacity needed for the duration of the stabilization period and deletes VMs only when there is sufficient capacity to meet the peak load. This might appear as a delay in scaling in, but it is a built-in feature of autoscaling.
11. After running the script again the node numbers increase from 2 to 3. Then from 3 to 4.

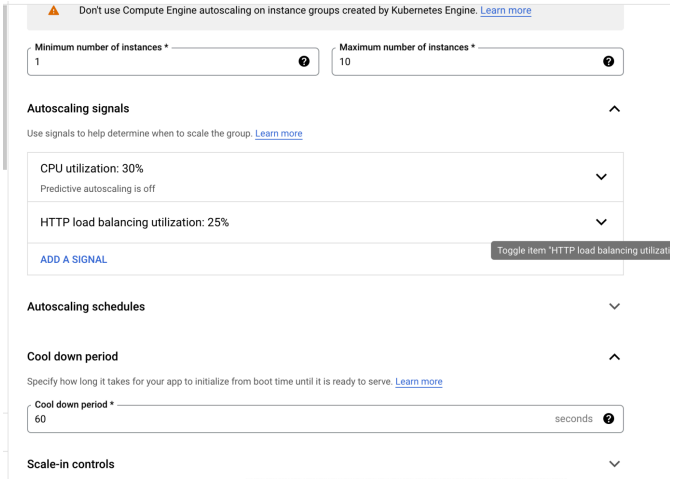


Figure 5: Autoscale configs set up in GCP.

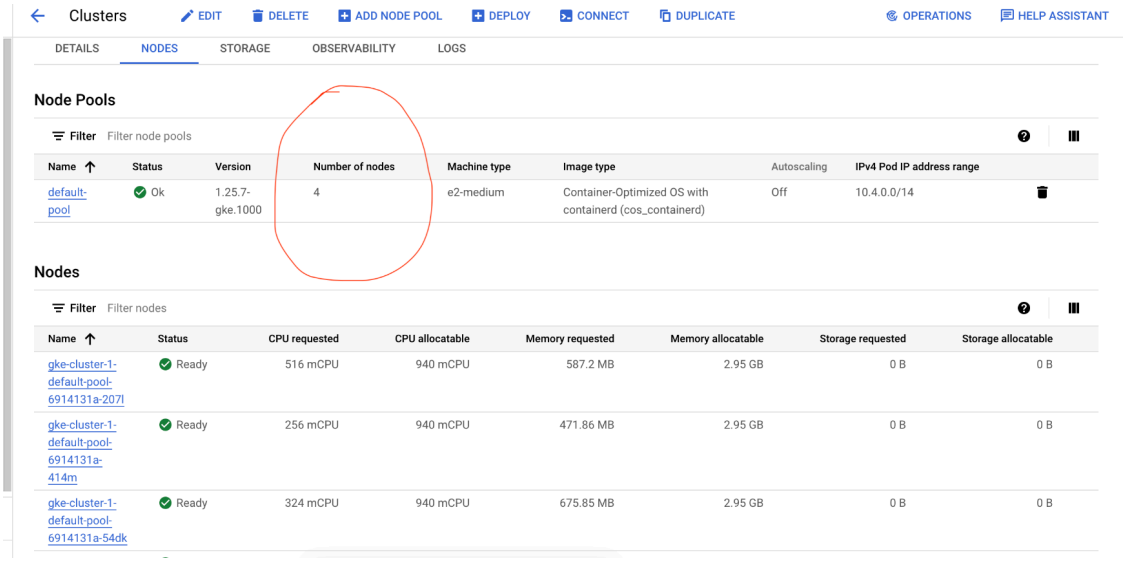


Figure 6: Increase in nodes after load was increased.

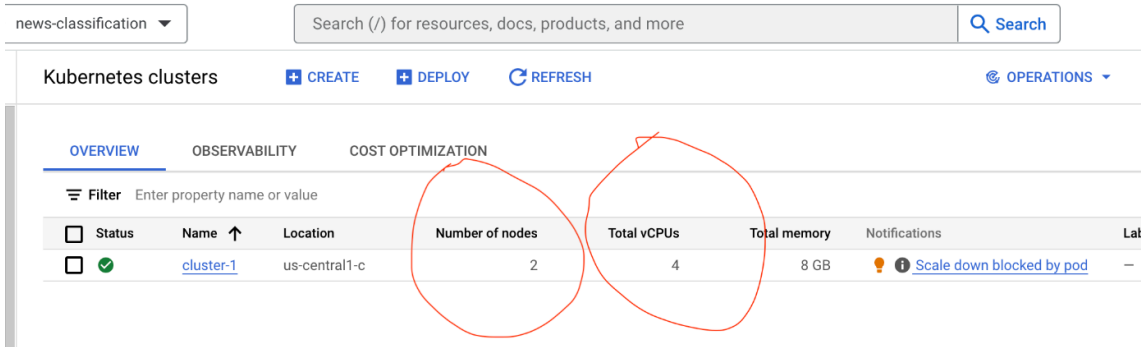


Figure 7: Decrease in number of nodes after endpoint is left unused for 4+ hours

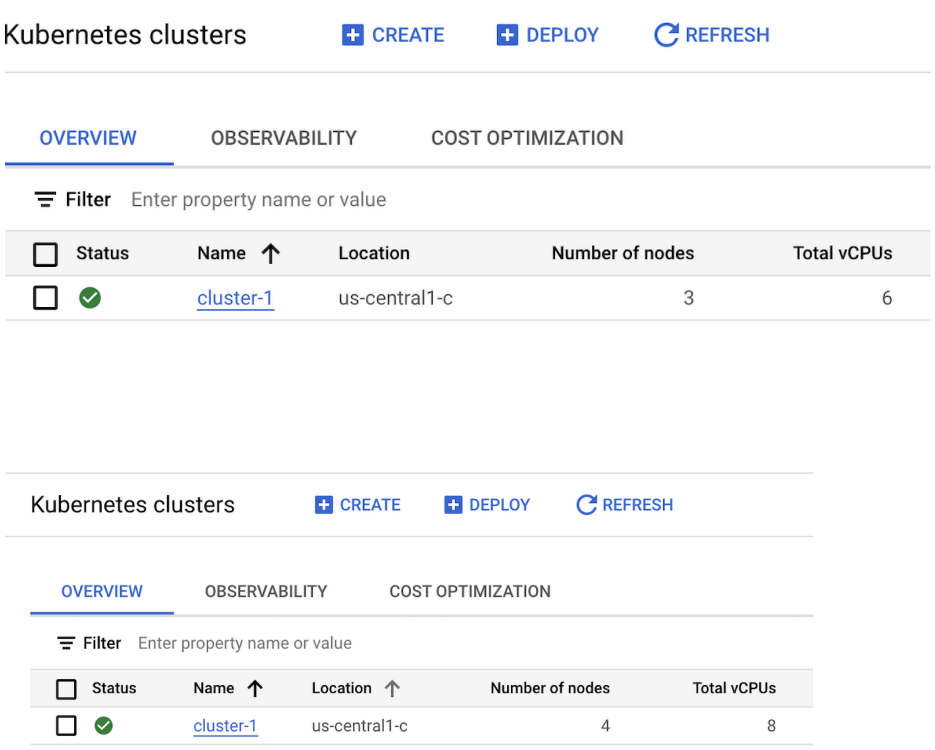


Figure 8: Increase in number of nodes from 2 to 4 (step 6)

5. Comparison (Qualitative Review)

Overall Kubernetes Strategy Remains Consistent

Kubernetes is a powerful tool to automate the deployment, management, and scaling of containerized applications, regardless of which cloud provider is used for resource provisioning. Kubernetes' declarative configuration describes the desired state of applications, and facilitates maintenance to achieve the desired state. As a result, Kubernetes provides a similar workflow across the different cloud providers. The

same Kubernetes YAML files are used on each cloud provider to describe the configuration and desired state of our application.

The core Kubernetes concepts and workflows remain the same across each cloud provider, but there are differences in implementation due to each cloud provider's specific setup.

Differences due to Cloud Provider Environment

GCP

Kubernetes is natively supported by Google Cloud, as it was originally developed by Google. The interface is more intuitively designed to support common Kubernetes workflows for deployment and management. Additionally, the cluster is accessed via GCloud CLI leading to a seamless developer experience. Someone at Google thought about the user experience and how the Kubernetes integration should work. Overall, GCP is our preferred cloud provider for hosting and managing a Kubernetes cluster.

AWS

Setting up the Kubernetes cluster on AWS is more complex than with the other cloud providers. Every object needs to have specific provisioning, and sensible defaults are not automatically installed on the cluster. The majority of time was spent installing defaults and policies that are automatically set up by the other providers. Additionally, the cluster is best accessed through Kubectl commands and Eksctl commands. These differences may be due to the large number of product offerings by AWS; consideration of the user experience with Kubernetes is lacking.

IBM

IBM has strong support for a streamlined setup of the Kubernetes cluster, but the developer community is smaller when compared to GCP and AWS, so documentation of specific features such as autoscaling is lacking. Having both a smaller presence in the market and a smaller community, it lacks some of the more complex integrations and workflows that GCP and AWS support. Namely, autoscaling is implemented via CLI which is less intuitive than with the other providers. The majority of time was spent configuring autoscaling for the cluster.

6. Results

Autoscaling Results

For all three cloud providers we used the same python script with inbuilt threading to generate concurrent requests such that 3600 users were sending 1 million requests to the endpoint to generate the news sentiment. We set the CPU Utilization limit as a

metric to test autoscaling. For GCP we set the utilization threshold as 30% and for IBM and AWS we set it as 10%. When the CPU workload hit the config set for auto scaling we saw that the number of nodes in the kubernetes clusters for all three cloud providers had increased and they kept on increasing as the load increased to the maximum nodes value that was set in the configurations. When the endpoints were left unused and uncalled, we could also see how resource wastage and reduction in cost happened as the number of nodes associated with the cluster decreased.

Costs

With Google, the cost structure is less predictable when compared to the other cloud providers. Expect to pay hourly for both compute and bandwidth utilized, and a monthly management fee. If costs are critical to your organization then it will be important to forecast expected use of the cluster. With AWS, expect to pay a flat-rate hourly fee based on the selected instances. Additionally, there is an hourly fee to enable EKS, which is similar to a management fee. With IBM, the cost structure is similar to AWS, but there is no management fee.

While it is not possible to select exactly the same instance type in a Kubernetes cluster across cloud providers, we provide an estimate for comparable instances. The minimum requirement across all three providers is for two nodes, one master node and one worker node, which is factored into the expected monthly prices. If there is a management fee, it is also reflected in the prices below.

	Instance Type	**vCPU**	**Memory (GB)**	**Monthly Cost ($)**
GCP	e2.medium	4	8	256
AWS	t2.xlarge	4	16	339
IBM	bx2.4x16	4	16	388

The table provides the minimum cost comparison based on the assumption that the cluster will have constant use for every hour of the month. It is important to keep in mind that the costs of GCP will increase if the demands of the cluster are high, given the flexible pricing scheme. Regardless of the variable costs, we still consider GCP as our preferred cloud provider for hosting Kubernetes-managed applications due to the ease of developer experience.

7. Conclusion

In conclusion, this project provided exploration into ML model deployment on three major cloud providers: Google, AWS, and IBM, all while leveraging Kubernetes. Through review of the metrics and logging visualizations on each cloud provider's UI, we evaluated the deployment, CPU utilization, and how the system managed resource-demanding loads.

Our study selected CPU utilization as the key metric for horizontal autoscaling, influencing the number of nodes associated with the cluster. We compared the experience of managing our Kubernetes deployment across the different cloud providers. Each provider has unique benefits and limitations, but all demonstrated the consistent strength and flexibility of cloud-based, ML application deployment and management with Kubernetes.

8. References

1) Kubernetes Cluster Autoscaler,*Last accessed on May 09, 2023 from* https://www.kubecost.com/kubernetes-autoscaling/kubernetes-cluster-autoscaler/
2) 5 Benefits of Horizontal Autoscaling in Kubernetes ,*Last accessed on May 09, 2023 from* https://medium.com/mindboard/5-benefits-of-horizontal-autoscaling-in-kubernetes-c95a0046dfab
3) TorchText*, Last accessed on May 09, 2023 from* https://pytorch.org/text/stable/index.html
4) Enabling the cluster autoscaler add-on in your cluster*, Last accessed on May 09, 2023 from* https://cloud.ibm.com/docs/containers?topic=containers-cluster-scaling-install-addon
5) Deploying machine learning models on Kubernetes*, Last accessed on May 09, 2023 from* https://cloud.google.com/community/tutorials/kubernetes-ml-ops
6) Autoscaling groups of instances*, Last accessed on May 09, 2023 from* https://cloud.google.com/compute/docs/autoscaler
7) Comparing the Top Eight Managed Kubernetes Providers*, Last accessed on May 09, 2023 from* https://medium.com/@elliotgraebert/comparing-the-top-eight-managed-kubernetes-providers-2ae39662391b
8) Kubernetes*, Last accessed on May 09, 2023 from* https://kubernetes.io/docs/home/
9) Text Classification With the Torchtext Library*, Last accessed on May 09, 2023 from* https://pytorch.org/tutorials/beginner/text_sentiment_ngrams_tutorial.html