Video Object Detection As A Service

Arnav Chakravarthy
Abhishek Chaudhary
Apeksha Rajkumari

1. Problem statement

The project aims to develop an application for object detection that incorporates edge computing and cloud services. Edge computing bridges the latency gap that is introduced by cloud computing. Moreover, it is more scalable and reliable. Through this project, we aim to gain an understanding of edge and cloud computing. Edge computing is achieved with the help of Raspberry Pi, which we aim to use as the edge computing device. As the aim of the project is to perform object detection, we will make use of camera module functionality along with Pi. We aim to make adequate use of the resources on Pi as well as of the EC2 instances, which will form the cloud computing component of the project. The goal of the project is to minimize the time required to process the videos, that is, the time from recording a video to receiving the result of the object detection performed. The processing will be divided amongst Pi and EC2 instances, which will be referred to as slaves throughout the report. This will help in decreasing the latency. The Pi may perform object detection on the video or send the video to the cloud for the same, based on the availability of Pi. The results are stored in a S3 Result bucket. The cloud processes the videos on EC2 instances that are dynamically spawned.

2. Design and implementation

2.1 Architecture

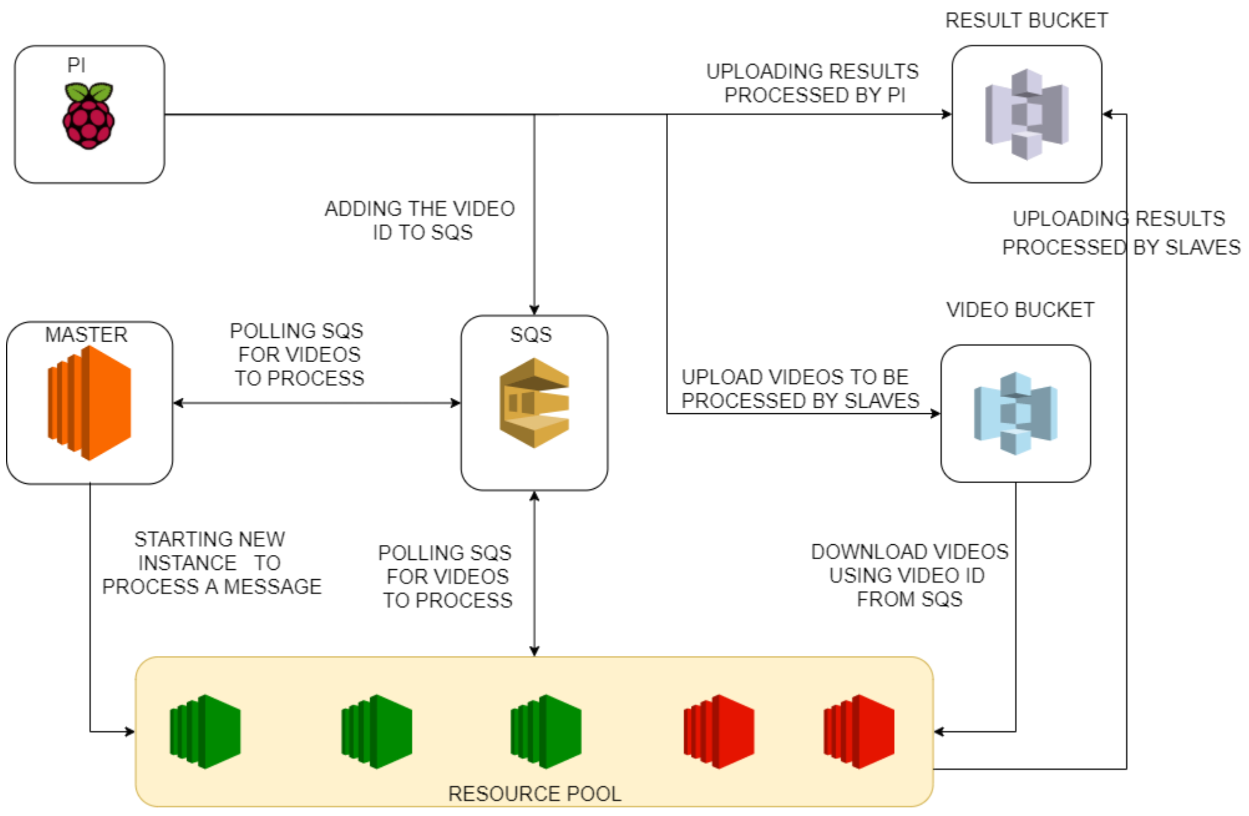
RESULT BUCKET
PI
UPLOADING RESULTS PROCESSED BY PI
ADDING THE VIDEO ID TO SQS
UPLOADING RESULTS PROCESSED BY SLAVES
VIDEO BUCKET
MASTER
POLLING SQS FOR VIDEOS TO PROCESS
SQS
UPLOAD VIDEOS TO BE PROCESSED BY SLAVES
STARTING NEW INSTANCE TO PROCESS A MESSAGE
POLLING SQS FOR VIDEOS TO PROCESS
DOWNLOAD VIDEOS USING VIDEO ID FROM SQS
RESOURCE POOL

Figure 1: Architecture of Proposed System

2.1.1 Explanation of Architecture

The proposed architecture consists of the following components:

1. Raspberry Pi
2. Raspberry Pi Camera
3. PIR Motion Sensor
4. Amazon EC2 Instances
5. Amazon Simple Storage Service (S3)
6. Amazon Simple Queue Service (SQS)

All these components interact with each other to deliver low latency results for real-time video object detection.

The Raspberry Pi is used as the edge computing device which records a video through the attached camera and either processes the video or sends it to our cloud architecture for processing

- In the case where the Raspberry Pi processes the video, it uploads only the results to S3 storage
- In the case where it sends the video to the cloud, it uploads the video to the S3 Video bucket and adds the video ID to SQS, which is polled by the master node

In order to process the videos and detect objects in them (either on the cloud or on the Pi), the system makes use of Darknet, which is an open source neural network framework written in the C programming language.

All the video IDs which are uploaded onto SQS, are picked up by the master node, during its periodic polling of SQS. During each poll, the master node checks the ***ApproximateNumberOfMessages*** and makes use of our **AutoScaling** logic to either scale out or in. In the case of scaling out, the master fetches a message from SQS, assigns a Visibility Timeout (so that it is not processed more than once) and assigns the message to a slave which is basically an EC2 instance. This assignment of message to each slave as well as starting the slave, is done on a seperate thread in order to allow concurrent processing. We SSH into the slaves using a python library called Paramiko and start the execution of the slave program. In the case of scaling in, the master stops the slave instance if it has completed the assigned processing job.

All the slaves (19 in number) are initially in the Stopped state. The master then starts and stops these instances based on our AutoScaling logic.

The slave instances are responsible for fetching the video file using the video ID, which is passed by the master, and processing these videos using the Darknet program. If any error occurs, the slave adds the message back to SQS, to be processed by other slaves, by changing its Visibility Timeout to 0. If the video processing is successful, it uploads the results to our S3 Result bucket. The slave then polls SQS to check whether there are any new messages for it to process. If yes, it processes them and if not, it exits. In case of exit, the master becomes aware of this and stops this instance.

2.1.2 Hardware

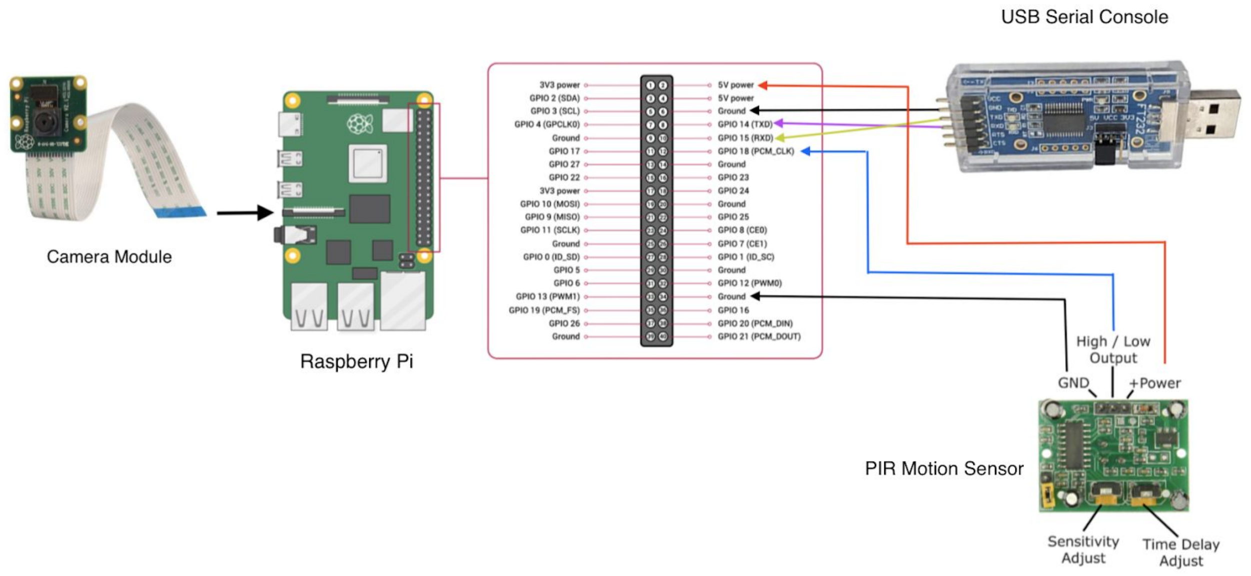

Figure 2: Hardware Setup

The project uses Raspberry Pi to incorporate edge computing. A PIR based motion detector signals the camera attached to the Pi to record and store a video of 5 seconds on detecting any movement. Using Darknet, the Pi performs object detection on the video and stores the result in our Result bucket in S3. If the Pi is already processing another video, the video will be sent to the cloud (Video bucket in S3) for processing.

There are several steps involved in setting up the hardware which have been discussed in *section - Installation Instructions*

2.1.3 AWS Services Used

1. AWS EC2

EC2 instances are virtual machines on which users can run their own applications. In this project, we have created a master instance that is responsible for spawning slave instances based on the number of messages in the SQS queue. A slave instance processes a video using Darknet. Then, it checks if there is any other message in the SQS. If it finds a message, it will process the corresponding video. Otherwise, the slave exits.

2. AWS SQS

SQS is a message queue service provided by AWS that can be used by distributed applications to exchange messages. This is done through a polling model. In this project, we have used a Standard SQS queue to store the keys of the videos stored in S3 Video bucket. The master instance checks the number of messages in this queue to decide the number of slave instances to spawn. The slave instance also checks the number of messages to decide if it should process another video or exit.

3. AWS S3

AWS S3 provides object storage services. In this project, videos received from Raspberry Pi are stored in S3 Video bucket. After object detection is performed on the videos by the EC2 slave instances, the results are again stored in S3 Result bucket. If the object detection is performed by Pi, it sends the results to S3 Result bucket.

2.2 Autoscaling

2.2.1 Video Processing By Raspberry Pi

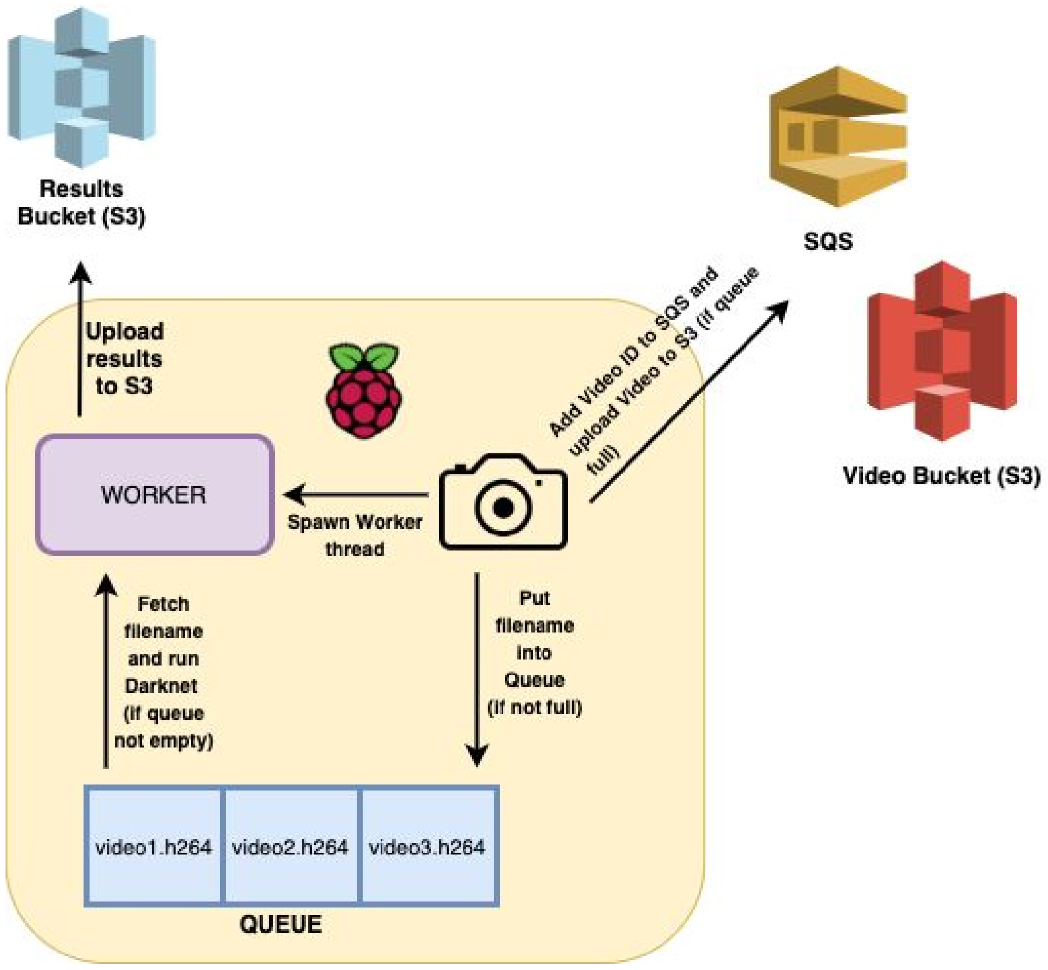
Results Bucket (S3)
Upload results to S3
SQS
Add Video ID to SQS and upload Video to S3 (if queue full)
Video Bucket (S3)
WORKER
Spawn Worker thread
Fetch filename and run Darknet (if queue not empty)
Put filename into Queue (if not full)
video1.h264
video2.h264
video3.h264
QUEUE

Figure 3: Proposed Producer-Consumer Architecture on Raspberry Pi

The goal of the project is to leverage cloud computing for scalability along with edge computing for responsiveness. Hence, in our project solution, we have spent a good amount of time planning on how we can leverage the edge computing on Raspberry Pi along with our cloud computing architecture to deliver low latency results.

We are using a **Producer-Consumer** architecture on the Raspberry Pi, with a fixed size and *thread-safe Queue* module provided by Python. It implements all the required locking semantics to handle thread-safety.

2.2.1.1 Producer

The producer is the camera module, which records a video and based on whether the queue is full or vacant, inserts the video filename into the queue, to be processed by a Consumer thread, or uploads the video to S3 along with adding the video ID to SQS to be processed on the cloud.

We are using a FIFO (first in first out) queue and have set the maximum queue size to 3. The reason we chose this number is that we observed that the Raspberry Pi takes ~1 minute to process a video whereas the EC2 instance takes ~4 minutes (including network latency and fetching video from S3) to process the same video. Hence at a time, the Raspberry Pi can handle 4 videos (1 processing and 3 in the queue), which would take ~4 minutes to process, the same time that an EC2 instance takes to process 1 video.

- If the queue is not full - We insert the filename and path of the video recorded by our Raspberry Pi into the queue, to be picked up by our Consumer (Worker) thread running on the Raspberry Pi.
- If the queue is full - We upload the video file onto our S3 bucket, and insert the video ID (same as that on the S3 bucket) onto SQS, to be assigned to an EC2 instance by our Master node.

2.2.1.2 Consumer

The Consumer or the Worker, is a thread running on the Raspberry Pi, which is listening for messages in the queue. While there are messages in the queue, it picks up a message, processes it using the Darknet program and uploads the results onto our Result S3 bucket. If the queue is empty, the thread remains asleep and does not do anything.

This helps save time as we do not waste time killing and spawning a new thread.

The number of worker threads running on the Raspberry Pi is predefined and configurable. We use only 1 worker thread for the demonstration and while reporting our end to end latency, but have also experimented with multiple worker threads.

2.2.2 Master-Slave Architecture

2.2.2.1 MASTER

This is the node which is responsible for supporting our cloud infrastructure and scaling out as well as scaling in based on our AutoScaling policy.

It is running a Python Script, which periodically polls SQS to check whether there are any new messages. We use the ***ApproximateNumberOfMessages*** attribute to check the number of new/unprocessed messages in SQS.

If there are new messages in SQS, it fetches those messages from SQS, and assigns each fetched message to an EC2 instance (Slaves) present in our Resource Pool which are in the *Stopped* or *Stopping* phase.

If messages are to be assigned to EC2 instances in the *Stopping* phase, we use a Waiter to wait for the Instance to reach the *Stopped* phase.

We also assign a visibility timeout of 4 minutes (based on the time it takes to process one message) to each fetched message, to avoid the same messages to be fetched more than once and therefore to be processed more than once.

We spawn a new thread which is responsible for handling the allocation of each new message to a specific EC2 Instance.

We have also set a maximum resource pool size of 19 EC2 instances (excluding the Master). The Master Node ensures that not more than 19 EC2 instances (Slaves) are running at a time.

We spawn 1 new slave for every new message in SQS, hence Scaling Out.

Every thread executes the following tasks:

1. Starts the instance allocated to it and uses a Waiter to wait for it to start.
2. Fetches the public DNS name of that instance.
3. Uses *Paramiko*, a Python module which implements the SSHv2 protocol, to SSH into its allocated instance.
4. It then starts the execution of the Slave program and assigns the allocated message to

   this Slave Program.
5. Once the program exits, it **Stops** the allocated instance and exits, hence scaling in

   when not required.

2.2.2.2 SLAVE

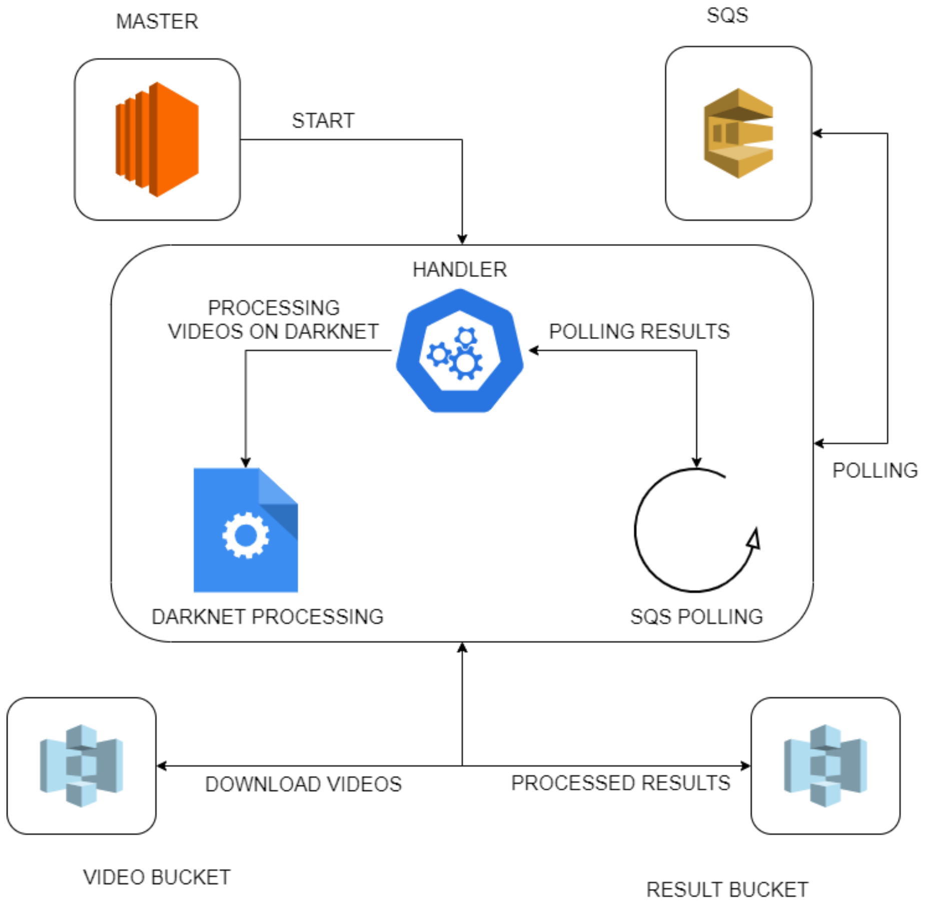

Figure 4: Working of the Master and Slave Instances

Each slave is an EC2 instance that is started by the master upon receiving a message from the SQS. The input message contains the video ID used to download the file from the S3 bucket.

After the file is downloaded, a subprocess is created to handle Darknet and its processing. Slave waits for the processing to get completed (or any exception if it occurs). After the Darknet processing is completed, the result file is parsed to obtain unique objects that were present in the video. The results are then uploaded back to S3.

After processing and uploading is done, we delete the message from SQS and one process cycle gets completed for the slave. One of the design choices we made here was not to terminate the slave but to make it poll SQS to check if it contains any messages or not. This helps in reducing latency significantly as we are not stopping instances after they are done processing and starting a new one for every message. If there are more messages in the queue that an instance can process, those messages are picked for processing using Darknet.

In case we get any exceptions we stop the instance and perform error handling and cleanup.

2.2.3 IAM ROLES AND PERMISSIONS

Our EC2 instances (both slaves and masters) are making use of SQS and S3 services. To make any query to either of these services we need to provide authorization to our instances. One way of accomplishing this is by setting credentials on every instance manually or via some scripts which deploy credentials on every instance. A better way, we thought would be to assign IAM user roles to the instances which would allow them *SQS and S3 access.* Using these permissions, we got a secure way of accessing the SQS and S3 buckets.

2.2.4 ERROR HANDLING

One of the main concerns while processing videos on slaves is error handling. What should happen in case we get an error?

We decided to '*put the message back into SQS*' when that happens. To understand this, we need to look at the SQS visibilityTimeOut parameter.

Let us assume we set visibilityTimeOut = X seconds while polling. So after it is polled, the message will not be seen by any other instance for exactly X seconds. That is we get X seconds of message ownership.

Using this parameter, we decided it is best to set the visibility back to 0 in case any error/exception happens which would allow other instances to process that message. Message receipt handle is used to modify messages in SQS. So, we use a receipt handle for updating the visibility timeout period.

This also helps with fault tolerance as messages are not lost or deleted if the slave is not working properly.

3. Testing and evaluation

For the purpose of testing our implementation we made use of the same strategy that was to be done for demo. We made use of images inside the darknet/data folder namely dog.jpg, eagle.jpg, giraffe.jpg, horses.jpg, and person.jpg. We triggered 10 motions using the motion sensor and recorded 10, 5-second videos. We set our queue size to 3 which means that Pi will process 1 video and have 3 in the queue and then the 5th one will be uploaded to the S3 Video bucket to be processed by the slaves.

This is the sequence we obtained for the 10 videos that were recorded :-

1. Video_0.h264, Video_1.h264, Video_2.h264, Video_3.h264, Video_6.h264 were processed on Pi
2. Video_5.h264, Video_7.h264, Video_8.h264, Video_9.h264 were sent to the S3 video bucket to be processed by slave ec2 instances.

Each slave instance takes around 3 minutes to process the video and there was one instance per video. So, a total of 4 instances were started by the master, each processing a single video and no two slave instances processing the same video.

This process of recording and uploading video files takes close to 2 minutes after which only the processing part remains pending.

Around the 4 minutes mark, Pi finished its processing and all the results are uploaded onto the S3 Result bucket. Then one by one all the slave EC2 instances also finish processing and upload their results to S3 Result bucket. Since there are no more messages in the queue for them to process, slaves are stopped one by one by the master. All the results are uploaded to the S3 Result bucket with the same key as the video ID.

The observed latency is **6 minutes 10 seconds**.

We experimented with the queue size. With queue size as 2, the average observed latency was over 7 minutes. With queue size 4, we didn't observe any advantage as videos on Pi were getting processed sequentially and hence it didn't offer any better latency. Hence, we decided to use a queue size of 3.

4. Code

We are making use of Boto3, which is the AWS SDK for Python to programmatically interact with our AWS services namely EC2, SQS and S3.

Master Service

1. poll_scaling.py

- It is the core logic for our AutoScaling.
- This program polls Amazon SQS every 10 seconds.
- It runs in a never ending while loop and polls Amazon SQS every 10 seconds.
- It calls a function *poll_for_scaling*, which checks Approximate Number of messages in SQS, the number of instances running, pending, stopped and stopping.
- It checks the number of messages in the queue. For each message it receives, it sets a Visibility Timeout of 11 minutes, hence preventing the message from being assigned and processed more than once.
- It checks which instances are stopped or stopping and assigns these messages to each of these instances on a new thread. (Scaling Out)
- In case the instances are stopping, we use a Waiter provided by Boto3, to wait for these instances to reach the *Stopped* state.
- To SSH into each slave we require the public DNS name. Hence, we have to wait for the instance to start, fetch the public DNS name and pass this onto the *slave_thread* function.
- In the *slave_thread,* we use Paramiko to SSH into each slave and execute the slave program which is responsible for processing the passed message.
- Once the processing on the slave is done, the Master closes its SSH connection and then **stops** this instance. (Scaling In)

2. create_instances.py

- This is the program used to instantiate our slave instances.
- We use 19 slave instances in our resource pool.
- Each of them are assigned an IAM role which gives them S3 and SQS access.
- We pass UserData, which clones the repository containing the code that runs on the slave program. UserData also contains commands to install necessary dependencies for the code to run.
- These instances are initialized with the AMI provided to us.
- This is only executed **once**.

Slaves

1. Process_queue.py

a. This forms the core logic of handling processing of videos on slaves.
b. The program receives the first message from the master and starts processing it.
c. The video_id is obtained from the message body and then used to download the video file from the Video bucket.
d. After the file is downloaded, it is processed by Darknet.
e. The program waits for the Darknet processing to be completed and then starts extracting unique objects from the result.txt file.
f. Result.txt file contains the output of Darknet execution, which is frame by frame listing of objects with a certain confidence score.
g. We use this file in function get_objects() to parse it and return a unique object present.
h. This result is then sent to the Result bucket using upload_results() function. If no object is detected then we send “no object detected”
i. After one processing cycle is done, the program once again polls the queue to check if there are any messages there to be processed.
j. If it gets another message, then the processing cycle is repeated again otherwise the program exists.
k. During a processing cycle if any error occurs then the message needs to be put back in the queue i.e we set the visibility to zero using **handle_visibility()** which sets **visibilityTimeOut** value to 0 using the **receipt_handle** value.

Raspberry Pi

1. Surv_threading.py

a. This program is responsible for implementing the producer-consumer architecture that we discussed in section 2.2.1
b. The program takes as an argument, the number of videos to process.
c. The queue size is set to 3 and is fixed.
d. The maximum number of Darknet instances that we can use on Pi is limited to 1. Hence, our maximum number of consumer threads is set to 1.
e. Consumer threads keep on checking the queue for any videos to process and send it to thread function on receiving a video.
f. In the producer section we first record a video using the take_snapshot program provided in the facedetect folder.
g. Then if the queue is not full, we add our video to the queue which will then be picked by the consumer threads.
h. If the queue is full then the video is sent to S3 bucket using uploadFile program in CloudComputingProj1 folder.

2. processPiResult.py

a. This program is responsible for parsing the results.txt file after Darknet execution on Pi is done.
b. We use this file in function get_objects() to parse it and return a unique object present.
c. This result is then sent to the Result bucket using upload_results() function. If no object is detected, “no object detected” is sent.
d. To obtain permissions for uploading results to S3 bucket, we place the cred.json file in the Pi, in a folder named *CloudComputingProj1* folder.

3. uploadFile.py

a. This program is used to upload the videos that need to be sent to the cloud for processing on slaves.
b. Similar to processPiResult.py we need authorization to upload files onto S3 bucket. The cred.json file inside the CloudComputingProj1 folder is used.

Installation Instructions

Setup for Master

- Spawn an EC2 Instance (which would be the master node)
- Assign an IAM role to allow it EC2 Access and SQS Access
- Copy *poll_scaling.py, create_instances.py* and *requirements_master.txt* to the EC2 Instance
- Create a Python3 virtual environment using pip install -p python3 venv
- Activate the virtual environment using source venv/bin/activate
- Run pip install -r requirements_master.txt to install all the dependencies
- Run python create_instances.py MASTER_INSTANCE_ID to start 19 slave instances
- Run python poll_scaling.py MASTER_INSTANCE_ID to begin the master polling

Setting up the Hardware

1. Flash Raspbian image to the microSD card using *balenaEtcher*
2. Serial Connection: Use USB serial console to connect to Pi
   a. Modify /boot/config.txt file, add the line
      ```
      dtoverlay=pi3-miniuart-bt
      ```
   b. Install CP210x USB to UART Bridge VCP Drivers from Silicon Labs
   c. Connect GND of USB serial console to GND of Pi
   d. Connect RXD of USB serial console to TXD of Pi
   e. Connect TXD of USB serial console to RXD of Pi
   f. After making the connections, power up the Pi and execute this terminal command
      ```
      screen /dev/cu.SLAB_USBtoUART 115200
      ```
   g. Pi is now accessible

3. Connecting the camera module:
   a. Connect the ribbon cable of camera to the Pi
   b. To check configuration settings, execute this terminal command
      ```
      sudo raspi-config
      ```
   c. Select Interfaces, Camera, Enable to enable the camera module
   d. Reboot the system
   e. Test the camera by recording a video
      ```
      raspivid -o test.h264 -t 50000
      ```

4. Connecting the motion sensor:
   a. Connect VCC to 5V of Pi (Pin 2)
   b. Connect GND to GND of Pi (Pin 14)
   c. Connect Output to GPIO18 of Pi (Pin 12)
   d. Test the motion detector by running the surveillance.py file on Pi

```
cd darknet
python surveillance.py
```

Setup on Pi

Our implementation uses the following directory structure on Pi.

- /home/pi/facedetect
    - Surv_thread.py
    - Take_snapshot.py
- /home/pi/CloudComputingProj1
    - Cred.json
    - processPiResult.py
    - uploadFile.py

The Boto3 module is required to interact with AWS.
Before starting the program, we need to make sure that cred.json has updated credentials.
Then we can run surv_thread.py as:

```
python surv_thread.py queue_size
```

Where queue_size is the command line argument given to the program.