CSE 546 — Project Report

Sainath Latkar, Ryan Kittle, Sharanya Banerjee

1. Problem statement

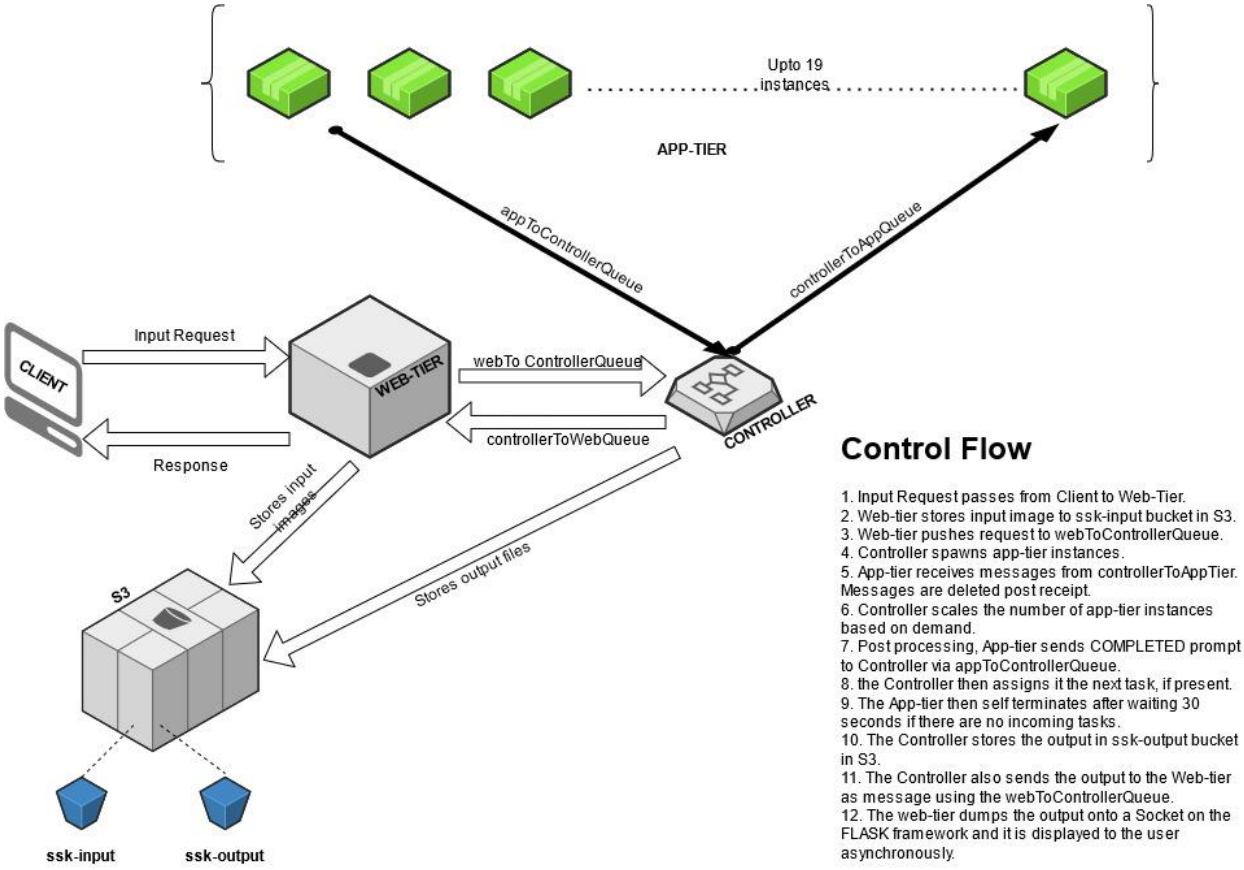
In this project, we aim to deploy a Cloud based model which has an underlying Image Classification Algorithm. The model consists of multiple segments as described in architecture below. The model should be able to asynchronously handle requests given by the user, who can feed any amount of images into the model through the webpage. The image classification algorithm processes those images and returns the result as a tuple (Filename, Classification), displaying results as they get calculated by the different EC2 instances. Depending on the traffic of incoming user requests, the system must be able to autoscale the number of app tier instances (Upscale for more traffic and Downscale for less traffic).

This problem is important for us to solve in order to understand the 5 key concepts of cloud computing: On-demand self-service, Rapid Elasticity, Broad Network Access, Measured Service and Resource Pooling. This gives us hands-on experience on AWS EC2, SQS and S3, which are industry leaders in providing cloud frameworks today.

2. Design and implementation

2.1 Architecture

The following architecture diagram depicts the system we have deployed:



From the above architecture diagram, we can describe each module as follows:

1. *Web-Tier:* The web-tier is the entry gateway to our system. The user uploads images from the deployed FLASK server and they pass on to the Controller through the Web-Tier. There are two queues between the Web-Tier and the Controller namely, webToControllerQueue and controllerToWebQueue. The Web-Tier stores the images input by the user into the S3 input Bucket (ssk-input). Then image names pass as messages to the Controller through the webToControllerQueue. Post processing, the Controller sends the output as messages to the Web-Tier by the controllerToWebQueue. The web tier then dumps those (image names, classification) output values to the FLASK server on a Socket, asynchronously and in real-time. The user can see the results directly from the webpage.

2. *Controller:* The second module in our system is the Controller. The Controller is connected to the Web-Tier by 2 queues (webToControllerQueue and controllerToWebQueue) and to the App-Tier by 2 queues (controllerToAppQueue and appToControllerQueue). Depending on the incoming traffic, the Controller deploys app-tier instances (up to 19) using our auto scaling algorithm. The controller fetches the messages from the web-tier and pushes them to the app-tier for processing. Post processing, it fetches the COMPLETED message from the app-tier and saves the Classification result to the S3 output bucket (ssk-output).

3. I The app-tier is our processing tier. It houses the underlying python image-classification system which takes images passed by the user as input and classifies them. App-tier instances are scaled based on incoming traffic, upscaled for high demand and downscaled otherwise. After completion, the app-tier sends a COMPLETED message to SQS and terminates itself.

4. *Simple Queue Service:* SQS is our Message passing service. There are 4 queues namely, webToControllerQueue, controllerToWebQueue, controllerToAppQueue and appToControllerQueue. It transmits messages to and fro the Web-Tier and Controller and Controller and App-Tier using a Publisher-Subscriber model. These messages tell each element what task they are supposed to do. After messages have been received by the corresponding Subscriber, they are auto-deleted to prevent redundancy.

5. *S3 Buckets:* In our system we have two S3 buckets buckets, one for input (ssk_input) and one for output (ssk_output). The input bucket stores the image files as passed by the user through the FLASK framework and the output bucket stores the output as .txt files.

2.2 Autoscaling

In our system, we have incorporated a system to autoscale based on incoming demand.

Situation: High demand.

Action: **[UPSCALING]** All the incoming demand is pushed into the controllerToAppQueue. At the onset, the Controller reads no more than the first 19 messages. The Controller then spawns no more than 19 app-tier instances to handle the corresponding requests. Once a request has been processed, the app-tier passes a COMPLETED message to the appToControllerQueue. Upon receipt of that message, the Controller then asynchronously assigns the next request to the next available app-tier for processing. This eliminates the time taken by an app-tier to spawn and reduces the Turn-around Time of the system.

Situation: Less demand.

Action: **[DOWNSCALING]** Once the number of incoming requests decreases to 18 or less, the App-tiers not assigned a task, wait for 30 seconds to read any incoming message. If the App-tier instance does not receive any new request within the time frame then it auto-terminates and sends corresponding feedback to the Controller. This way, the Controller knows at any given time, the number of app-tier instances that exist as spawned. All available resources are managed efficiently in this manner, while keeping the rapid-provisioning capability of the system intact.

3. Testing and evaluation

For testing our system, we implemented the following test cases:

1. *Passing 1 request -*

   The Web-tier stores the image into the S3 input (ssk-input) bucket and passes the request to the Controller via the webToControllerQueue. The Controller then spawns 1 app-tier instance using the AMI. It then sends the image name as a message through the controllerToAppQueue to the app-tier instance. The instance processes the request while having sent a PROCESSING request onto the appToControllerQueue. Post processing, the instance sends a COMPLETED message along with the outcome onto the appToControllerQueue to the Controller. The app-tier instance then waits 30 seconds for any new requests. If there are no new requests, it auto-terminates itself. The Controller then pushes the outcome into the S3 output (ssk-output) bucket and sends it to the socket on the web-tier via the webToControllerQueue. The output is displayed on the webpage.

2. *Passing 19 requests -*

   The Web-tier stores the images into the S3 input (ssk-input) bucket and passes 19 requests to the Controller via the webToControllerQueue. The Controller then spawns 19 app-tier instances using the AMI. It then sends the image names as messages through the controllerToAppQueue to the app-tier instances. The instance processes the requests while having sent 19 PROCESSING requests onto the appToControllerQueue. Post processing, the instance sends 19 COMPLETED messages along with the outcomes onto the appToControllerQueue to the Controller. The app-tier instances then wait 30 seconds for any new requests. If there are no new requests, they auto-terminate themselves. The Controller then pushes the outcomes into the S3 output (ssk-output) bucket and sends it to the socket on the web-tier via the webToControllerQueue. The output is displayed on the web page asynchronously.

3. *Passing >19 requests* -

   The Web-tier stores the images into the S3 input (ssk-input) bucket and passes all the requests to the Controller via the webToControllerQueue. The Controller then reads the first 19 requests and spawns 19 app-tier instances using the AMI. It then sends the image names as messages through the controllerToAppQueue to the app-tier instances. The instance processes the requests while having sent 19 PROCESSING requests onto the appToControllerQueue. Post processing, the instance sends 19 COMPLETED messages along with the outcomes onto the appToControllerQueue to the Controller. The app-tier instances then wait 30 seconds for any new requests. The Controller then pops the next request and assigns the job to the next available app-tier; and the cycle continues. If there are no new requests, they auto-terminate themselves. After all the requests are processed, the Controller pushes them into the S3 output (ssk-output) bucket and sends it to the socket on the web-tier via the webToControllerQueue. The output is displayed on the web page asynchronously.

4. Code

Instruction to Run the code:

To run the code, please follow the steps listed below -

$pwd

/home/ubuntu

$cd Cloud_WEBTIER

$python3 sqsListener.py &

$python3 start.py &

$cd ../controller/CLOUD-ControllerFinal/

$python3 controller.py &

Controller Code:

We first initialize our global variables at the top, such as the SQS queues and S3 bucket, essentially creating the connections necessary for our program to run. Next we create a helper function that checks for free instances that are available for message processing, which returns true when there is at least one instance available, and false when the max number of instances are already processing requests. The main functionality for our controller lies in the “while True” loop, which checks for any messages within the appToController or webToController queues.

- If there is a message in the appToController queue, this means that an App tier instance has begun processing an image, finished processing an image, or is ready to be shut down. When an instance has begun processing an image, no further action must be taken beyond updating the local list of instances on the

controller. However, when an app tier instance has finished processing an image, the controller must update the local list, save the result to a file which gets uploaded to the S3 bucket, and send a message to the controllerToWeb queue which states that a new result is available to be displayed. Finally, if the message states that an app tier instance is ready to be shut down, the local list of instances is updated to reflect the loss of one machine. Once the appropriate action has been taken by the controller (Processing, Completed, or Destroy), the message is removed from the appToController queue.

- If there is a message in the webToController queue, it means a request has been received and an image is ready to be classified. The controller initially forwards the message from the webToController queue to the controllerToApp queue, and then looks to spawn a new instance if necessary in order to handle the request. Once the controller has forwarded the message and spawned an instance if necessary, it removes the message from the webToController queue.

Web Tier Code:

We have three files that work together in order to send and receive user requests and results: submit.html, sqsListener.py, and start.py.

- submit.html: We use Flask in order to take user requests and images, connecting to a socket and sending/receiving data through POST requests with multipart/form-data. When a new image result is received, the page automatically updates to include it.
- sqsListener.py: We initially create our global variables, which are our connections to the Flask socket and controllerToWeb SQS queue. The primary functionality of this code is executed through the “while True” loop, which listens for messages on the controllerToWeb queue. When a message is received, this means an image result has been calculated and is ready to be displayed for the user. The result is sent as a POST request through the socket, with Flask updating the page to include it. Once the message has been forwarded it is removed from the SQS queue.
- start.py: Again, we start by creating the global variables, which include the webToController and controllerToWeb queues, the S3 bucket, and allowed file extensions for the user to upload. Towards the bottom of the program we have multiple helper functions working with Flask to specify what actions should be taken when files are uploaded. The primary function of this file is the upload_file method, which uploads files to the S3 bucket and sends messages to the webToController queue.

App Tier Code:

Similar to the others, this code begins by creating the necessary global variables, including the SQS queues. Again, the bulk of the code takes place in the “while True” loop, which checks for messages in the controllerToApp queue. When a message is received, it sends a message to the controller, notifying it that the instance has begun processing an image. Next, it downloads the image file from the S3 bucket, and runs the image classification program on the downloaded file. Once the classification is complete, it sends another message to the controller, this time notifying it that it has finished processing the image. Afterwards, the program checks for more messages in the controllerToApp queue, signifying additional images to be processed. If there are messages, it begins classifying the next image. If there are no more messages, the instance waits 30 seconds for more to be sent. If those 30 seconds pass with no more requests being received, the instance begins to shut itself down, notifying the controller through the appToController queue.

5. Individual contributions (optional)

Sainath Latkar: My name is Sainath Latkar (ASU ID: 121943096). Below is listed my individual contribution to the project, with respect to Design, Implementation and Testing of the system.

Design - In the system that we have developed, I came up with the design to build the Controller. The Controller receives message requests from the web-tier and spawns the required amount of app-tier instances (no more than 19). It then assigns the app-tier instances with tasks from the queue and pushes a PROCESSING message into the SQS. Once an app-tier completes a task, the controller pushes the COMPLETED message into the SQS. This is how the Controller aids in auto-scaling the system.

Implementation -

Controller code: We first initialize our global variables at the top, such as the SQS queues and S3 bucket, essentially creating the connections necessary for our program to run. Next we create a helper function that checks for free instances that are available for message processing, which returns true when there is at least one instance available, and false when the max number of instances are already processing requests. The main functionality for our controller lies in the “while True” loop, which checks for any messages within the appToController or webToController queues.

If there is a message in the appToController queue, this means that an App tier instance has begun processing an image, finished processing an image, or is ready to be shut down. When an instance has begun processing an image, no further action must be taken beyond updating the local list of instances on the controller. However, when an app tier instance has finished

processing an image, the controller must update the local list, save the result to a file which gets uploaded to the S3 bucket, and send a message to the controllerToWeb queue which states that a new result is available to be displayed. Finally, if the message states that an app tier instance is ready to be shut down, the local list of instances is updated to reflect the loss of one machine. Once the appropriate action has been taken by the controller (Processing, Completed, or Destroy), the message is removed from the appToController queue.

If there is a message in the webToController queue, it means a request has been received and an image is ready to be classified. The controller initially forwards the message from the webToController queue to the controllerToApp queue, and then looks to spawn a new instance if necessary in order to handle the request. Once the controller has forwarded the message and spawned an instance if necessary, it removes the message from the webToController queue.

Testing - The Web-tier stores the image into the S3 input (ssk-input) bucket and passes the request to the Controller via the webToControllerQueue. The Controller then spawns 1 app-tier instance using the AMI. It then sends the image name as a message through the controllerToAppQueue to the app-tier instance. The instance processes the request while having sent a PROCESSING request onto the appToControllerQueue. Post processing, the instance sends a COMPLETED message along with the outcome onto the appToControllerQueue to the Controller. The app-tier instance then waits 30 seconds for any new requests. If there are no new requests, it auto-terminates itself. The Controller then pushes the outcome into the S3 output (ssk-output) bucket and sends it to the socket on the web-tier via the webToControllerQueue. The output is displayed on the webpage.

Ryan Kittle: My name is Ryan Kittle (ASU ID: 1212804526). Below is listed my individual contribution to the project, with respect to Design, Implementation and Testing of the system.

Design - In the system that we have developed, I came up with the design to build the Web-tier, which consists of three main files. One file is necessary as the html for the user input, while the other two allow for the communication and data transfer. This allows for the web-tier to accomplish its main tasks, which I would consider to be the intake of user images, saving these images to the appropriate S3 bucket, notifying the controller through the SQS queue, and displaying the image classification results on the webpage for the user. Keeping the code modular and fault-tolerant was something that I wanted to keep at the forefront of my mind throughout the entire implementation.

Implementation -

Handling the first task, the intake of user images could be easily accomplished using Flask. It can take in multiple files at once, which was one of the most important requirements for user input that was designated for us. Once the user inputs their images, Flask is able to send the files to the EC2 instance through multipart POST requests that hold the files as data. They are sent over a socket, which is easy to connect to through a specific port. Once the message has been output through the socket, the web-tier EC2 instance saves the input image(s) to the “input-ssk” S3 bucket. This is done through multiple functions that work with Flask, which are housed in both the submit.html file and and start.py. As each file gets saved to the input S3 bucket, messages are individually sent out over the webToController queue. Every individual message contains the filename for one image, and gives the controller knowledge about which images in the S3 bucket must be processed by the app-tier EC2 instances. Once the messages are sent out over the queue, the web-tier has given the controller the information necessary to get the app-tiers running. The final task of the web-tier, displaying the image classification results for the user, consists of two main steps, and is handled by the sqsListener.py file. It uses a “while True” loop to listen to the controllerToWeb queue, and whenever a new message is received, it sends the result tuple as a POST request to Flask through the aforementioned socket. Once Flask receives this message, it displays the results on the webpage for the user to view.

Testing - In my opinion, there were two aspects that I would need to test: the web-tier specifically, and our design as a whole. Testing the web-tier specifically did not require too much work, and was mostly spent making sure the socket and queue configurations were correct. In addition to this, I also had to ensure that images were being uploaded correctly, and only specific file types could be uploaded. For testing our design, I handled the test-case that included passing exactly 19 requests. For this, we would need to activate exactly the max amount of app-tier instances (19), and each one would shutdown after the waiting period passes, without any one instance doing more than one image calculation. This required some adjustments in terms of timing, and some specific lines had to be reordered in order to ensure messages were deleted at the proper time, but once these adjustments were made, our design was able to perfectly handle the uploading of 19 images, and subsequently scale themselves down after the waiting window has passed.

Sharanya Banerjee: My name is Sharanya Banerjee (ASU ID: 1219403952). Below is listed my individual contribution to the project, with respect to Design, Implementation and Testing of the system.

Design - In the system that we have developed, I came up with the design to build the App-Tier. The app-tier consists of the underlying image classification program. It receives a task from the Controller and processes it. Once it is done with processing, it sends the output to the Controller and waits 30 seconds for a new task. If no new task comes to it within the specified time frame, it auto terminates. This is how it ensures most efficient utilization of available resources and aids in autoscaling.

Implementation -

App-tier code: Similar to the others, this code begins by creating the necessary global variables, including the SQS queues. Again, the bulk of the code takes place in the “while True” loop, which checks for messages in the controllerToApp queue. When a message is received, it sends a message to the controller, notifying it that the instance has begun processing an image. Next, it downloads the image file from the S3 bucket, and runs the image classification program on the downloaded file. Once the classification is complete, it sends another message to the controller, this time notifying it that it has finished processing the image. Afterwards, the program checks for more messages in the controllerToApp queue, signifying additional images to be processed. If there are messages, it begins classifying the next image. If there are no more messages, the instance waits 30 seconds for more to be sent. If those 30 seconds pass with no more requests being received, the instance begins to shut itself down, notifying the controller through the appToController queue.

Testing - The Web-tier stores the images into the S3 input (ssk-input) bucket and passes all the requests to the Controller via the webToControllerQueue. The Controller then reads the first 19 requests and spawns 19 app-tier instances using the AMI. It then sends the image names as messages through the controllerToAppQueue to the app-tier instances. The instance processes the requests while having sent 19 PROCESSING requests onto the appToControllerQueue. Post processing, the instance sends 19 COMPLETED messages along with the outcomes onto the appToControllerQueue to the Controller. The app-tier instances then wait 30 seconds for any new requests. The Controller then pops the next request and assigns the job to the next available app-tier; and the cycle continues. If there are no new requests, they auto-terminate themselves. After all the requests are processed, the Controller pushes them into the S3 output (ssk-output) bucket and sends it to the socket on the web-tier via the webToControllerQueue. The output is displayed on the web page asynchronously.