CSE 546 — Project 1 Report

Nithish Moudhgalya, Raghavendar Thiruvopadi

1. Problem statement

Edge computing is an emerging paradigm for enabling computation on or near the smart devices and Internet of Things (IoT) that produce the input data and provide feedback to users and the physical world. With the explosive growth of data contributed by various edge devices deployed every day, the cloud is not able to meet all the computational demands, especially when the response time requirement is tight. The continuous advancement of System-on-Chips has enabled edge devices to conduct more complicated workloads, leading to the emergence of edge computing which extends the computing from the cloud to the edge. One main advantage of edge computing is the low response time since computation is conducted close to where the input is generated and the response needs to be produced. Therefore, cloud and edge computing are two complementary paradigms for enabling many challenging applications such as AI, AR/VR, and autonomous driving, where cloud resources can provide scalability and edge devices can deliver responsiveness. In this project, we will utilize both cloud (using AWS) and edge (using Raspberry Pi) to develop an application that provides real-time object detection. AWS is the most widely used IaaS provider and offers a variety of compute, storage, and messaging cloud services. Raspberry Pi is a low cost, credit-card sized IoT development platform.

2. Design and implementation

2.1. *Architecture*

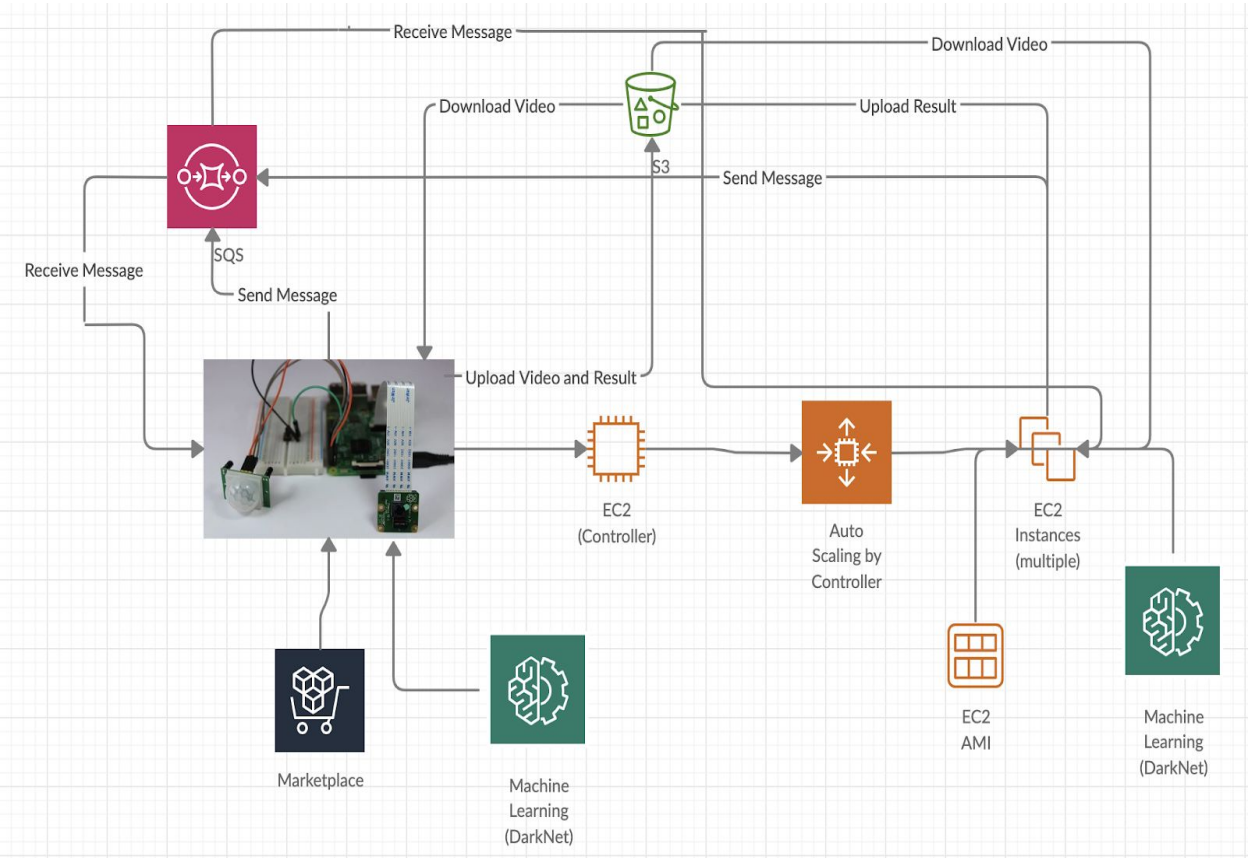


Fig 1: System Architecture

We have 2 main components in this project as mentioned earlier - edge computing and cloud computing. We have a Raspberry Pi 3 B+ board (Pi) for the edge computing part. The Pi is connected with a motion detector PIR sensor and a camera module to record videos on intruder detection. The cloud platform used as the second component of the architecture is Amazon Web Services (AWS). In specific we use the Elastic Compute (EC2), Simple Queuing Service (SQS) and Storage (S3) services from AWS in this project. The figure above shows the architecture of our project. The main workflow of the architecture is as follows:

a) The PIR motion detector detects the presence of an intruder.
b) The PIR sensor triggers the camera module connected to Pi to record a video for 5 seconds. The videos are recorded using 416x416 resolution to reduce upload-download latency. Also, as this is the same resolution that YOLOv3 uses to process the videos, it also reduces the resizing frames latency thus giving a minimal execution time for the entire system.
c) The recorded video is uploaded with a unique key into a S3 bucket.
d) Along with uploading the video to S3, the Pi also adds a new request to a request FIFO queue in SQS. The request contains the bucket and video name.
e) Based on the number of requests in the queue, the controller we implement would instantiate appropriate number of EC2 instances.
f) Both Pi and EC2 instances are viewed as consumers of the queue. Any or all of them can take messages/requests from the queue to handle them concurrently.
g) The Pi and EC2 instances run an object detection script of darknet and extracts all the objects in the videos.
h) The results of object detection are saved into another S3 bucket and once handled, a response message gets pushed into a response queue to keep track of the requests handled and load balance.
i) Every EC2 instance waits for a small duration to receive requests after which self-terminates.

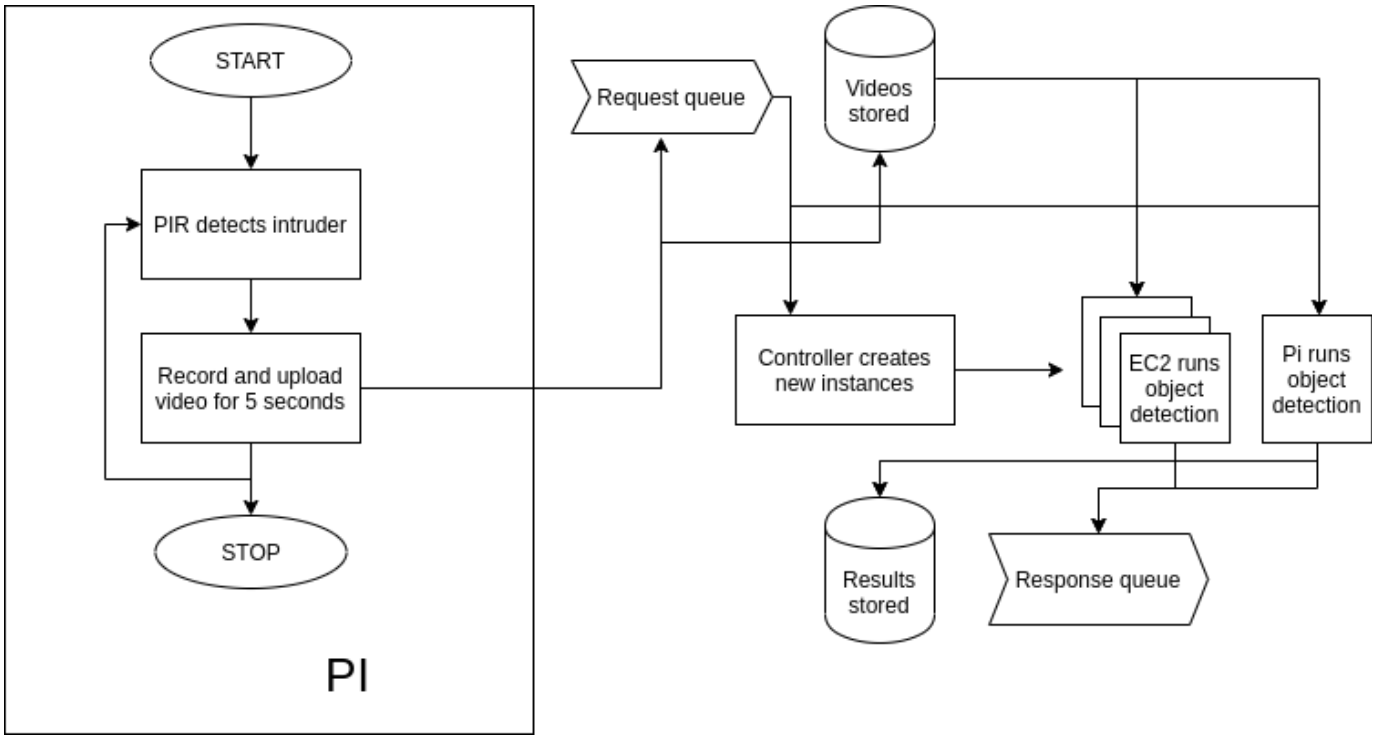


Fig 2: Flowchart for system architecture

2.2. *Autoscaling*

The autoscaling part is the most essential part of this architecture. It manages the number of EC2 instances being used to handle the requests in the request queue. The auto scaling component is a python code deployed on a dedicated EC2 instance. The controller instance runs the following steps to effectively manage the number of EC2 resources being used for object detection. For easier reference, the controller is referred to as master and the other instances as slaves in this report.

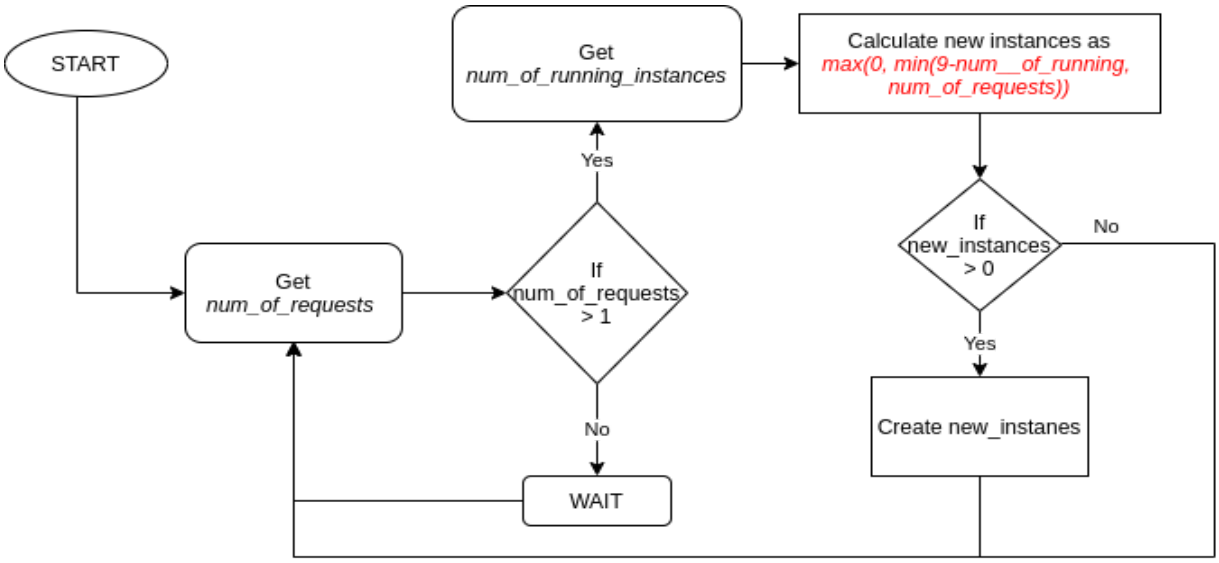


Fig: 2 Flowchart of Autoscaling Controller

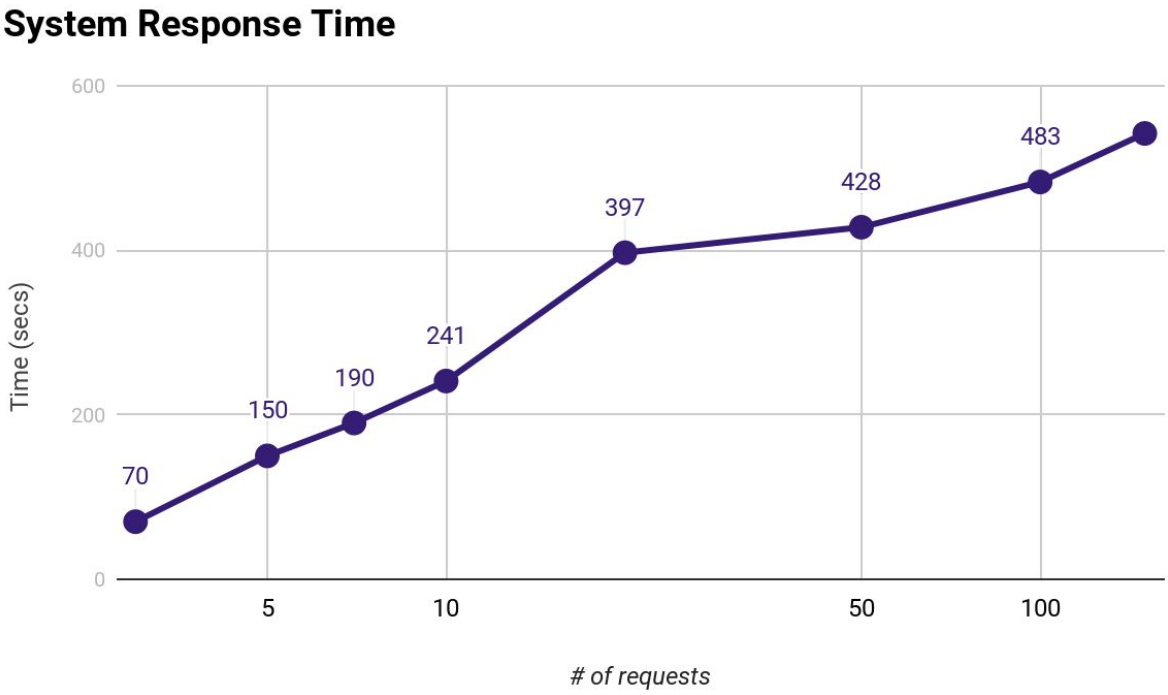
a) Check the request queue for the number of messages in it.
b) Check the existing number of running slave instances.
c) If any new instances can be created, then the master creates them. The number of new instances is calculated as *max(0, min(9-num__of_running, num_of_requests)).*
d) Repeat steps a-c till no more requests are in the queue.

3. Testing and evaluation

To test the application we record 10 videos from the camera module of Pi using motion detectors of some known images. The Pi records the videos of 10 objects and sends it to the S3 bucket and Request queue. The controller which keeps running on a dedicated EC2 instance then checks the number of requests in the queue and spawns new slave EC2 instances to handle them. Simultaneously, Pi also runs the object detection code on it and uploads the results to the output S3 bucket. The only way to compare our architecture is the accuracy of the predictions and the time it takes to achieve the same. These are tabulated in the tables below.

Total # of images	# of images Pi handled	# of images EC2 handled	Time Taken by Pi (s)	Time Taken by EC2 instances (s)	Total Time (s)
3	3	-	70	-	70
5	3	2	128	150	150
7	4	3	146	190	190
10	6	4	167	241	241
20	11	9	211	397	397
50	20	30	306	428	428
100	40	60	439	483	483
150	60	90	503	542	542

The Pi is much faster than EC2 instances in object detection. So it can handle more requests than the EC2 in the same time period. Also the instances take time to get instantiated and to start running the object detection code by which time the Pi can handle the videos. The overall system can handle a huge number of requests. Only the initial phase is slow where the instances have to be created by the controller. But once all of them are created, the requests get handled at great speeds. The plot below shows the plot of the number of requests and time taken by the system. We can observe that the curve is exponential initially but then becomes flat. Also, we observed that reducing the recording resolution to the size of YOLOv3's input resolution reduces the time manifold as opposed to the high resolution videos. The improvement was observed to be almost 2x.



4. Code

4.1. recorder.py

This script constantly checks whether a motion is detected by the motion sensor. If there is a motion detected by the motion sensor, the Camera Module starts recording a video for 5 seconds. The recorded video is of reduced resolution that YOLOv3 uses to process frames thus reducing upload-download latency and resizing frame latency of the system. The recorded video is saved in raspberry pi with filename as current timestamp.

4.2. uploader.py

This script constantly checks for videos in raspberry pi. If videos are present those videos are uploaded to S3 request bucket and a message of the video filename is sent to SQS request queue. The sent video is deleted from the raspberry pi.

4.3. pi_run.py

This script checks for messages in the SQS request queue. If a message exists, it reads the messages and deletes it. The message contains the filename of the video which has to be downloaded from S3 request bucket. Then the darknet command is run on the downloaded video and the command execution is saved to the SQS response queue i.e., whether the command is run successfully or not. The result which we get from darknet command is sent to S3 response bucket and saved using the same filename as that of the video. All the files i.e., the downloaded video and text files used to save the output are deleted from the raspberry pi.

4.4. cred_transfer.sh

A shell script to transfer all the aws credentials and necessary scripts from local to raspberry pi and controller (ec2 instance).

4.5. controller.py

This script is run in a static ec2 instance. It checks the SQS request queue for the total number of messages and also takes the count of the total number of running instances. Using this information it spawns new instances according to the messages in the queue and one more important thing is that it cannot have more than 20 running instances in total.

4.6. ec2_run.py

The controller spawns the ec2 instance and the cron job within the ec2 instance runs this script. This script checks for messages in the SQS request queue. If a message exists, it reads the messages and deletes it. The message contains the filename of the video which has to be downloaded from S3 request bucket. Then the darknet command is run on the downloaded video and the command execution is saved to the SQS response queue i.e., whether the command is run successfully or not. The result which we get from darknet command is sent to S3 response bucket and saved using the same filename as that of the video. All the files i.e., the downloaded video and text files used to save the output are deleted from the ec2 instance. The script checks for messages in the SQS request queue and if no messages are present for the next 20 seconds, the script terminates the ec2 instance in which it is running.

4.7. panda.py

This script runs a shell command which uses gnome terminal command to run recorder.py, uploader.py and Pi_run.py in raspberry pi at the same time.

Contributions of Nithish Moudhgalya

The main tasks in this project can be klisted down as follows,

1) Setting up AWS educate account (queues, buckets and instances) - Nithish
2) Setting up camera module in Pi to record videos and uploading to S3 and SQS - Raghav
3) Setting up motion sensor in PI to trigger video recording - Raghav
4) Writing a script to run darknet command and extract results in desired format - Nithish
5) Writing an auto scaling script to control the number of EC2 instances being used concurrently to handle requests. - Nithish, Raghav
6) Collecting results, drawing architecture and flow diagrams, report and README files. - Nithish, Raghav

Out of these tasks, I worked on tasks **1, 4, 5 and 6**.

For task 1, I used my AWSEducate account and set up the desired services to be used. Created 2 S3 buckets to hold the inputs and outputs of the system. I also created 2 FIFO queues in SQS service for adding messages as requests and responses to and from the system respectively. All services are made public for ease of access for all members of the team, which was pretty hard given the AWS Educate account gave session based tokens/credentials that expired every 2 hours.

For task 4, I wrote a script to run the darknet shell command inside a python code and used file and string operations to extract the objects detected by the system. The first part involved using a simple system function call in the os package of python while the latter involved line by line processing of the result file and extracting the object name from the darknet predictions using string formatting and manipulations. The script also uploaded the results into the output bucket and a response into the response queue.

For task 5, we approached the problem in 2 ways. The controller can be the only authority in deciding which slave handles which request or the controller merely acts only as a mediator in this process. The first idea involved a more cumbersome multiprocessing approach in which we SSH into every slave instance and run a command over ssh on them. This would be gainful as in the SSH can be done in parallel into multiple instances at the same time so the requests can get handled at a very fast rate parallelly and the controller knows which instance is handling which request. However due to certain restrictions in multiprocessing, even a failed child process has to wait for the successful ones to try again which delays the entire system by a lot in case of large requests. So we went with the second approach, wherein the controller merely acts as a facilitator to manage the number of EC2 slave instances running. The controller has no command over which instance handles which request. This way, every slave EC2 runs infinitely trying to access the queue and obtain requests. If it finds a request in 20 secs, it runs the object detection else it auto terminates itself. This was a huge improvement in time as every instance if it fails can try to process another request without waiting for other instances to finish their requests.

For task 6, I worked on collecting results and drawing flow diagrams. I also helped my team members write up the report so that the tasks that I handled alone could be written more lucidly.

Contributions of Raghavendar Thiruvopadi

The main tasks in this project can be klisted down as follows,

1) Setting up AWS educate account (queues, buckets and instances) - Nithish
2) Setting up camera module in Pi to record videos and uploading to S3 and SQS - Raghav
3) Setting up motion sensor in PI to trigger video recording - Raghav
4) Writing a script to run darknet command and extract results in desired format - Nithish
5) Writing an auto scaling script to control the number of EC2 instances being used concurrently to handle requests. - Nithish, Raghav
6) Collecting results, drawing architecture and flow diagrams, report and README files. - Nithish, Raghav

Out of these tasks, I worked on tasks **2, 3, 5 and 6**.

For task 2 and 3, I setted up raspberry pi by following the instructions given by our TA. Then raspberry pi is connected to the necessary module i.e., the camera module and the motion sensor module to the raspberry pi. I wrote a python script for the raspberry pi to record videos when a motion is detected. This script will continuously monitor for a response from the motion sensor. I wrote another script which constantly checks for videos in raspberry pi. If videos are present , then those videos are uploaded to S3 request bucket and a message of the video filename is sent to SQS request queue. The sent video is deleted from the raspberry pi so that raspberry pi memory is used efficiently.

For task 5, we approached the problem in 2 ways. The controller can be the only authority in deciding which slave handles which request or the controller merely acts only as a mediator in this process. The first idea involved a more cumbersome multiprocessing approach in which we SSH into every slave instance and run a command over ssh on them. This would be gainful as in the SSH can be done in parallel into multiple instances at the same time so the requests can get handled at a very fast rate parallelly and the controller knows which instance is handling which request. However due to certain restrictions in multiprocessing, even a failed child process has to wait for the successful ones to try again which delays the entire system by a lot in case of large requests. So we went with the second approach, wherein the controller merely acts as a facilitator to manage the number of EC2 slave instances running. The controller has no command over which instance handles which request. This way, every slave EC2 runs infinitely trying to access the queue and obtain requests. If it finds a request in 20 secs, it runs the object detection else it auto terminates itself. This was a huge improvement in time as every instance if it fails can try to process another request without waiting for other instances to finish their requests.

For task 6, I worked on collecting results and drawing architecture. I also helped my team member write up the report.