**CHAPTER 1: CONTENTS OF THE BASE PAPER**

**Title:** Distributed analytics in fog computing platforms using tensorflow and kubernetes.

**Journal name**: IEEE Transactions on Edge Analytics

**Publisher:** IEEE

**Year:** 2017

**INTRODUCTION:**

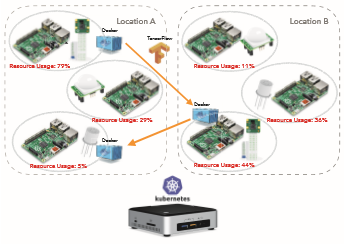
Fog computing and edge computing are effectively the same thing. Both are concerned with leveraging the computing capabilities within a local network to carry out computation tasks that would ordinarily have been carried out in the cloud. The main difference between edge computing and fog computing comes down to where the processing of that data takes place. So, with Fog computing, the data is processed within a fog node or IoT gateway which is situated within the LAN. As for edge computing, the data is processed on the device or sensor itself without being transferred anywhere.

## Related Work:

Edge Analytics have been becoming popular since the evolution of IoT. Cisco proposes fog computing, which has been employed for sensor intensive applications and computational intensive applications. Many researchers start to leverage the concept of fog computing to address IoT’s issues. However, these studies do not consider running applications in a distributed way. For example, our earlier work considers the deployment problem of light weight fog applications, which can be hosted by a single fog devices. In contrast, this paper presents a platform capable to split any fog application into smaller pieces for multiple fog devices. There are several studies presenting their programming models for distributed fog computing. They also discuss how fog computing benefits IoT regards its mobility, short response time, and low cost. In particular they proposed a PaaS programming model for IoT applications. It supports heterogeneous fog devices and allows applications to be dynamically scaled in terms of computing resources. They proposed a distributed data flow programming model for IoT applications in fog platforms. Their framework provides an efficient way to develop IoT applications and coordinate distributed resources. They proposed a programming model that is implemented using container technologies for fog platforms. It provides APIs split applications to communicate with one other. It also includes the algorithms that take the computing resources into account for deployment. Because of the mobility of the fog devices, QoS and work load sensitive migrations are supported. The programming models proposed by these studies are not designed for complex analytics applications in their programming model.

.

**System Overview:**



**Fig. 1. Overview of fog computing platform.**

The Internet-of-Things (IoT) is becoming popular in our daily life. We see IoT applications everywhere, such as smart homes, smart factories, and smart cities. The ubiquitous IoT applications signiﬁcantly change human lifestyle. For example, Echo is a new IoT application manufactured by Amazon, which connects humans to other IoT devices via voice commands. With the growing IoT market, a forecast from Gartner says that 8.4 billion IoT devices will be in use worldwide in 2017, which is 31% more than that in 2016; and the number will increase to 20.4 billion by 2020 . When the number of IoT devices is getting higher, the amount of data is also getting larger. Because IoT devices usually have limited resources, sensor data tend to be sent to powerful data centers for processing. Sending large sensor data to the data centers may congest the networks and overload the data centers. To solve this problem, collaborating several IoT devices and pre-processing the data before transmitting them over the Internet are the keys. We adopt fog computing, which integrates resources from data centers to end devices to run applications in a distributed way. That is, in fog computing platforms, we leverage resources from data centers, edge networks, and end devices. For brevity, we call these devices as fog devices. We need virtualization technologies to transparently deploy IoT applications. Third, to manage large numbers of IoT applications and fog devices, a centralized server in the headquarter is required to keep track of their status.

**Open Source softwares used:**

1. TensorFlow

2. Docker

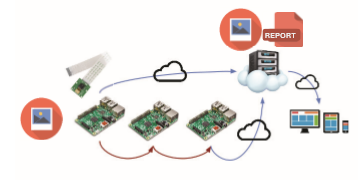
3. Kubernetes

## TensorFlow :

TensorFlow is a library for numerical computations using data ﬂow graphs. This idea of graphs suits the structure of distributed IoT devices, in which fog devices are nodes and the sensor data ﬂows are edges in a graph. Hence, a cluster of weak devices can work together to perform resource-consuming analytics. Moreover, TensorFlow supports a wide spectrum of state-of-the art DL approaches, which are used to analyze the large amount of Sensor data.

TensorFlow helps us build the data ﬂow graph by splitting an application into multiple smaller operators. Operators are the units of deployments. TensorFlow deploys those operators on distributed fog devices and allow them to communicate with one another using multi-dimensional data arrays, called tensors. TensorFlow has a ﬂexible architecture that can deploy applications on desktops, servers, or mobile devices with the same API. Our fog platform leverages on that and deploys various operators on heterogeneous fog devices. With TensorFlow, each applications can be written as a graph of multiple operators. Such multi-operator applications have several advantages, compared to the single-operator ones. First, multi-operator applications solve the problem that fog devices don’t have enough computing resources, since they allow IoT devices to only perform the jobs they are good at. Second, multi-operator applications save the resources and reduce the data transfer amount. For example, if there are two operators on different fog devices needing the same input data, they can get the data from the same operator, and reuse the operator rather than collecting data separately. Last, preprocessing the raw data reduces the burden of networks and servers. IoT applications usually transmit the data to the cloud side, but the incredible amount of data could congest the network and overload the server.

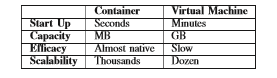
Fig. 2. An example that pre-processes the images before sending them to the cloud data center



# Docker:

Every operator may require different libraries and we can’t install all the libraries on every fog device, which takes too much time and storage. For example, some multimedia libraries, such as OpenCV is quite large for fog devices. Therefore, our platform adopts Docker containers to package the operators with their required libraries into Docker images. Hence, we do not need to install libraries on our fog devices. The operators and libraries are packaged as a container for on-demand deployment.

Table1: Comparisions Between Container and Virtual Machine



Similar to traditional virtualization technologies, container technology is the solution to transparently deploy tasks on heterogeneous devices. However, different from traditional virtualization technologies, container is lighter weight and more scalable as summarized in Table 1. It is suitable for fog devices that don’t have enough computing resources. Besides, the short startup time is also helpful for real-time IoT applications and fast deployments. Furthermore, if developers want to change some algorithms of an application or the conﬁguration of operator graphs, they can easily launch a whole new container, since the overhead of restarting a container is small.

# Kubernetes:

For those containers running on heterogeneous devices at different locations, we need a tool to manage and monitor them, such as Kubernetes, OpenStack, Docker Swarm, and SaltStack. We selected Kubernetes because of its popularity. We gain full control over all the operators in every application using Kubernetes. Kubernetes has an extension, named Heapster, which enables container cluster monitoring and resource monitoring. It also provides error recovery. For example, when fog devices run out of batteries or are disconnected from the Internet, Kubernetes can automatically relaunch new containers to keep serving our applications.

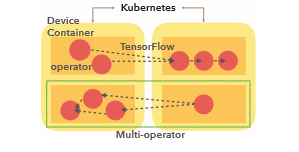


Fig. 3. The architecture of our fog computing platform.

As in Figure 3, Kubernetes plays the role of the centralized server in the headquarter. It monitors the status of fog devices and deploys virtualized operators on them according to the monitored information. Every fog device runs multiple containers, which are part of different applications. If one of the applications needs to be modiﬁed, we just replace that container, which avoids interrupting other apps.

# CHAPTER 2: MERITS AND DEMERITS OF THE BASE PAPER

**Merits:**

1. Easy to setup architecture which can be rolled out in minutes.

2. Distributed analytics results in the auto scaling of the application.

3. Tensorflow helps in integrating multiple operators under the same platform.

4. Extensive library support for mathematical computations.

5. Low weight softwares and binaries from Docker and Kubernetes.

6. Monitoring tools like Prometheus and grafana can be easily integrated.

**Demerits:**

1. The requirements of the cluster like wifi and continuous power supply are not feasible everywhere.

2. The cluster is not mobile, if we are willing to change the location of the cluster, it has to be completely tear down and has to be rebuilt at a newer location.

3. We cannot implement shell communication to the cluster for high level instructions.

4. The infrastructure cost of the setup is high.

5. The chance of getting hacked in the edge platform is very high.

**CHAPTER 3: SOURCE CODE**

**Requirements:**

1. Raspberry pi (3 No. )

2. USB hub for power supply

3. Continuous wifi supply

**Steps**:

1. Creating the K8S cluster

2. Creating a service in the cluster

3. Building and deploying applications in the cluster

**Creating the kubernetes cluster**

1.Add “cgroup\_enable=cpuset cgroup\_memory=1 cgroup\_enable=memory” at the end of the file /boot/cmdline.txt

Now reboot the pi

2. Among all the raspberrypis, pick any one of the node as server and run the command “curl –sfl get.k8s.io –sh|”

3.Now collect the TOKEN at /var/lib/rancher/k8s/server/node-token

The token looks like

**“**K1089729d4ab5e51a44b1871768c7c04ad80bc6319d7bef5d94c7caaf9b0bd29efc::node:1fcdc14840494f3ebdcad635c7b7a9b7**”**

The server is up and running by this time

**Joining the worker node**

On the worker nodes do the following:

$ export K8S\_URL="https://<SERVER\_NODE\_IP>:6443"

$ export K8S\_TOKEN="K1089729d4ab5e51a44b1871768c7c04ad80bc6319d7bef5d94c7caaf9b0bd29efc::node:1fcdc14840494f3ebbcad635c7b7a9b7"

Now install kubernetes on the worker node after exporting the environment variables

The cluster is up and ready with master and worker nodes

**List the nodes:**

$ kubectl getnodes –o wide

**Save: Kubernetes-service-svc yaml**

apiVersion: v1

kind: service

metadata:

name: YOLO

labels:

app: YOLO

spec:

type: NodePort

ports:

-port: 8080

Protocol: TCP

targetPort: 8080

nodePort: 31111

selector:

app: YOLO

**Save: Kubernetes-deployement-dep.yaml**

appVersion:apps/v1

kind: Deployement

metadata:

name: YOLO

labels:

app: YOLO

spec:

replicas: 1

selector:

matchLabels:

app: YOLO

template:

metadata:

labels:

app: YOLO

spec:

containers:

-name: YOLO

Image: functions/figlet:latest-armhf

imagePullPolicy: Always

ports:

-containerPort: 31112

Protocol: TCP

Now apply the configuration:

$ kubectl apply-f Kubernetes-deployement-dep.yaml Kubernetes-service-svc.yaml

Deployemet.apps/YOLO created

Service/YOLO created

The cluster is up running with YOLO container.

**Deploying Code to the YOLO Container:**

We have selected python environment as a deployment, follow the below steps to create a "hello-python" deployment

**Create a new folder for your work**:

$ mkdir -p ~/YOLO && \

cd ~/YOLO

**Now let's create a new Python function using the CLI:**

$ python new --lang python3-armhf hello-python

**This creates three files for you**:

hello-python/handler.py

hello-python/requirements.txt

hello-python.yml

**Now edit the handler.py file**:

def handle(req):

print("The cluster is up and running python" + req)

**Checkout the YAML file “hello-python.yml”:**

provider:

name: YOLO

gateway: http://127.0.0.1:31112

functions:

hello-python:

lang: python

handler: ./hello-python

image: chaitumaverick/hello-python:latest

**let's build the deployment:**

$ docker build -f ./hello-python.yml

**YOLO Application code:**

This code made use of OpenCV library from the Tensorflow.

**Steps:**

1. Open camera

2. Capture the scene using the camera

3. Draw the rectangles for the localized area on the image

4. Close the camera if esc is pressed.

1 import cv2

2 face\_cascade = cv2.CascadeClassifier('haarcascade\_frontalface\_default.xml')

3 eye\_cascade = cv2.CascadeClassifier('haarcascade\_eye.xml')

4 cap = cv2.VideoCapture(0)

5 while 1:

6 ret, img = cap.read()

7 gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

8 faces = face\_cascade.detectMultiScale(gray, 1.3, 5)

9 for (x,y,w,h) in faces:

10 cv2.rectangle(img,(x,y),(x+w,y+h),(255,255,0),2)

11 roi\_gray = gray[y:y+h, x:x+w]

12 roi\_color = img[y:y+h, x:x+w]

13 eyes = eye\_cascade.detectMultiScale(roi\_gray)

14 for (ex,ey,ew,eh) in eyes:

15 cv2.rectangle(roi\_color,(ex,ey),(ex+ew,ey+eh),(0,127,255),2)

16 cv2.imshow('img',img)

17 k = cv2.waitKey(30) & 0xff

18 if k == 27:

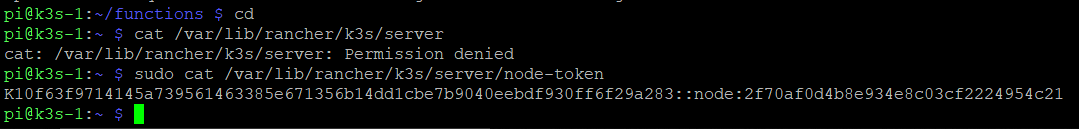
19 break

20 cap.release()

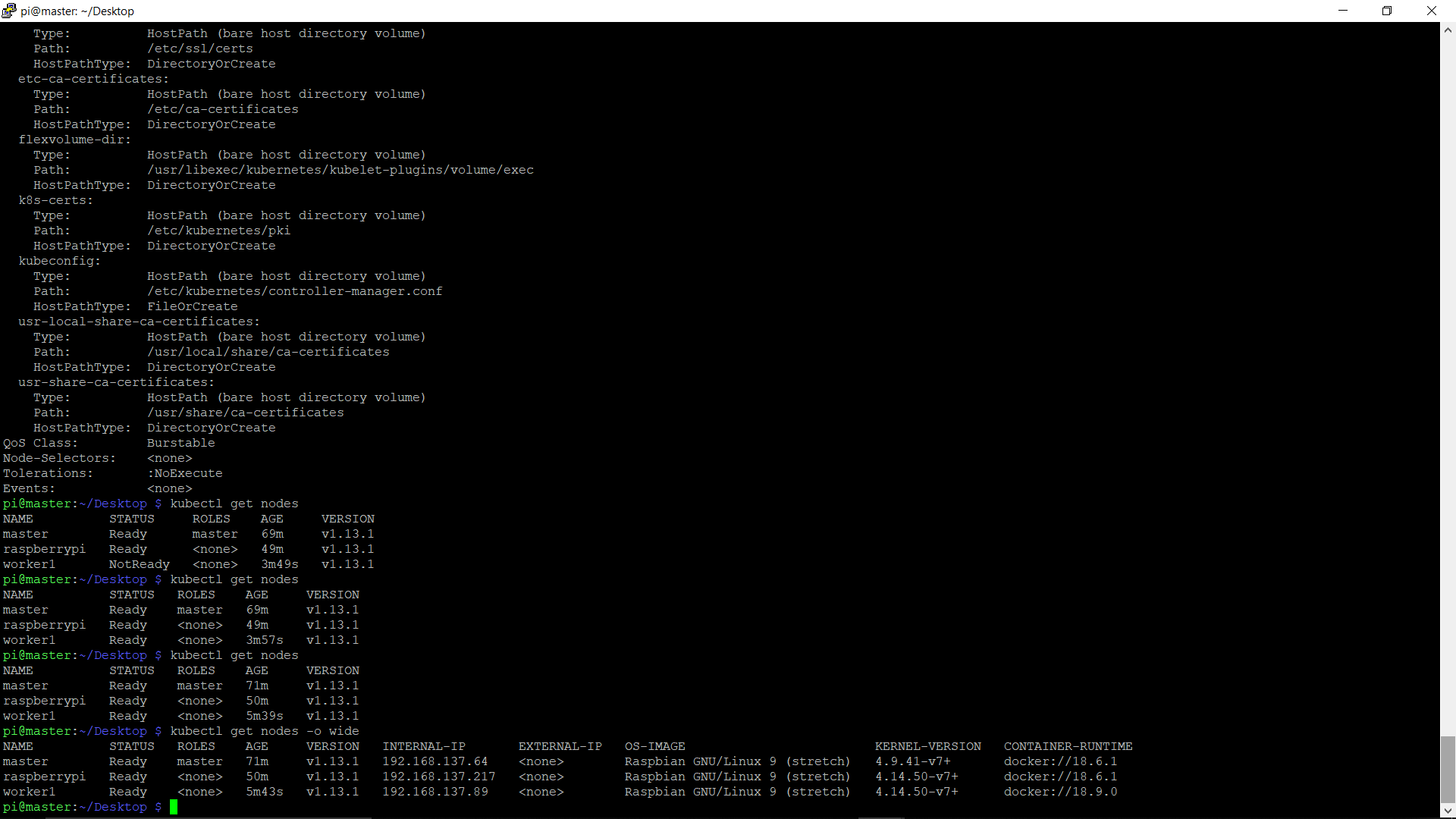
21 cv2.destroyAllWindows()

**CHAPTER 4: SNAPSHOTS AND RESULTS**

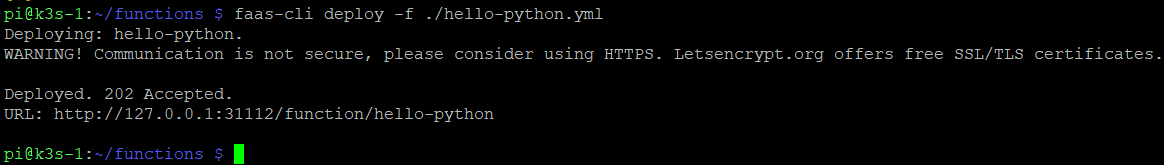
1. Authentication Token for worker nodes:



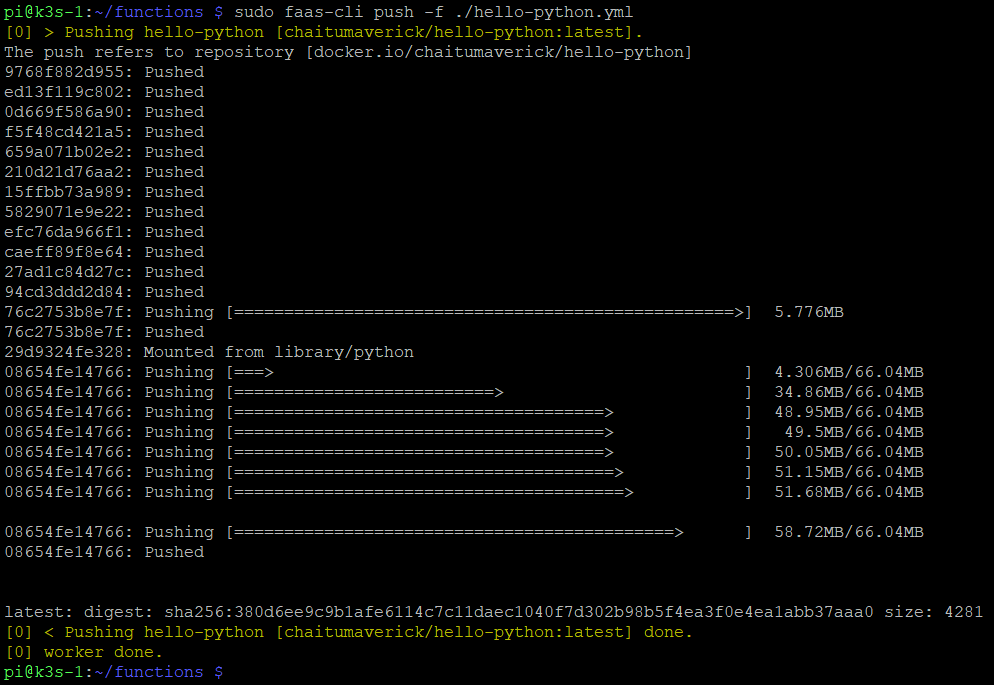
2. The cluster and the nodes in it:



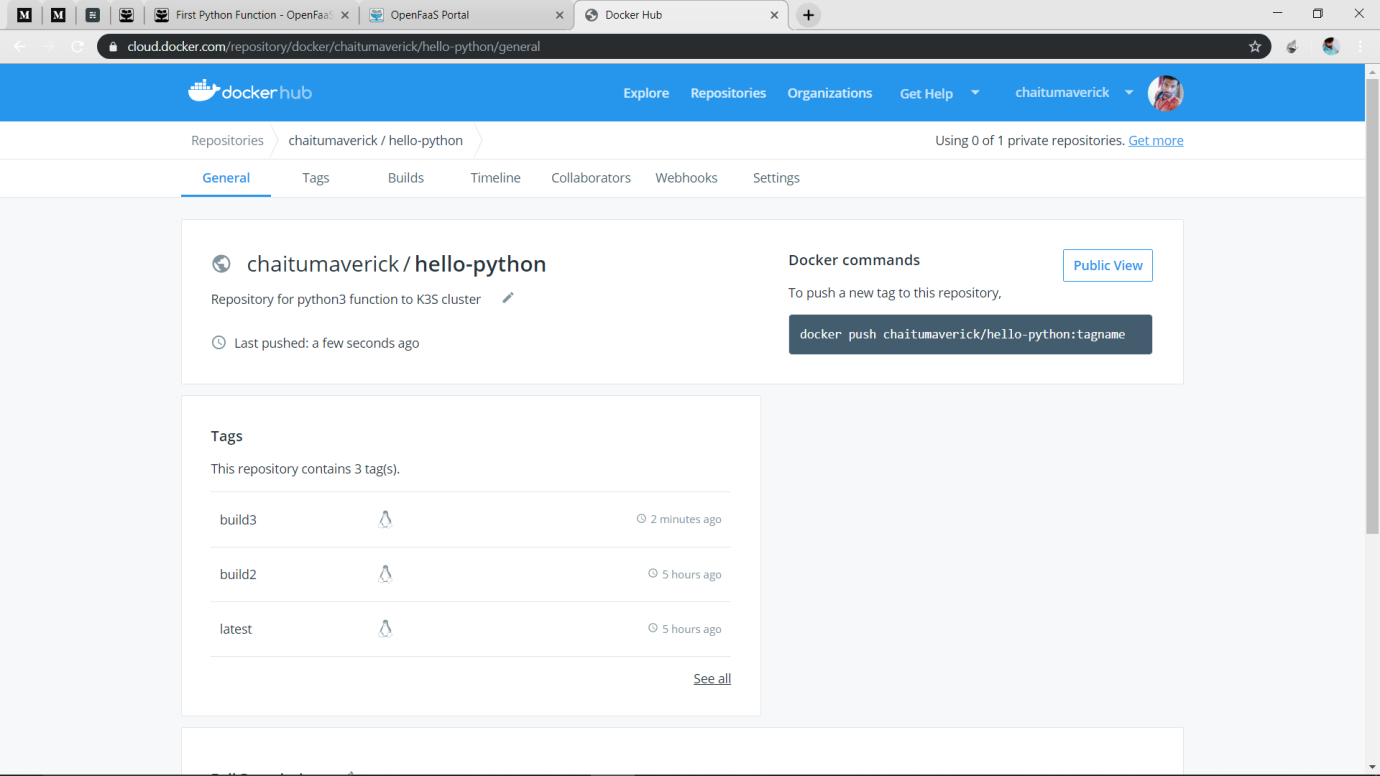
3. Running the containers:



4. Pushing the deployment to the Docker Hub:

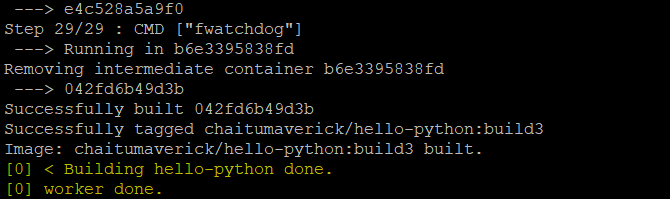


5. Docker Hub repository:



15

6. Docker container running:

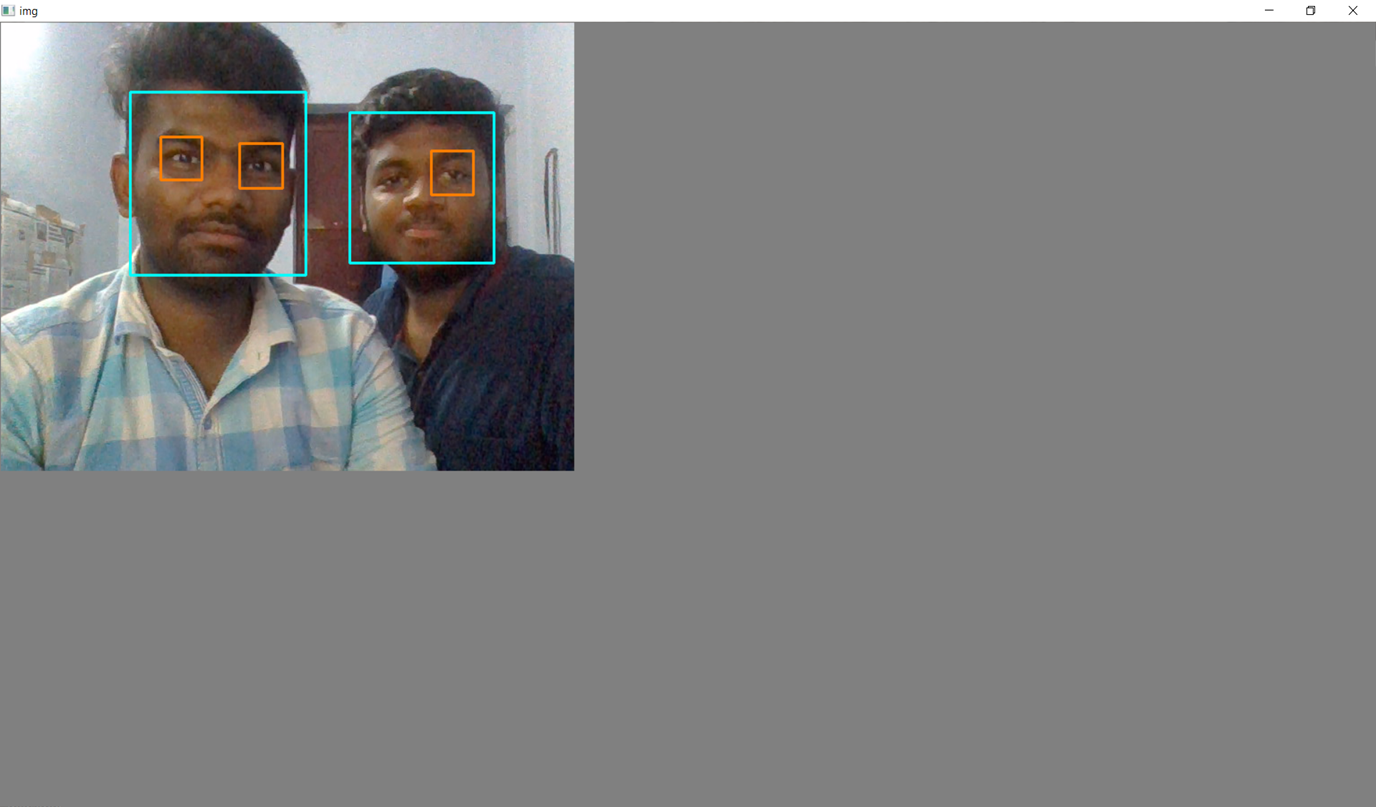


7. Total set up of the cluster:



16

8. YOLO Application results



17

# CHAPTER 5: CONCLUSION AND FUTURE PLANS

This platform integrates resources from data center to end devices. We leverage these devices to run IoT and multimedia applications, which take different sensor data, including camera, microphone, and etc. We focus on implementing a platform, which supports complicated analytics, such as Deep Learning to avoid sending a large amount of raw data to powerful data centers for analyzing. We adopt three popular open-source projects, including TensorFlow,Dockerand Kubernetes to implement our fog computing platform. We conduct extensive measurement studies to quantify the performance.

However, there are some critical issues that have to be considered as our future work. First, the decisions of splitting an application into operators is tricky because different decisions result in differentperformance and overload. Second, required resources of each operator will be affected by makes/models of fog devices and different QoS levels, such as sensing frequency. Third, we need to make optimal deployment decisions to serve more IoT applications on our fog computing platform.

The future work can be done using OpenFaaS , it is a function as a service platform where all the services are deployed as a functions on the cluster using either command line or OpenFaaS GUI. The functions are dynamic in nature, any changes in the function will be effected just by rebuilding the function with the modified. The functions will be assigned an URL through which the computational resource are available. We can map a tunnel to the cluster either using ngrok or Inlet TNS , these applications do port forwarding of the localhost url to the external url, this extended url can be used for remote access.

# 18

# CHAPTER 6:REFERENCES

**1)** “Amazon Echo,” <https://www.amazon.com/Amazon-Echo-BluetoothSpeaker-with-WiFi-Alexa/dp/B00X4WHP5E>**.**

**2)** “Gartner says 8.4 billion connected ”things” will be in use in 2017, up 31 percent from 2016,” <http://www.gartner.com/newsroom/id/3598917>.

**3)**  “Fog computing and the Internet of Things: Extend the cloud to where the things are,” http://www.cisco.com/c/dam/en us/solutions/trends/iot/ docs/computing-overview.pdf, 2015.

**4)** “TensorFlow,” https://www.tensorﬂow.org.

**5)** “Docker,” <https://www.docker.com>.

**6)** “Kubernetes,” http://kubernetes.io/.

**7) “** Xen,” http://www.xenproject.org/.

8**)** “KVM,” <http://www.linux-kvm.org/>.

**9)** F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog computing and its role in the internet of things,” in Proc. of ACM Workshop on Mobile Cloud Computing (MCC), 2012.

**10)**  H. Hong, J. Chuang, and C. Hsu, “Animation rendering on multimedia fog computing platforms,” in Proc. of IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Luxembourg, December 2016.

**11)** H. Hong, P. Tsai, and C. Hsu, “Dynamic module deployment in a fog computing platform,” in Proc. of Asia-Paciﬁc Network Operations and Management Symposium (APNOMS), 2016.

**12)** D. Wu, I. Arkhipov, M. Kim, L. Talcott, C. Regan, A. McCann, and V. N, “ADDSEN: adaptive data processing and dissemination for drone swarms in urban sensing,” IEEE Transactions on Computers, vol. 66, no. 2, pp. 183–198, 2016.

**13)** K. Giang, M. Blackstock, R. Lea, and C. Leung, “Developing IoT applications in the fog: A distributed dataﬂow approach,” in Proc of IEEE International Conference on Internet of Things (IOT), 2015.

19

**14)** S. Shin, S. Seo, S. Eom, J. Jung, and H. Lee, “A Pub/Sub-Based fog computing architecture for Internet-of-Vehicles,” in Proc. of IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2016.

**15)** K. Hong, D. Lillethun, U. Ramachandran, O. B, and B. Koldehofe, “Mobile fog: a programming model for large-scale applications on the internet of things,” in Proc. of ACM Workshop on Mobile Cloud Computing (MCC), 2013.

**16)**  N. Giang, M. Blackstock, R. Lea, and V. Leung, “Developing IoT applications in the fog: A distributed dataﬂow approach,” in Proc. of IoT, 2015.

**17)** E. Saurez, K. Hong, D. Lillethun, U. Ramachandran, and B. Ottenwlder, “Incremental deployment and migration of geo-distributed situation awareness applications in the fog,” in Proc. of ACM International Conference on Distributed and Event-based Systems (DEBS), 2016.

**18)** “OpenStack,” <https://www.openstack.org/>.

**19)** “Docker swarm,” <https://docs.docker.com/swarm/overview/>.

**20)** “Saltstack,” <https://saltstack.com/>.

**21)** M. Y. S. Uddin, A. Nelson, K. Benson, G. Wang, Q. Zhu, Q. Han, N. Alhassoun, P. Chakravarthi, J. Stamatakis, D. Hoffman, L. Darcy, and N.

Venkatasubramanian, “The SCALE2 multi-network architecture for iot-based resilient communities,” in Proc. of IEEE International Conference on Smart

Computing (SMARTCOMP), May 2016, pp. 1– 8.

**22) “**MQTT,” <http://mqtt.org>.

**23)** J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Uniﬁed, real-time object detection,” in Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 779–788.

**24)** “HypriotOS,” <https://blog.hypriot.com>.

20