Performance Analysis of Image Generative Models on leading Cloud Platforms

Abstract: The use of image generative models has grown significantly in recent years, with applications ranging from image generation and style transfer to image super-resolution and object detection. To support the deployment of these models in production environments, many cloud platforms now offer Kubernetes services for container orchestration. In this paper, we perform a performance analysis of image generative models on different Kubernetes services provided by popular cloud platforms. We compare the performance of models such as Generative Adversarial Networks (GANs) and Deep Convolutional Generative Adversarial Networks (DCGANs) on these platforms and evaluate factors such as model accuracy, inference time, and cost. Our results provide insights into the trade-offs and considerations for selecting a Kubernetes service for deploying image-generative models in the cloud.

CCS Concepts: • **Generative Adversarial Networks**; • **Machine Learning**; • **Unsupervised Learning**; • **Kubernetes**;

Additional Key Words and Phrases: Docker Images, deployment, containers, Google Kubernetes Engine (GKE), IBM, ContainerD

1 INTRODUCTION

The use of image generative models has grown significantly in recent years, with applications ranging from image generation and style transfer to image super-resolution and object detection. To support the deployment of these models in production environments, many cloud platforms now offer Kubernetes services for container orchestration. In this paper, we perform a performance analysis of image generative models on different Kubernetes services provided by popular cloud platforms. We compare the performance of models such as Generative Adversarial Networks (GANs) and Deep Convolutional Generative Adversarial Networks (DCGANs) on these platforms and evaluate factors such as model accuracy, inference time, and cost. Our results provide insights into the trade-offs

Author's address:

and considerations for selecting a Kubernetes service for deploying image-generative models in the cloud.

1.1 GANs

GANs (Generative Adversarial Networks) is a type of deep learning model that is used to generate synthetic data. They are made up of two neural networks, a generator, and a discriminator, that are trained at the same time. The generator generates synthetic data, while the discriminator attempts to distinguish it from real data.

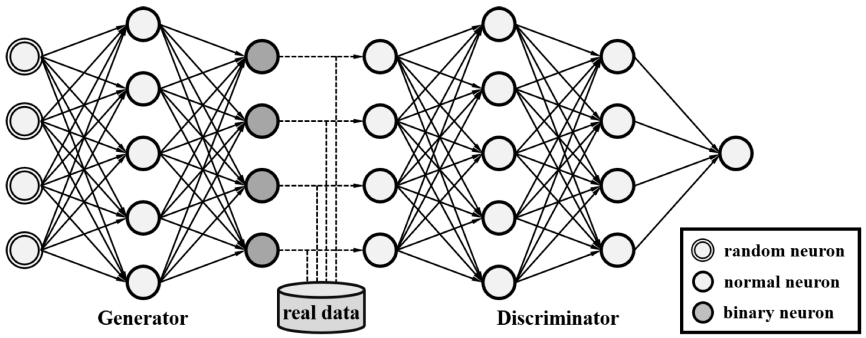


Fig. 1. GAN Architecture

The two networks are trained in an adversarial manner, with the generator attempting to generate data that the discriminator cannot distinguish from real data and the discriminator attempting to correctly classify the data as either real or synthetic. As the training progresses, the generator becomes more adept at generating realistic synthetic data, and the discriminator becomes more adept at distinguishing real data from synthetic data. GANs have been used for a variety of tasks, including generating images, text, and audio.

1.2 DCGANs

Deep Convolutional Generative Adversarial Networks (DCGANs) are a type of GAN in which convolutional neural networks are used as both the generator and the discriminator. Convolutional neural networks are especially well-suited for image generation tasks because they can effectively learn and capture image spatial structure.

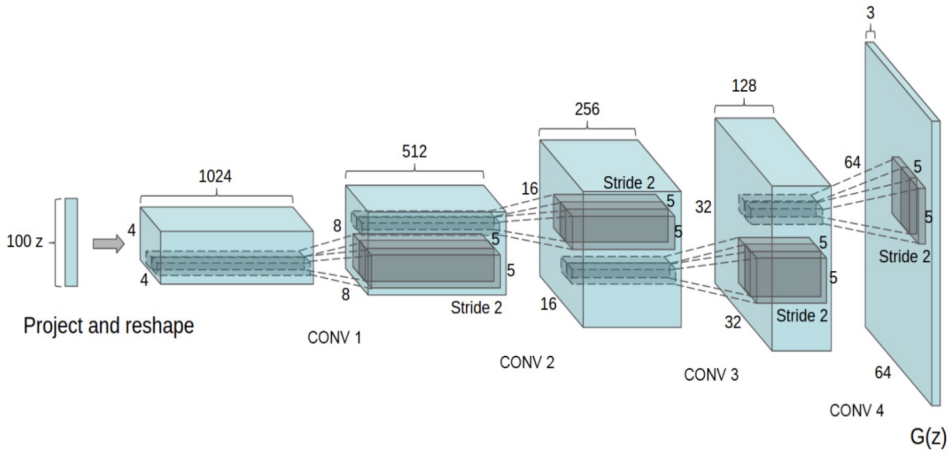


Fig. 2. DCGANs Architecture

The generator network in a DCGAN is made up of a series of transposed convolutional layers that upsample an input noise vector to generate an image. A series of convolutional layers in the discriminator network downsample the input image and classify it as real or synthetic. As with regular GANs, the generator and discriminator are trained concurrently, with the generator attempting to generate realistic images and the discriminator attempting to correctly classify the images as real or synthetic.

DCGANs have been used to generate realistic images of faces, animals, and other objects, among other things. They've also been used for image style transfer and image super-resolution, where the goal is to change an existing image in a specific way.

2 RELATED WORK

There has been a growing interest in the use of image generative models on cloud platforms, as these models have the potential to enable a wide range of applications, such as image synthesis, image style transfer, and image super-resolution. However, the performance of image-generative models on cloud platforms has not been extensively studied, particularly in the context of Kubernetes services.

Several studies have analyzed the performance of machine learning models on cloud platforms, but most of these studies have focused on traditional machine learning models,

such as support vector machines and decision trees, rather than image generative models. Some studies have analyzed the performance of image classification models on cloud platforms, but these models are typically less computationally intensive than image generative models, which makes them less suitable for comparison.

A few studies have analyzed the performance of image-generative models on cloud platforms, but these studies have typically focused on a single cloud platform, rather than comparing the performance of different platforms. For example, one study analyzed the performance of GANs [8], while another study analyzed the performance of DCGANs on GCP.

To the best of our knowledge, there have been no comprehensive studies that have analyzed the performance of image-generative models on multiple cloud platforms in the context of Kubernetes services. This is the main contribution of our study.

3 IMPLEMENTATION

For the implementation part, we first started with modeling the GAN and DCGAN models. We divided the code into two different files, one for training the model and storing it and the other for using the stored model to test it and generate images. We used two image datasets, CIFAR-10 and MNIST separately for training our model. For our cloud platforms, we use the Kubernetes services provided by IBM Kubernetes, Google Kubernetes Engine, and Amazon Elastic Kubernetes Service. We created and deployed images on an online docker registry called docker hub and then used those images to perform our training and testing on Kubernetes engines.

3.1 DATASET

Machine learning practitioners frequently use the MNIST (Modified National Institute of Standards and Technology) dataset [3], particularly for image categorization. It comprises of a test set of 10,000 photos and a training set of 60,000 grayscale images of handwritten numbers (0–9). There are 784 total pixels in each image, which is 28 by 28 pixels. The photos have been preprocessed to guarantee that the pixel values are between 0 and 1, which is known as normalization.

Due to its relative simplicity, while still being difficult enough to necessitate the employment of machine learning techniques, the MNIST dataset serves as a benchmark for

many machine learning algorithms. It is frequently used as a foundation for the creation of more sophisticated picture classification algorithms.

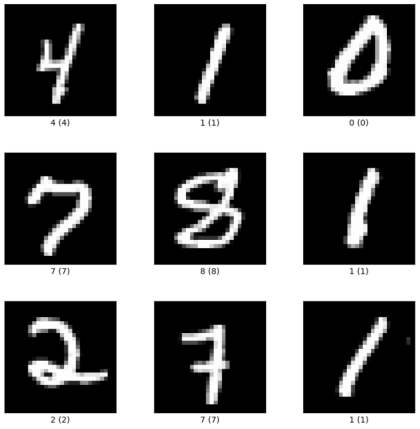


Fig. 3. MNIST Images.

The fact that the MNIST dataset has been extensively analyzed and used means that individuals who are interested in using it can access a variety of information and resources. The dataset is also simple to use and download, making it available to researchers and students just starting out in the field of machine learning.

The dataset of raw images known as CIFAR-10 (Canadian Institute For Advanced Research) is utilized for image classification tasks. It has 10 classes, 60,000 32x32 color training images, and 10,000 test images (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck). The photos are preprocessed to ensure that the pixel values are between 0 and 1 after being obtained from numerous sources, including the internet.

The machine learning community frequently uses the CIFAR-10 dataset as a benchmark for image classification techniques. It is a better choice for evaluating the effectiveness of machine learning models on real-world data because it has a more complicated dataset than MNIST and more varied and realistic images.

The small (32x32 pixel) and potentially noisy images in the CIFAR-10 dataset [5] present one of the dataset's issues because it can be challenging for machine learning models to appropriately categorize the images. It is therefore a useful dataset for creating and testing more sophisticated image categorization methods.

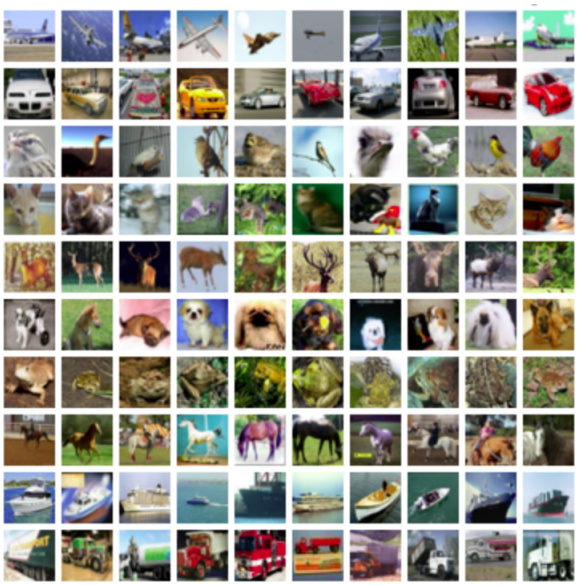


Fig. 4. CIFAR-10 Images.

3.2 MODEL ARCHITECTURE

Generative adversarial models (GANs) are a type of neural network architecture that are used for generating new, synthetic data that is similar to a training dataset. They consist of two main components: a generator network and a discriminator network.

The discriminator network, on the other hand, is responsible for distinguishing between real and synthetic data. It takes a data sample as input and processes it through a series of layers to output a probability that the sample is real. The discriminator is trained to correctly classify real data as real and synthetic data as synthetic.

During training, both the generator and discriminator networks are optimized simultaneously. One main challenge during GAN training is that the training process has to maintain a kind of balance in training the generator and discriminator. This is because the training of a discriminator is relatively easier as its job is only to classify into 2 classes: Fake or real. Now this means that if the discriminator's loss becomes 0, the generator will not be able to learn how to produce more realistic images.

The generator network is responsible for generating new data samples. It takes a random noise vector as input and processes it through a series of layers to produce a synthetic

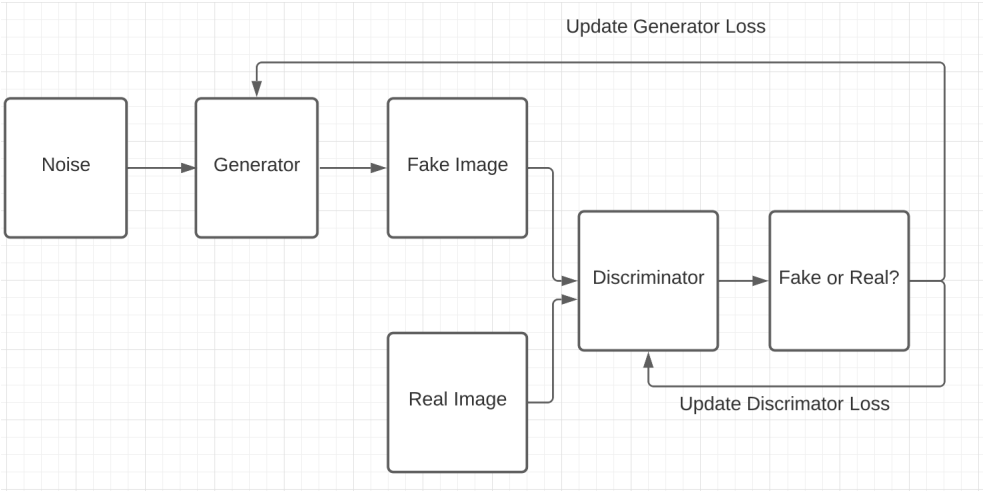


Fig. 5. GAN Architecture

data sample. The generator is trained to produce data that is similar to the training data, so it tries to "fool" the discriminator network into believing that the synthetic data is real.

Each generator block in the GAN architecture our model uses has there are following 3 layers: Linear layer followed by batch normalization and ReLu activation. The final layer consists of a Linear layer followed by a sigmoid.

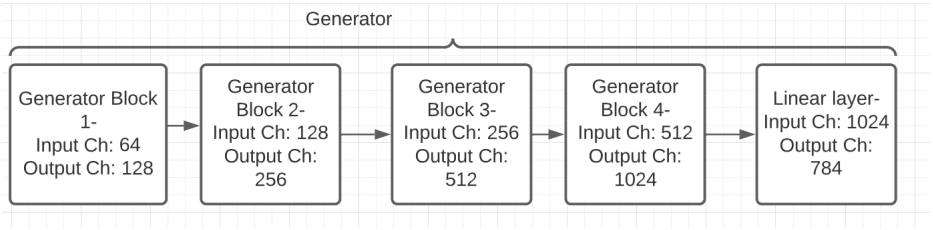


Fig. 6. GAN Generator

The discriminator network, on the other hand, is responsible for distinguishing between real and synthetic data. It takes a data sample as input and processes it through a series of layers to output a probability that the sample is real. The discriminator is trained to correctly classify real data as real and synthetic data as synthetic.

Each discriminator block in the GAN architecture our model uses there are following 2 layers: Linear layer followed by Leaky ReLu activation. The final layer consists of a Linear layer.

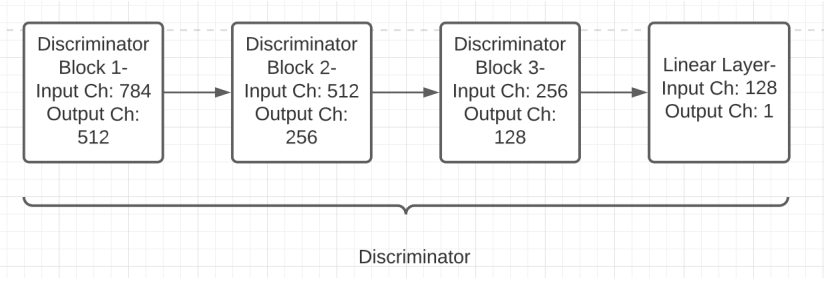


Fig. 7. GAN Discriminator

Deep convolutional generative adversarial networks (DCGANs) are a variant of GANs that are specifically designed to generate synthetic images and use convolutional layers instead of Linear layers. Following is the architecture of the DC-GAN generator and discriminator we have used for our models:

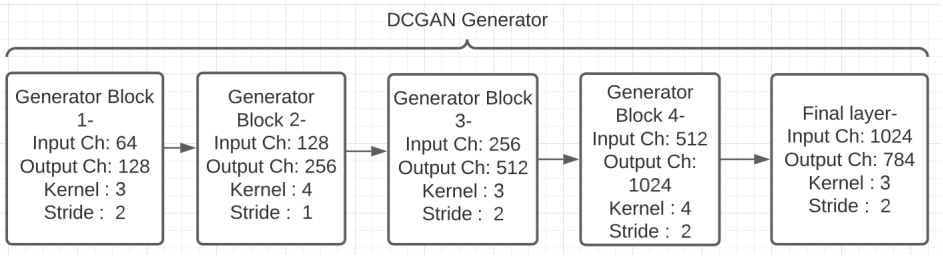


Fig. 8. DC-GAN Generator

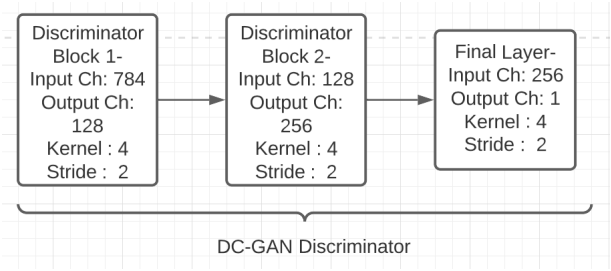


Fig. 9. DC-GAN Discriminator

3.3 DOCKER

A tool called Docker is intended to make it simpler to develop, distribute, and run applications using containers. A program's runtime, libraries, dependencies, and application code are all contained in a lightweight, independent, executable package known as a container.

Developers may quickly distribute and deploy their apps and dependencies on any system that has Docker installed by packaging them using Docker. As a result, there is no longer a need to worry about variations in development environments or system settings, which makes it simpler to create and deploy applications.

Fig. 10. DCGAN Images Repository on Dockerhub

Docker containers are made using Docker images as templates. They include metadata like the container's name and version, the application code, and dependencies. Each layer that makes up a Docker image [4] represents a modification or update to the image. Due to the fact that only the layers that have changed need to be transferred when updating or sharing an image, Docker images may now be built and shared efficiently.

A Docker registry, which is a repository for storing and sharing Docker images, is where Docker images can be kept. The availability of numerous public Docker registries, like Docker Hub, makes it simple to find and use pre-built images. To store and exchange pictures within their company, developers can also design and manage their own private Docker registries.

For our implementation for each scenario, we create one docker image for testing and training. For our docker images, we used the prebuilt image of PyTorch and built our required image on it. Initially, we created the docker images on our local system and then deployed them to the DockerHub registry so that they would be publicly available and can

be pulled into our Kubernetes project. The same images were used for all different cloud platforms. We performed the above steps for all different scenarios with two training models(GAN and DCGANs) and image datasets (MNIST and CIFAR-10).

3.4 CONTAINER RUNTIMES

Container runtimes are responsible for executing containers. The lifespan of a container, including starting, halting, and cleaning up containers, is managed by a container runtime.

Some of the most common runtime environments used are Docker and ContainerD [2]. These runtimes provide the infrastructure needed to manage containers, allowing users to launch containers, pull images from a registry, and monitor their health.

For our implementation, we used containerD as the runtime for our containers. One of the primary advantages of using container runtimes is that they enable applications to be packaged and deployed as self-contained units, making it easier to run the same application consistently across multiple environments. This is especially useful in cloud environments, where applications may need to be distributed across multiple servers or regions.

4 EXPERIMENTS

In this project, we have developed two models on two datasets each:

- `IBM KUBERNETES SERVICE`: trained GAN and DCGAN models on MNIST and CIFAR-10 datasets and generated images using docker containerization on IBM Kubernetes service. IBM Kubernetes cloud cluster has 8 vCPUs of 16GB memory each.
- `GOOGLE KUBERNETES ENGINE`: trained GAN and DCGAN models on MNIST and CIFAR-10 datasets and generated images using docker containerization on Google Kubernetes Engine. Google cloud cluster has 8 vCPUs of 16GB memory each.
- `AMAZON ELASTIC KUBERNETES SERVICE`: trained GAN and DCGAN models on MNIST and CIFAR-10 datasets and generated images using docker containerization on Amazon Elastic Kubernetes service. Our cloud cluster have 8 vCPUs of 16GB memory each.

- `LOCAL M1 MAC`: trained GAN and DCGAN models on MNIST and CIFAR-10 datasets and generated images using docker containerization on Local M1 Mac Machine. Local M1 Mac has 8 cores CPU of 16GB memory each inclusive of 4 high-performance cores. We ensured to use of these environments at the same specifications to have the compute resources constant for the performance comparison.

4.1 METHODOLOGY

Several steps are involved in training a Generative Adversarial Network (GAN) and DCGAN on a Kubernetes cluster:

- `Creating a cluster on cloud provider`: To begin, you must create a Kubernetes cluster on which to run your training of both models. Typically, this entails installing and configuring the Kubernetes software on a cluster of machines known as nodes.
- `Deploy Docker Image`: Prepare a docker image for both your training and generating new images. The Docker image would be containing the code and any dependencies that are required to train and test the models.
- `Create Persistent storage`: Because GAN training generates a large amount of data, you'll need to configure persistent storage for your training data and model checkpoints. This can be accomplished by using a persistent volume claim (PVC) in your Kubernetes cluster. Here we use 20GB as the storage capacity for our cluster and it can be managed and updated later as per requirements.
- `Create a Kubernetes training pod`: Once your code is prepared and your storage is set up, you can create a Kubernetes training pod to manage the process of training your GAN and DCGAN models.
- `Kubernetes service`: Next we create a service on our cluster that will export our pods as a service on a network. Our deployment pod while running will listen to TCP port 5000.
- `Run the Deployment pod`: Lastly we will run the deployment pod which we use the model created during the train and generate images. These images created are then stored in a location specified. The deployment pod will listen at TCP port 5000.

- `Built a frontend for User Interaction`: We also built a webpage to display our functionalities and help users interact and compare images generated from clusters created on different Cloud Platforms. We can compare the image quality (naked eye) and also compare the training and generating image time.

4.2 YAML

YAML (YAML Ain't Markup Language) is a human-readable data serialization language that is used to represent complex data structures in a format that is simple to read and write. It is commonly used to store configuration files, but it can also store structured data. YAML is designed to be easy to read and write, with a syntax reminiscent of programming languages and a structure very similar to JSON. It uses indentation to define the data structure and key-value pairs to represent data elements.

deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mydeployment
  labels:
    app: guru_app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: guru_app
  template:
    metadata:
      labels:
        app: guru_app
    spec:
      containers:
        - name: mydeployment
          image: gurmehrsohi/sohi_test
          ports:
            - containerPort: 5000
          volumeMounts:
            - name: myvolume
              mountPath: /training_model
      volumes:
        - name: myvolume
          persistentVolumeClaim:
            claimName: gsohi
```

Fig. 11. Deployment YAML for DCGAN

For our implementation, we use YAML files to give configurations to the Kubernetes clusters that we have created and are running on the cloud platforms. We define four different YAML files

- `pvc.yaml`: To create persistent volume storage for our clusters. We define 20GB for the storage and a directory to be mounted on this storage.
- `training.yaml`: To create and run a training pod inside the cluster and then save the trained model to the directory created on a volume. This model will be later used to generate images. Our training.YAML file uses the train images present on the docker hub to load libraries and the training code. This model training code is then run on each of the clusters.
- `service.yaml`: This YAML file helps create a service for our cluster which will link our backend to a template frontend that we create for user usage. The backend will create a new image every time the POST method is called. For our project, we use 5000 as our TCP port for getting responses from our backend.
- `development.yaml`: This will create a deployment pod to run our flask application which will be used to generate new images from our model which is stored inside our volume storage.

4.3 KUBECTL, DASHBOARD

As your GAN training progresses, you'll want to monitor its progress and potentially scale up or down the number of nodes you're using. You can use Kubernetes tools such as kubectl and the Kubernetes dashboard to do this. Kubectl is a command-line tool used to manage and deploy applications on a Kubernetes cluster [1]. Kubernetes is an open-source container orchestration system that automates the deployment, scaling, and management of containerized applications. kubectl allows you to run commands against Kubernetes clusters to deploy applications, view the status of your deployments, and view and troubleshoot your applications. It is an essential tool for anyone working with Kubernetes.

Some of the kubectl commands we used.

- `kubectl get pods`: list all pods in the current namespace

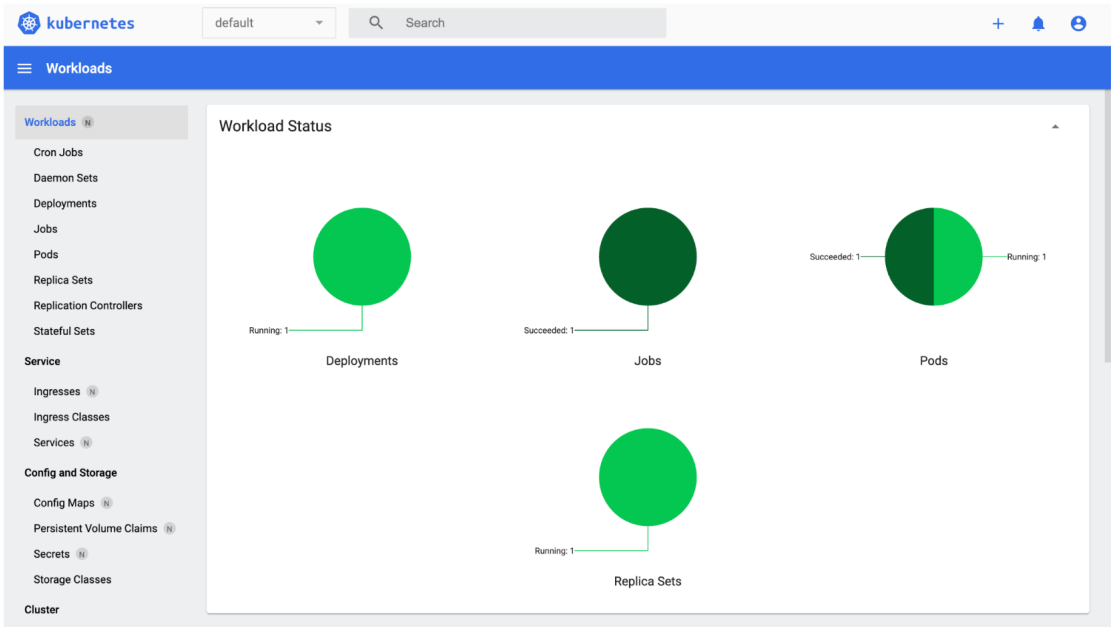


Fig. 12. Live Dashboard Snapshot of DCGAN deployment on IBM Kubernetes Cluster

- `kubectl apply -f <file.yaml>`: apply the configuration in a YAML file to the Kubernetes cluster
- `kubectl logs <pod-name>`: show all the logs related to the pod
- `kubectl pod describe`: list all pods in the current namespace

Kubernetes dashboard is used to visualize and see logs of parts to the clusters like nodes, pods, services, and storage. The Kubernetes dashboard is a web-based management and monitoring interface for Kubernetes clusters and applications. It displays a graphical representation of your deployment's clusters, nodes, pods, and containers, as well as their resources and resource usage. The dashboard allows you to deploy and scale applications, view and manage cluster resources, and troubleshoot application issues. You must have a Kubernetes cluster up and running to access the Kubernetes dashboard. If you don't already have a cluster, you can create one through a managed Kubernetes service like Google Kubernetes Engine (GKE), Amazon Elastic Container Service for Kubernetes (Elastic Kubernetes Service), or IBM Kubernetes service. For our implementation we

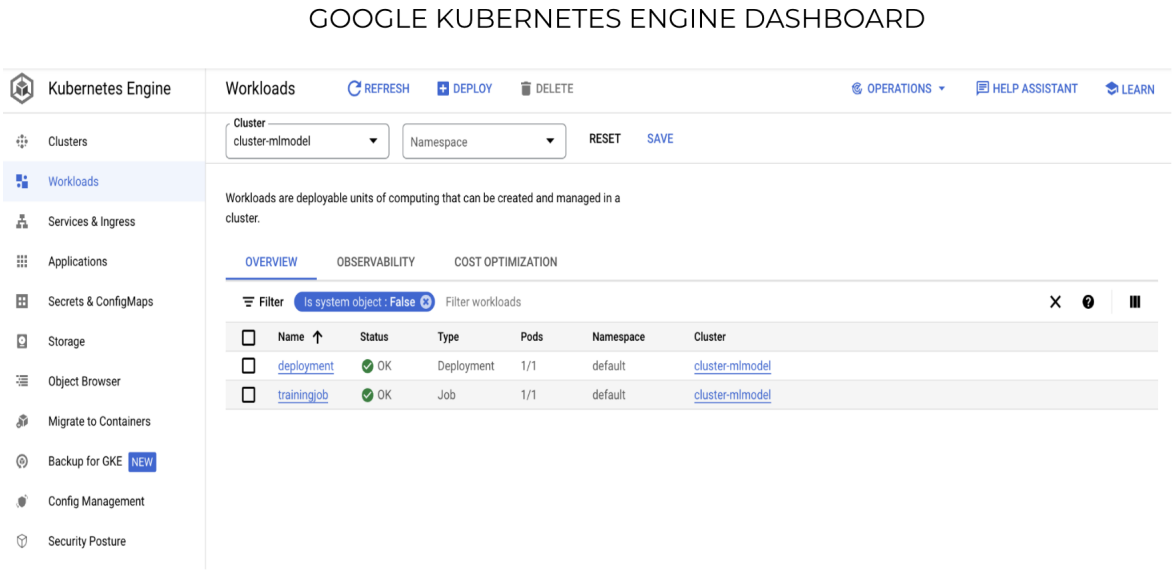


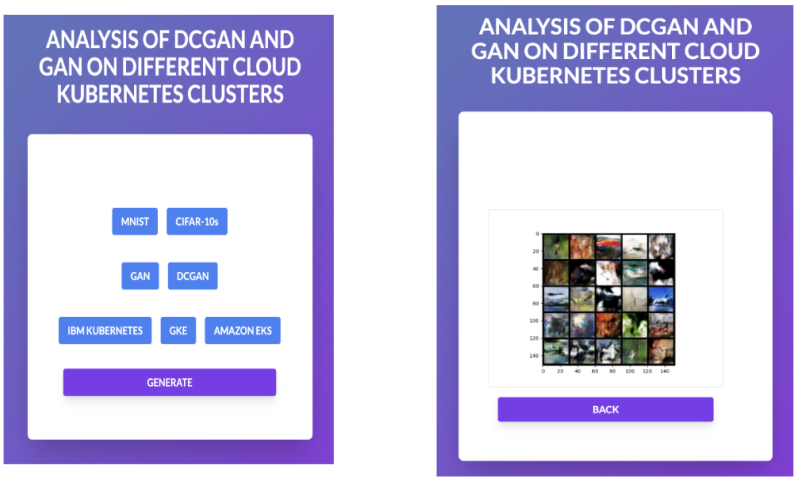
Fig. 13. Live Dashboard Snapshot of DCGAN deployment on Google Kubernetes Cluster

created a cluster for each scenario and the figures here can show how our dashboard looks for IBM and Google Kubernetes engines.

4.4 FLASK AND UI

For creating the web page we use Flask as it is one of the most popular Python Frameworks to build web applications [6]. Using Flask we create a POST endpoint for our application to send a request with the required selections: which dataset to use, model to train, and Cloud Platform to run our application. After sending a POST request we get the response which is a generated image from our trained model. We use the image and the training and testing time and memory to make our analysis i.e. comparisons.

The web page is simple and user-intuitive. It takes the dataset, model and the cloud service as the inputs to run the pre-trained model on the cloud cluster. After running the pre-trained model on the cloud, it displays the generated image with the help of an API on the web page as seen in Figure 14.



(a) Front-end to select dataset, model and environment

(b) Generated Image

Fig. 14. Frontend using Flask

5 EVALUATIONS AND RESULTS

In this section, the model performance across all three training environments is discussed. We have GAN, and DCGAN models running on CIFAR10 and MNIST datasets. We ran these models on Google Kubernetes Service, IBM Kubernetes Service, and Local M1 Mac (16GB RAM).

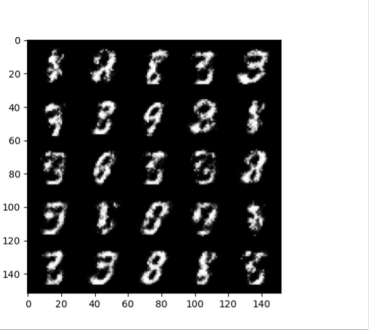


Fig. 15. GAN model running on MNIST dataset

The generated images are reported environment independent as all the generated for the same model on a given dataset was having similar results across all three training

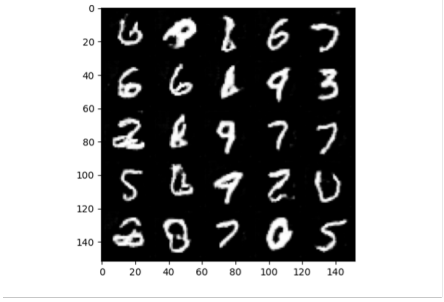


Fig. 16. DCGAN model running on MNIST dataset

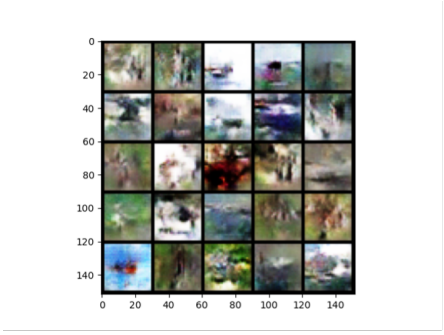


Fig. 17. GAN model running on CIFAR10 dataset

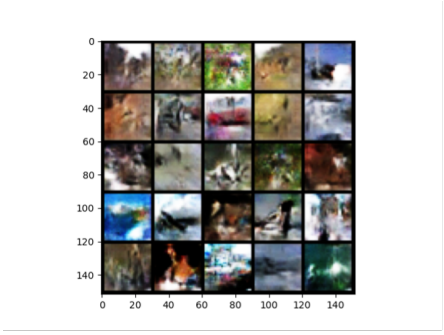


Fig. 18. DCGAN model running on CIFAR10 dataset

environments - IBM Cloud Kubernetes Service, Google Cloud Kubernetes Engine, and M1 Mac - all having 8 vCPUs with 16GB memory each.

The major distinguishing parameter across all three training models was the training time required to train the model (on a given dataset). This factor is well documented in

Figure 19 which compares the metric-average training time per epoch across all three training environments for all the models tested.

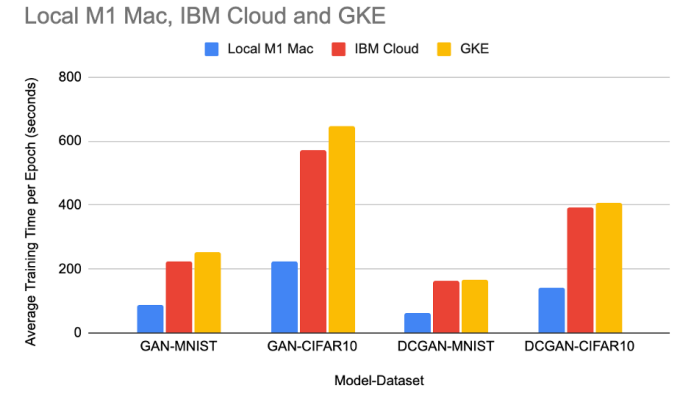


Fig. 19. Average Training Time Per Epoch for GKS, IBM Cloud, and Local M1

6 INFERENCES

- All three training environments generated images of similar sharpness for GAN and DCGAN after 50 epochs.
- Per epoch training time trend stays consistent across all 3 environments (GKS, IBM Cloud, Local M1).
- The average training time required per epoch trend is observed as, Google Kubernetes Engine (GKS) > IBM Kubernetes Service > Local M1 Mac
- CIFAR10 models took longer to train than MNIST models as the CIFAR10 dataset has 3 channels whereas the MNIST dataset has 1 channel.
- Local M1 Mac (8 cores CPU, 16GB memory) took the least time owing to its 4 high-performance cores whereas GKS and IBM Kubernetes had only 8 vCPU cores.
- IBM Kubernetes Service performed better than Google Kubernetes service by 15 percent. The IBM Kubernetes Service performed better than GKS due to its higher IOPS in storage than GKS. IBM Kubernetes Service have 48K IOPS in contrast to maximum 30K storage IOPS in GKS [7]. Since Generative Adversarial networks are memory intensive with all intermediate epochs storing data in the storage, a

higher storage IOPS leads to lesser total training time which potentially explains 15 percent lesser per epoch training time in IBM Cloud than GKS .

7 CONCLUSION AND FUTURE WORK

- In the future, we can expect to see continued growth in the use of Kubernetes for running generative models in production. This is likely to be driven by the increasing adoption of containerized applications and the growing need for robust and scalable infrastructure to support them.
- One of the main challenges in running image-generative models on Kubernetes is ensuring that the models can handle large amounts of data while remaining efficient. Researchers are developing techniques for improving the performance of image-generative models on Kubernetes, such as the use of hardware accelerators such as GPUs.
- There is some research going on these days to integrate these image-generative models with data visualization tools like a Tableau.
- Also in the future, we can add autoscaling as a feature to our clusters that are currently running. Autoscaling can help assign and reassign resources to the clusters according to load. This can help to ensure that the models have enough resources to handle the workload while minimizing cost.
- Overall, the use of Kubernetes for running generative models is likely to become increasingly common in the future, as it offers a powerful and flexible platform for deploying and managing these types of models in production.

REFERENCES

[1] Command line tool (kubectl). *Command line tool (kubectl)*. URL: https://kubernetes.io/docs/reference/kubectl/.

[2] Containerd. *Containerd*. URL: https://github.com/h-mehta/GANs.

[3] Li Deng. "The mnist database of handwritten digit images for machine learning research". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142.

[4] Docker. *Develop with Docker*. URL: https://docs.docker.com/develop/.

[5] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. "CIFAR-10 (Canadian Institute for Advanced Research)". In: (). URL: http://www.cs.toronto.edu/~kriz/cifar.html.

[6] Pallets. *interfaces of Flask*. 2010. URL: https://flask.palletsprojects.com/en/2.2.x/api/.

[7] Principled Technologies. *Public cloud infrastructure comparison: IBM Cloud vs. Google Cloud*. 2019. URL: https://www.principledtechnologies.com/IBM/Cloud-vs-Google-Cloud-research-0419-v2.pdf.

[8] Abraham Wu and Rudi Stouffs. *Generative Adversarial Networks in the built environment: A comprehensive review of the application of GANs across data types and scales*. 2022. URL: https://www.researchgate.net/publication/362754176_Generative_Adversarial_Networks_in_the_built_environment_A_comprehensive_review_of_the_application_of_GANs_across_data_types_and_scales.