Mobile Edge Computing Using Raspberry Pi for Small-Scale Testbed

Soo Min Kwon*, Manish Kewalramani* and Natalie Kim*
* Rutgers University, New Brunswick, NJ 08901, USA
EMAIL: {smk330, mkewalra, nkk35}@scarletmail.rutgers.edu

***Abstract*—Mobile Edge Computing (MEC) is a network architecture that extends cloud computing capabilities to the edge of any cellular framework. The MEC paradigm is an iteration of a technology cycle that begins with centralized processing and then utilizes the notion of distributed computing for faster calculation time. To reduce latency, this architecture leverages mobile networks by running and processing data closer to the cellular customer. Latency is a significant factor in real-world applications, such as those coupled with the Internet of Things (IoT). IoT applications such as object detection, face recognition, and language processing rely on machine learning algorithms that requires substantial amounts of data. By processing at the edge, we show that we can handle large amounts of data and make models more robust to run sophisticated, time-critical applications. In this paper, we will implement a small-scale MEC testbed consisting of a Raspberry Pi mobile node and the Microsoft Azure edge server to run and process real-time applications, demonstrating the advantages of MEC.**

***Index Terms*—Mobile Edge Computing, Internet of Things, Raspberry Pi, testbed, machine learning, sensors, facial recognition**

I. Introduction

The advent of Mobile Edge Computing (MEC) has spurred a reinvention of the wireless landscape. Based on the principle of distributed computing, MEC allows applications and services to run in closer proximity to the end user, resulting in decreased latency and higher computational capacity. The architecture is demonstrated in Figure 2.

As the number of connected devices approaches the scale of tens of billions, the necessity of MEC becomes apparent [1]. Although the centralized nature of traditional cloud computing (Fig. 1) offers powerful computational and storage resources, it is not well-suited as the sole wireless architecture given the technological trends we see today. For example, internet connectivity is no longer limited to just smartphones and computers; the internet is anticipated to host an abundance of devices such as cars, televisions, home appliances, parking meters, smart watches, real-time sensors, and virtual reality applications. This emerging phenomenon is referred to as the

Fig. 1. Cloud Computing Model [2]

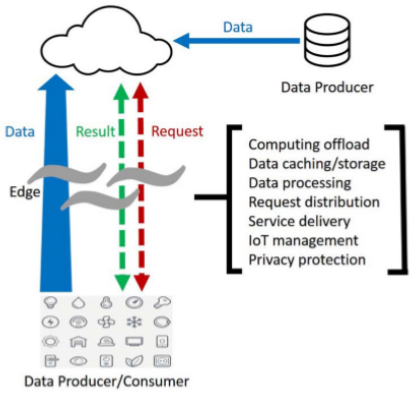

Fig. 2. Mobile Edge Computing Model [2]

Internet of Things (IoT). Not only do several of these IoT applications require low latency (which is explicitly achieved by the reduced physical distance of data transmission in MEC), but the growing ubiquity of internet connectivity will also produce an enormous quantity of raw data. In this sense, these IoT devices are both data consumers and data producers, whereas in the past, they were primarily data consumers [2]. The sheer quantity of this data demand would inevitably result in a bottleneck in the centralized cloud computing paradigm [2]. The proposed solution is to assign computational tasks to edge nodes that exist in between the end device and the central cloud data center, such as micro data centers and cloudlets [3]. In this scheme, much data will never have to be transmitted to the central cloud, resulting in a faster end-user experience, as well as improving overall computational efficiency, power efficiency, and energy consumption [4].

Mobile Edge Computing, alongside Network Functions Virtualization (NFV) and Software-Defined Networking (SDN), is recognized by The European 5G Infrastructure Public Private Partnership as a cornerstone of 5G [5], an emerging cellular network technology that aims to improve wireless speeds, capacity, and latency. Plans for 5G include transmission over millimeter waves, coupled with more densely-packed, smaller cell antennas to support these higher bandwidths. This architecture will address problems faced by older generation networks such as radio frequency spectra availability, energy efficiency, spectral efficiency, high data rate demands, and coverage requirements [6].

Motivation: Motivated by the increasing advances in MEC deployment, our project will explore the application side by demonstrating use cases. Some of the major advantages of MEc that motivate our work is as follows:

- With cloud services in closer proximity to the end user, we can enable sophisticated IoT applications to run with better latency.
- By utilizing computing resources at the edge, we enable a way to offload energy-consuming applications without having to travel to the centralized cloud, solving the trade-off between device battery life and computational delay [7].
- By computing at the edge, we can reserve central cloud capacity for more high-volume, time-insensitive tasks while simultaneously providing more data storage to users at the edge.

Challenges: We highlight some of the main research challenges in the area of MEC, as well as the difficulties of our current contributions in this paper:

- One major challenge of edge computing is guaranteeing that the processed data will be delivered back to the data source. MEC requires reliable connectivity and thorough coverage.
- Computing all applications on the edge may be energy consuming and even worse in terms of latency if handled incorrectly. Making a proper decision on whether to offload the computation to the edge versus the central cloud is a difficult issue to resolve.
- There may be cases where the user is constantly moving while performing applications such as video streaming. MEC must provide mobility management and guarantee service continuity [7].
- Using pre-defined servers such as Microsoft Azure can lead to issues in downloading packages. For example, programs may not execute due to insufficient downloading of necessary materials.

Our Contributions: We summarize the major contributions of our work so far as follows:

- We propose the implementation of a small-scale MEC testbed consisting of a mobile node (Raspberry Pi) and an edge server (Microsoft Azure IoT).
- We connect our Raspberry Pi to Microsoft Azure IoT Hub to send and receive messages to the edge server.
- We design two computationally expensive experiments, weather prediction and facial recognition, to compare the processing times between the Azure server and the Raspberry Pi.
- We utilize BME280 sensor data and machine learning algorithms to predict whether it will rain, given the temperature and humidity of an area.
- We utilize the Raspberry Pi camera and computer vision techniques for facial recognition. Once images are processed, we will compare and analyze offloading speeds of the Raspberry Pi and the edge server.

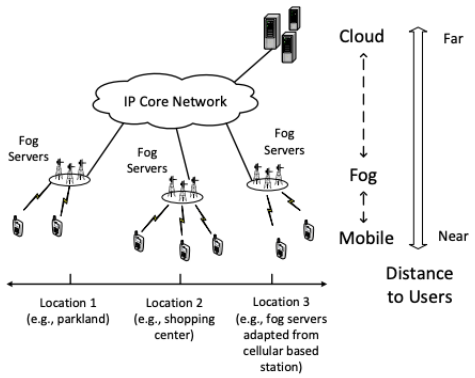

Fig. 3. Fog Computing Architecture [10]

II. Related Works

There has been a considerable amount of related work that addresses the issues of the Internet of Things with MEC. For instance, the work in [8] discusses scalability issues within the Internet, claiming that as more mobile devices require Internet connection, it will become increasingly difficult to manage IoT data. To counter this, they propose 'EdgeIoT', a framework that will bring computing resources closer to IoT devices so that processing at the core is significantly reduced. However, in this proposed architecture, users must send pictures/videos to a fog node to process potentially sensitive and confidential applications, such as facial recognition. This framework poses potential privacy risks. As a solution, they proposed that each user gets a proxy VM, though it may be cumbersome to handle. In our paper, we only consider a small-scale testbed for MEC, and will generally not be handling private data. Rather, we will be first working with temperature sensors and their respective readings, discussed in detail in the following section.

Moreover, [9] introduces a deep learning approach to withdrawing raw sensor data from IoT devices, which will then be moved to the centralized cloud upon extraction. However, it is well known that deep learning models have large parameters and often battle issues with latency. Such models can be optimized to have smaller data capacity, but still have larger storage and complexity than other machine learning models. In our paper, we extensively exploit statistical machine learning models (e.g. Support Vector Machines) when classifying our sensor data to keep data processing minimal.

Another related proposal is Fog computing, discussed in [10], an architecture that places the contents and application services as close to the consumer as possible, as visualized in Figure 3. The purpose and end result of Fog computing is very similar to that of MEC - both paradigms attempt to solve the issue of latency. However, although Fog computing places additional resources at the edge, there are substantial challenges with this framework. To begin, the network operator needs to anticipate the demand of each server for a given task in order to allocate its resources. This becomes a burgeoning challenge as applications increase in scale and run simultane-

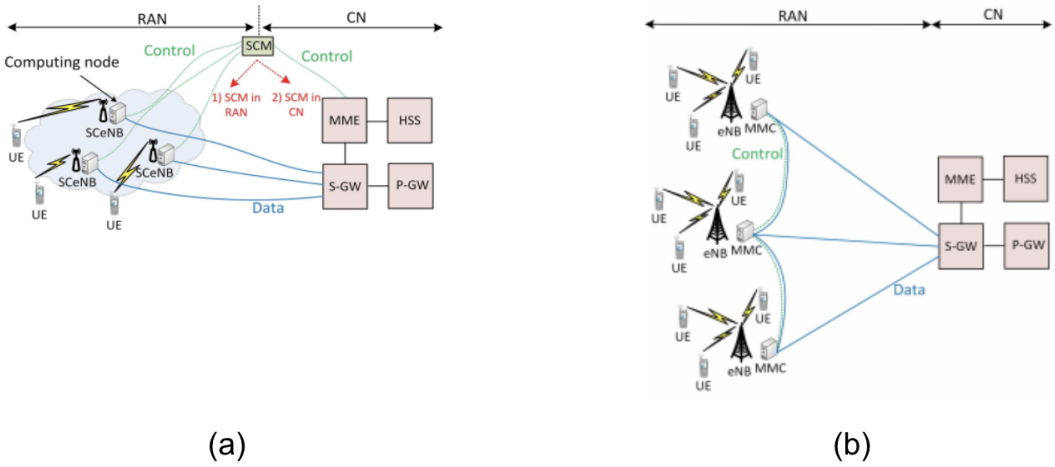

Fig. 4. (a) Small Cell Cloud (SCC), (b) Mobile Micro Clouds (MMC) Architecture [7]

ously. Additionally, the network operator needs to customize the applications in each server based on local demand. Scalability again presents itself as an issue in this case. On the other hand, MEC solves the challenge of scalability at its core, and thus our project will adopt the method of MEC.

Our project is best described by the content in [11]. This paper summarizes the importance of MEC architecture in supporting a robust, high-capacity Internet of Things. The work in [11] is primarily concerned with deploying applications for surveillance, safety, and connectivity for moving IoT devices. However, in our paper, we instead focus on indicating the potential of using the Raspberry Pi as a mobile node. We demonstrate some initial experiments with stationary sensors in section IV.

III. MEC Architecture and Standardization

We dedicate this section to discuss some of the core components in the Mobile Edge Computing architecture. In specific, we will be analyzing the frameworks of the Small Cell Cloud (SCC) and Mobile Micro Clouds (MMC). These architectures are critical in examining the overall idea of MEC.

A. Small Cell Cloud (SCC)

The general idea of SCC is to enhance small cells, like microcells, by the addition of computation and storage capabilities [7]. With small cells being frequently deployed in mobile networks, SCC can provide the necessary computational power and meet the delay requirements of users performing intensive tasks. The significance in the SCC architecture lies in the Small Cell Manager (SCM). The SCM supervises the overall network, and makes decisions on whether a given application requires new deployment or when to continue in an on-going computation to maximize efficiency for the network and the user. This process is shown in (a) of Figure 4, where the SCM block is denoted at top.

B. Mobile Micro Clouds (MMC)

The concept of MMC is very similar to that of SCC. The primary goal of both architectures is to decrease latency for mobile users by giving users instantaneous access to cloud services. The difference is that in the MMC architecture, there is no supervising node, and control is assumed to be fully distributed. This is shown in (b) of Figure 4, where the user equipments (UEs) have equivalent control.

IV. Preliminary Experiments

To familiarize ourselves with the functionalities of the Raspberry Pi, we conducted two initial preliminary experiments. These experiments were primarily comprised of connecting the Raspberry Pi to two different sources for real-time applications. The first source was the Google Cloud Vision, in which we were able to run applications such as facial recognition using the Raspberry Pi camera. In the second source, we were able to collect simulated sensor data using the programs provided by Microsoft Azure. We describe in detail our initial experiments and their results in the following sections.

A. Connecting the Raspberry Pi to Google Cloud Vision

Firstly, we connected the Raspberry Pi to Google Cloud Vision [12]. Using the Google Cloud Platform allows us to run applications with the Raspberry Pi camera for object detection and facial recognition.

To begin, we first summarize our steps in connecting to the cloud. To access Google Cloud, we made an account to communicate with the Google API. Once enabled, we were able to acquire a JSON key to make credentials available for our shared account. We tested our account to ensure proper connectivity with Google's Vision API. We created a "bucket" on the cloud storage, where we stored a random image for object detection, as shown in Figure 5. The picture on the left of the figure was the image used for detection, and the right


```
  "mid": "/m/09g5pq",
  "description": "People",
  "score": 0.9594623,
  "topicality": 0.9594623
},
{
  "mid": "/m/07bxq",
  "description": "Tourism",
  "score": 0.7592621,
  "topicality": 0.7592621
},
{
  "mid": "/m/03qtwd",
  "description": "Crowd",
  "score": 0.74990344,
  "topicality": 0.74990344
```


Fig. 5. Image Used for Object Detection and Results with Confidence Scores

shows the results. For example, the program detected that there were people in the image with a confidence score of roughly 96%.

Secondly, we cloned the Google Vision Tutorials repository off of GitHub and downloaded the necessary packages for Python. We were able to successfully take pictures of different famous logos and classify them using the programs located in the repository. Furthermore, we were also able to use the camera to take pictures of people expressing different emotions for facial recognition processing. For example, if we were to capture an image of someone smiling, the results would indicate high unlikeliness that the person is angry. An example of these results are shown in Figure 6.

B. Connecting the Raspberry Pi to Microsoft Azure Hub

We begin this subsection by summarizing our objectives for this experiment. Our objectives are as follows:

- Our learning objective was to learn how to create an Azure IoT hub and to learn how to connect our Raspberry Pi with a BME280 sensor.
- Once connected, our goal was to collect sensor data by running a sample application on the Raspberry Pi.
- Upon collecting data with our Raspberry Pi, our next step was to transfer our sensor data to our IoT hub.

To create an IoT hub, we connected the Raspberry Pi to the Microsoft Azure Hub, similarly to how it was connected to Google Cloud Vision in the previous section. The IoT hub receives the temperature data sent from our Raspberry Pi. Once the data was sent to the IoT hub, we were able to to offload tasks that were too computationally-intensive to perform on the Raspberry Pi itself. The server then computes the tasks and returns the results.


```
  "angerLikelihood": "VERY_UNLIKELY"
"blurredLikelihood": "VERY_UNLIKELY"
    "boundingPoly": {
        "vertices": [
```


Fig. 6. Results for Facial Recognition

V. Methodology

In this section, we discuss our implementation of a testbed utilizing a Raspberry Pi as our node and the Azure platform as the computing edge. To demonstrate the process of data handling between the Raspberry Pi and the Azure server, we designed and performed two experiments: a weather predictor and a face recognizer.

A. Weather Prediction Using Azure Stream Analytics

For the weather predictor, we deployed a sensor onto a breadboard to record the temperature, humidity, and pressure of a confined room. The recorded data was sent to the Azure IoT Hub to measure the differences in processing capabilities and computational advantages. From the IoT Hub, we were able to access the Azure weather prediction model as a web service and subsequently observe the output of the model. Figure 7 illustrates the unified process for the weather prediction experiment, which can be summarized as the following:

- We connect the BME280 sensor to the Raspberry Pi and verify our connection to the Azure IoT Hub with Visual Studio.
- We collect temperature and humidity sensor data to send to the IoT Hub.
- We create an Azure Stream Analytics job to access the weather prediction model.
- Our model uses a logistic regression machine learning algorithm, as indicated in Figure 11. Logistic regression for binary classification finds a hyper-plane that separates our data in two classes: rain or no rain. More information about the specifics of logistic regression can be found here [18].
- The output of the overall process is the prediction of whether it will rain or not. In our Microsoft Azure Storage, we can see the input (sensor readings) and the output (weather prediction) of our model.

This process is detailed in the following subsections. We also discuss the preliminary software downloads we needed in order to read and send messages directly from the device to the cloud, and vice versa.

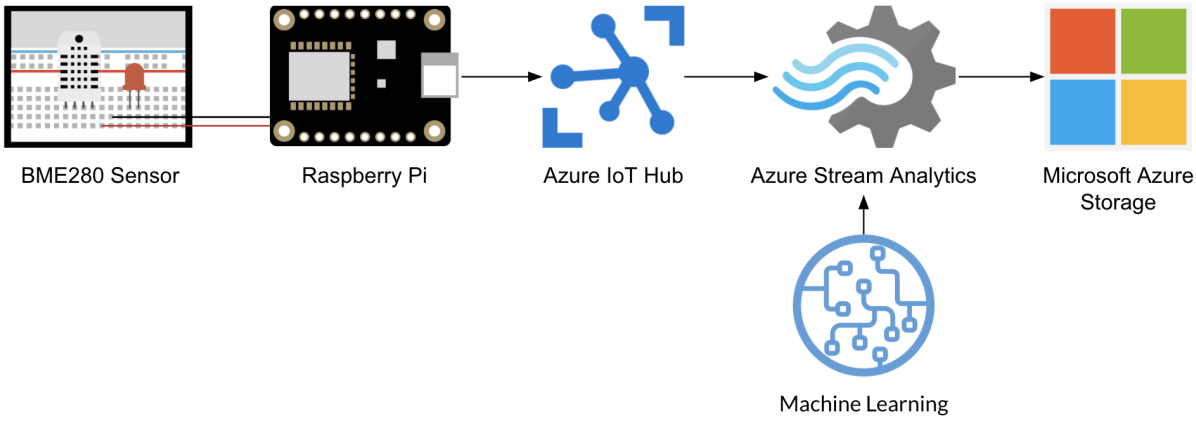

Fig. 7. Testbed Setup for Weather Prediction

1) Raspberry Pi and Sensor Setup: In order to collect sensor data, we installed the BME280 Temperature Humidity Pressure Sensor onto a breadboard using jumper wires as shown in Figure 8. As demonstrated in the figure, the jumper wires connect directly from the Raspberry Pi onto the ports of the sensor. To power the breadboard from the Raspberry Pi, we connected the first pin of the Raspberry Pi to the fifth pin on the sensor labeled "VDD". Next, we grounded the breadboard by connecting a wire from the sixth pin of the Raspberry Pi onto the seventh pin labeled "GND" on the sensor. The next two port connections were responsible for the operation of the sensor: we connected pins three and five from the Raspberry Pi onto sensor pins denoted by SDI and SDK, respectively. The LED was placed arbitrarily on the breadboard, connected to VDD and GND ports. Finally, the sensor was soldered onto the board for stability. The details of these connections can be found in [13], [14].

2) Retrieving Data from Sensor: To validate our sensor implementation, we ran sample BME280 sensor scripts to measure the temperature, humidity, and pressure of the room. However, to run the scripts, we first had to verify that the sensor was deployed onto the correct address. In [15], we had the address of the sensor to be 0x77, but the sample scripts required the sensor to be addressed in 0x76. To fix this, we edited the program to run on our current address by troubleshooting through the script provided. The breakdown of the script is as follows:

- The script imports functions and libraries in Python that enables the sensor to read the temperature, humidity, and pressure of an area.
- The script calls the 'readBME280All' function, which sets the oversampling and mode of the sensor, reads the raw data recorded, calibrates the readings of the sensor, and prints the readings after refining the data.

After running the script, we can observe the readings of the sensor from Raspberry Pi terminal, as demonstrated in the table of Figure 9. The sensor recorded an average temperature of $25.01°C$, which equates to approximately $77.02°F$, close to the temperature of the room.

3) Sending Data to the Microsoft Azure Server: After retrieving data from the sensor, it is crucial to confirm that the data is successfully sent and received by the storage of

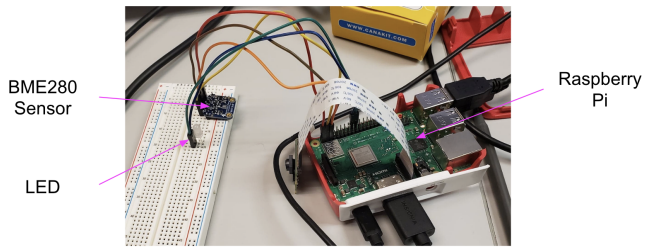

Fig. 8. Raspberry Pi to BME280 Sensor Implementation

```
[IoTHubMonitor] [2:42:47 PM] Message received from [RaspberryPiNode]:
"{\"temperature\":25.01,''humidity\":18.0105}"
[IoTHubMonitor] [2:42:48 PM] Message received from [RaspberryPiNode]:
"{\"temperature\":25.01,''humidity\":18.0105}"
[IoTHubMonitor] [2:42:49 PM] Message received from [RaspberryPiNode]:
"{\"temperature\":25.01,''humidity\":18.0105}"
[IoTHubMonitor] [2:42:50 PM] Message received from [RaspberryPiNode]:
"{\"temperature\":25.01,''humidity\":18.0105}"
[IoTHubMonitor] [2:57:27 PM] Message received from [RaspberryPiNode]:
"{\"temperature\":25.01,''humidity\":18.0105}"
[IoTHubMonitor] [2:57:27 PM] Message received from [RaspberryPiNode]:
"{\"temperature\":25.01,''humidity\":18.0105}"
```

Time	Temperature	Humidity
2:42:47 PM	25.01	18.0105
2:42:48 PM	25.01	18.0105
2:42:49 PM	25.01	18.0105
2:42:50 PM	25.01	18.0105
2:57:27 PM	25.01	18.0105
2:57:27 PM	25.01	18.0105

Fig. 9. Messages Received and Temperature & Humidity Recorded

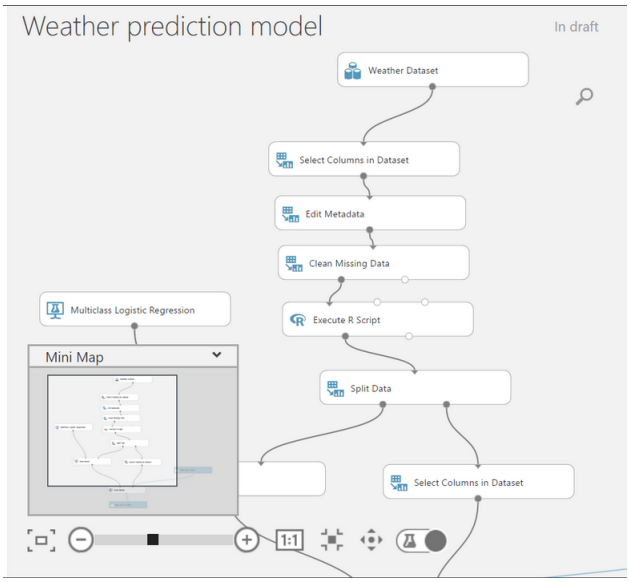

Fig. 10. Steps in the Azure Weather Prediction model

our Azure IoT Hub server. To handle incoming and outgoing messages, we downloaded Visual Studio Code and connected the software to our IoT hub account as described in [16]. This software allows us to send messages from our CPU to the server, even in situations where our Raspberry Pi device fails to send messages directly to the IoT Hub. Because these situations commonly arise in data transmissions, it is important to verify that the data messages are successfully received. In our case, we verified our connection to the server by manually sending temperature and humidity readings recorded from the previous section, as shown in Figure 9. The Visual Studio software prints the time that the messages were received, as well as the content of the messages.

4) Machine Learning Using Azure Stream Analytics: After successfully confirming our connection to the Azure server, we were tasked with deploying the weather prediction model as a web service, as described in [17]. To deploy our model as a web service, we needed to access the 'Azure Stream Analytics' block as previously indicated in the unified process in Figure 7. The importance of this block is that we can continually record and send data to the web service for predictions. Once deployed, the weather prediction model can then be accessed in order to process the data-stream generated by the sensor.

The complete weather prediction model can be observed in Figure 10. Once the web service receives the sensor data from the Raspberry Pi, we begin the data cleaning process. For data cleaning, it is necessary that we look and impute for any missing data values. For example, it is possible that a temperature reading could have not been received from the sensor when sending data to the Azure IoT hub. To impute this value, we take the empirical average of all the temperature data and fill in the missing temperate values. This same process would be replicated if a humidity reading were to be absent. Once data cleaning and processing was completed, we execute our script for weather prediction. The machine learning algorithm that we used for weather prediction was logistic regression. Since our task was to measure computation time rather than achieving accuracy, the model was not cross validated for accuracy improvement. However, given our test results, we can conclude that the model was quite accurate and efficient.

After running this job was ran, it returned a query to our designated Azure IoT Hub storage location. Here, we can view the data inputs and prediction outputs, as well as the specific confidence level. Sample outputs and predictions are shown in the Results section.

B. Facial Recognition Using Azure Virtual Machine

Facial recognition is the second experiment we performed to demonstrate the advantages of using the Microsoft Azure server as the computing edge. Because facial recognition is computationally-expensive, the advantages of using MEC architecture become readily apparent. The objective was to process images of different faces taken by the Raspberry Pi camera on the CPU and record the time it took to process these images. To analyze the differences in computational performance, we sent the same images to the edge server and recorded the time it took to process the same task. In the following sections, we discuss in detail the setup of this experiment.

1) Raspberry Pi Camera Setup and Taking Images: In order to begin training our facial recognition model, we first needed to setup the Raspberry Pi camera to take images and store the images into the Pi storage and edge server. For each person in our group, we took 5 images, each with different angles (i.e. facing up, facing down, etc.). Taking images from different angles can aid our model to make more accurate predictions when testing. The setup for the camera is quite simple and can easily be implemented by following the instructions in [21]. However, it is important to note that the Raspberry Pi camera can result in images with high dimensions, and must be downsized before processing. This can be achieved by either cropping the images to be smaller, or using software such as OpenCV for downsizing. Once proper images are taken and stored, the images are also sent to the Azure IoT hub to later be accessed for processing.

2) Sending Images to Microsoft Azure Server: The images taken for training the model needs to be successfully sent to the Microsoft Azure server to be later used for processing. Since our dataset is larger and in different format than the messages from the Weather Prediction experiment, it is difficult to send the data through Virtual Studio as previously done. There are two options in sending the images: (1) use linux commands to send a .zip file to the Azure server, (2) save the images to a Github repository and clone them on the server. (1) is a rather difficult task, as managing from the local server to the edge server can be challenging if not performed correctly. In this paper, we employ (2) to store in the images on the edge server.

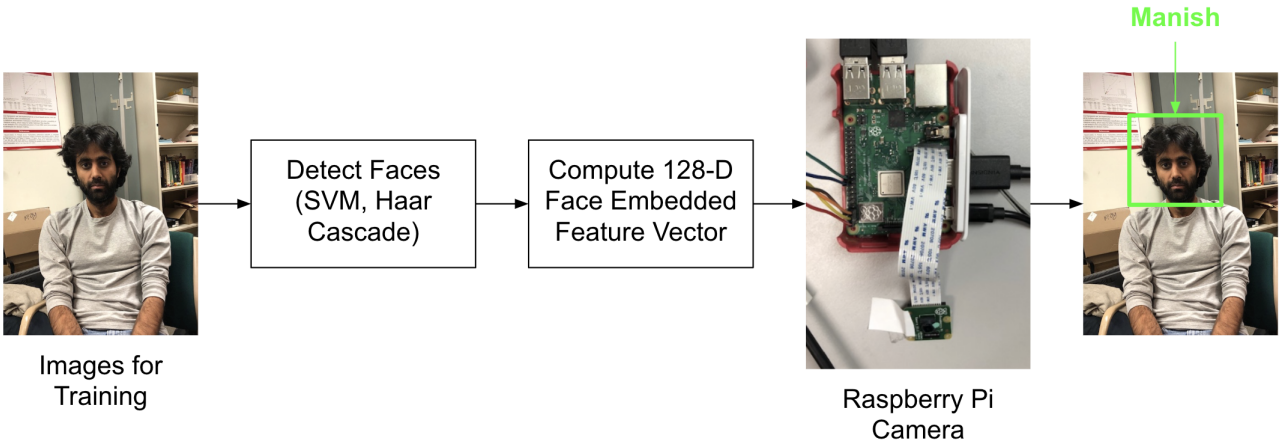

Fig. 11. Facial Recognition Process

3) Processing Images for Facial Recognition: The facial recognition process is shown in Figure 11. Once offloading and storing of pictures is complete, we begin the image processing step for our model. In order to process the images, we utilize two machine learning mechanisms, linear support vector machines (SVM) and Haar cascade to detect the faces in the images. Once the facial features are recognized from the training images, we embed the features into a 128-dimensional vector as an input to our model. These vectors are compared with the other feature vectors to make a finalized facial prediction. More information about the setup for this model can be found in [22].

As one can infer, the processing of these images from machine learning to a 128-dimensional feature vector can be a computationally intensive task, especially for a Raspberry Pi. The time to process these images will be recorded on the Raspberry Pi and on the edge server to compare the differences. The results are shown in the following section.

VI. Results

In this section, we observe the results from deploying a computationally-intensive machine learning task in two different ways. One method will involve deploying the task to Microsoft Azure in order to process the data. The other method will carry out the computation directly on the Raspberry Pi microprocessor. We will plot the computation times to compare the performance of the two different methods in order to determine the most efficient strategy.

```
'Weather Prediction Model' test returned
["25.55", "18.059", "NO", 0.4899410003791]

Result:['Temperature', 'Humidity', "Scored Labels", "Scored Probabilities"]
```

Fig. 12. Weather Predictions with Sensor Data

A. Weather Prediction Results

Figure 12 shows weather prediction results - given a temperature reading of $25.55°C$ and a humidity reading of 18.06%, it is unlikely that it would rain, with a scored probability of 0.4899. The decision process of rain is done by thresh-holding the scored probabilities. If the scored probability was greater than 0.50, then it would predict that it would rain, and a scored probability of less than 0.50 would indicate no rain. A scored probability of exactly 0.50 would be chosen at random.

As described in the methodology section of this paper, the data in this experiment is a livestream of temperature and humidity readings from each individual sensor to the edge node. This setup allowed us to circumvent the cost of storing the temperature data on the mobile node itself, allowing us to use cloud storage to store the readings. While efficient in storage, this experiment was inefficient in comparing the performance of the two methods, as the task of weather prediction was not cumbersome enough to cause a significant difference in computation times. When processing on the Raspberry Pi itself, we get the same results, with nearly the same time to process the data. This can link back to one of the research challenges faced when exploring MEC. It is difficult to infer which tasks should be offloaded to the edge server, and which tasks should remain to be computed on the mobile node.

B. Face Recognition Results

In Figure 13, we can observe the results from our face recognition model. In that given image, our model was able to make 100% accurate predictions on recognizing each person. These results are made from comparing the 128-dimensional feature vectors created from processing the images on the Raspberry Pi. To measure the computational differences, the images were processed from a Microsoft Virtual Machine, available in the Azure portal. All the scripts and images needed for training were offloaded to the server as stated in the previous sections. Once the virtual machine was implemented, we ran the codes to measure the differences in the time it took

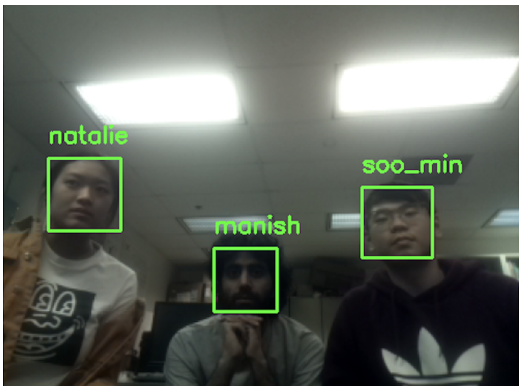

Fig. 13. Face Recognition Model Predictions

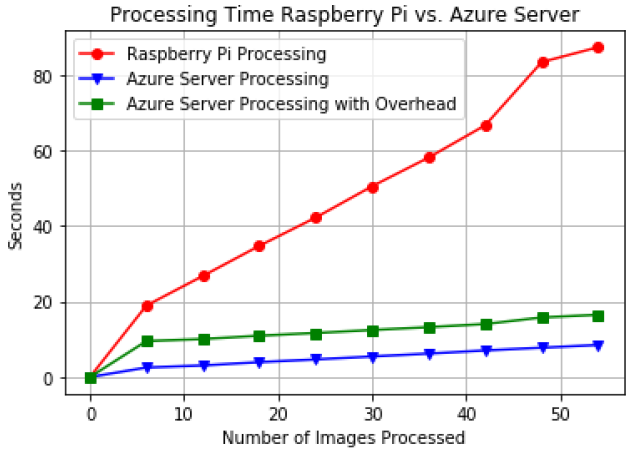

Fig. 14. Computational Differences in Processing Time

to process the feature vectors. The plots of these times are displayed in Figure 14.

Figure 14 clearly shows that processing on the edge server for face recognition is much better than processing on the Raspberry Pi. As the number of images to be processed increases, the processing time linearly increases. At 54 images, the time it took for the Raspberry Pi to process the images was approximately 87.32 seconds, and as for the edge server, it took approximately 8.491 seconds. It is also important to note the overhead when dealing with the edge server. The overhead is the time it takes for the images to be sent from the Raspberry Pi to the Microsoft Azure server. In each instance, it took approximately 6 to 7 seconds to send the data, which added to the Azure server processing time is still significantly faster than the Raspberry Pi.

VII. Conclusion & Future Work

In this paper, we began to explore real-time applications accessible to a Raspberry Pi, such as image recognition and weather prediction. Our goal was to highlight the advantages of offloading certain tasks to the edge of the network rather than performing them locally.

The weather prediction task was not computationally demanding enough to produce a dramatic difference in computing locally on the Pi, versus using MEC. Subsequently, we chose to explore the task of facial recognition. In this case, the computational power of our Raspberry Pi was not sufficient to quickly perform image recognition on the photos it captured. We worked around this limitation by sending the image data to the Microsoft Azure server. We observed the advantages of this setup by measuring the time needed to compute the results; using the server to perform facial recognition took significantly less time than computing directly on the Raspberry Pi, even with the overhead of data transmission.

Our future work in this realm would investigate ways to improve the efficiency of facial recognition and related applications. We would like to test different algorithms to perform the identical task. For example, future work could include performing face recognition with something more computationally expensive like a deep neural network rather than machine learning algorithms. Doing so would help us to isolate the most computationally-expensive components in the MEC pipeline, and ultimately point us towards a more efficient computing paradigm. Additionally our future work would include adapting our model of an MEC architecture to better fit reality by using a less powerful node as our edge node, as edge nodes will typically have less resources available to them than the plethora of power available to the cloud. This would allow us to test the feasibility of MEC architecture with more reasonable real world constraints in future works.

Appendix

In this section, we will explain how to complete the face recognition as shown in Figures 13 and 14. We will assume that the necessary packages (such as Python and OpenCV) are already downloaded and ready to execute. Note that this file is written to be executed for the Raspberry Pi camera. If using an external camera, simply comment out the video stream initialization with the Pi camera in the 'pi-face-recognition.py' script.

1) The scripts will only run if the necessary packages are correctly installed and updated. Run the following lines of code if not already installed: '*pip install dlib*', '*pip install face_recognition*', '*pip install imutils*'
2) Open the '*pi-face-recognition.zip*' file and locate the '*dataset*' file.
   - In the 'dataset' file, we have images of the authors (Soo Min, Natalie, and Manish) for training. If you would like to change the training dataset, make a new file with your name (or the person being recognized) and place the images in the file.
   - For accuracy, we took six images from six different angles of each person for training. For reference, please look at the images of the different angles we used.
   - **Note:** If the images taken for training are too large, there is a chance that the Pi will crash upon execution. Make sure the images are of reasonable size (resolution) before execution.

3) Run the following line of code: '*python encode_faces.py –dataset dataset –encodings encodings.pickle –detection-method hog*'
   - **Note:** The first three hyphens in the code are actually two hyphens, not one.
   - Running this code with return a encodings.pickle file, while contains the 128-dimensional face embeddings for each face in our dataset.
   - If you would like to generate the graph in Figure 14, record the time it takes to process these images into the pickle file. This can be done using time.time() in Python.
   - Repeat this process for different amounts of images to be processed and use Matplotlib to plot.
4) Run the following line of code: *python pi_face_recognition.py –cascade haarcascade_frontalface_default.xml –encodings encodings.pickle*
   - **Note:** There are double hyphens in front of "cascade haarcascade" and "encodings encodings.pickle".
   - Running this line of code should access the Raspberry Pi camera and make face recognitions based on the trained images.
5) To run this code on the server, use Microsoft Azure's Virtual Machine and upload all the necessary files needed to compile. Repeat all steps and record the overhead and time it takes to process the images.

REFERENCES

[1] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B. Letaief. A Survey on Mobile Edge Computing: The Communication Perspective. *IEEE Communications Surveys & Tutorials*, vol. 19, issue 4, Aug. 2017.

[2] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal*, vol. 3, no. 5, October 2016.

[3] Yaser Jararweha, Lo'ai Tawalbehb, Fadi Ababneha, Abdallah Khreishahc, Fahd Dosari. Scalable Cloudlet-based Mobile Computing Model. *Procedia Computer Science* vol. 34, pg. 434-441, 2014.

[4] Ke Zhang, Yuming Mao, Supeng Leng. Energy-Efficient Offloading for Mobile Edge Computing in 5G Heterogeneous Networks. *IEEE Access* vol. 4, pg. 5896 - 5907, Aug. 2016.

[5] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher and Valerie Young. Mobile Edge Computing: A key technology towards 5G. *ETSI White Paper No. 11* Sep. 2015.

[6] Cheng-Xiang Wang, Fourat Haider, Xiqi Gao, and Xiao-Hu You. Cellular Architecture and Key Technologies for 5G Wireless Communication Networks. *IEEE Communications Magazine*. Feb. 2014.

[7] Pavel Mach, Zdenek Becvar. Mobile Edge Computing: A Survey on Architecture and Computation Offloading. *IEEE Communications Surveys and Tutorials*. Mar. 2019.

[8] Xiang Sun, Nirwan Ansari. EdgeIoT: Mobile Edge Computing for the Internet of Things. *IEEE Communications Magazine*. Dec. 2016.

[9] He Li, Kaoru Ota, Mianxiong Dong. Learning IoT in Edge: Deep Learning for the Internet of Things with Edge Computing. *IEEE Network*. Feb. 2018.

[10] Tom Luan, Longxiang Gao, Zhi Li, Yang Xiang, Limin Sun. Fog Computing: Focusing on Mobile Users at the Edge. *Comput. Sci.*. Feb. 2015.

[11] Dario Sabella, Alessandro Vaillant, Pekka Kuure, Uwe Rauschenbach, Giust Fabio. Mobile-Edge Computing Architecture: The role of MEC in the Internet of Things. *IEEE Consumer Electronics Magazine*. Oct. 2016.

[12] Use Google Cloud Vision On the Raspberry Pi and GoPiGo. https://www.dexterindustries.com/howto/use-google-cloud-vision-on-the-raspberry-pi/

[13] Connect Raspberry Pi to Azure IoT HUB https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-raspberry-pi-kit-node-get-started

[14] Raspberry Pi 2 and 3 Pin Mappings https://docs.microsoft.com/en-us/windows/iot-core/learn-about-hardware/pinmappings/pinmappingsrpi

[15] Using the BME280 I2C Temperature and Pressure Sensor in Python https://www.raspberrypi-spy.co.uk/2016/07/using-bme280-i2c-temperature-pressure-sensor-in-python/

[16] Use Azure IoT Tools for Visual Studio Code to send and receive messages between your device and IoT Hub https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-vscode-iot-toolkit-cloud-device-messaging

[17] Weather forecast using the sensor data from your IoT hub in Azure Machine Learning https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-weather-forecast-machine-learning

[18] Joanne Peng, Kuk Lee, Gary Ingersoll. An Introduction to Logistic Regression Analysis and Reporting. *Journal of Educational Research*. Sept. 2002

[19] J. Hagenauer, E. Offer, and L. Papke. Iterative decoding of binary block and convolutional codes. *IEEE Trans. Inform. Theory*, vol. 42, no. 2, pp. 429—445, Mar. 1996.

[20] T. Mayer, H. Jenkac, and J. Hagenauer. Turbo base-station cooperation for intercell interference cancellation. *IEEE Int. Conf. Commun. (ICC)*, Istanbul, Turkey, pp. 356–361, June 2006.

[21] Raspberry Pi Camera Configuration https://www.raspberrypi.org/documentation/configuration/camera.md

[22] Raspberry Pi Face Recognition https://www.pyimagesearch.com/2018/06/25/raspberry-pi-face-recognition/