CSE 546 - Project Report

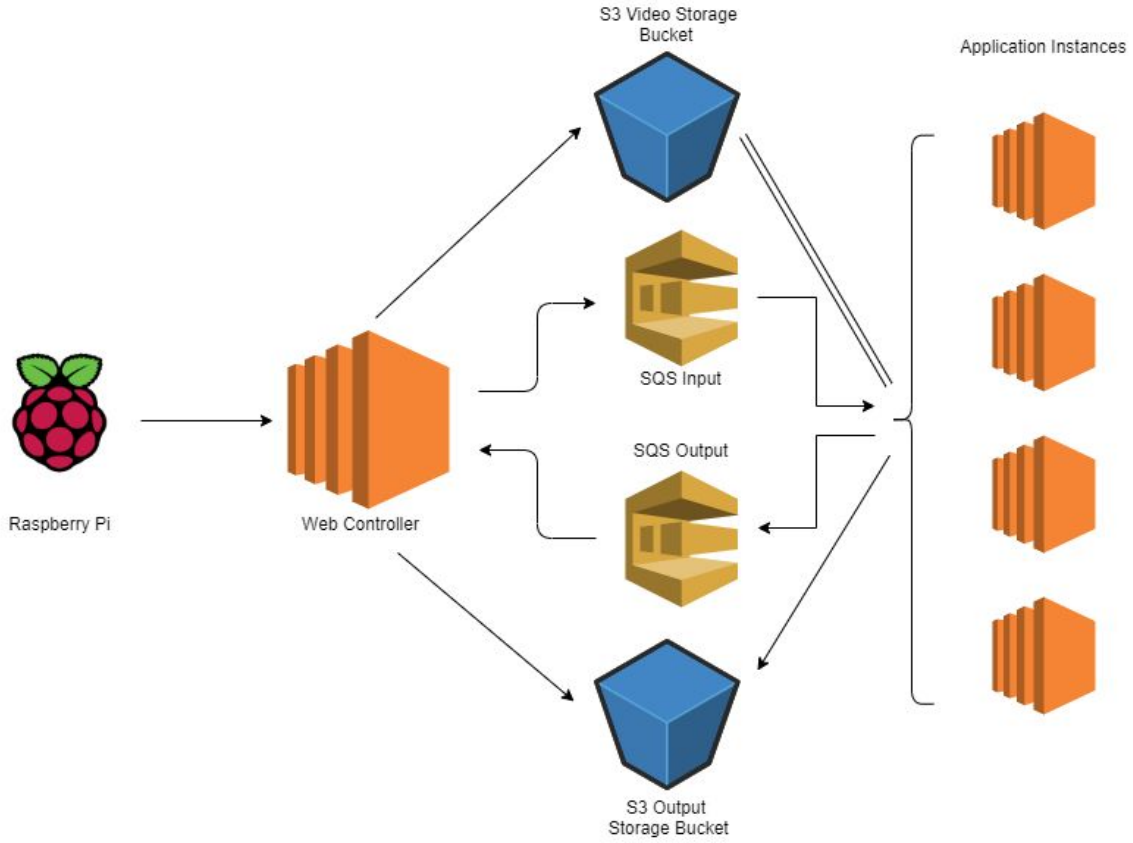
Rahul Advani (1217160594)
Shantanu Purandare (1217160516)
Shreya Darak (1215202846)

1. Problem Statement

With the explosive growth of data contributed by various edge devices and IoT, the cloud is not able to meet all the computational demands, especially when the response time requirement is tight. For example, in self-driving cars the decision to accelerate or not is instantaneous, and needs to be made in less than a second, due to which the sensor/video data generated by the self-driving car needs to be processed on the computer present on the car itself. The on-board computer in this example is known as an edge device. To reduce the computational load on the cloud, edge computing, an emerging paradigm, provides low response times since the computations are performed closer to where the input is generated and the response needs to be produced, but it fails at scaling the applications to accommodate the workload demand due to limited computing resources. As a solution, we aim at using Edge computing and Cloud computing as complementary paradigms where the edge device (Raspberry Pi) delivers responsiveness and the cloud (AWS) provides scalability to develop an application that provides real time object detection.

2. Design and Implementation

2.1. Architecture



2.1.1. Components

- **Raspberry Pi**- Raspberry Pi is a low cost, credit-card sized IoT development platform. The Pi continuously monitors the status of the motion sensor and once a motion is detected it records a 5 second video using a camera attached to it. All the videos are sent to the Web Controller to store them in the Video Storage Bucket present on AWS S3 (AWS Simple Storage Service). The Pi runs the Darknet deep learning object detection program on the recorded video. The results obtained from Darknet are then sent to the Web Controller to store it in the Output Storage Bucket on AWS S3. Simultaneously, the Pi makes a decision to hand over the processing of a video if it is currently processing an earlier video. In the latter scenario, it will send the video to an

application running on an EC2 instance i.e. Web Controller , which will scale up instances as required to process the newly acquired video.

- **Web Controller**- The Web Controller is an EC2 instance which plays the role of a LoadBalancer. We have exposed 2 REST APIs which either accept a video file or a video file and its output. Based on the API hit, it either places the video file and its output file into separate S3 buckets, or just places the video file in an S3 bucket and the video file's metadata in an SQS queue. Based on the number of messages in the SQS queue and number of EC2 instances running, the Web Controller decides whether to launch a new App Instance instance or not.

- **Amazon SQS**- SQS is a distributed queuing service that supports programmatic sending of messages as a method of communication between web service applications. The Web Controller adds metadata about the video files to be processed by the cloud onto the Input SQS queue for consumption by the Application Instances. The Application Instances is responsible for picking up the meta data from the SQS queue and processing its corresponding video.

- **Application Instances-** Application Instances are EC2 instances that are used to process videos. A custom AMI was created having specific services which run during boot up. The Web Controller monitors the Input SQS queue to spin new Application Instances depending on the number of pending videos in the queue. Each Application Instance picks up a message from the input queue and fetches the corresponding video from an S3 bucket and runs the deep learning Darknet classification program on the video assigned to it and places its output in the S3 Output Bucket. It also sends responses back to the LoadBalancer using the Output SQS queue and terminates itself upon completion of these tasks.

- **Amazon S3-** Amazon S3 (Simple Storage Service) is a cloud storage service designed to store large chunks of data. In our system, we have 2 bucks namely, Video S3 and Output S3. The video S3 Bucket is used to store the videos recorded by the Pi, while the Output S3 Bucket is used to store the outputs of the videos processed by the Pi as well as the Application Instances.

2.1.2. Workflow

When a motion is triggered, the Raspberry Pi records a 5 second video, which is then stored in a specific folder. There is a second script which is continuously looking at the contents of this folder. Whenever a new video appears in this folder, the Pi either runs the deep learning object detection model on it or passes it forward to the Web Controller. We are using a multi threaded program to make this decision, if the Pi is busy processing a video, a new thread is created to POST the video via a REST API to the Web Controller. Once the Pi has completed the deep

learning process on a video, it's essentially free and starts processing the next available video. The system architecture employs the Web Controller as the load balancer.

As mentioned earlier, the Web Controller has exposed 2 APIs. Through the first API, it receives a video and its output (as the video was processed on the Pi itself), it stores the video and output files in their respective S3 buckets. Through the second API, the Web Controller just receives a video from the Pi and stores it in the Video bucket on S3 and places the video file name in the Input SQS. The Web Controller is also continuously looking at the number of messages in the input queue and the number of deployed App Instance EC2 Machines. Based on these parameters, the autoscaling logic spins up a new App Instance if required.

After boot up, these Application Instances dequeue a message(video name) from the Input SQS, and fetch the corresponding video from the S3. After which, it runs the deep learning object detection model on it. After running the object detection program, the Application Instances store the output in the Output S3 Buckets in the form of {key, value} pairs where key=[video file name] and value=[result of the object detection program]. The Application Instances then add the video file name as a message to the Output SQS queue, which signifies the videos which have completed processing.

2.2. Autoscaling

Our application supports autoscaling of the EC2 instances to handle the peak traffic. We added this functionality to provide low latency to the customer. This logic looks at the difference in number of messages in the queue and number of EC2 instances running at that time and spins up more EC2 instances to handle the new object detection requests. We also have the functionality of scaling down at the app-instance. Whenever an app-instance completes processing of a video, it will check for new messages in the sqs queue. If there are no new messages, the instance will terminate itself. Howerer, we have a cap on maximum app-instances running which is 19. Hence total new EC2 instances launched will be a minimum of 19 and totalEC2InstancesToLaunch. We have paused the execution for 5millisec using thread to make sure instances won't be terminated while still consuming messages from the input queue.

3. Testing and Evaluation

The Testing phase involved unit testing, integration testing, regression testing and end to end testing. All the individual components of the system were first tested to confirm that they worked as expected. The components were then exposed to integration testing to see how well they worked with each other. After any changes were made to the program to modify its behaviour we had to regress test it to make sure that no new bugs were introduced to the system.

When the system passed all the above tests we exposed it to end to end testing. It included feeding the camera module with images of the subjects of object detection and verifying that the output produced is correct. The main components tested were the autoscaling logic, in particular if the desired number of EC2 instances were deployed and terminated or not without any wastage of resources, the SQS queues and whether they worked properly as a messaging interface between the LoadBalancer and the Application Instances. The S3 buckets were also tested to see if all the videos and outputs were stored in the expected format.

The evaluation metrics involved both the correctness of object detection and the end to end latency. Steps were taken to improve the end to end latency so that it passed the expected baseline latency.

4. Code

4.1 Python script on the PI:

1) Multithreading module: Receives a video file from the camera module. Uses multithreading logic to decide whether the video should be processed on the Pi or on the Web-Controller. This is based on the availability of Pi for video processing indicated using a boolean variable ‘piFree’
2) Classification on Pi module: This module is called if the threading logic delegates a video to be processed on the Pi. This module then runs the Deep Learning script on the Pi, generates the output and sends the video and the output files to the Web Controller using POST requests for storing in the S3 buckets.
3) Classification on Controller module- This module is called if the threading logic delegates a video to be processed on the Web-Controller because the Pi is already busy processing another one. This module then uses POST requests to send the video file to the Web-Controller for storage and further processing.

4.2 Web-Controller Tier :

1) This has a web- controller tier which accepts web requests from pi. Http post endpoint in this module will receive getvideo(video) and getVideoOutput(video, output) requests from pi and will publish it to sqs and s3 input buckets according to the need.
2) We have used Spring Boot framework for creating beans of aws sdk clients EC2, SQS and S3. We are reading these property names from application.properties so that it can be configured and changed over time.
3) This tier has an Auto-scaling module which creates new EC2 instances with custom AMI which already has an app-instance deployed on it.

4.3 Application Instances:

1) Listener Module- This module continuously monitors the input SQS queue using the specified request parameters like the visibilityTimeout, for any new messages and picks up the messages from the queue. The metadata from the messages is then used to fetch the video file from the s3 bucket for stored videos. If no message is available after multiple tries to fetch the message using long polling, the listener module terminates the instance to avoid any wastage of resources.
2) Deep Learning Module- This module takes a video file as an input and runs the deep learning script from inside our java program and parses the result of the object detection performed on the video file into the requested format. This result is then forwarded to the output S3 bucket to store it as a key, value pair where key is the video file name and value is the result obtained.

4.4 Custom AMI Creation:

For running the App Instance we have created a custom AMI. To start off, we have used the base AMI image provided with the problem statement (ami-0903fd482d7208724). Using this AMI we deployed an EC2 instance, which was a base for creating our custom AMI image. We copied an executable jar of the App Instance onto this machine and wrote a shell script which executed this runnable jar. Further, we wrote a service which would run the previously mentioned script on boot up. After enabling this service we created an image of the current EC2 state through the AWS Console and created a new AMI (ami-0e3e384b94d4235c2). Once the image is created we use that particular AMI Id while spinning up instances.

5. Instructions for Project Execution:

5.1 Pi setup and execution-

- Set up the pi wire configuration for serial connection using instructions given in the project doc.
- As pi needs to send the videos to the AWS environment , set up the wifi on pi and ssh into the pi using assigned IP.
- Change the IP of the webcontroller on 17th line in pi3.py
- Run the DISPLAY command specified in the project documentation to enable the display and to capture the output.
- Ensure that the ~/Desktop/darknet/videos folder is empty before running the script.
- For executing and capturing videos on pi, first run python ~/Desktop/darknet/pi3.py, in another terminal run python ~/facedetect/load_balance_surveliance.py.
- After which, for every motion detected, a video will be recorded and processed.

5.2 Web-Controller Setup and execution:

- Configure and setup AWS environment with region us-east-1.
- Create input and output s3 buckets as well as set up two sqs queues, input and output. Create a EC2 instance manually for running a web-controller server.
- Build the code given in the web-controller directory using gradle commands ***./gradlew clean build*** and ***./gradlew build Jar.*** Now transfer this built jar to EC2 instance using scp command. I.e. scp -i <pem file> ./build/libs/<jar>.jar ec2-user@<ip>:~
- Create the java environment by installing default-jre (jdk) packages on the EC2 instance.
- Create an output log file via 'touch ~/webControllerOp'. Run this jar using command ***java -jar AutoScaleWebController-0.0.1-SNAPSHOT.jar > webControllerOp 2>&1 &*** which will redirect the output to the webController file.

5.3 App- Instances set-up and Execution:

- Configure and setup AWS environment with region us-east-1.
- This Web-controller will use custom ami-0e3e384b94d4235c2 which will scale up/create the App Instance. If the App Instance needs to be manually deployed, it can be chosen from the 'My AMI' tab under 'Create Instance'. The general way to create a custom AMI can be created by following the steps mentioned in section 4.4. For this.Use the code in the directory app-instance to build the jar using gradle and deploy it in ami.

5. Individual Contribution

Rahul Kamalkumar Advani (1217160594)

Design

Designed the script running on the Pi to process or send videos to the web controller. Programmatically compared the performance of multiple approaches, each approach varied the number of threads being spawned through the script running on the Pi. The final decision to spawn just 2 threads was made, taking into consideration the processing power of the Pi is limited. Structured the workflow for creating and deploying custom AMI, which behaved as a fully functional App Instance immediately on boot up. Also, theoretically compared different end-end system designs initially, before starting to build the application. These designs were mainly centered around the location of the web controller and the different ways to create the App Instance without any manual intervention.

Implementation

Scheduling Python Script - Implemented the python script, which starts processing a video once it has finished recording. Every video which was recorded was either processed on the Pi or sent to the Web Controller. This was achieved through a multi threaded program. One thread was responsible for running the deep learning model on the pi and the other thread was responsible for sending the files to the web controller. For which, I made use of a shared boolean flag, ‘piFree’ ,which essentially made the decision to process the video on the Pi or not.

Web Controller - Used Spring Boot framework in Java to build the Web controller. I specifically worked on writing code for the baseline structure. i.e. creating specific beans (java objects) on startup of the code, which could be injected/used by any class across the application. Eg: Beans for the AWS SQS, S3 and EC2 client. I also wrote code which exposed multiple REST APIs to accept video files from the raspberry Pi and further process them to the different AWS services.

Custom AMI Creation - I wrote a shell script which downloaded the necessary libraries and ran the executable jar on the App Instance. This shell script needed to be executed on boot up, i.e. during scale up when the App instance was created. For which, I wrote a service which ran the above mentioned script immediately after boot up. Once the necessary scripts and libraries were in place, I took a snapshot of the instance and created a custom AMI.

Testing

I primarily tested the functionality and performance between the hardware and software components on the pi, i.e. the workflow between the motion sensor, camera and scheduling script. I also regularly stress-tested the entire system with my teammates after we completed integrating all the components. Through which I noted the performance and problems in different components of the system, and collectively we reduced the initial latency down by 4 minutes.

Shreya Darak (1215202846)

Design

I worked on initial configuration of PI setup and running Darknet Model on pi with given video or image. I was also involved in understanding the requirements of this project and designing the architecture with the goal of “low latency for customers” in mind. The architecture of the project was the result of many brainstorming sessions with my project mates in which I played a big part. I have implemented core logic for the web-tier of the application which is responsible for autoscaling and handling the load of the object detection requests.

Implementation:

During the course of this project, we used github for team collaboration and version control. I worked on a web-Controller application and implemented functionalities that involved setting up connections with SQS,EC2 and S3. I implemented functionalities like sending incoming video and output files from pi to output s3 bucket. As well as sending input video files to input s3 bucket and filename to SQS to handle load of incoming requests. I made the application configurable so that bucket names, AMI Image ID, SQS queue names can be changed over time.I implemented auto-scaling logic to scale for peak traffic and provide object detection output to customers with minimum delay. I tried to compare different approaches to get the optimized result for object detection. Auto scaling module has logic for spinning new EC2 instances depending on the difference in number of messages in SQS and number of EC2 instances running. To make sure, instances won't be terminated while still consuming messages from the input queue, I have paused the execution for 5millisec using thread.

After setting up the AWS environment including security groups and IAM user roles, I deployed the Web-Controller application on an already created EC2 instance. Later I moved on to integrate it with App-Instance as my teammates started developing functionalities in it as our design evolved.

Testing and Debugging:

Before starting with the design phase, I tested the Darknet model given to us. I configured and tested the performance in terms of processing power and time required to run the model on both pi and EC2 instances. After integrating all the components, we tested complete flow without scaling up logic and with one EC2 instance. When this was successful, we used auto scaling logic to enhance performance and reduce latency.

While testing auto scaling logic, we encountered many corner case scenarios, to address that I optimized a few parameters and performed stress testing multiple times.In this testing and debugging phase, I helped in fixing the bug and improved performance.

Shantanu Purandare (1217160516)

Design:
Designed the structure of the individual components that form the Application Instances and the interfaces that were used for interaction between these components. Compared different approaches that focused on efficiency to specifically minimize the time each instance was active to finalize on the design of these instances. Collaborated with my team members to come up with the architecture for the system by comparing between the proposed system designs that focused on placement of the individual components like the Web Controller and Listener for maximum efficiency.

Implementation:
Application Instances-
Developed the Application Instances using the Spring Boot framework in Java. Wrote the code for bean creation and injection to be used by the Application Instance modules including the beans for setting up the AWS environment, SQS, S3 and EC2 clients. I wrote the code for the lifecycle of each Application Instance, for the streamlined performance of these instances, essential for the overall robustness of the system. This involved the code for monitoring Input SQS queue for new incoming messages/requests. Reducing the visibilityTimeout attribute of the input SQS queue and deleting messages from queue immediately after the Instance picked it, we were able to speed up the process of fetching metadata from the queue to further process it in the Application Instance's lifecycle. The instance also had to terminate itself in case of an empty Input SQS queue in accordance with the autoscaling logic and to avoid wastage of AWS resources.

Implemented components of the Application Instances and their interaction with each other. Developed the functionalities for retrieving video files from S3 buckets and messages from the Input SQS queue one by one as well as sending the appropriate output messages using Output SQS queue. Worked on the code that ran the Deep Learning model script through a Java program in the Application Instances system and wrote the parsing logic for generating the output files in the specified format before storing them in the output S3 bucket.

I worked on the Python script along with my team member to reduce the overall latency of the system by deciding on the strategies for making the decision of processing the video on the Pi or sending it to the Web Controller

Testing:
Performed unit testing and integration testing on the Application Instances iteratively improved the response times on these instances. After integrating and establishing an end to end connectivity of the whole system, I worked with my teammates to test the system under higher workload and noted down the performance metrics to pinpoint the bottlenecks and find better ways to implement and integrate the changes into the system. Eventually we were able to reduce the end to end latency of the system by 4 minutes.