Building and Evaluating a Prototype Edge Computing System

by
Arnav Bhutani

A THESIS

submitted to

Oregon State University

Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Associate)

Presented June 1, 2021
Commencement June 2021

AN ABSTRACT OF THE THESIS OF

Arnav Bhutani for the degree of Honors Baccalaureate of Science in Computer Science presented on June 1, 2021. Title: Building and Evaluating a Prototype Edge Computing System.

Abstract approved:_____

Scott Fairbanks

As fifth generation telecommunications equipment becomes more viable and reliable, demand for high-speed, low-latency, viewpoint specific data analysis is expected to dramatically increase. Systems such as self-driving cars, traffic cameras, warehouses and other commercial buildings will be using fifth generation telecommunications to form 'smart cities', driving demand for the edge. The purpose of this project is to create a proof-of-concept edge network that can deploy various high intensity machine learning and AI tasks, from image recognition to algorithm generation. A system leveraging Kubernetes was shown to work using both the cloud and the edge, however true comparisons cannot be made, as there is a lack of data, and various issues that still have to be tested. This project can serve as a pretested model that will work on both the edge and the cloud for a future comparison.

Key Words: Edge Computing, 5g, Cloud Computing, Kubernetes

Corresponding e-mail address: Bhutania@oregonstate.edu

Building and Evaluating a Prototype Edge Computing System

by
Arnav Bhutani

A THESIS

submitted to

Oregon State University

Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Associate)

Presented June 1, 2021
Commencement June 2021

Honors Baccalaureate of Science in Computer Science project of Arnav Bhutani presented on June 1, 2021.

APPROVED:

_____

Scott Fairbanks, Mentor, representing Department of Computer Science

_____

Kirsten Winters Committee Member, representing Department of Computer Science

_____

Rahul Khanna, Committee Member, representing Intel, Corp.

_____

Toni Doolen, Dean, Oregon State University Honors College

I understand that my project will become part of the permanent collection of Oregon State University, Honors College. My signature below authorizes release of my project to any reader upon request.

_____

Arnav Bhutani, Author

Contents

I. INTRODUCTION

In today's world, there are two main compute resources, on-premise (on-prem) and cloud services (AWS, Google Cloud, Azure). On-premise refers to hardware that a specific company maintains and runs for their own webservices, for example, Amazon hosts their online marketplace through Amazon Web Services. Much of the internet began with on-premise servers. However, building-up web and server infrastructure is expensive, and large companies realized that they could subsidize the cost of their own hardware by renting out their computer power to host other people's websites during non-peak traffic hours. Thus, the cloud was born. These two modes of service serve as the backbone for most internet architecture, as most major internet services use their own hardware and minor players use the cloud[1]. However, as technology grows, a third mode of service is emerging, the edge.

Compute resources are always changing hands, as companies and individuals optimize their resource needs and expenditures. The internet has moved from using individual server mainframes in the early days, to servers, the cloud, and indeed the edge. These movements are driven by the need for more computational power, or the potential lower cost. Take for example, the migration to the cloud, companies have embraced the cloud because cloud services distribute the cost of maintenance of networks, reducing the need for each individual company to pay for high-cost maintenance staff. As with all tech, innovation will make certain advantages universal, and further advantage others, leading to a never-ending race for the best price/performance for software solutions. The edge stands to gain ground in this race when we look at future software solutions that rely on low latency, and other location-based concerns[2].

5g promises to bring ultra-low latency, high-bandwidth, and high-speed telecommunications to large service areas. Allowing for the rise of the 'smart city' or a city with a multitude of interconnected devices using 5g technologies. These interconnected devices will have to rely on localized computing power. This localized compute has been dubbed 'the edge', as the edge functions like the cloud, but is closer to service areas than a traditional datacenter-based cloud. The edge offers advantages in that it is close to its served devices, allowing for security and privacy as required, alongside ultra-low latency for highspeed tasks. Services such as autonomous vehicles, city-wide surveillance, and augmented reality are well poised to take advantage of the benefits of the edge[2]. As the edge gets built out to handle the maximum capacity that a smart city can handle, these servers will need to be optimized for everyday usage and needs. Companies and cities will have to grapple with how to best utilize the always running resources that compose the edge.

The purpose of this project is to create a proof-of-concept system that can take advantage of the always online network that the edge uses to run machine learning tasks such as image recognition. The project will coordinate a number single-threaded containers utilizing Docker, Kubernetes, and related technologies. The project will be defined as a success should it be able to perform complex machine learning tasks in a replicable environment on an 'edge-like' device.

II. BACKGROUND

This project's sponsor, Intel Corp., is an international semi-conductor manufacturer. Intel produces systems for a variety of use cases, including the edge. This project serves as an entry point for researchers who are interested in advancing edge systems as a proven way to setup an edge system for comparison between the edge and the cloud.

This project started as a senior capstone project between a partner, Ty Cole, and I, supported by Rahul Khanna from Intel. Ty and I created the design of the project and ensured that it ran on the cloud. Following the conclusion of the capstone project, I continued to develop the project so that it was in a usable state on an example edge system using consumer grade technology.

A. Docker

Docker is an open-source containerization software that was developed and released in 2013. Docker was created as a reliability layer and abstracts the physical properties of hardware away from the software dependencies of a program. This allows a single Docker program to run any hardware and operating system combination as long as the combination has the base Docker program [3]. Docker programs, defined in dockerfiles, create containers which are almost entirely independent of the operating system and hardware that it is running on. Containers run using their own operating system and have all the necessary programs and dependencies built in. As a result, Docker containers are usually single threaded. The project uses Docker heavily throughout the project as all the individual components of the program are containerized.

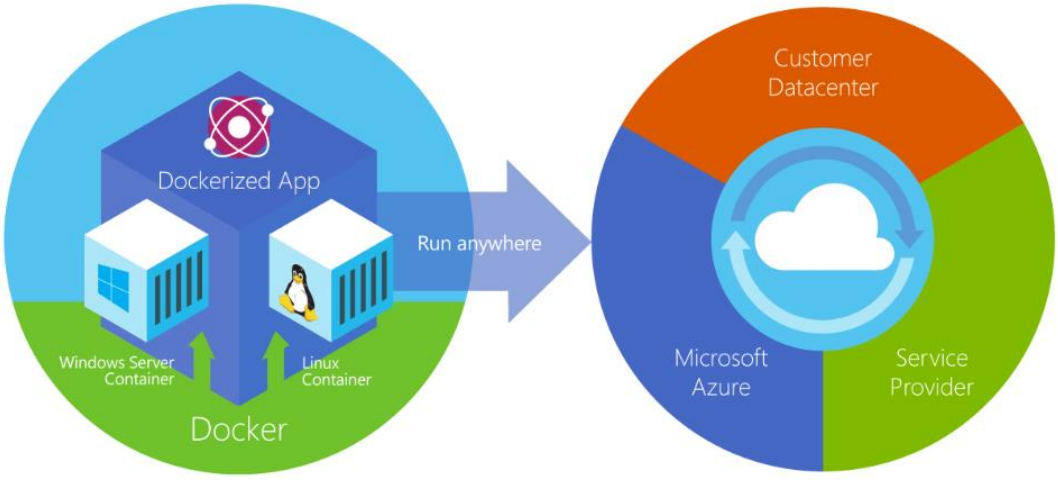

Figure 1: Illustration of Docker[1]

B. Kubernetes

Kubernetes is a piece of software developed by Google using the Go programming language to help coordinate Docker containers in the cloud. Kubernetes was made open source in 2014 and is managed by the Cloud Native Computing Foundation (CNCF). Kubernetes builds upon Docker's networking by introducing a hierarchical schematic to containers that allows for the developer to create relationships between containers in simple and sustainable ways[4]. A large challenge that Kubernetes overcomes is managing container deployments on different physical pieces of hardware, as before Kubernetes managing Docker deployments on multiple physical servers was quite difficult.

The project utilizes Kubernetes as the primary medium of deploying containers over the servers present on the edge network. The containers in this project were deployed as deployments of pods and services. With pods being analogous to a docker container, deployments controlling a set of pods to scale up or down, and services serving as a front-end for inter-pod communication. For example, an ingestion pod could be controlled by a deployment that manages three pods, and all three pods use a single service to communicate with a queuing pod by asking for the queuing pod's service.

C. Kafka

Kafka is a distributed queueing service which allows for different containers to pull 'jobs' or data from a queue[5]. Kafka is deployed in two containers and a service. The message broker holds the messages which it is passed through an accompanying service. Another managing container makes sure that the broker is functioning properly. The project used Kafka as the primary method of communication between containers, by setting up a queue in a a first in first out fashion for the processing containers and as a job assignment tool for worker containers.

D. Other Technologies

There were various other technologies used during different points of the project, however these technologies were not integral to the creation of the platform and served to demonstrate a purpose.

The machine learning algorithm You Only Look Once (YOLO) was also used as a sample for a distributed classification workload. This algorithm was based upon the idea of only using one 'look' or pass at an image for recognition. The project used YOLO as an evaluation of whether the model usage pipeline functioned[6].

Genetic algorithms, are algorithms that take principles of genetic evolution and apply them to a use case. Genetic algorithms create 'genes' in the form of candidate models, and after fitting each model, the best performing algorithms are advanced to the next trails. Models are then mutated and the models are trained again. This process is repeated until a satisfactory

result has been achieved [5]. The project used a genetic algorithm to evaluate the model creation pipeline and create models for the model usage section.

III. Model Implementation

The project used Kubernetes as a base orchestration platform, and for ease-of-use purposes was developed in the default namespace. The project was then split into two distinct parts: model implementation, and model creation. Each of these parts was composed of layers including: ingestion and preprocessing, transmission and queuing, data processing, training, and working. Each of these layers corresponds to a different set of docker containers. Containerizing the project allowed for each container to run as a single thread on a node/machine.

The purpose of the model implementation portion of the project was to verify that the platform worked in a usage scenario using a pretrained algorithm. The implementation started with an ingestion container. Which then parsed data to the Kafka broker. The Broker handled taking the data from each individual ingestion container queued it as a task to be queried by the worker containers. While the last docker container requested the data from the Kafka broker and processed it. In our processing example, we used the YOLO algorithm, however any other containerized image recognition or broader machine learning algorithm can be used so long as it works with the dataset. By the end of the project, the algorithm had been changed to one made by the model creation portion of the project. By breaking up the algorithm into ingestion, transmission and processing, the project was able to scale each section as needed on as many devices as needed, or to how much compute power is necessary. Figure 2 below illustrates a broad three pipeline implementation of this design, which was made to demonstrate an earlier understanding of the project that was using USB cameras.

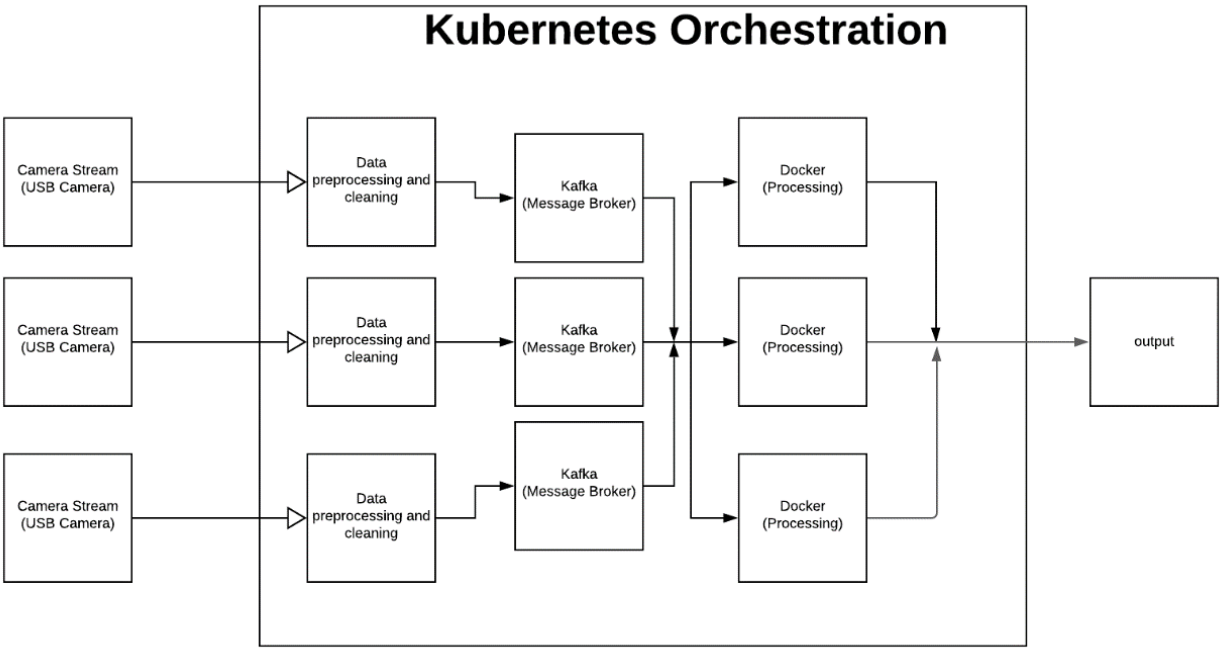

Figure 2: Model implementation diagram.

A. Preprocessor

The preprocessor segment of the model implementation was composed of three parts, an application, the containerization of the application and then a Kubernetes deployment.

The application was a simple python application, which can be viewed in the seventh listing of the appendix. The application made use of the Keras package for the database, and the Kafka package to appropriately connect and send data to the Kafka broker. The application downloaded the Modified National Institute of Standards and Technology database (MNIST) and then sent the first fifty testing datasets to the Kafka broker through a socket connection. After the first fifty testing datasets are sent, the connection is closed, and the application terminates.

The containerization of the above application was done using docker. The exact code can be seen in the sixth listing of the appendix. The container was quite simple, and just included the base python container, with the non-standard kafka-python package installed in the OS. The application is then added to the root directory and then launched using python at the containers' runtime.

The Kubernetes deployment was created in a yaml (a notation format like JSON) file. The exact code can be seen in the fifth listing of the appendix. The file follows the standard deployment format listed on the Kubernetes website, with the additional fields defining an environment variable that lists the name of the service that serves the Kafka broker container. The deployment ensures that each container is running healthily and can scale the number of containers up and down as necessary.

B. Kafka/Message Broker

Kafka itself is split into two different containers, each of which has a service which stands as an abstraction in front of the Kubernetes deployments. The first container is called the Kafka-Broker, and the second is called the Zookeeper. As Kafka is a completed open-source product, all components are provided by the Apache organization, and the defined as specified on their website. The Broker is used to send and hold messages, while the Zookeeper ensures that each message is only sent once and to an appropriate consumer.

C. Processor

The processor like the preprocessor is composed of three parts, an application, the containerization of the application and a Kubernetes deployment.

The application is a python application, which can be viewed in the tenth listing of the appendix. The application makes use of the sockets package to receive a model from the model creation portion of the project in the form of a h5 file. The application then uses the Kafka package to receive the images. It then uses the model to classify the images.

The containerization of the application was done in docker. The dockerfile can be viewed in the appendix as listing nine, and used the Tensorflow container as a base. The Kafka Keras and Networkx dependencies were installed on the OS. The application was then run at the containers' runtime.

The Kubernetes Deployment, much like for the preprocessor was standard, except for the inclusion of the Kafka Broker port. The implementation can be seen in the eighth listing of the appendix.

IV. MODEL CREATION

The model creation portion of the project was centered around using the edge to improve the development and fit of models to ingested data. A genetic algorithm was used to see if distributing the model training to multiple containers was possible. The genetic algorithm was split into three sections much like the model usage portion of the project, with a trainer/master container, a transmission container, and a set of 'work' containers alongside corresponding services. To achieve this split, the genetic algorithm was split. The trainer container ran the initial dataset setup and created the potential candidates for the genetic algorithm, instead of training each candidate algorithm locally, the candidate for training was then sent to the Kafka broker with the candidate's index, and a training dataset. Worker(s) containers then requested and trained their algorithm using the python package Keras. The trained models were then sent back to the Kafka broker, and requested by the initial container, which used their index for genetic selection. Note that a key difference between the model creation and other portions of the project is that the model creation pipeline is designed in a 'master-slave' relationship, with a singular ingestion or control container, and many worker containers. This is an important distinction as this relationship requires consistent two-way communication between the 'master', control container and the many

‘slave’, worker, containers. Figure 3 illustrates this setup with nine workers and a trainer node, the communication done by Kafka is not shown in this model.

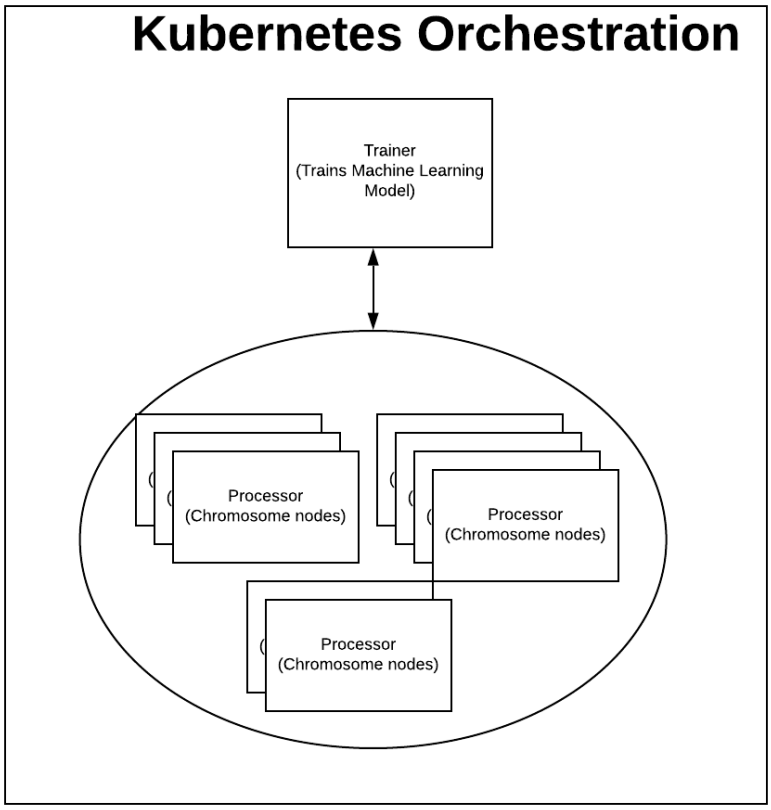

Figure 3: Model Creation Diagram

A. Trainer

The trainer was composed of four sections, an application, a containerization of the application, a deployment for the container and a service in front of the deployment.

The application was derived from an example implementation of the Keras-CoDeepNeat package. This python application makes use of the Keras-CoDeepNeat library to create a set of populations based on the training data. These populations are subdivided into blueprints on which mutations are applied. These blueprints are then fitted by Keras to the training data, and the best model is then selected for the next generation. The project modified the Keras-CoDeepNeat package such that the actual fitting of blueprints is not done by the Keras package, but rather by our own custom code that sent each blueprint is sent to a Kafka broker alongside the blueprint’s index. We also modified the package so that the model selected is then sent to the model implementation portion of the project, should it be running, alongside an index, the training data, and other metadata about the sample being trained. An excerpt containing the modifications to the example application can be seen in the fourteenth listing of the appendix.

The containerization for the trainer application was done using docker. The dockerfile can be viewed in the thirteenth listing of the appendix. The base container varies based on the hardware used, specifically the vendor that supplies the graphics processing unit. However, in our case the base TensorFlow container was used in Intel/Nvidia systems, and the Rocm/TensorFlow package was used in Intel/AMD systems. Further discussion on this topic is in the Project Specifications section of the paper. After that the packages used by the Keras-CoDeepNeat are installed the program is run.

The Kubernetes deployment was created using the yaml notation. The exact code can be seen in the eleventh listing of the appendix. It is again a standard deployment, with the exception of the Kafka port, and an additional port for the completely trained model to be sent to the model implementation portion of the project.

The Kubernetes service was created using the yaml notation. The exact code can be seen in the twelfth listing of the appendix. It is a standard service and stands in front of the trainer. This service is used for the model implementation portion of the project to lookup the completed model used by the model creation portion of the project.

B. Kafka

Again, the Kafka deployment is as advised by the Apache Kafka group. There is one exception in that the message size allowed by the containers has been expanded to reflect the size needed to transmit the various blueprints. Two different Kafka sets were created as a way to independently develop the two segments of the project, and in the future these individual brokers could be combined into one, so long as the increased message size buffer is retained. The Kubernetes deployment and service for this Kafka deployment can be viewed in appendices fifteen through eighteen.

C. Worker

The worker, like the preprocessor and the processor, is composed of three parts, the application, the containerization, and the Kubernetes deployment.

The application is made using python and can be viewed in the twenty first listing of the appendix. The application receives the model and its metadata from the Kafka broker, and unpacks the model. The metadata is then used to unpack the training data, and the data is trained using the Keras function, ‘model_fit’. The application then sends the model back, alongside its fitting score, so that the trainer can select the best scores for the next generation.

The container is made using docker, and the dockerfile can be viewed in appendix twenty. The dockerfile is quite simple and uses a base Tensorflow container and adds some additional packages that are used by the application. The application is then run at the container’s runtime.

The Kubernetes deployment was created using the yaml notation and can be viewed in appendix nineteen. It is functionally identical to the deployment used for the trainer. It is to be noted, that this deployment would need to be slightly modified if one wishes to scale the number of containers used. As, in this design there was just one container deployed.

V. Project Specifications

The project was designed in two different environments, using different base machines. The project was first developed on the cloud as a part of a capstone program for the Oregon STate University Computer Science program. It was then extended through individual work with an edge device, an Intel NUC compute platform. The cloud was composed of Intel Haswell era server SOCs, and very limited graphical processing unites (GPUs) running a Linux operating system. While the NUC was equipped with an Intel Comet-Lake consumer i7-8700k CPU and an Advanced Micro Dynamics (AMD) Vega 56 GPU also using a Linux operating system.

A. Cloud Setup

The project was initially developed on the G-cloud suite, which allowed for free access to a limited Kubernetes network. The project was developed on this network, using Linux, Kubernetes, Yaml, Docker, Python 3.6 and a set of Python packages, including but not limited to: TensorFlow, NetworkX, and Keras. Most of the individual applications, such as the ingestion container, etc. were written using Python 3.6, and then were containerized using Docker. Kubernetes configuration files were then written in Yaml. The project is self-contained and can be run on any Kubernetes network using Nvidia or Intel based hardware using a simple set of Kubernetes commands.

B. Edge Setup

Additionally, the project was extended to be used on AMD and consumer hardware when used with edge devices. The AMD ROCm platform (an AMD substitute for Nvidia's CUDA platform) had to be installed. In addition, Kubernetes had to be deployed on the NUC platform, which required the installation of Kube admin (a Kubernetes software), and a network solution, Flannel which was chosen for its simplicity. The docker containers for all the deployments that used the Python package TensorFlow had to be changed to use those developed by the ROCm project, in this case a docker container with CentOS, Python3.6 and TensorFlow 0.13.1. After all these adjustments there were a few Kubernetes adjustments, such as allowing the master node to be used by containers (a default setting turned off by Kubernetes) and modifying the ingestion containers to be built with the dataset preinstalled (due to network signing issues).

C. Comparision

While the outcome of both solutions was similar, there were driver level issues with running required packages on different hardware. Specifically, Tensorflow a popular machine learning Python package, was difficult to implement due to specific package drivers being written with nvidia hardware in mind. These issues should be kept in mind for a network rollout, as they may preclude certain server setups from being used in the network.

VI. Results and Discussion

The proof of concept was able to run on both a cloud infrastructure and on an edge platform, opening the subject to further testing and investigation. Both the model creation and the model implementation algorithms accurately ran on both the cloud and the edge. This model can be used as a base for further testing in the field of edge computing, as the basic infrastructure for any machine learning task has been created. Containerizing seems to be a natural solution to the problem posed by a highly connected smart city, with many edge devices working cohesively to power next generation technologies such as autonomous vehicles, or large area private security.

While the project is successful there are number of issues that need to be explored. Current testing was very limited, and only extended as far as the implementation of the project on the cloud and the edge. There was no empirical performance data collected from either service, and thus no conclusions can be drawn as to the performance of the cloud or the edge. Assumptions such as, the edge having lower latency, and the relative computer power of the edge and the cloud must be tested before any conclusions can be drawn about the model.

In addition, there are potential problems with some of the technologies used. Project performance, in terms of both time and overhead, must be measured to see if this multi-device approach improves upon just a simple singe-device. Should performance not be improved then there are very few real-world situations where this project can be deployed.

Fifth generation technologies have not been verified to work with the project either. The data transfers were quite small in this example and should fall well below the 1 gigabit per a second threshold that early 5g networks are capable of. However, if the packets transmitted within applications grow in size, then multi-system edge networks using 5g technology could slow down and make edge processing inefficient, when compared to the cloud. This can be mitigated using more preprocessing steps to evenly distribute the data over the network, however there is an absolute limit that needs to be tested.

Kubernetes will have to be further tested to see if there is an upper limit to the number of nodes and containers that it can run. Should this limit not be large enough, a deployment of this model could run into issues when scaling to a large number of devices.

Power usage will have to be measured to see if docker is a viable cost-effective containerization technology. As the edge grows, power usage starts to be a concern, as the large number of always running devices can put a strain on a cities' power infrastructure. Optimizing the edge for performance per a watt could help alleviate some of these concerns.

VII. Conclusions

This project was a valuable way of testing the feasibility of edge computing in the future. Current popular trends such as containerization and cloud-based orchestrators can be adapted to run well on the edge. This solution could serve as a valuable first step in testing the viability of the edge in comparison to the cloud.

VIII. Acknowledgements

I would like to thank my mentor, Instructor Scott Fairbanks for his patience and his guidance through the thesis process. I would also like to thank Dr. Rahul Khanna, who was a sponsor for my project for his assistance with constructing the project and with providing the edge computing resource for the project. I also would like to thank Instructor Kirsten Winters for helping me start out the Thesis processes, and would also like to acknowledge Ty Cole, my partner from the initial capstone project, for his contributions to the project.

IX. References

[1] K. Foote, “A Brief History of Cloud Computing - DATAVERSITY,” DATAVERSITY, Jun. 22, 2017. https://www.dataversity.net/brief-history-cloud-computing/.
[2] S. Yi, Z. Hao, Z. Qin and Q. Li, "Fog Computing: Platform and Applications," 2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb), 2015, pp. 73-78, doi: 10.1109/HotWeb.2015.22.
[3] Nish Anil, “What is Docker?,” *Microsoft.com*, Aug. 31, 2018. https://docs.microsoft.com/en-us/dotnet/architecture/microservices/container-docker-introduction/docker-defined.
[4] Kubernetes, “What is Kubernetes?,” *Kubernetes*, Apr. 30, 2021. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/#why-you-need-kubernetes-and-what-can-it-do (accessed Jun. 03, 2021).
[5] Amazon Web Services, “What is Apache Kafka?,” *Amazon Web Services, Inc.*, 2021. https://aws.amazon.com/msk/what-is-kafka/.
[6] Open Data Science, “Overview of the YOLO Object Detection Algorithm,” *Medium*, Sep. 25, 2018. https://medium.com/@ODSC/overview-of-the-yolo-object-detection-algorithm-7b52a745d3e0#:~:text=YOLO%20is%20a%20clever%20convolutional%20neural%20network%20%28CNN%29 (accessed Jun. 03, 2021).

X. APPENDIX

A. *Essential Code Listings – Model Implementation*

Listing 1: Kubernetes deployment for the Kafka broker

```yaml
kind: Deployment
apiVersion: apps/v1
metadata:
  name: kafka-broker-pre
spec:
  selector:
    matchLabels:
      app: kafka-pre
      id: "0"
  template:
    metadata:
      labels:
        app: kafka-pre
        id: "0"
    spec:
      containers:
      - name: kafka-pre
        image: wurstmeister/kafka
        ports:
        - containerPort: 9092
        env:
        - name: KAFKA_ADVERTISED_PORT
          value: "9092"
        - name: KAFKA_ADVERTISED_HOST_NAME
          value: kafka-service-pre
        - name: KAFKA_ZOOKEEPER_CONNECT
          value: zoo1:2181
        - name: KAFKA_BROKER_ID
          value: "0"
        - name: KAFKA_CREATE_TOPICS
          value: images:1:1
```

Listing 2: Kubernetes service for the Kafka broker

```
apiVersion: v1
kind: Service
metadata:
  name: kafka-service-pre
  labels:
    name: kafka
spec:
  ports:
  - port: 9092
    name: kafka-port
    protocol: TCP
  selector:
    app: kafka-pre
    id: "0"
  type: LoadBalancer
```

Listing 3: Kubernetes deployment for the Kafka Zookeeper container

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: zookeeper-deployment-1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: zookeeper-1
  template:
    metadata:
      labels:
        app: zookeeper-1
    spec:
      containers:
      - name: zoo1
        image: digitalwonderland/zookeeper
        ports:
        - containerPort: 2181
        env:
        - name: ZOOKEEPER_ID
          value: "1"
        - name: ZOOKEEPER_SERVER_1
          value: zoo1
```

Listing 4: Kubernetes service for the Kafka Zookeeper container

```
apiVersion: v1
kind: Service
metadata:
  name: zoo1
  labels:
    app: zookeeper-1
spec:
  ports:
  - name: client
    port: 2181
    protocol: TCP
  - name: follower
    port: 2888
    protocol: TCP
  - name: leader
    port: 3888
    protocol: TCP
  selector:
    app: zookeeper-1
```

Listing 5: Kubernetes deployment for the preprocessing container

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: preprocessor
spec:
  selector:
    matchLabels:
      app: preprocessor
  template:
    metadata:
      labels:
        app: preprocessor
    spec:
      containers:
      - name: preprocessor
        image: coletyl/preprocessor
        env:
         - name: KAFKA_HOST_NAME
           value: kafka-service-pre
```

Listing 6: Docker container for preprocessor application

```
FROM python:3
RUN pip3 install kafka-python
ADD app.py /app.py
ENTRYPOINT ["python3", "app.py"]
```

Listing 7: Python application for preprocessor

```
from kafka import KafkaProducer
import os
import sys

from keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Connect to Kafka
kafka_host = os.getenv('KAFKA_HOST_NAME')
if(kafka_host == None):
    print("Failed, no KAFKA_HOST_NAME environment variable was set")
    sys.exit(1)

# Setup Kafka Producer
producer = KafkaProducer(bootstrap_servers=kafka_host)
numIterations = 0
for x_test as test:
    numIterations = numIterations+1
    # Stop after 50 iterations
    if(numIterations >= 50):
        exit(0)
    b = test1.tobytes()
    print("Message sent")
    response = producer.send('images', b)
    print("Response = " + str(response))
    result = response.get(timeout=30)
    print("Fetched Message = " + str(result))
```

Listing 8: Kubernetes deployment for the processor container

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: processor
spec:
  selector:
    matchLabels:
      app: processor
  template:
    metadata:
      labels:
        app: processor
    spec:
      containers:
      - name: processor
        image: coletyl/processor
        env:
         - name: KAFKA_HOST_NAME
           value: kafka-service-pre
```

Listing 9: Docker container for the processor application

```
FROM rocm/tensorflow:latest
LABEL maintianer="Tyler Cole"


ENV DEBIAN_FRONTEND noninteractive

ADD app.py /app.py

RUN pip3 install kafka-python
RUN pip3 install keras
RUN pip3 install networkx


ENTRYPOINT ["python3",  "/app.py"]
```

Listing 10: Python processor application

```
from kafka import KafkaConsumer
import os
import sys
import time
import socket
from tensorflow import keras
import numpy as np

kafka_host = os.getenv('KAFKA_HOST_NAME')
if(kafka_host == None):
    print("Failed, no KAFKA_HOST_NAME environment variable was set")
    sys.exit(1)

if __name__ == "__main__":

    HOST, PORT = "trainer-model-service", 80
    data = "my data"
    # Create a socket (SOCK_STREAM means a TCP socket)
    while 1:
        try:
            # Connect to server and send data
            sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            sock.connect((HOST, PORT))
        except socket.error:
            print("Couldn't connect to server, no model available");
            time.sleep(5);
            sock.close();
            continue
       sock.sendall(bytes(data + " ", "utf-8"))
        # Receive data from the server and shut down
        print("Receiving the model:");
        received = b'';
        numReceived = 0 ;
        while True:
            data = sock.recv(1048576);
            print("Recevied data, len = " + str(len(data)) + ", this is byte num
" + str(numReceived) + "\n");
            numReceived+=1;
            if not data: break
            received += data;
        sock.close()
        break
```

```
    print("Len = " + str(len(received)));
    f = open('/model.h5', 'wb')
    f.write(received)
    f.close()
    print("Model receieved");
    model = keras.models.load_model("/model.h5")

    consumer  = KafkaConsumer("images", group_id="processor",  request_timeout_ms
=120000,session_timeout_ms=100000, bootstrap_servers=kafka_host)

    for message in consumer:
       bytestream = bytes(message.value)
       newer = np.frombuffer(bytestream, dtype=np.uint8)
       newer = newer.reshape((28,28))
       newer = np.expand_dims(newer, axis=0)
       newer = np.expand_dims(newer, axis=3)
       scores = model.predict(newer);
       print("Prediction = " + str(scores));
```

B. *Essential Code Listings – Model Creation*

Listing 11: Kubernetes deployment for the training container

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: trainer
spec:
  selector:
    matchLabels:
      app: trainer
  replicas: 1
  template:
    metadata:
      labels:
        app: trainer
    spec:
      containers:
        - name: trainer
          image: coletyl/trainer
          ports:
            - containerPort: 9999
          env:
           - name: KAFKA_HOST_NAME
             value: kafka-service-train
```

Listing 12: Kubernetes service for the training container

```yaml
apiVersion: v1
kind: Service
metadata:
  name: trainer-model-service
spec:
  selector:
    app: trainer
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9999
```

Listing 13: Docker container for the training application

```
FROM rocm/tensorflow:rocm3.3-tf1.13-centos-mkl-dev
LABEL maintianer="Tyler Cole"

ENV CENTOS_FRONTEND noninteractive

ADD Keras-CoDeepNEAT /Keras-CoDeepNEAT
ADD run_mnist.py /run_mnist.py

RUN yum install -y python3.6-dev graphviz* libgraphviz-dev

RUN pip install kafka-python
RUN pip install -Iv keras==2.2.5
RUN pip install -Iv networkx==2.3
RUN pip install -Iv pydot==1.4.1
RUN pip install graphviz
RUN pip install pygraphviz
RUN pip install sklearn
RUN pip install matplotlib==3.2.1



ENTRYPOINT ["python3",  "/run_mnist.py"]
```

Listing 14: Excerpt from the python training application

```
if __name__ == "__main__":

    generations = 2
    training_epochs = 2
    final_model_training_epochs = 2
    population_size = 1
    blueprint_population_size = 10
    module_population_size = 30
    n_blueprint_species = 3
    n_module_species = 3

    def create_dir(dir):
        if not os.path.exists(os.path.dirname(dir)):
            try:
                os.makedirs(os.path.dirname(dir))
            except OSError as exc: # Guard against race condition
                if exc.errno != errno.EEXIST:
                    raise

    create_dir("models/")
    create_dir("images/")

    run_mnist_full(generations, training_epochs, population_size, blueprint_popul
ation_size, module_population_size, n_blueprint_species, n_module_species, final_
model_training_epochs)


    print("##############################################");
    print("### Loading top performing model ....  #####");
    print("##############################################");

    filename = "best_generation_" + str(generations-1) + ".h5";
    model = keras.models.load_model("models/" + filename)

    print("##############################################");
    print("### Manually verifying the test scores #####");
    print("##############################################");

    img_rows, img_cols = 28, 28
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    y_test = keras.utils.to_categorical(y_test, 10)
    x_test = x_test.astype('float32')
    x_test /= 255
```

```
    scores = model.evaluate(x_test, y_test, verbose=1)

    print("Setting up kafka application ... ");

    os.rename("models/"+filename, "/model.h5");


    HOST, PORT = socket.gethostname(), 9999

    # Create the server, binding to localhost on port 9999
    server = socketserver.TCPServer((HOST, PORT), MyTCPHandler)

    # Activate the server; this will keep running until you
    # interrupt the program with Ctrl-C
    server.serve_forever()
```

Listing 15: Kubernetes deployment for the Kafka broker container

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: kafka-broker-train
spec:
  selector:
    matchLabels:
      app: kafka-train
      id: "0"
  template:
    metadata:
      labels:
        app: kafka-train
        id: "0"
    spec:
      containers:
      - name: kafka-train
        image: wurstmeister/kafka
        ports:
        - containerPort: 9092
        env:
        - name: KAFKA_MESSAGE_MAX_BYTES
          value: "2000000000"
        - name: KAFKA_ADVERTISED_PORT
          value: "9092"
        - name: KAFKA_ADVERTISED_HOST_NAME
          value: kafka-service-train
        - name: KAFKA_ZOOKEEPER_CONNECT
          value: zoo2:2181
        - name: KAFKA_BROKER_ID
          value: "0"
        - name: KAFKA_CREATE_TOPICS
          value: "models-to-fit:1:1,models-fitted:1:1"
```

Listing 16: Kubernetes service for the Kafka broker container(s)

```
apiVersion: v1
kind: Service
metadata:
  name: kafka-service-train
  labels:
    name: kafka
spec:
  ports:
  - port: 9092
    name: kafka-port
    protocol: TCP
  selector:
    app: kafka-train
    id: "0"
  type: LoadBalancer
```

Listing 17: Kubernetes deployment for the Kafka Zookeeper container(s)

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: zookeeper-deployment-2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: zookeeper-2
  template:
    metadata:
      labels:
        app: zookeeper-2
    spec:
      containers:
      - name: zoo2
        image: digitalwonderland/zookeeper
        ports:
        - containerPort: 2181
        env:
        - name: ZOOKEEPER_ID
          value: "1"
        - name: ZOOKEEPER_SERVER_1
          value: zoo2
```

Listing 18: Kubernetes service for the Kafka Zookeeper container(s)

```
apiVersion: v1
kind: Service
metadata:
  name: zoo2
  labels:
    app: zookeeper-2
spec:
  ports:
  - name: client
    port: 2181
    protocol: TCP
  - name: follower
    port: 2888
    protocol: TCP
  - name: leader
    port: 3888
    protocol: TCP
  selector:
    app: zookeeper-2
```

Listing 19: Kubernetes deployment for the worker container(s)

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: worker
spec:
  selector:
    matchLabels:
      app: worker
  template:
    metadata:
      labels:
        app: worker
    spec:
      containers:
      - name: worker
        image: coletyl/worker
        env:
         - name: KAFKA_HOST_NAME
           value: kafka-service-train
```

Listing 20: Docker file for the worker application

```
FROM rocm/tensorflow:rocm3.3-tf1.13-centos-mkl-dev
LABEL maintianer="Tyler Cole"



ENV CENTOS_FRONTEND noninteractive

ADD app.py /app.py

RUN pip3 install kafka-python
RUN pip3 install keras
RUN pip3 install networkx



ENTRYPOINT ["python3",  "/app.py"]
```

Listing 21: Python application for the worker

```
from kafka import KafkaConsumer
from kafka import KafkaProducer
import os
from tensorflow import keras
import numpy as np
import io
import tarfile
import json
kafka_host = os.getenv('KAFKA_HOST_NAME')
if(kafka_host == None):
    kafka_host = "kafka-service-train";

consumer  = KafkaConsumer("models-to-
fit", group_id="worker",  fetch_max_bytes=2000000000, request_timeout_ms=120000,s
ession_timeout_ms=100000, bootstrap_servers=kafka_host)

while(1):
    print("Starting the worker process");
    for message in consumer:
        print("Waiting for message");
        bytestream = bytes(message.value)
        file_like_object = io.BytesIO(bytestream)
        tar = tarfile.open(fileobj=file_like_object)
        for member in tar.getmembers():
            print("F = " + str(member.name));
        tar.extractall()
        break;

    print("Model receieved");
    model = keras.models.load_model("/model.h5")

    input_x = np.fromfile("input_x", dtype = np.uint32);
    input_y = np.fromfile("input_y", dtype = np.uint32);
    print("X and Y are loaded", str(input_x.shape), str(input_y.shape));

    metadata_file = open("metadata");
    metadata = json.load(metadata_file);
    metadata_file.close();
    print("Metadata has been received, = " + str(metadata));
    index =  metadata["index"];
    training_epochs = metadata["training_epochs"];
    validation_split = metadata["validation_split"];
    x_shape = metadata["x_shape"];
    y_shape = metadata["y_shape"];
```

```
    input_x = np.reshape(input_x, x_shape);
    input_y = np.reshape(input_y, y_shape);
    print("New shape = " + str(input_x.shape));
    score = model.fit(input_x, input_y, epochs=training_epochs, validation_split=
float(validation_split), batch_size=128)
    producer = KafkaProducer(bootstrap_servers=kafka_host)
    print("index, score = ", + str(index) + " "  + str(score));
    producer.send('models-fitted', {index: score})
    print("Resposne = " + str(response))
    result = response.get(timeout=30)
    print("Result = " + str(result))
```