CSE 546 — Project1 Report

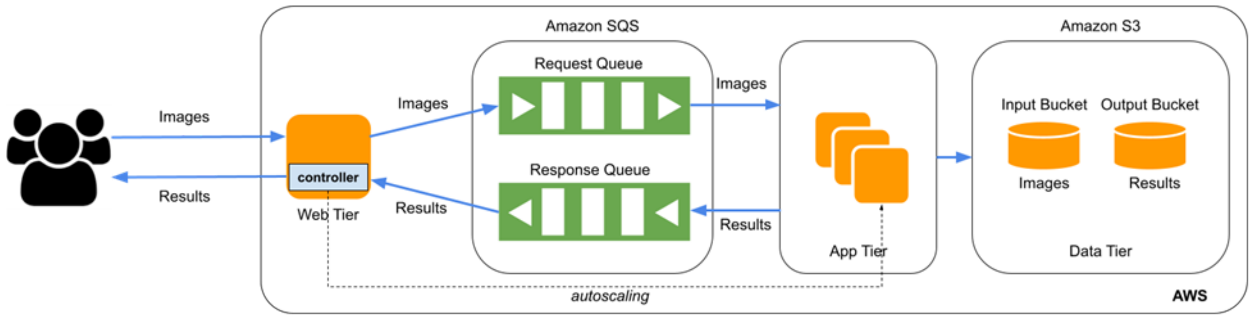
Ezedine Kargougou, Tanner Greenhagen, Mark Pop

1. Problem statement

The problem we were trying to solve is that of a low tier face recognition application. Essentially this was about taking in a picture of a person and being able to quickly and accurately output the name of the person by using modern day tools such as AWS and machine learning. This is important because it could influence the lives of people in the future. Although this is a simple introductory project, cloud computing and face recognition has the potential to affect the lives of millions of people and open doors to future technologies.

2. Design and implementation

2.1 Architecture



The idea behind this design is that images will be fed via the Web Tier and those requests will wait in the SQS queues (Request Queue) until they are ready to be processed. Then they will be passed to the App Tier. This is where the bulk of the computations will be done. The images will be passed to an S3 bucket called the input bucket. Depending on the result of the App Tier, a result that is stored in another S3 bucket called the Output bucket will send a response that will go through the Response Queue and back to the Web Tier. The major components of this design include:

- **S3 buckets**: We used the 2 S3 buckets to store images and the result of those images.
- **SQS:** We used SQS queues to handle incoming and outgoing requests.
- **EC2 instance:** Finally we used EC2 instances to handle the computations. We implemented autoscaling that will create or delete instances depending on the requests stored in the queues.

2.2 Autoscaling

A controller was created specifically to implement the Auto Scaling feature on the web tier. The method was to first look at the number of messages in the request queue, if there were messages in the queue and the number of instances in either 'running' or 'pending' state were below twenty, then we created an app tier instance. There was another method in place to monitor the number of instances and delete the most recently created one if for some reason the number of app tier instances exceeded twenty. Finally the last method looked at both the number of messages in the request queue and the number of messages in flight from the request queue and terminated all instances if both were at zero.

These three functions were run on a loop continuously, scaling up or down depending on the number of images requested of our system.

3. Testing and evaluation

For our application the actual testing was broken down into four testing phases: testing the app tier jar, testing everything locally, testing autoscaling remotely with request handling locally, and finally testing everything like it would actually be run. The actual code for our app tier was written in java and took out messages from the response queue, saved the image that came from the string data of the request queue, ran the classification python script with the saved image, saved the image and classification output to s3 and sent the outessage to the response queue. We sent image data in the queues and then ran a local python script that would provide some constant output and see if the image would be stored in s3 along with the output and if an output response would be stored in the response queue. If the image was saved and the output made it to the right locations then it was considered to be working. The next test worked by checking if the loop from the initial request to the ending response would make it. Our python flask webtier would be run locally and exposed via the local ip address and port 8081. We would run the java apptier code locally and have it wait to read messages from the request queue and send messages back to the response queue. We would then send 5 images to the local webtier using the workload generator script. We then expected to get the same name for each image as we still weren't actually using the classification python script and had a script that would just return a constant name. The correctness of the run was judged by if every request got a response and if every image was stored to the input bucket. The next test worked on actually autoscaling. The app tier jar was pushed to an ec2 instance that had an image created from it. Then, on the webtier the controller.py file was run that handled the scaling up and scaling down of the app tier instances. The flask server was still run locally, but now we were using the "--with-threads" options which would allow for natural flask multithreading to occur. This was now useful as there would be multiple app tiers to process data rather than just the single locally run jar. Requests were sent from the verifying multithreaded workload generator and the results were then judged based on the correctness of the classification and the time taken to get those results. We started by just testing 5 images, then 10, then 20, then 50, and finally 100 images. We would verify that the correct number of images and output text was stored and then we would clear the input and output bucket. We also looked at the increase and decrease of the app tier instances. We checked that the number of app tier instances would scale up to 20 and then back down to 0 after completion. The final test was to just put the flask application on the web tier instance and run it there with the elastic ip address rather than the local ip address we had been using. The same measures as before were used for evaluation. It was found that we would always correctly classify each image which is expected as we did not modify the classifier and we were matching the responses to the correct requests. The time it took for classification however varied. The biggest time loss was starting up the instances which took from 2 to 4 minutes from our observations. It also was found that the more we ran the webtier the slower and slower it got which seems to be linked to our aws CPU utilization. However the results for 100 images would usually finish in about 9 minutes which we considered to be acceptable.

4. Code

The controller.py was our controller code that was in charge of managing the Auto Scaling functionality. It was coded in Python and used the Boto3 library to interface with the AWS APIs. The first one it implemented was the SQS queuing service where we referenced the 'inQueue' as our request queue. We used this resource to query the number of instances in the queue and enter one of two methods that were on a continuous loop. Those methods were createInstance and stopInstance.

The criteria for entering createInstance was that there were more than zero messages in the request queue and that the count of 'pending' or 'running' instances were less than twenty. The createInstance function used the AWS API for EC2 clients where we passed in specific parameters to launch an EC2 instance from a pre-made app tier AMI. One of the parameters was user data that ran a command line to run the code as soon as the new app tier instance was created, which was:

```
#!/bin/bash
cd /home/ec2-user
java -jar /home/ec2-user/AppTier.jar
```

The stopInstance method has two states based on a variable passed in (0 or 1) depending on where the method entered from. The initial loop looked at the number of EC2 instances and ran the stopInstance method with a 0 if the number of 'running' instances exceeded twenty. In this state a list of AMIs were retrieved and the most recent EC2 instances above twenty were terminated. Alternatively, the outer loop checked the request queue and if both the number of messages and number of messages in flight were zero, then the program would enter the stopInstance method with a parameter of 1. In this state the controller would terminate all active instances since the lack of messages would mean the job is approaching the end stages.

To run this application we secure copied (scp) the file onto the web application and ran the process in the background with the command:

```
python3 controller.py &
```

The actual AppTier java application would continuously loop and try to grab messages from the request queue. If there were any messages that it grabbed, it would look to work on the first message and return the other messages right back to the request queue to be handled by other app tier instances while it was working on the one message. This was done by changing to a low visibility timeout to allow it to be accessed quickly by another instance. The message body had the image name, the width, the height, and then groupings of the rgb values of all of the individual pictures. The components of the string could be separated by splitting the string on spaces. Each rgb value would then be converted into a singular colored pixel value by adding the red value, and the green value multiplied by 256 and adding the blue value multiplied by 256 squared. These integer pixel values would then be used to create an image file that would get sent to the python file as input. The thread would then be frozen until the result of the classification was obtained. The image and output would then be stored in the correct s3 buckets and the image and the classification name would be sent back to the response queue. The application itself was actually a maven application and it was compiled into a runnable jar via "mvn clean compile assembly:single". The project can be imported as an eclipse maven project to actually be ran locally. The jar was then run on app tier startup. The jar can be run via 'java -jar AppTier.jar' assuming you are in the same directory as the jar.

The final component was the web tier application which ended up being a python flask application. The application would look for post requests sent to the ip address of the machine running the file with just a slash (http://44.193.255.131:8081/ in our case). It would then get the image file name from the filename associated with the bytes of the 'myfile' key. The actual image came from the stream value of the same 'myfile' key value. A string would then be created that would have the image name, image width, and image height stored as the first three values. They were separated by spaces for delimiting . purposes. The rgb values of each pixel were then grabbed and stored together only separated by commas to keep them grouped together (r1,g1,b1 r2,g2,b2 …). The string would then be sent as a message to the request queue. Then it would wait until messages were available in the response queue. When messages were available there they would be grabbed and checked to see if they had the image name stored with them. The response queue message body would have the classified name along with the image name. If the image name matched then the classification name also in the message would be sent back to the client. If the message did not have the correct image name it would be returned back to the response queue for the correct thread to grab it. To deal with multiple requests at once, we made use of the built in option for flask to multithread our application with the '--num-threads' option. This made it so we could switch between different requests and not have to deal with each request sequentially. Before running the application the environment variables needed to be set via "export FLASK_APP=flaskr" and "export FLASK_ENV=production" which would tell flask where to find the __init__.py starting file and what debugging options to use. The application itself would be ran with flask run -p 8081 --host=0.0.0.0 --with-threads where '--host=0.0.0.0' just set it to run on whatever the web tier elastic ip address was (44.193.255.131 in our case). After running that command on the web tier requests could be sent to the web tier for image classification. To actually run the file you would need to copy over the entire flaskr directory and run the flask command above from the parent folder of the flaskr directory.