CSE 546 — Project Report

Sai Teja Padakandla (1217167588)

Ashmi Chheda (1217018972)

Nishant Washisth (1217130460)

1. Problem statement

Clearly describe the problem that you are solving and explain why it is important.

Description:

In this project, we are building an elastic and responsive application that utilizes cloud resources and IoT devices to achieve real time object detection on videos recorded by the devices. We have developed this project using Amazon Web Resources (AWS) and Raspberry Pi. In this project we are using both cloud (AWS) and Edge (Raspberry Pi) to develop an application that provides real time object detection using a lightweight deep learning framework, Darknet to perform object detection on Raspberry Pi.

Importance:

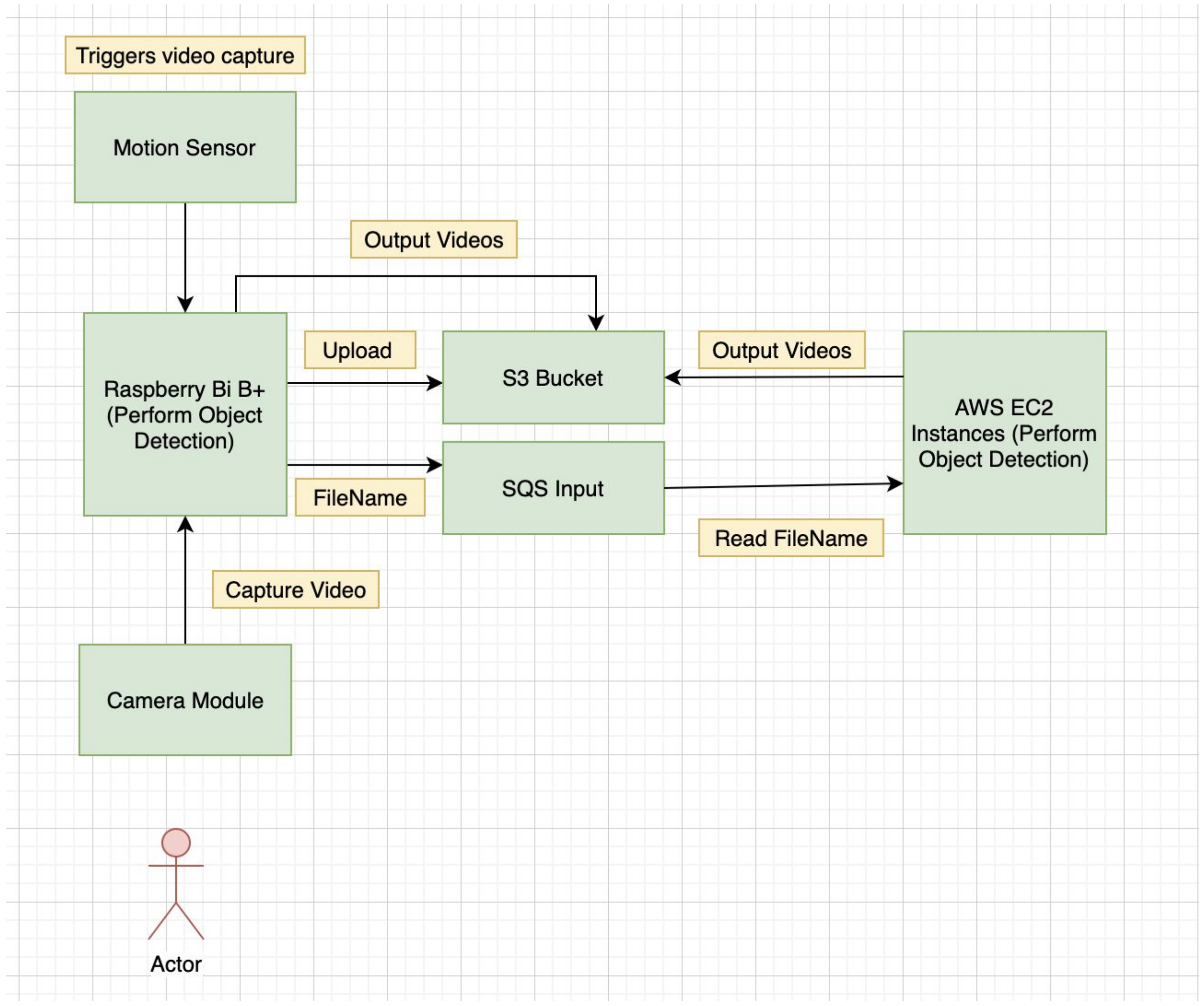
Edge computing is a distributed computing model in which computing takes place near the physical location where data is being collected and analyzed, rather than on a centralized server or in the cloud. This new infrastructure involves sensors to collect data and edge servers to securely process data in real-time on site, while also connecting other devices, like laptops and smartphones, to the network. Edge computing works hand in hand with the cloud to provide a flexible solution based on the data collection and analysis needs of each organization. For real-time collection and analysis, the edge is ideal for certain workloads. At the same time, the cloud can provide a centralized location for large scale analytics.Together they provide real-time and longer term insights into performance and asset performance management thus achieving lesser end to end latency.

2. Design and implementation

2.1 Architecture

Provide an architecture diagram including all the major components, and explain the design and implementation of each component in detail.

Architecture Diagram :



2.2.1 AWS Services used in the project:

1. **AWS EC2 (Elastic Cloud Computing)**

EC2 allows users to create their virtual machine in Amazon cloud for running any application. In our project, we are using EC2 instances (Linux environment images with deep learning framework) for implementing our application at Ras[berry pI and App Tier.

2. **AWS SQS (Simple Queue Service)**

We are using AWS SQS (standard FIFO) in our project for outgoing and incoming messages in our architecture. SQS makes it easy to scale messages effectively and keep the data secure.

3. **AWS S3 (Simple Storage Service)**

In our project, we are using S3 to store results from app instances for persistency. Amazon S3 facilitates, highly-scalable, secured and low-latency data storage from the cloud. Due it's simple web interface, it makes it easy to store and retrieve data on Amazon S3. We can store in buckets with (key, value) object format with any kind of file.

2.1.2 Frameworks used:

We have used the Java Spring Boot framework for our application at App Tier. We have implemented shell scripts for running our logic for auto-start and termination of App instances.

2.1.3 Architecture Explanation

Our architecture consists of an edge computing device which is the Raspberry pi (B+ version). It consists of a camera module and motion sensor. Whenever a motion is sensed by the motion sensor request for video capture is triggered to the Raspberry pi. The application running on the Raspberry pi will start the camera module and a recording for 5 sec is done. All these captured videos will be later uploaded to AWS s3 bucket. Whenever a video is uploaded to S3 bucket corresponding Video file name and file path are stored as a key value pair in SQS.

Now, processing of all these videos is distributed among the Raspberry Pi and AWS instances based on the availability. Raspberry Pi once it processes the video it uploads the results to the results folder in the S3 bucket. Similarly on the cloud side, once the SQS gets messages,AWS EC2 instances equivalent to the number of messages in the SQS will get started. The SQS will be kept polling and an equivalent number of AWS EC2 instances will be started/allocated whenever new messages are seen in the SQS. There are 20 EC2 instances created in total in order to achieve lesser end to end latency. Darknet applications are installed on all these instances. Once the EC2 instance completes its task of object detection it will store the result under the results folder in S3 bucket and wait for a new task for around 40 seconds. If no new message processing request is received this will go back to the stop state. The Load balancing algorithm is efficient and will process maximum upto 20 requests at a time and when there are more than 20 messages in the SQS the additional messages are kept waiting and will be allocated to a EC2 instance once any of them gets free. AWS SQS provides a nice feature of visibility timeout which makes messages invisible to other EC2 instances when one EC2 instance is reading from the SQS. We have set the visibility time out of messages in SQS to 5 minutes considering the maximum time for the darknet model to execute. If anything goes wrong and the EC2 instance terminates, then after visibility timeout, that message in SQS will be visible to other EC2 instances. In other cases, if it runs well the message will be deleted from the SQS. Visibility timeout feature provides contention avoidance as well as fault tolerance.

2.2 Autoscaling

Scaling in and Scaling out are done at the application tier. Whenever the new input video to process is uploaded to S3, a message input is put into the SQS respectively. The important thing which we have to make sure is that the scale out logic does not try to create more than the maximum number of instances (i.e. 20).

Most difficult part of auto-scaling was to create an environment which is thread safe. For handling the auto-scaling we have implemented a python script that takes into account two inputs, number of running instances and approximate number of visible SQS messages. The aim of our logic is to match the number of running instances and the number of visible SQS messages,if we can do the processing within 20 resources available.

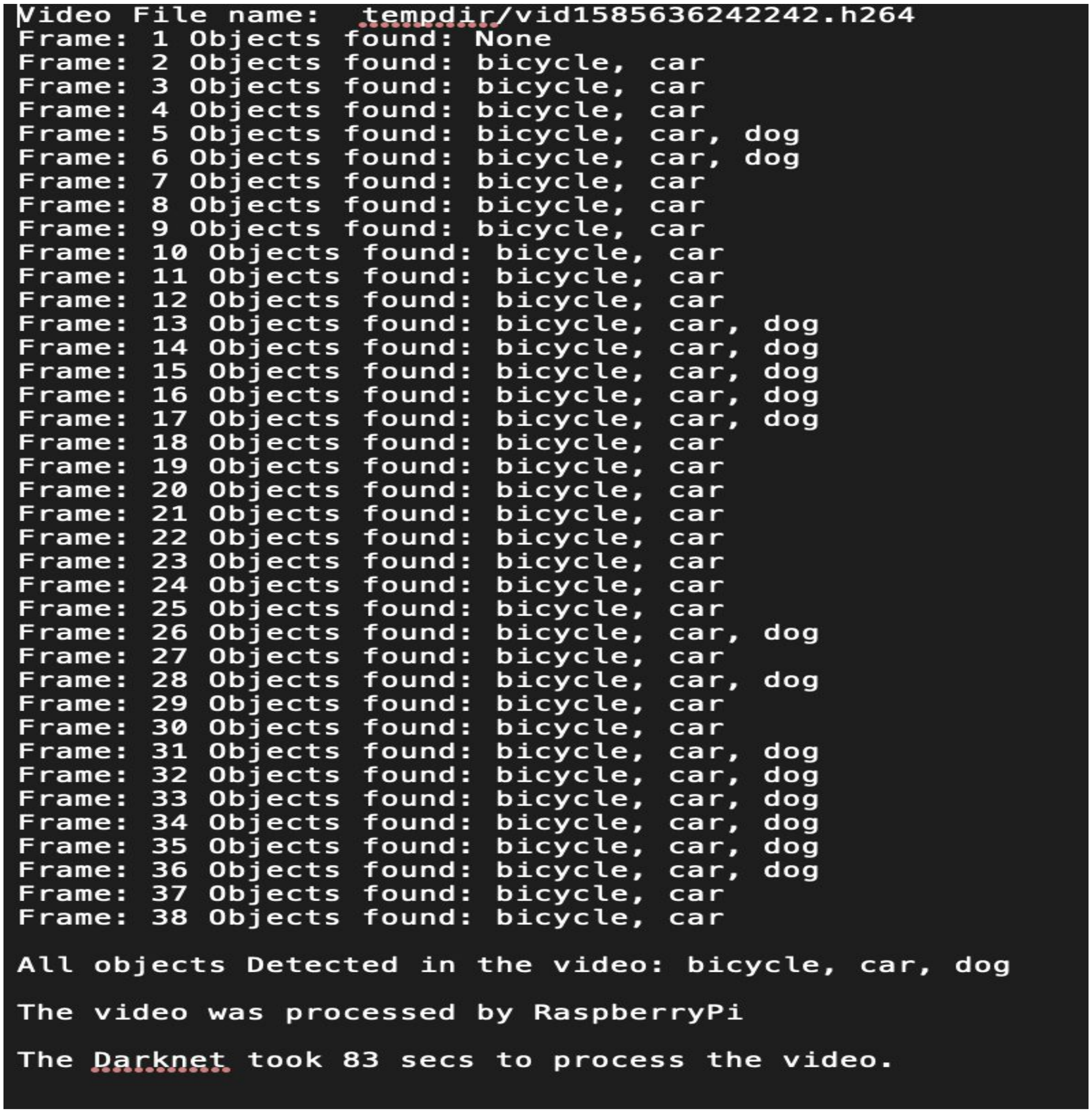
So, our logic checks the number of messages (queue length) in the SQS queue and compares with the number of stopped EC2 instances. Thus it launches the EC2 instances equal to the number of messages in SQS. If the message queue length is more than 20, then we cannot create more than 20 EC2 instances, hence we create 20 EC2 instances that are running and handle the requests. We can say that it is almost like the greedy solution to achieve less end to end latency. In our custom AMI which we are using to create the app Instances have the capability to run the java application using shell script every time it boots up. Then the app instance will start doing the work it needs to do. This is the scale out logic we have implemented at the application tier. We are trying to provide service to the users as fast as possible. Once the EC2 instance finishes processing the video and stores the result in S3 results folder it will be ready to take new requests if available for around 40 seconds. If there are no new message requests, the instances will be terminated. This is the scale-in logic implemented as part of this project.

3. Testing and evaluation

Explain in detail how you tested and evaluated your application.

1. First we created a Spring boot application and tested the pollSensor logic, to check whether the sensor is detecting the videos recorded by the camera and it’s getting uploaded to S3 bucket, and messages getting uploaded to SQS..
2. Then for autoscaling we implemented a Python script that checks the running and stopped instances and created instances according to the SQS length. We tested the script considering scenarios where if the SQS length is > 20 say 25, then according to the 20 instances limit, 20 EC2 (maximum) should be created. For scale in, we added a 40 seconds timeframe for the EC2 instance to terminate.

3. Following is the output obtained after processing the video and we have stored the output result.txt file in S3 bucket.



```
Video File name:  tempdir/vid1585636242242.h264
Frame: 1 Objects found: None
Frame: 2 Objects found: bicycle, car
Frame: 3 Objects found: bicycle, car
Frame: 4 Objects found: bicycle, car
Frame: 5 Objects found: bicycle, car, dog
Frame: 6 Objects found: bicycle, car, dog
Frame: 7 Objects found: bicycle, car
Frame: 8 Objects found: bicycle, car
Frame: 9 Objects found: bicycle, car
Frame: 10 Objects found: bicycle, car
Frame: 11 Objects found: bicycle, car
Frame: 12 Objects found: bicycle, car
Frame: 13 Objects found: bicycle, car, dog
Frame: 14 Objects found: bicycle, car, dog
Frame: 15 Objects found: bicycle, car, dog
Frame: 16 Objects found: bicycle, car, dog
Frame: 17 Objects found: bicycle, car, dog
Frame: 18 Objects found: bicycle, car
Frame: 19 Objects found: bicycle, car
Frame: 20 Objects found: bicycle, car
Frame: 21 Objects found: bicycle, car
Frame: 22 Objects found: bicycle, car
Frame: 23 Objects found: bicycle, car
Frame: 24 Objects found: bicycle, car
Frame: 25 Objects found: bicycle, car
Frame: 26 Objects found: bicycle, car, dog
Frame: 27 Objects found: bicycle, car
Frame: 28 Objects found: bicycle, car, dog
Frame: 29 Objects found: bicycle, car
Frame: 30 Objects found: bicycle, car
Frame: 31 Objects found: bicycle, car, dog
Frame: 32 Objects found: bicycle, car, dog
Frame: 33 Objects found: bicycle, car, dog
Frame: 34 Objects found: bicycle, car, dog
Frame: 35 Objects found: bicycle, car, dog
Frame: 36 Objects found: bicycle, car, dog
Frame: 37 Objects found: bicycle, car
Frame: 38 Objects found: bicycle, car

All objects Detected in the video: bicycle, car, dog

The video was processed by RaspberryPi

The Darknet took 83 secs to process the video.
```

4. Code

Explain in detail the functionality of every program included in the submission zip file.

We have used Java SpringBoot and Python in this project. The application which records the video from RaspberryPi and the application which runs on the EC2 instance has been written in Java SpringBoot, while the auto scaling application has been written in Python. The are two java projects

CSE546pi and CSE546piListener, and a python file SQS.py. CSE546pi and SQS.py run on the RaspberryPi while CSE546piListener runs on the EC2 instances.

Java Packages/Class/Files in the CSE546pi:

1. Cse546piApplication.java: The main SpringBootApplication class from where the application starts running.
2. pollSensor.java: This component continuously checks for input from the GPIO pin where the motion sensor has been connected. If it finds the input value as 1, it starts recording the video for 5 seconds and then it calls the asynchronous function putonS3SQS from S3SQS.java which uploads it to S3 and puts a message with the video name in SQS.
3. S3SQS.java: This component contain functions to provide access to the AWS services i.e. SQS and S3. The asynchronous function localContEvaluation continuously checks for one message from the input SQS Queue, it receives the video name from the message and takes the video from local storage and provides the video to the deep learning framework, Darknet for object classification. A Process builder will execute the deep learning computation instructions. The classification results are stored in a text file that will be written to the S3 Storage bucket using AWS S3 APIs.The respective message from the SQS gets deleted after the processing is completed.
4. Executor.java: This component configures the executor with 8 active threads to support the asynchronous execution.

Java Packages/Class/Files in the Cse546piListenerApplication:

1. Cse546piListenerApplication: The main SpringBootApplication class from where the application starts running.
2. PollSQS.java: This component contains the function keepPollingSQS which executes once the application starts up and then the functions call the fuction retrieveFromS3SQS from S3SQS component.
3. S3SQS.java: This component contain functions to provide access to the AWS services i.e. SQS and S3. The function retrieveFromS3SQS continuously checks for one message from the input SQS Queue, it receives the video name from the message and downloads the video from S3 and provides the video to the deep learning framework, Darknet for object classification. A Process builder will execute the deep learning computation instructions. The classification results are stored in a text file that will be written to the S3 Storage bucket using AWS S3 APIs.The respective message from the SQS gets deleted after the processing is completed. If the application is unable to find any message in the SQS for 40 seconds it will stop itself, which is followed by shutting down the machine.

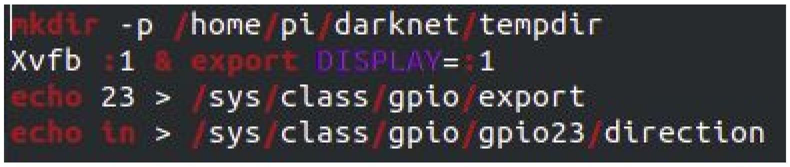
SQS.py:

Auto scaling script which polls the SQS and looks if there are any messages. If there are any messages found, an equivalent number of AWS EC2 instances will be started for computation. This is considered a Scale-up process. Whenever the EC2 instance completes its task it waits for 40 seconds if there is any new job assigned for it. In case no job is assigned, it will stop.

Explain in detail how to install your programs and how to run them.

1. RaspberryPi setup:

   Jar is built from the project CSE546pi and put it in the darknet folder and the following commands are added to the /etc/rc.local file to enable an output channel for the GPIO pin where motion sensor has been connected (pin 25 in this case) and the command to enable the execution of darknet without a display when the RaspberryPi boots up.



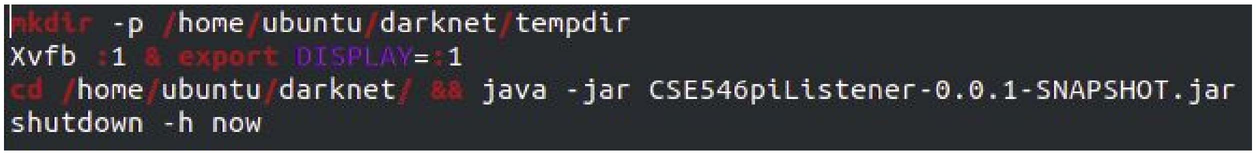
```
mkdir -p /home/pi/darknet/tempdir
Xvfb :1 & export DISPLAY=:1
echo 23 > /sys/class/gpio/export
echo in > /sys/class/gpio/gpio23/direction
```

   The SQS.py file is copied to the raspberryPi to make the RaspberryPi as the autoscaling controller

   To start the application we need to run the jar file and the python file.

2. EC2 instance setup:

   We create an EC2 instance and a Jar file from the project CSE546pi and we put the Jar file in the darknet folder and the following commands are added to the /etc/rc.local file to enable the execution of darknet without a display and then the executing the Jar file and automatic shutdown of the instance once the execution of Jar finish to ensure the autoscaling. Java runtime should be installed on the EC2 instance in order to run Jar file.



```
mkdir -p /home/ubuntu/darknet/tempdir
Xvfb :1 & export DISPLAY=:1
cd /home/ubuntu/darknet/ && java -jar CSE546piListener-0.0.1-SNAPSHOT.jar
shutdown -h now
```

   Finally, we create an AMI from the configured EC2 instance and using that AMI we create 19 other instances to support the auto scaling. Initially all the EC2 instances will be in the stop state.

5. Individual contributions (optional)

For each team member who wishes to include this project in the MCS project portfolio, provide a one-page description of this team member's individual contribution to the design, implementation, and testing of the project. One page per student; no more, no less.