CSE 546 - Project 1 Report

Darshan Dagly (ASU ID: 1215180174)
Smruti Berad (ASU ID: 1214907915)
Viken Shaumitra Parikh (ASU ID: 1215126783)

1. Architecture

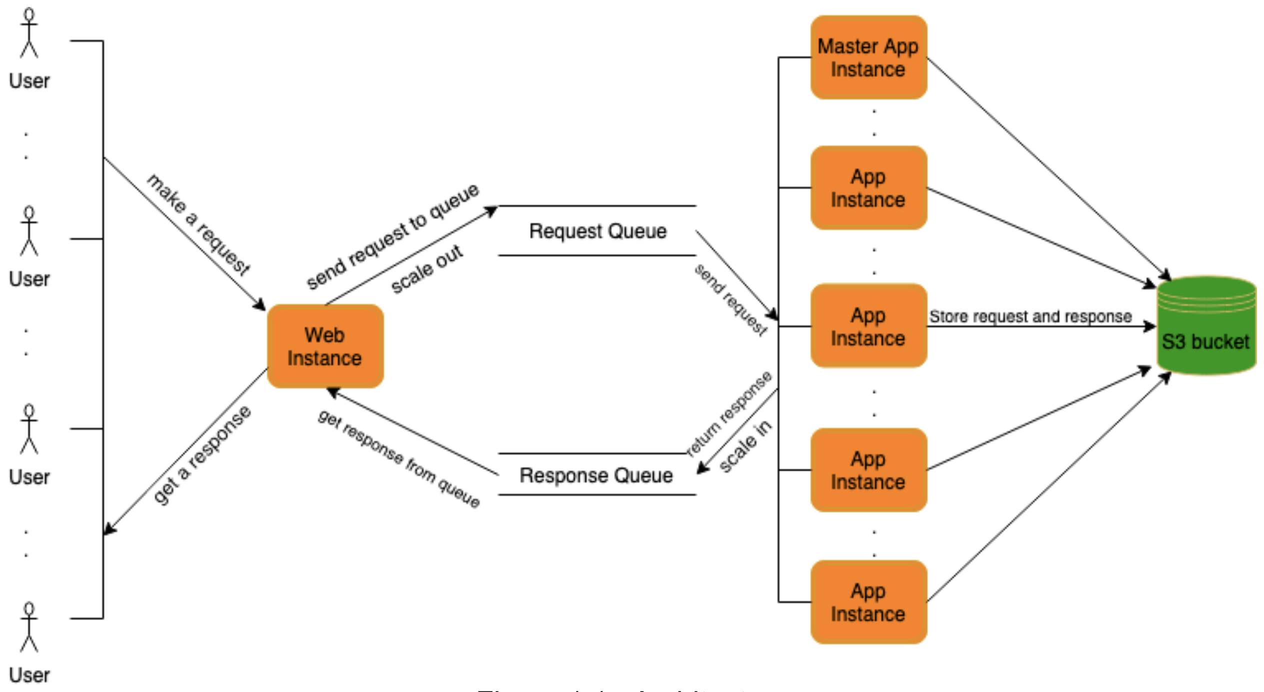


Figure 1.1 : Architecture

Amazon Web Services used for the project are:

- **Elastic Compute Cloud** - EC2 is a web service which allows users to create a virtual machine for running an application. It provides a secure and a resizable compute capacity in the cloud. We have used the EC2 for creating our Web and Application instances.

- **Simple Queue Service** - SQS is a fully managed message queuing service that enables you to decouple and scale microservices, distributed systems, and serverless applications. We have used the SQS to handle our requests and responses to and from our different instances.

- **Simple Storage Service** - S3 is the largest and most performant, secure, and feature-rich object storage service. The response from our application is stored in the S3 buckets in the form of a key-value pairs where the video name is the key and the detection result is the value.

Architecture Overview:

Web Instance:
The Web Server accepts requests from the user and stores it in the request SQS queue. Once the results are available in the response SQS queue, it responds back to the user with the results of the object detection. This is built using the Python Flask Framework.

The AutoScaler module scales out and scales in the system depending on the load on the web instance. This is built using the Spring Boot Framework in Java

Application Instance:
The architecture consists of 1 Master Application Instance and 18 self-stopping worker Application Instances. These requests pushed into the request SQS queue are then picked by one of the 19 Application instances, which then process the request by downloading the video and running darknet object detection on it. The video file name and the objects identified by darknet are pushed to a Response SQS queue and a S3 Bucket by the Application Instance.

As per our architecture, we initially start with 2 instances (1 Web Instance, 1 Master Application Instance) which are always running and are never terminated during scaling in. 18 worker Application Instances are spawned by the AutoScaler module **(Explained in detail below)** when the request load increases **(Scale Out)**. These worker application instances poll the request queue and process the request one by one. If the request queue is empty, the worker application instances self terminates **(Scale In)**.

2. Autoscaling

The Autoscaling functionality is broken down and handled separately at the Web and Application Instances. The Scaling Out feature through which new Application Instances are spawned is handled at the Web Instance. Scaling-In of the Application Instances is handled by each of the Instances individually.

The Scale out logic at the Web Instance is implemented by keeping a count of the following 3 parameters - number of messages in the request queue, number of running application instances and number of stopped application instances. The Web Instance checks the count of messages in the request queue every 40 seconds. For the scope of this project, the maximum number of instances is set to 20. If the maximum number of instances are not already in use and if it finds the number of messages in the queue is greater than the number of running instances, the web instance scales out using the below approach:

1) First, it calculates the additional number of instances required to process the messages in the queue by subtracting the number of running application instances from the number of messages in the queue.
2) It then checks if it has enough stopped instances to fill in the deficit of instances calculated above.
3) If it has enough stopped instances, these instances are started otherwise it starts all the available stopped instances and creates the remaining required instances.

The Scale In logic is included in each of the spawned worker application instances. We have implemented the worker model where each worker instance when spawned will long poll the request queue for requests to process. Upon completion of the request processing, the instance will poll the queue again for new requests to process. If the queue is empty, the worker instance will wait for 10 seconds for incoming requests, if no requests are found, the worker application instance will stop itself.

3. Code

Functionality of Programs

1) **Functionalities of Web Server**
   a) The Flask Server has a thread which is listening to user requests and assigns worker threads for each request.
   b) A unique user request id is generated for each request which will help us map the requests with the responses.
   c) This unique id is sent to the request SQS queue which would be used by the app instances.
   d) There is one another worker thread which is listening to the responses put by the app instances in the response SQS queue. It uses long polling of 10 seconds wait time so that too many requests are not sent to SQS queue on polling.
   e) There is a global dictionary object which is used by all the threads.
   f) The responses read from the SQS queue are mapped by using the unique id as key and its answer as value.
   g) When a response has been received, the user thread reads that the unique id is present in the dictionary, so it now sends the answer mapped to this user id to the user and deletes the answer from the dictionary and terminates the user request worker thread.

2) **Functionalities of AutoScaler**
   a) This package/program monitors the request SQS queue and the number of running and stopped application instances every 40 seconds.
   b) Based on the above counts, appropriate number of instances are created or started by the AutoScaler module if required.

3) **Functionalities of the Master Application and Application Instances**
   a) This program continuously long polls the request queue for 10 seconds waiting for user requests.
   b) If a request is found, it will set a 1-minute visibility timeout on the request so that it is not picked up by any other application instance.
   c) The program will then invoke detect_object.sh file stored in the home/ubuntu/darknet directory which runs the object detection commands and returns the video file name and its corresponding results.
   d) The results along with its unique request id is concatenated together and pushed to the response queue where it is picked up by the Web Server.
   e) The request in the request queue is now deleted by the Application Instance.
   f) The results are finally added to the S3 bucket in the following format: [video-name, result]

g) Request queue is again polled for new request and the entire process from b to f is repeated if a request is found. If no request is found the following 2 possible functionality occurs depending on whether it is the Master Application Instance or the Application Instance.
   i) Master Application Instance – Continues polling the request queue until a new request is found.
   ii) Application Instance – Instance Stops is not new requests are found.

4) Functionalities of AWS Services

a) EC2 service: It is used to host all the different modules in the system with a loosely coupled architecture. We have used a total of 20 instances. One of the EC2 instances runs the Web Server and AutoScaler, and another one of the EC2 instances runs the processing module object detection and both of the instances are always running. The AutoScaler starts up to 18 other EC2 instances based on number of pending requests on the request SQS queue. The EC2 instances shut down on their own if they do not find any pending requests in the request SQS queue after a wait time of 10 seconds.
b) Simple Queuing Service: The SQS queue acts as a buffer until the app or web instance is free to process the pending user requests. One of the SQS queues is used for adding the user requests, while the other SQS queue is used to add the responses from the app instances with the results of the inputs.
c) Simple Storage Service: To have the results persistently at a place, S3 bucket is used to store the name of the video file with the list of objects detected in it. The app instance inserts this entry of the result after it has finished processing the video for the user request.

Code Setup and Execution Steps

1) Setup of the AWS Services

a) To set up the AWS services, we install the AWS CLI and configure the AWS credentials for the EC2 instance.
b) We setup request and response SQS standard queues and S3 bucket on the AWS console.

2) Setup of the web instance

a) To start the web server, start an ec2 instance and run the __init__.py python3 script on it. It requires python3 (And required libraries: boto3 and Flask) and nohup services installed on the EC2 instance.
b) We assign an elastic IP address from AWS to this instance so that we have a static IP address associated with our web server instance even after it has been rebooted.

3) Setup of the master app instance

a) Create an EC2 instance using the Image ID - ami-0e355297545de2f8 under location us-west-1
b) Create a JAR file of the Master Application Instance files submitted and deploy it on this instance along with the detect_object.sh file to the darknet directory.
c) Install Java JDK and xvfb using the below two commands.
   i) sudo apt-get install default-jdk
   ii) sudo apt install xvfb

4) Setup of app Instance

a) Create an EC2 instance using the Image ID - ami-0e355297545de2f8 under location us-west-1.
b) Create a JAR file of the Application Instance submitted and deploy it on this instance along with the detect_object.sh file to the darknet directory.
c) Install Java JDK and xvfb using the below two commands.
   i) sudo apt-get install default-jdk
   ii) sudo apt install xvfb
d) Add the below 2 lines to etc/rc.local file. This will ensure the application instance code starts running whenever the instance is spawned by the AutoScaler Instance.
   i) cd /home/ubuntu/darknet
   ii) java -jar Application-Instance-v2.jar
e) Create an image of this instance by selecting the selecting the EC2 instance on the AWS portal and clicking on Actions → Image → Create Image.

5) Setup of the AutoScaler module

a) Add the image id obtained in the above step to application.properties file of AutoScaler module.
b) Create a JAR file of the Java files submitted and deploy it to the EC2 instance on which the web server is already running.

6) Execution of the project

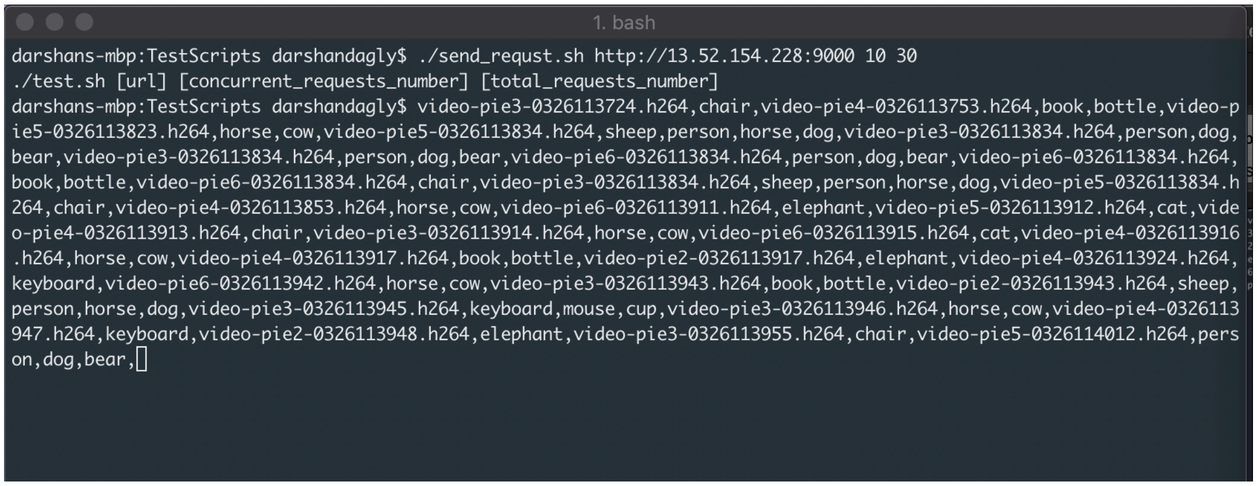
a) Run the following 2 commands on the Web Instance to start the Web Server and the AutoScaler Service.
   i) nohup python3 __init__.py &
   ii) nohup java -jar ScaleOutService-v1.jar &
b) Run the following command on the Master Application Instance to start the Master Application Instance.
   i) nohup java -jar Master-Application-Instance-v3.jar &
c) This command will run the Python Flask web server and the AutoScaler Java application in detached mode until the instance has been shut down.
d) Run the send_requst.sh test script provided with the public IP address of the Web Instance to send requests to the Web Instance. Alternatively, requests can also be sent by entering the IP address in a web browser.
e) Run the list_ec2.sh and list_buckets.sh scripts to monitor the number of running EC2 instances and data in the S3 bucket.

4. Project Status

We have achieved the following:

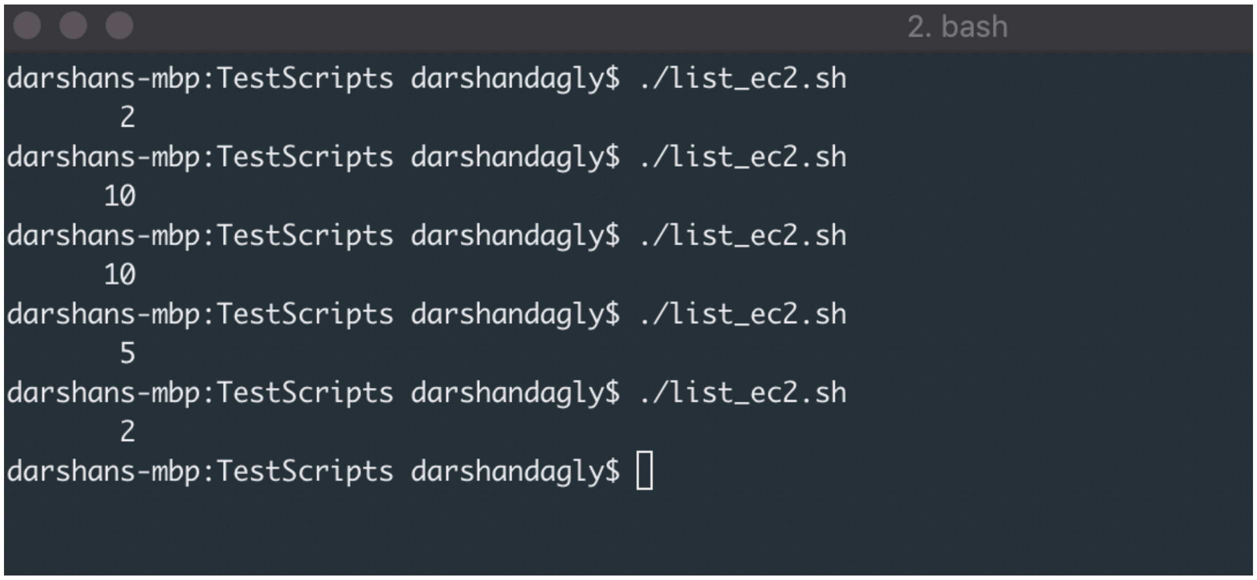
- The user can make multiple requests to the web instance which in turn sends them to the app instance to get responses from the video surveillance system.
- The application is capable of handling multiple requests and scale in and out with respect to the demand.
- The video names and corresponding results are stored in S3.
- The app handles all requests quickly without missing any request.

<u>Screenshots</u>:



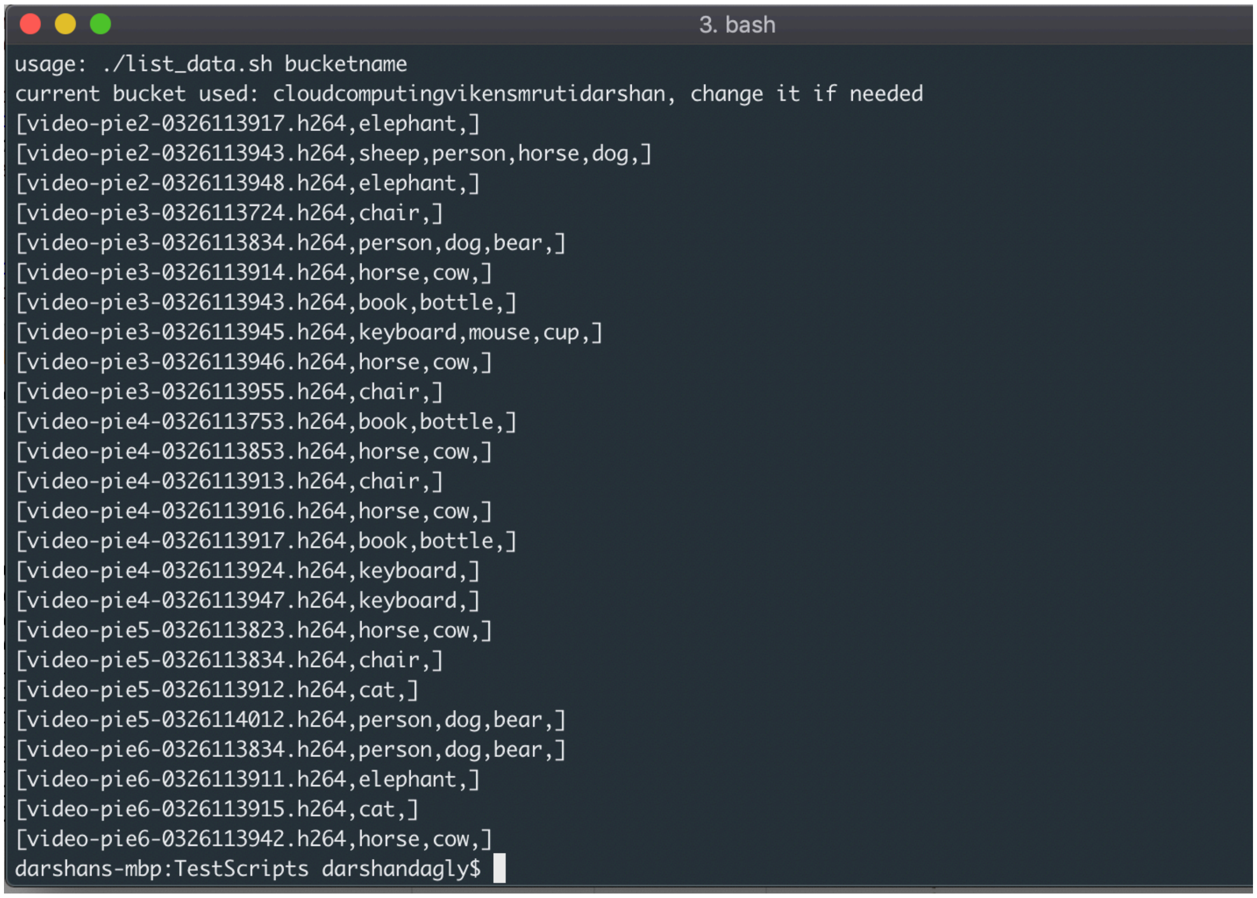
```
darshans-mbp:TestScripts darshandagly$ ./send_requst.sh http://13.52.154.228:9000 10 30
./test.sh [url] [concurrent_requests_number] [total_requests_number]
darshans-mbp:TestScripts darshandagly$ video-pie3-0326113724.h264,chair,video-pie4-0326113753.h264,book,bottle,video-p
ie5-0326113823.h264,horse,cow,video-pie5-0326113834.h264,sheep,person,horse,dog,video-pie3-0326113834.h264,person,dog,
bear,video-pie3-0326113834.h264,person,dog,bear,video-pie6-0326113834.h264,person,dog,bear,video-pie6-0326113834.h264,
book,bottle,video-pie6-0326113834.h264,chair,video-pie3-0326113834.h264,sheep,person,horse,dog,video-pie5-0326113834.h
264,chair,video-pie4-0326113853.h264,horse,cow,video-pie6-0326113911.h264,elephant,video-pie5-0326113912.h264,cat,vide
o-pie4-0326113913.h264,chair,video-pie3-0326113914.h264,horse,cow,video-pie6-0326113915.h264,cat,video-pie4-0326113916
.h264,horse,cow,video-pie4-0326113917.h264,book,bottle,video-pie2-0326113917.h264,elephant,video-pie4-0326113924.h264,
keyboard,video-pie6-0326113942.h264,horse,cow,video-pie3-0326113943.h264,book,bottle,video-pie2-0326113943.h264,sheep,
person,horse,dog,video-pie3-0326113945.h264,keyboard,mouse,cup,video-pie3-0326113946.h264,horse,cow,video-pie4-0326113
947.h264,keyboard,video-pie2-0326113948.h264,elephant,video-pie3-0326113955.h264,chair,video-pie5-0326114012.h264,pers
on,dog,bear,
```

Figure 4.1: send_requst script file



```
darshans-mbp:TestScripts darshandagly$ ./list_ec2.sh
       2
darshans-mbp:TestScripts darshandagly$ ./list_ec2.sh
      10
darshans-mbp:TestScripts darshandagly$ ./list_ec2.sh
      10
darshans-mbp:TestScripts darshandagly$ ./list_ec2.sh
       5
darshans-mbp:TestScripts darshandagly$ ./list_ec2.sh
       2
darshans-mbp:TestScripts darshandagly$
```

Figure 4.2: list_ec2 script file



```
usage: ./list_data.sh bucketname
current bucket used: cloudcomputingvikensmrutidarshan, change it if needed
[video-pie2-0326113917.h264,elephant,]
[video-pie2-0326113943.h264,sheep,person,horse,dog,]
[video-pie2-0326113948.h264,elephant,]
[video-pie3-0326113724.h264,chair,]
[video-pie3-0326113834.h264,person,dog,bear,]
[video-pie3-0326113914.h264,horse,cow,]
[video-pie3-0326113943.h264,book,bottle,]
[video-pie3-0326113945.h264,keyboard,mouse,cup,]
[video-pie3-0326113946.h264,horse,cow,]
[video-pie3-0326113955.h264,chair,]
[video-pie4-0326113753.h264,book,bottle,]
[video-pie4-0326113853.h264,horse,cow,]
[video-pie4-0326113913.h264,chair,]
[video-pie4-0326113916.h264,horse,cow,]
[video-pie4-0326113917.h264,book,bottle,]
[video-pie4-0326113924.h264,keyboard,]
[video-pie4-0326113947.h264,keyboard,]
[video-pie5-0326113823.h264,horse,cow,]
[video-pie5-0326113834.h264,chair,]
[video-pie5-0326113912.h264,cat,]
[video-pie5-0326114012.h264,person,dog,bear,]
[video-pie6-0326113834.h264,person,dog,bear,]
[video-pie6-0326113911.h264,elephant,]
[video-pie6-0326113915.h264,cat,]
[video-pie6-0326113942.h264,horse,cow,]
darshans-mbp:TestScripts darshandagly$
```

Figure 4.3: list_buckets script file

5. Individual Contribution

Darshan Dagly (ASU ID: 1215180174)

- **Design:** I collaborated with the rest of the team to design the overall architecture of the entire system. This involved deciding which AWS resources to use, where each component will reside in the architecture, and how will they communicate with each other. I also finalized the architecture of the Application Instances which uses the self-stopping worker logic. We also decided to run the Web Server and AutoScaler on same instance to optimize available number of EC2 instances. I also decided on various fault tolerance strategies to be implemented on the Application Instance to ensure that no requests are lost if any instances go into failure or crash while processing a request.

- **Implementation:** I developed the entire Application Instance module and the AutoScaler module of the Project. The Application Instance module consists of the Master Application Instance and the 18 self-stopping worker application instances. The Application Instance module and the AutoScaler module were both developed using the Spring Boot Framework in Java. A fault tolerance system on the application instance was also implemented by me which ensured that no request was missed or lost if any of the instance failed/crashed. This was done by setting a visibility timeout on requests which were picked by the application instance to ensure that the same request is not picked by any other instance. Once the results are successfully delivered to the user, only then the request is deleted. In case of an instance failure, the visibility timeout would eventually expire, and the request would then be picked up by another instance. The AutoScaler module which ensured that the system successfully scaled out and scaled in upon increase or decrease of load on the web server was implemented by me. I also developed a shell script which was executed on the Application Instances to invoke the darknet detection functionality. This shell script returned the detection results and the video file name to the application instance which was then stored on S3 by the application instance. I also deployed the Application Instance Jar file and the Shell Script on the darknet EC2 image instance provided and created a snapshot and image of the instance after completing the deployment and the setup. This image was used to spawn worker application instances by the AutoScaler module.

- **Testing:** I was in-charge of the unit testing of the modules developed by me. This includes the unit testing of the Application Instance module and the AutoScaler module. Upon successful unit testing, the modules developed by various team-members were integrated and multiple rounds of integration testing was done by me and my team members. I was also involved in the standalone testing of various AWS services to ensure that they were running as expected. I also did the load testing of the entire system using the scripts provided by the TA to ensure that all the requests were processed under high load and none of the requests timed-out.

Viken Shaumitra Parikh (ASU ID: 1215126783)

- **Introduction:** The aim of the Project was about developing a cloud platform using Amazon web services that can take requests from url of a video from the user and return the list of objects detected in the video along with the name of the video. The cloud platform would facilitate the scaling in and out of instances to provide efficient results for varying number of user requests at any time. The AWS services used were Elastic Compute Cloud, Simple Queue Service, Simple Storage Service to serve this purpose.

- **Design :** I contributed to design by collaborating with team members to develop an architecture of the system which consisted of a web server, an AutoScaler, 2 SQS queues, a persistent storage, a master app server and other 18 instances which would be used to scale up and down as resources as per number of user requests in SQS queue. I came up with the idea for architecture with my teammates. We also decided to handle fault tolerance for the system by various strategies so that requests are not lost in unusual behavior of the system. We decided to run the Web server and AutoScaler on same instance to optimize available number of EC2 instances.

- **Implementation:** I collaborated with the team to develop the Web server, to configure the AWS services required for the system, and implementation of the app instances that would process the requests and provide object detection as a list of objects on the video as response. I developed the Web server in Python using Flask framework that asynchronously receives requests and assigns a thread worker to process each request and add them to the request SQS queue, and background thread would also be running to receive the requests long polling the response SQS queue and send response to the user requests. The Web server ran on the same EC2 as the AutoScaler and this EC2 instance would be always running. The response would be added to a global dictionary. A unique id would be generated to map the user request and responses. The request SQS queue would be used as a buffer for the app instances to receive the requests when they are free to process it, and the response SQS queue would be used as a buffer to the web server to receive responses from the app instances. We wrote the app instance and AutoScaler code in Java. The app server instances would receive the requests and delete the requests from the request SQS queue when they have processed them and send the result to the response SQS queue. The app instances would upload the processed results to AWS s3 for persistent storage of the results. The individual app instances would shut down if there are no requests in the request SQS to process, while the AutoScaler will start the app instances if there are more requests than currently running instances can process at this time. The master app instances would always be running. On booting up, the app instances would start listening to the requests of the request SQS queue.

- **Testing:** I was involved in different stages of testing. I started with unit testing of the individual modules and testing individual AWS services to check if they are running as expected. Then I was involved in integrating the system modules and AWS services and testing them after integration. Lastly, testing the system for various loads and faults was done by using the provided test scripts and testing them in different load scenarios.