CSE546 — Project Report

Aditya Krishna Sai Pulikonda
Kshitij Jayprakash Sutar
Yash Kotadia

1. Problem Statement

The problem we are facing these days is the massive increase in data generated by various edge devices deployed everyday. This explosive growth of data results in the cloud not able to meet the computational requirements, especially when the response time requirement is tight. This results in an increase in the end to end latency which in the context of IoT and cloud services is the network delay of transmitting the data to the cloud, the processing time of the cloud service and finally the network transport delay of transmitting the information to the IoT system. Companies who had been using the cloud for many of their applications have discovered that costs in bandwidth due to the increase in latency were higher than they expected.

Consider the example of an internet-connected video camera that sends live footage from a remote office. While a single device producing data can transmit across a network quite easily, problems arise when the number of devices transmitting data at the same time grows. Instead of one video camera transmitting live footage, multiply that by hundreds or thousands of devices. Not only will quality suffer due to latency, but the costs in bandwidth can be tremendous.

Edge computing is one way of solving this problem which extends the computing from the cloud to the edge. One main reason behind the development of edge computing is the exponential
growth of IoT devices, which connect to the internet to either send information to the cloud or receive back data from the cloud.

It is a part of distributed computing topology in which information processing is located close to the edge-where things and people produce or consume that information. In simpler terms, edge computing brings computation and data storage closer to the devices where it's being gathered, rather than relying on a central location that can be thousands of miles away. This is done so that data, especially real-time data does not suffer latency issues that affect the application's performance. Moreover, companies can save money by processing data locally, reducing the amount of data that needs to be processed in a centralized or cloud-based location. Edge computing works by allowing data from the internet of things devices to be analyzed at the edge of the network before being sent to a data center or cloud.

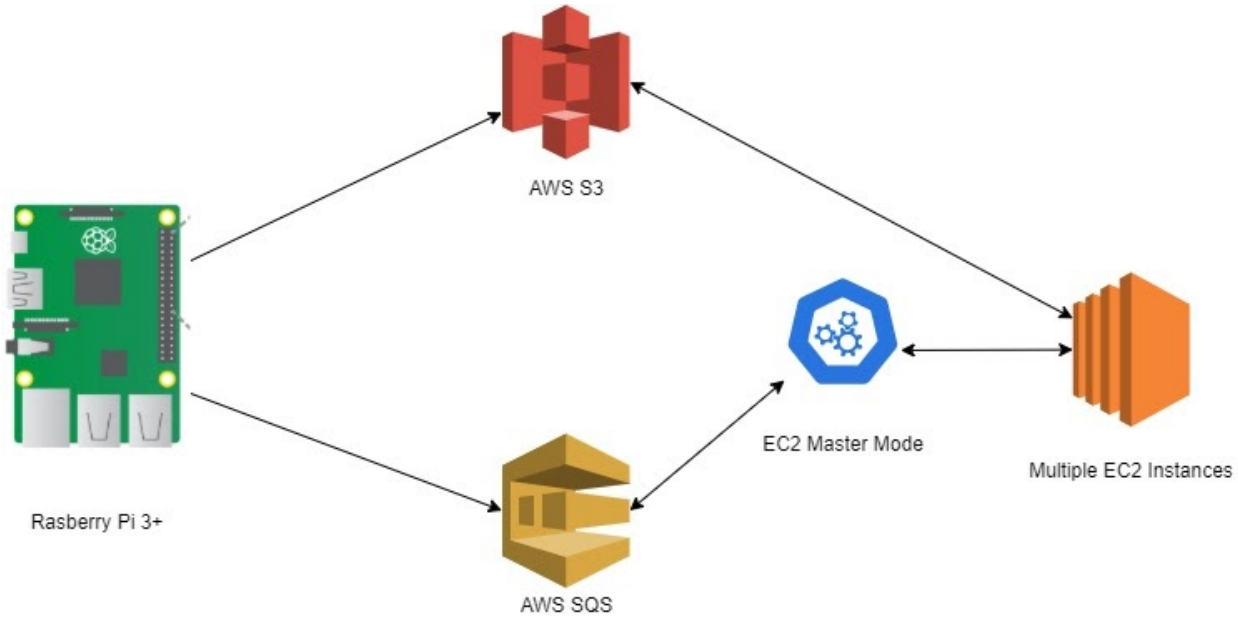
The biggest benefit of edge computing is the ability to process and store data faster, enabling more efficient real-time applications that are critical to companies. Before edge computing, a smartphone scanning a person's face for facial recognition would need to run the face detection algorithm using the cloud-based service which takes a lot of time to process. With an edge computing model, the algorithm could run locally on an edge server or even on the smartphone itself, given the increasing power of smartphones. Applications such as virtual and augmented reality, self-driving cars, smart cities, and even building-automation systems require fast processing and response.

AI algorithms require large amounts of processing power, which is why most of them run via cloud services. The growth of AI chipsets that can handle processing at the edge will allow for
better real-time responses within applications that need instant computing.

In this project, we have made use of this idea to perform real-time object detection using raspberry pi that records the videos using its attached camera. We have used a lightweight deep learning framework, Darknet as our object detection model that can run on either the pi or cloud. The objective of this system is to minimize the end to end latency of object detection, i.e the time between motion is detected, video getting recorded, processed and the result is produced. Another important factor is resource utilization, the cost of processing on the cloud must be minimized and resources must be relinquished when not required.

2. Design and implementation

2.1 Architecture



We have used three AWS based cloud services in our architecture.
1. Amazon Elastic Compute Cloud (EC2)
2. Amazon S3
3. Amazon Simple Queue Service (SQS)

We have used **EC2** to perform object detection on the requests that the Raspberry pi cannot handle in time.

SQS is used to couple the various components in the system and achieves auto-scaling. As soon as the sensor detects any motion, the camera attached to the pi starts recording the video.As soon as the length of the video reaches 5 seconds, we are initiating a thread that processes the video on the pi using the darknet code we have dumped on it before. If the thread is already in process and a new request comes in, the thread uploads the video to the s3 bucket and adds the key of the bucket as a message to SQS.

S3 is used to store all the videos captured on the Pi, and the object detection results of all videos. EC2 uses the key of the s3 bucket stored in SQS to download the video from the s3 bucket, processes the video using the darknet code dumped on the instance and stores the results back in the s3 bucket as a key-value pair (video_name, object_detection_results)

The motion sensor attached to the pi records the video for a duration of 5 seconds as soon as it detects any motion. If the pi is idle, we are initiating a thread that passes the video recorded to the pi which starts processing the video using darknet code that we installed on it. If a new processing request comes in and the pi is already processing some video, the thread uploads the video to the s3 bucket and adds a message to SQS with the key of the bucket in s3. As soon as pi processes the video, the object detection results will be stored in the s3 bucket as key-value pairs (video_name, object_detection_results). If a new video comes in, we check whether the pi is idle or not and decide whether we want to process the video on the pi or upload the video to the s3. We have created an Ec2 master node that stays active all the time and continuously monitors the SQS queue. It takes the responsibility of starting and stopping instances based on the number of messages in the SQS, assigns a video to Ec2 based on its status(running or idle) and as soon as a message gets processed, it removes that message from the queue. We had to make the master node run all the time, to ensure that we are not leaving other instances idle and wasted. As we have a limit of 20 instances, we queue all the pending requests when it reaches the limit. When there is no request, our design only uses the pi without using EC2 instances making pi an edge component.

2.2 Autoscaling

To ensure that instances will not be wasted, we came up with master-slave architecture wherein the master node takes the responsibility of starting and stopping an instance based on the demand. We keep the master node running all the time which does the autoscaling based on the number of messages in the queue. The master node or master instance regularly monitors the SQS and if -

1. The number of messages in the queue is less than the number of idle instances, it starts the required number of instances and assigns a message to every idle instance. As soon as an instance processes the video, it saves the results in the S3, and informs the master node that it is idle. If the SQS still contains messages, the master node assigns a message to the idle instance. If not, the master node waits for a few seconds and stops the instance.

2. The number of messages is more than the number of idle instances, the master node starts all the instances that are idle and assigns a message to each instance. As soon as an instance completes processing a video, it informs the same to the master node and if there are still messages in the SQS, the master instance assigns a new job to that particular instance. If not, the master instance terminates the instance.

In this way, based on demand the master node starts and stops the instances. The cost we are paying to achieve this is by keeping the master node running all the time so that other instances will not be wasted and all the instances will be properly utilized.

3. Testing and evaluation

As suggested in the project specification, we recorded multiple videos and showed the example images to the camera. To ensure that the detection results are correct, we cross-checked with each of the images whether the object that is detected is reasonably correct. According to our observations, the videos are recorded, uploaded to the bucket and processed correctly. The results are correctly passed and uploaded to the output folder in the s3 bucket. We have successfully managed to always have at least one thread of object detection running on the Pi all time thus, making our Pi an edge computing component. Overall, our observation is that all the videos are processed correctly. We make use of the Pi's processing power to reduce the latency of object detection while offloading the extra videos to the cloud. The master node effectively manages the number of instances that are required at any point in time and the resources are utilized effectively.

The following table summarizes the latency observed in each of the components of our system:

Component	Time Taken	Explanation
Recording Video	5 seconds	The duration of each video is 5s.
Uploading video	30 seconds	The default resolution of the videos is 720p, thus the size of each video is around 10Mb.
Object Detection on Pi	1:30 minutes	The Pi can process around 1.2 frames per second.
Autoscaling Frequency	30 seconds	Increase/Decrease the number of required instances based on length of SQS queue.
Starting / Stopping Instance	30 seconds	An EC2 instance takes around 30 seconds to change its state.
Object Detection on EC2	3 minutes	An EC2 instance can process around 0.6 frames per second, half of Pi.

Uploading Results	Negligible	The output text file is of a negligible size which doesn't take much time to upload

Based on the above-mentioned latency figures, we conclude that the approximate latency of our system is 2 minutes if the video is processed on Pi and 5 minutes if it is outsourced to the cloud.

Along with an example video showing time taken in various stages of the model processing the video, we have also performed other tests where we checked the model behaviour on edge cases. These involve checking model's response when we record only 1 video where the entire processing happens on the pi and more than 10 videos where we start the idle instances based on the number of messages in the SQS and wait for the instances to completely process the videos and then we assign a new video if we still have unseen messages in the queue.

One observation we want to make note is the same video takes different time to upload during different runs of the model and network bandwidth we use to upload the video from the pi to s3 plays a huge role in the overall latency. At times the 5 seconds video we record takes 30 seconds to be uploaded and there are times where it takes more than 3 minutes. We can't control these varied times even though we come up with the most optimal auto scaling technique. The interesting observation is once the video gets uploaded , it takes very little time for the video to be processed giving accurate results and wasting no resources which we think is the true goal behind this project.

4. Code

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

We have made 4 files

1. surveillance.py
2. get_video.py
3. send_video.py
4. autoscaling.py

1.surveillance.py - This file takes the input from the motion sensor. If the input is 0, the sensor prints " No intruder is detected" and if the input is 1, the sensor prints "Intruder detected" and starts recording the video for 5 seconds and makes a subprocess call to send_video.py

- This file runs on Raspberry pi which records the video when a motion is detected

2.send_video.py - This file takes the input video recorded by the camera, and uploads the video to the S3 bucket.

- This file runs on the pi and uploads the video to the s3 bucket

3.get_video.py - This file has 3 functions

1. getJobs - gets one message stored in the sqs queue

2. process - downloads the video from the s3 bucket using the key stored in the SQS and processes the video, parses the output to get the object detection results and stores the results in the S3 bucket as a (key, value) pair - (video_name, object_detection_results)
3. parseOutput - parse the output returned by the darknet and stores the unique objects detected throughout the video as a list
    - This file runs on every Ec2 instance that processes a video and stores the results in the s3 bucket by parsing the output returned by the YOLO model.

4.autoscaling.py - This file contains the code for performing autoscaling. The functions in this file are -

1. getLengthOfQ - computes the length of the queue
2. getRunningIds- returns the list of instances that are running and are different from the master node. (As the master node runs all the time)
3. getStoppedIds - returns the list of instances that are in stopped state and are different from the master instance
4. processVideo - processes the video by running the darknet code.
5. main - gets the length of the queue and list of instances that are in running state and stop state.

    If the length of the queue is more than the number of instances that are idle, it starts all the instances and assigns a job to each newly started instance. As soon as an instance processes a video, the master node assigns a new message to that instance if the queue still contains messages.

    If the length of the queue is less than the number of instances that are idle, it starts the instances based on the length of the queue and assigns a job to each newly started instance.

    After processing the video, slave instances inform the master instance and if there are no more messages in the queue, the master instance waits for a few seconds and stops all running instances. This way we ensure that all the instances are completely utilized and we are not wasting any instances.

    - This file runs on the master node which divides the work among the free instances based on the messages in the queue thereby ensuring that no instance is being underutilized.

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

How to Install the Libraries required:

Python libraries required to run the above program are:

1. Boto
2. Boto3
3. Paramiko

The command for installing these libraries is:
pip install boto boto3 paramiko

Note: Python version used for running these programs is 2.7.13

How to Run the Programs:

For Raspberry Pi

We run the surveillance.py using the command ‘python surveillance.py’.

For Master EC2 Instance

Here, we have written a python script autoscaling.py which requires two arguments: AWS SQS URL and AWS Region Name.

The command is written ‘python autoscaling.py <SQS URL> <AWS Region Name>’

5. Individual contributions

Yash Kotadia

Designing the Pipeline

The proposed system consists of multiple components, i.e, Pi, AWS EC2, AWS SQS, AWS S3. Leveraging the functionalities of all these components and configuring their interfacing is key to have an efficient surveillance system. The Pi served as an IOT device as well as an edge computing component in our system. The EC2 instances were used for outsourcing the object detection onto the cloud. The SQS queue was an important component that effectively chained the Pi and the EC2 instances. It ensures that no messages are lost between the IoT device and cloud. In case if there is a failure in processing a message by an EC2 instance, the message is again made available for processing. The S3 bucket is a <key, value> store which is used for storing the videos to be processed by EC2 instances and all the outputs as well. The Pi uses threading to run object detection on a video along with recording newer videos. The master EC2 node uses multiple threads to communicate with slave nodes and keep a tab on their status.

Auto-Scaling

Auto-scaling is the most crucial component of the system for minimizing resource utilization. It ensures that the resources are only provisioned as and when required. It increases/decreases the number of EC2 slaves depending on the demand which is quantified as the length of the SQS queue. The auto-scaling feature is maintained by the master EC2 node. The master EC2 node throughout each loop determines the approximate length of the SQS queue. It maintains a list of stopped, idle(running) and busy(running) EC2 slave instances. Thus, based on all these parameters, we start new instances or stop idle instances. The master node interacts with slaves with threading. Thus for each video to be processed, it creates a new thread that communicates with the slave instances. This thread ensures that the master is able to connect to the slave, if not then it reattempts to connect to the slave. Once it connects with the slave, it instructs the slave to retrieve a video, process it and upload the predictions. In case, if the slave instance fails, the message again becomes visible in the SQS queue since it wasn't deleted by the failing instance.

Designing the Tests

The efficiency of the entire system is key to the success of the surveillance system. The tests were designed to quantify the latency and resources utilized. Since some videos are processed on the Pi while others on the cloud, the overall latency of the system is calculated as an average of the two. The EC2 instances are provisioned and relinquished based on the length of the SQS queue, thus maximizing resource utilization.

Kshitij Sutar

Raspberry Pi Setup and Configuration

The Raspberry Pi given had to be configured with the Pi Camera module as well as PIR motion sensor. For setting up the Pi, the Raspbian OS was flashed on an SD card through Balena Ethecher, then connected the Camera and Motion Detector with Pi's GPIO pins with connectors. To control the Pi, the connection using ethernet was used. The Ethernet cable was used to connect it to the laptop and using Internet Connection Sharing(ICS), the auto-generated IP is used to connect to Pi via PuTTY.

Recording Video Mechanism

The main functionality of this project is to detect the Intruder by detecting a motion through the motion sensor and then recording the video for processing through Darknet. These functions were implemented using a Python script as the Python provides Raspberry Pi libraries to communicate with these sensors. As the objective was continuous monitoring for Intruder, the code ran till manually terminated and whenever the sensor goes to high state the camera is triggered to record a 5 second video and save locally. The loop consisted of two cases where as follows:
1) As low latency is required, 1 video can be assigned to be processed by Pi itself, saving the uploading and downloading video cost, so a thread is created to handle the process_video operation and the Intruder Detection continues.
2) If the Pi is still processing a video and another recorded video is completed then this triggers the code to send a message to SQS and upload the recorded video in S3 bucket.
As latency is to be maintained minimum these two cases were run on thread so the intruder detection works on Main thread.

Send Video and Get Video Methods

The Auto-Scaling feature is implemented using a Master-Slave architecture, the Auto-Scaling code uses helper python scripts to send and retrieve videos. The first part is written in send_video.py where the Boto3 library is used to upload the recorded video stored locally on the Pi to S3 bucket and send a message to the SQS queue in which the body is written in the format of ['process', s3BucketName, s3Input, s3Output, videoName]. The function is called when the Pi is busy processing another video to offload the detection to EC2 instances. The second script i.e. get_video.py is called when the controller EC2 instance sends an SSH request to other instances to run Darknet. This requires the arguments SQS Queue Name to read the message and run the process of object detection using the Darknet command and then write the output in S3 bucket. The output is also parsed before uploading to S3 and finally the message is deleted from the SQS Queue. If the message is not deleted within Visibility Period then the Message is returned to Queue.

Aditya Krishna Sai Pulikonda

Designing the system on the cloud

The system uses three major AWS based services - Amazon EC2 to process the video that Raspberry pi cannot process in time, Amazon S3 to store the videos uploaded by pi and Amazon SQS to couple the various components in the system and achieve auto-scaling. EC2 uses the key of the bucket stored in the SQS to download the video from the bucket and processes the video by running the darknet code that was already dumped on the instance and stores the results back in s3 as (key,value) pairs - (video_name, object_detection_results)

Auto Scaling

Autoscaling is the most important feature of this model, which ensures that the latency (time between uploading the video, processing and storing the results in the bucket) will be as low as possible. To achieve auto-scaling, a master node is used which runs a script using which it starts or stops the slave instances based on the number of messages in the SQS queue and only removes the message from the queue after the video is fully processed. As soon as a slave node is done processing the video, it uploads the results to s3 and changes its state from running to idle. If the slave node fails to process a video, the message will not be deleted from the SQS and that message will be assigned to another idle instance by the master node. The master node achieves all this by keeping note of all the running, idle instances and using these details, it dynamically assigns video to an idle instance and also waits for a running instance to fully process the video. The cost paid for this implementation is keeping the master instance running which monitors the SQS queue all the time so that other instances will not be wasted and all the instances will be properly utilized.

Testing the model

Testing is as important as designing the model. To check whether the system designed fully performs autoscaling, the model was tested using multiple input configurations. First, it was tested using edge case configurations such as 0 videos where no instance must start running or 1 video where the entire processing should be done on the pi making pi an edge computing component by saving the time of uploading the video to the cloud and retrieving data after processing back from the cloud. It was also tested for 10 videos where one video must be processed on pi and all other videos will be uploaded to s3 by pi and the master node starts those number of instances that are idle and based on the number of messages in the SQS. Then, the model was tested for other test cases varying the number of videos recorded, content of the video keeping the duration constant so that the results can be compared. The latency and object detection results are saved for all the test configurations.