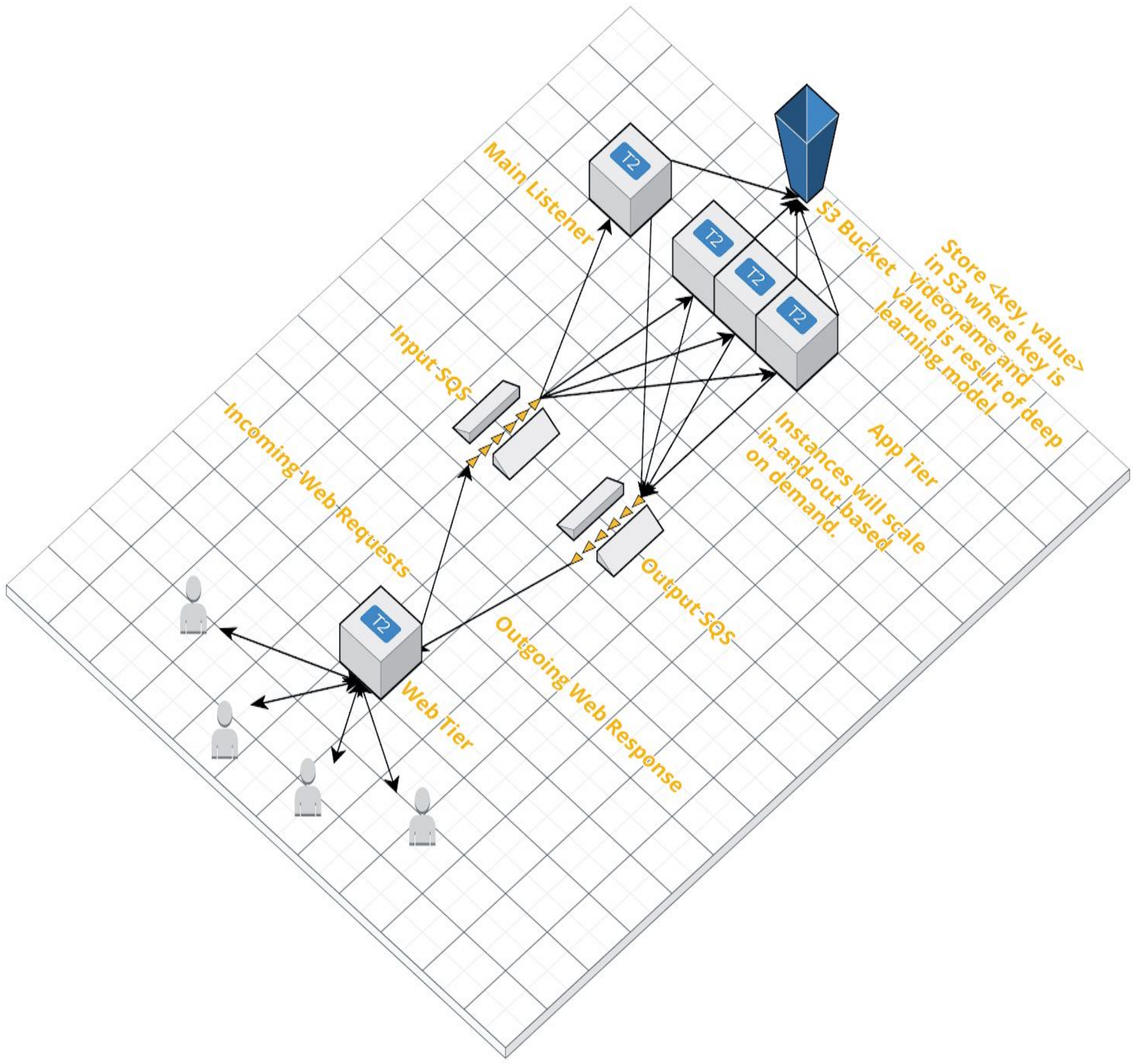
CSE 546 — Project 1 Report

Arsh Deep Singh Padda
Rushikesh Sargar
Suhas Venkatesh Murthy

1. Architecture


Main Listener
S3 Bucket
Store <key, value> in S3 where key is videoname and value is result of deep learning model
Input SQS
App Tier
Instances will scale in and out based on demand.
Incoming Web Requests
Output SQS
Web Tier
Outgoing Web Response
T2

Different AWS Components used in the projects are the following:

- Amazon Elastic Compute Cloud (AWS EC2)
    - EC2 is the main processing unit for both the web tier and app tier. In the web tier, we are using an EC2 to handle the requests and send the message to SQS. This EC2 is also responsible to deliver the response to all the incoming requests. In the app tier, we areusing EC2 to do video retrieval and video recognition.
- Amazon Simple Queue Service (AWS SQS)

    - We are using an SQS to maintain an input queue and an output queue. The role of input queue is to generate a message id for each request and push it to queue which will be later retrieved by EC2 for processing. The role of the output queue is to deliver the response back to the request received.
- Amazon Simple Storage Service (AWS S3)
    - We are using an S3 for storing the video name as the key and the result of the deep learning model as value.

We had two architecture in our mind:

- Architecture 1
    - Web Tier Role
        - Receive the request and download the video. Save the video into an S3 bucket (video bucket) with the key as the video name and the value as the video. The web tier then sends a message in the queue with the video file name as the message body. The web tier will then keep trying to check another s3 (answer bucket) with the filename as the key to look for answers. Once it receives the answer, it sends it back as a response in the form <filename, result>.
    - Queue Role
        - Receive the message from web tier and hold them till app tier is ready to receive the message. This architecture doesn't require an output queue.
    - App Tier Role
        - Once the app tier EC2 instance receives a message from the queue, It fetches the video from the s3 bucket (video bucket). It then runs the video recognizing model and feteches the result. The results are then stored in the s3 bucket (answer bucket). The instance terminate themselves if there are no messages in the queue.
    - Problem with the Architecture:
        - We were using two S3 buckets which was an overhead when we could have achieved the desired result using only one S3 bucket.
        - The continuous polling for looking into the answer bucket was redundant.
        - If we have concurrent requests then there is a lot of work for the web tier to do. This can cause a bottleneck as the web tier has to download and then push the file to S3.
        - All the above reasons forced us to change the architecture.
- Architecture 2
    - Web Tier Role
        - Receive the request and send a message in input queue to notify the EC2 that a request has been made. Listen for messages in the output queue. The responses were sent to which every thread receives the message first from the output queue. The web tier plays the role of scale out. It checks the number of messages in the queue and the number of instances that are turned on (this also includes the number of instances that are pending i.e. in process of turning on).

- Queue Role
    - Receive the message from web tier and hold them till app tier is ready to receive the message. Similarly, we have an output queue which will hold the filename and result to the request posted once EC2 has done it’s work.
- App Tier Role
    - Once the app tier EC2 instance receives a message from the queue, it sends a request to the url and downloads the video. It then runs the video recognizing model and feteches the result. The results are then stored in the s3 bucket (answer bucket). The instance then delete the video downloaded and sends the message to the output queue. The instance will then keep looking in the queue for the next 30 seconds and will start the process of recognition if it finds a message. If there is no message then it will terminate itself. In order to serve request immediately, we kept one instance on EC2 which will be always running. The maximum number of instances that can be used at any time is 19 instances.
- Fault Tolerance
    - Our architecture is fault tolerance as once an EC2 instance will pick up a message, then it will make the message invisible to others in the queue. If the EC2 instances runs in some problem than the message will be available back in the queue after 10000 seconds. If the message is processed properly, then the message will be deleted. This ensures there is no duplication.

2. Autoscaling

Explain in detail how your design and implementation support automatic scale-up and -down on demand.

We made scale up the responsibility of the web tier since it’s main responsibility is to handle the response and send them to the input queue. The number of messages in the queue acts as a measure of how many instances we need to start in order to scale out. The web tier will check the number of messages in queue and will check how many instances are running (this includes the instances which are in processing stage also), if the difference of the number is positive, then that many instances are turned on. If there are already 19 instances running, then no action is taken. The App tier instances will perform their task and pick the next request from the queue if there is any available.

Scale down was done by the EC2 instances. Once they process a request from the queue, the instances look into the queue to see if there are any more requests. If there aren’t any more requests, then the instance looks every second for the next 30 seconds and shutdown if there are no requests. This 30 seconds delay was intentionally decided by the team because we felt that this way we should be able to handle sudden recurring burst of requests more efficiently.

3. Code

3.1 Code Modules

A. Web-Tier Module

a. Web-Tier module is a spring boot app following MVC framework.
b. We have a main thread which monitors the number of instances to be created based on the number of requests to be served.
c. Requests are served using different thread which hits the controller.
d. When a request is made by the client, we add a message to the input SQS for the app tier to process it.
e. Web tier retrieves the message ID from the message sent to input SQS in order to map client request and response from app tier.
f. Web tier continuously monitors the output SQS for response. The response message has a message body in this format - "message-id:video-name:result".
g. Web tier maintains a HashTable to store messages since the web app tier continuously polls the output SQS.
h. If a client request's result is available in the output SQS, web app polls it, compares message ID and returns it to the client if the message ID of the request and message ID in the response message body is same. If not, the result is saved in the HashTable and returned from HashTable when that request is being served.
i. Web app deletes the message from the output queue once it polls it from the SQS.

B. Load Balancer Module

a. Scale out is implemented at the web tier. Scale out creates instances on demand based on the number of running instances and number of messages in the input SQS.
b. We have two running instances at all times - Web tier and Main Listener. Apart from these we can create a maximum of 18 more, summing up to 20 on a whole.
c. Web tier creates instances using the Listener's AMI, which when boots up, picks up the message from the input SQS and puts the result to output SQS.
d. Web app tier's primary thread continuously monitors the number of messages in the input SQS and number of running instances at the moment.

   e. If the number of messages to be served is greater than the number of instances running and lesser than 20 (Limit), load balancer triggers a request to create more instances with Listener’s AMI to serve the requests.
   f. At most, we create 18 more instances. If there are more than 20 requests, load balancer triggers a request to create maximum allowed number of instances which is 18 more and others requests have to wait in the input SQS.

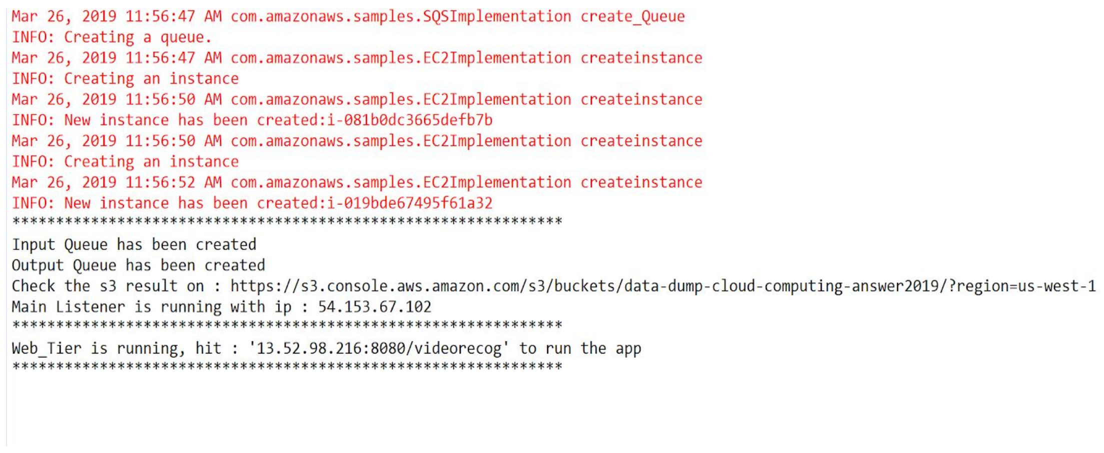
C. Listeners Module

   a. This module is mainly executed by the Listeners in App-Tier which has 2 type of Listeners : Main Listener and App Listeners
   b. All listeners will receive message containing the url from the input SQS i.e input_queue and it will download the video (on EC2 listener) and pass the video path to the deep learning module to predict the objects in the video.
   c. Call for the deep learning module is made using process builder where the path of the video and the weight is passed.
   d. Output of the deep learning module is pushed on s3 bucket with the file name as the video_name and predicted value as the content of the file.
   e. The output is also passed to the output SQS i.e output_queue in the format (input_message_id : video_name : predicted_output)
   f. Once the output is sent, the message from the input_queue is deleted.
   g. And App_Listener looks for available message to pick from the input_queue, if it doesn’t receive any message, it waits for a small amount of time for the next messages, if it doesn’t get any , it terminates and performs the scale-in operation. t2.micro is a smaller instance and latency of turning an instance on is more, to avoid that delay, the Listeners wait before terminating.
   h. The Main Listener remains on all the time, and looks for messages in the input queue.

D. Build Infrastructure

   a. This module builds the entire infrastructure for the project.
   b. AMI of Web_Tier_Image, Main_Listeners Image, Bucket Name, Queue Names and other credentials are provided to the module
   c. A java code, creates the necessary components of the project i.e Web_Tier, Main_Listener, Input_Queue and Output_Queue and S3 bucket
   d. And returns the ip of the web_tier which can be used to use the application.

e. The spring boot application takes 15secs to start the server, after which the web tier is ready to take the requests.



```
Mar 26, 2019 11:56:47 AM com.amazonaws.samples.SQSImplementation create_Queue
INFO: Creating a queue.
Mar 26, 2019 11:56:47 AM com.amazonaws.samples.EC2Implementation createinstance
INFO: Creating an instance
Mar 26, 2019 11:56:50 AM com.amazonaws.samples.EC2Implementation createinstance
INFO: New instance has been created:i-081b0dc3665defb7b
Mar 26, 2019 11:56:50 AM com.amazonaws.samples.EC2Implementation createinstance
INFO: Creating an instance
Mar 26, 2019 11:56:52 AM com.amazonaws.samples.EC2Implementation createinstance
INFO: New instance has been created:i-019bde67495f61a32
***************************************************************
Input Queue has been created
Output Queue has been created
Check the s3 result on : https://s3.console.aws.amazon.com/s3/buckets/data-dump-cloud-computing-answer2019/?region=us-west-1
Main Listener is running with ip : 54.153.67.102
***************************************************************
Web_Tier is running, hit : '13.52.98.216:8080/videorecog' to run the app
***************************************************************
```

3.2 Project Setup

A. Setup of Web-Tier

a. Instruction given in the project description were followed to setup the project, Set default region to us-west-1 and ami-0e355297545de2f82 to create the app tier.

b. Create a jar of the MainListener module and move it on the EC2 instance using scp command on the local terminal i.e scp -i key_pem <Web_Tier_file_name>.jar ubuntu@<public_ip>:/home/ubuntu/

c. Log into the EC2 instance and change the permission of the jar file using: chmod +x <Web_Tier_file_name>.jar

d. Setup java environment by installing default-jre(jdk) on the instance

e. To execute this jar file on boot, add the execution command in crontab using : crontab -e , a file will be opened, press **i** to edit it and add "**java -jar /home/ubuntu/<Web_Tier_file_name>.jar**" in it save and close the file using **:wq**

f. Create Image of this instance and mark it as Web_Tier_Image.

B. Setup of App-Tier

a. Instruction given in the project description were followed to setup the project, Set default region to us-west-1 and ami-0e355297545de2f82 to create the app tier.

b. Create a jar of the MainListener module and move it on the EC2 instance using scp command on the local terminal i.e scp -i key_pem <Main_Listener_file_name>.jar ubuntu@<public_ip>:/home/ubuntu/

c. Log into the EC2 instance and change the permission of the jar file using: chmod +x <Main_Listener_file_name>.jar
d. Setup java environment by installing default-jre(jdk) on the instance
e. To execute this jar file on boot, add the execution command in crontab using : crontab -e , a file will be opened, press **i** to edit it and add "**java -jar /home/ubuntu/<Main_Listener_file_name>.jar**" in it save and close the file using **:wq**
f. Create Image of this instance and mark it as MainListener_Image.
g. Similarly, Create a jar file for the AppListeners and follow the same steps to execute it on boot and create image of it as AppListener_Image.

3.3 Execution Steps

A. Execute the Build_Infra java code from any local machine, it will setup the infrastructure and return the web_tie's public ip.
B. You will be able to see Web-Tier and Main_Listener running on the console.
C. You can execute send_request.sh script to make requests to the web_tier, console will print the output for each request. Syntax usage - ./send_request.sh <url> <number_of_concurrent_requests> <total_number_of_requests>.
D. Scale Out and Scale in results can be observed in the console.

3. Project status

We met all the requirements of the projects which were the following:

- Web Tier is implemented successfully, which accepts the request and responds back with the result of the deep learning model. We have only one EC2 instance which is running this module.
- App Tier has been implemented successfully - It has 2 modules with one main listener which is always in an ec2 instance. The other listener comes up based on demand. Web tier scales up - triggers create request with Listener AMI to process requests. Further, we also implemented the scale-in at app tier to shut-down instances if it is done processing the request.
- Load Balancer module - we have implemented the scaling up successfully. The module continuously monitors the input SQS and scales up accordingly to handle the requests gracefully.

4. Individual contributions (optional)

Arsh Deep Singh Padda

- **Design** -
  - I was the main lead who designed the two architecture. I understood the functionality of the of the component discussed in the class and came up with the plan to design the initial build of the architecture. My initial design had a lot of loopholes. My first architecture would have made use of two S3 storage and redundant call to S3 buckets to check for answers. This would eliminate the free tier of AWS very quickly and wasn't a cost effective solution. I later worked on the feedback from my teammates and build a better architecture. The new architecture made the web tier light and quick to handle requests. All the heavy data task such as video download and running the deep learning model was pushed to app tier. The SQS queues were used to decouple the app tier and web tier.
- **Implementation** -
  - We had two tier in the architecture. I build a python web server (web tier) using flask to help test the app tier while we were making changes to web tier. I first build the architecture 1 web tier to test how it will hold to multiple requests. This was a good idea since architecture 1 was flawed and helped in building a better architecture for both the web tier and app tier.
  - I build the AMI image for the web tier and app tier. Once we started testing the whole architecture together, I was responsible to push the jar and write a script to run the jar executable at startup. Both the images of web tier and app tier were prepared by me.
  - The startup script was something which we had a lot of troubles.
- **Testing** -
  - I was responsible to check the status of the SQS and the number of EC2 instance that were running while we did the testing. There were times when our web tier was broken and I had to manually feed data into the SQS to test the app tier. The python web tier helped us in testing the app tier while we made changes to the java web tier.

Suhas Venkatesh Murthy

- **Design** -
    - I came up with the design for the web tier module from both infrastructure and programming perspective - framework, interfaces and classes.
    - The design phase involved making two crucial decisions regarding the way to send input client request to app tier for further processing and way to match input request to response.
    - I collaborated with the rest of the team to discuss advantages and disadvantages of each of the techniques and finalized the one which was ideal.
- **Implementation** -
    - Web tier is built using java spring framework which follows MVC architectural pattern.
    - I made sure that the primary thread which is the main thread in the web tier continuously monitors the number of running instances and number of input requests and scales up - triggers instance create request accordingly. The thread which handles client requests is a separate thread which hits the controller directly.
    - Whenever there is a request from the client, the code is implemented in such a way that the web tier sends a message to the input SQS with the url - for downloading the video at the app tier.
    - Once a message is sent to the input queue, I implemented the web tier to retrieve the Message ID of the sent message in order to match it with the response so that each of the clients get a response for the same request they made. I implemented the web tier to continuously monitor the output SQS for response with polling at every 10 seconds.
    - Once a response is acquired, we have the message body in this format - “message id of input req : video-name : output”.
    - I split the message body based on “:” and then compared the message ID which I obtained from the message body with the message ID retrieved from the input send message to SQS.
    - If the message ID matches, the output is quickly sent back to the client. If the IDs don’t match, web tier adds it to in memory thread safe data structure - Hash Table with key as the message ID.
    - Since the output response for few requests are already in the hash table, the corresponding thread serving the request, picks the result from the hash table and returns it to the client.
    - Once a message is been polled from the output SQS, I implemented the web tier module to delete it from the queue since either the response is sent back to the client or response is stored in the hash table.
- **Testing** -
    - I collaborated with the team members to test the app end to end.
    - I was also actively involved in the troubleshooting, debugging and rectifying errors from both coding and setup perspective.