CSE 546 — Project1 Report

Bishnupriya Pradhan(1222307502), Karan Dabas(1219496811), Swetaswini Nayak(1219505573)

1. Problem statement

As part of this project, we developed a cloud application that does face recognition. The entire application was built using available AWS Iaas services(Elastic Cloud Compute), messaging service(SQS) and storage service(S3). Such services are used because of their ability to adapt to scaling out or in. The EC2 component allows us to run the face recognition system in parallel and helps us handle multiple requests that increase the overall application performance and efficiency. It also makes the system on-demand and deals with large requests without waiting for manual intervention. SQS is used as the only communication bridge between the web server(Flask) and the compute instances; such channels are independent of any system dependencies of webserver and compute instances and operate at high communication speed. Fig-1 shows the overall system design, this approach with all its components gives us all the key characteristics of the cloud system and achieves the most reasonable implementation.

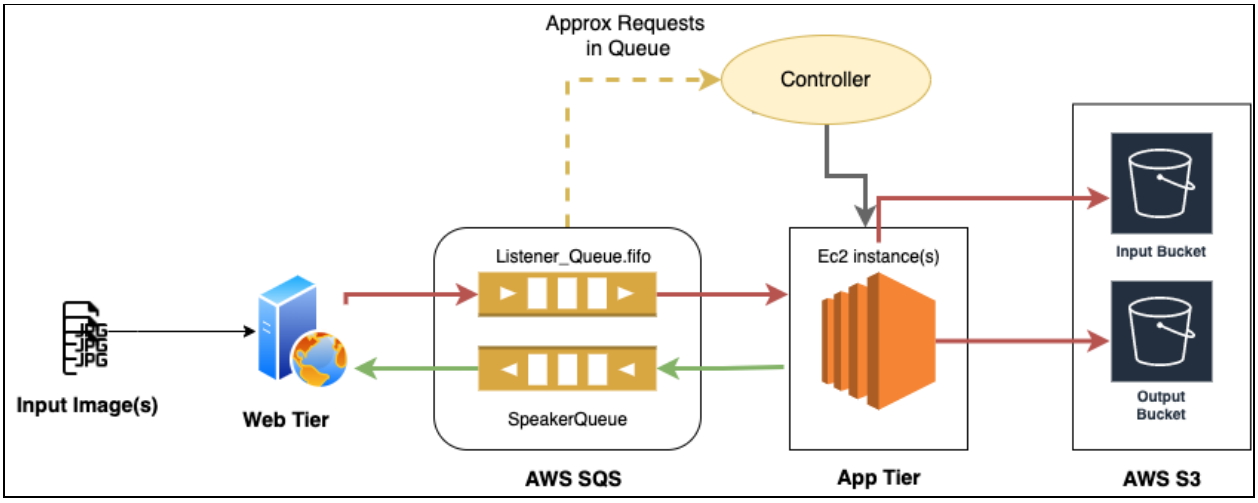

Fig-1: Overview of cloud application design.

2. Design and implementation

2.1 Architecture

The detailed system architecture can be seen in Fig-2, the entire setup is based on two components: a consumer and a producer. The producer creates queries and the consumer processes the incoming query and renders results. The two components are web-tier(producer) and app-tier(consumer). Talking about the web-tier, the web server is developed using Flask in python and is hosted on an independent ec2 instance alongside a controller script. This instance is named web-tier and is always running irrespective of the available requests. The controller script is used to manage the app-tier ec2 instances, this controller also has access to the listener sqs that hosts the incoming requests in a FIFO type queue. At any given time, the controller checks for approximate available messages in this queue and based on it makes a decision to launch new instances or restart the instances that are in the stopped status. The

app-tier consists of individual ec2-instances of the same nature and underlying code called app-instance. The image and the filename of the input request is communicated through the listener sqs, to achieve this, first the image is converted into a 3d NumPy array and serialized into a string and passed forward by the web-tier. At the receiver end, the app-tier instance reads the message from the queue and deserializes it back to the 3d image array and uses it to reconstruct the image using PIL fromarray method and pushed down the provided DL model for classification. Once the classification label is ready, the app-tier instance first writes the input image with its filename into an s3 storage under the input-response bucket and the classification label is written to the output-response bucket. The app-instance is also connected to the speaker queue and the class label is passed to the queue for web-tier output.

The sqs is used here to decouple our application and to execute asynchronous behavior in our application. Another crucial part of app-instance is that of its ability to auto-shutdown once it's done working, the app-instance is stopped if it sits idle for over 5 mins.

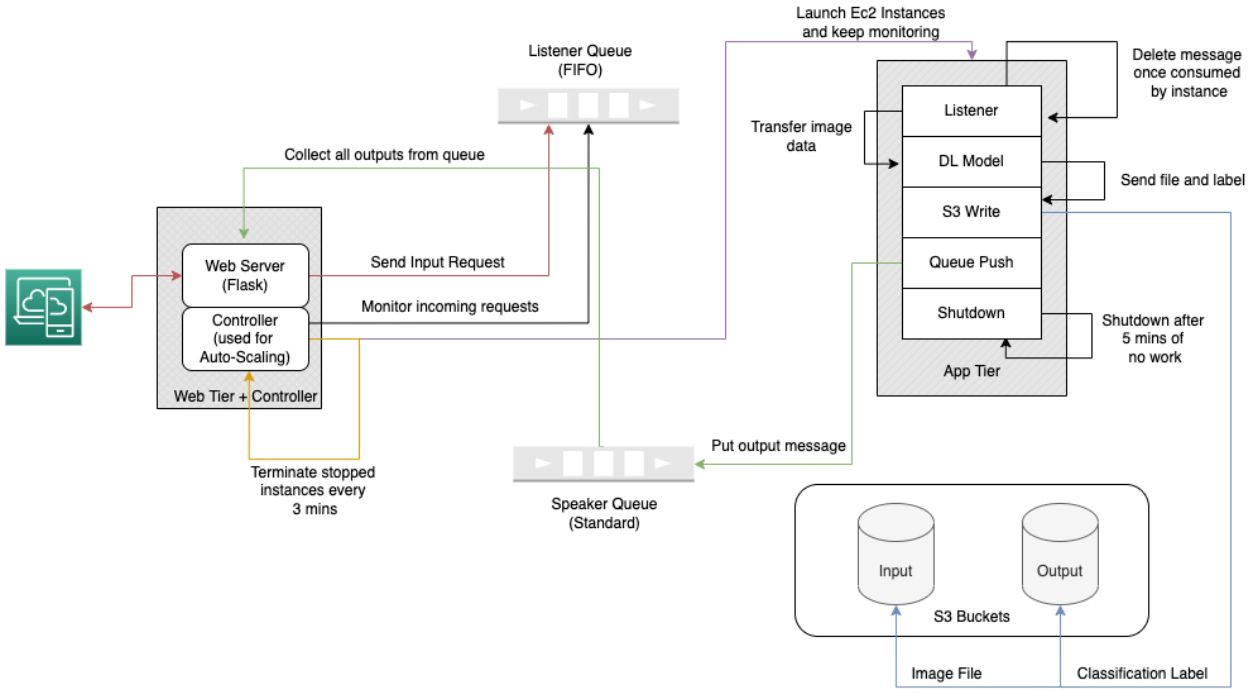

Fig-2: Architecture and workflow of the entire cloud application.

2.2 Autoscaling

The autoscaling feature is developed in-house instead of using AWS services such as a load balancer. The feature is mainly implemented into the controller. The controller is responsible for launching and terminating instances where-in the stop protocol is only carried by the individual app-instances. The controller makes sure that the instances are limited to a max limit of 19, this includes new, running, and stopped instances. This limit is never exceeded, given N requests the controller first checks for available instances and if it cannot find any, it checks the stopped instances if available they are restarted and if none are available new instances are launched this is called scaling up. If N is < 19 then N new instances will be created else only 19 instances are launched. If X instances are already running or in the stopped state, then N-X will be the new number and based on the value comparison with 19 will decide a suitable

number of new instances to launch. For a scenario where the total input requests are less than the running instances, all the idle instances will be first stopped based on the app-instance logic(with 5 mins idle time limit) and in every 3 mins, the controller collects all the stopped instances and terminates them. This is called scaling down.

3. Testing and evaluation

The cloud app is tested on two fronts, we compare the number of requests with only one instance which is the case for the simple workload_generator and the multithread approach where we send out the request of a number of messages over multiple threads. The Table below shows the setting of these two test runs and are evaluated based on the number of instances the cloud app launches and how much time does it take to get the complete result from the command when run from zero state(where no app-tier instances are running initially). In summary the multi threaded approach launches multiple instances and does in fact return results much faster compared to one after another request approach. On average, an approach with multithreading is handled 134sec faster compared to the simple sequential approach.

Number of requests(Images)	Number of Web tier threads	Number of instances launched	Time taken to get results(*) [seconds]
10	1	1	132.26
10	10	10	102.64
30	1	1	147.43
30	30	19	126.84
100	1	1	490.72
100	100	19	140.81

Apart from testing overall time taken for results, we also looked into our messaging bridge(SQS) and found that even though our app is super fast and gets results quickly(once instances are running) the only bottleneck is that of mapping the results in the Speaker queue to the right web-tier thread. We are using a random id to keep a track of it but the issue is that even though we know the random id for a given thread run we cannot directly use it in an API call to get that exact message from the standard queue(Speaker), we need to read the message and compare the id with the author name and if it's a match return the label and delete the message and if not a match the thread looks for another message in the queue and this goes on until every thread receives their message. Due to this sometimes we end up in a situation where the responses for a command can take longer compared to the time taken by the cloud app to write the results in the S3 buckets.

4. Code

App-Tier: We have created a new custom AMI(id:) on top of the given AMI for the project. The new AMI has an app-instance file that holds the logic for app-tier instances. We also have another instance that has the same underlying OS and is used for app controller and hosting flask web app(web-tier).

1. app-instance.py- This is the only script needed to implement the logic for app-tier instances that consume on listener sqs and process for classification label, saves the input images and labels to a s3 and labels are returned to a standard sqs calles Speaker that holds the output that is later used by web-tier in return of the input post request.

   The script's **main()** function holds the entire implementation such as reading messages from Listener sqs, using PIL library to reconstruct the image from the numpy array and makes a call to the given DL model in face_match function. The face_match returns the classification label and after this the function called **save_s3image(img, filename)** that writes the image and filename to input-response bucket. The **save_s3result(label,filename)** is used to write the classification label with filename as the key to the output-response bucket in S3. Another function is **send_result(label,filename)** that pushes the label and filename into the Speaker queue for final output of the web-tier. One last function is **string2array(str1)** that takes in the Listener queue message(of type string) and converts it back to a numpy array.

   The calls to these functions are put in a while loop which is always true and keeps feeding on the incoming messages, the logic for time calculations are also done in it. The time difference between the latest message processed with the current time is used as a measure to estimate the idle time. As this idle time goes to 5mins the instance calls the **stopInstance()** function that simply uses the boto3 api that stops the instance and puts itself to the stopped state.

2. app-controller.py- This script contains the logic for the controller that manages the app-tier instances. This script holds the **main()** function that has a while loop that runs infinitely and within its block we make calls the the boto3 api to check for the approximate available messages in the Listener queue using **queue.attributes['ApproximateNumberOfMessages']** after this step we get the integer value of number of messages to process and now we use this to make a decision on how many instances to launch/restart. The controller has the ability to do actions like create instances using the function called **create_instance(id,maxLimit)** where id is some integer value used for naming the instance and maxLimit is the upper limit on the number of instances which is set to be 19. The choice of 19 is due to the fact that we already need an instance that will run the web-tier and controller in it. The **terminate_protocol(stopped_instances,maxLimit)** function is used to terminate the already stopped instances in every 3 mins interval. The function takes two arguments, the first being the list of ids for the stopped instances and the second is the maxLimit. The **update_stopped_instances(maxLimit)** is a function we use to add in the newly stopped instances to the list. This is necessary because each of the app-tier instances stop on their own and we need to keep track of it so that we can maybe later decide to restart or terminate these instances. We also have a function to restart the stopped instances that uses the boto3 api call to achieve the task, this method is called **restart_stopped_instances(num,stopped_instances,maxLimit)** and takes in the amount of instances to restart, list of stopped instances and maxLimit variable. Finally, a function

called **update_instances()** that simply updates the active instances list and fills them with the instance ids. This function is called in a while loop as well. A time delay is also added whenever we launch instances and is set to 45secs and a 5sec delay is added right after the termination protocol is called for the instances.

Web-Tier: This a Flask server that takes in an image and writes it to the Listener queue for further processing. Once the image is sent it waits for the response from the cloud side app-tier for results in the Speaker queue.

1. app.py- This script holds the entire implementation of the flask server that takes input images. The script takes the input image and assigns a random id to it along with the filename info from the input image. The next step is to convert the image file to a numpy array that we can later serialize to a string for sending it to app-tier. The function is called array2string(arr) that does the array to string conversion. The allowed_file(filename) function is used to cross check the file type to make sure the only specific file type is processed. After sending the image to the queue , a while loop that is always true keeps on checking the Speaker queue for available classification responses and the message is deleted only when the random id assigned with the input image matches the id that comes with the label in return. If the match is found the label is returned as text.

Project Setup: Before we begin, following are the required python libraries for successful setup of the project:

1. Flask: used for the web server, use command- $pip install flask
2. Boto3: it is a python sdk for AWS, use command- $pip install boto3
3. Pillow: used for getting image array from image file, use command- $ pip install Pillow

App-tier- Launch the instance similar to the web-tier+controller step and now only scp app_instance.py to the instance home directory. After this we need to make sure to add a periodic run of the app instance using crontab.

Add script to crontab:

crontab -e

-open a vim, type i to enter insert mode

type: * * * * * python3 /home/ec2-user/app_instance.py

-press esc and then type :wq to save and exit from the vim

After this you may exit from the ssh and go to the EC2 dashboard and select the instance and create an AMI temperate from it. Save the AMI id, this **needs to be added to the AMI_ID variable in app_controller** for the next part.

Web-tier + controller- Launch the instance for the given AMI(ami-0a55e620aa5c79a24) in project description pdf. After this access the instance using ssh. Make sure that you download the accessKey file.

SSH command: ssh -i <accessKey file> ec2-user@<public id of instance>

After this access the folder where the app.py and app_controller.py is located using the terminal and transfer files to this instance. You can use scp command for this,

SCP command: scp -i <accessKey file> <python script name> ec2-user@<Public IPv4 DNS>:~

Now go to the terminal window where you accessed the ec2 instance using ssh and do ls command to check the files in its home directory. Next step is to use screen command to run both files in separate screen sections so that even when you disconnect from the instance the scripts keep running.

Screen based run:

screen -S <give some session name>

python3 app.py

now do ctrl+A+D to detach the session

again create a new session for app_controller and run it using python3 app_controller.py and then detach from it. Now you may disconnect from the ec2-instance.

Lastly we need to make sure that our web-tier can be accessed from public ip. Select the instance in the ec2 dashboard and follow instructions from this blog.

5. Individual contributions (optional)

For each team member who wishes to include this project in the MCS project portfolio, provide a one-page description of this team member's individual contribution to the design, implementation, and testing of the project. Different students' individual contributions cannot overlap. One page per student; no more, no less.

Karan Dabas (ASU ID- 1219496811)

Following are my contributions for the Face recognition cloud applications project:

- Design- Design is one of the most crucial steps and the first step of the project. I started by listing out the requirements and covering the literature review of the AWS components that will be used for this project. This also includes the much-needed controller component that monitors and manages the ec2 instances. The process includes figuring out the individual component settings and how to incorporate them with specified project guidelines. The Fig-2 system architecture includes various design choices such as the selection of FIFO queue for Listener and standard queue for Speaker SQS. This was done mainly to impose the idea of always processing the input requests that come in first. And for the results, we need a queue that can be accessed in a standard manner irrespective of the input request order. This choice helps us make an output of an async style system. After preliminary implementation, I found that if we run the app using multi threads the results get mix-matched, so I came up with the idea of assigning a unique random id to each incoming request and when reading for the output in the Speaker queue I only delete a message if it contains the same random id thus giving us the correct match for the request.
- Web Tier- I was responsible for creating the web application that takes input image requests using Flask. The Flask selection was mainly because of its efficiency and easy implementation that can also support multithreading. I started with creating a UI based app where a user can select one or many images and upload them for the results. But later also added a simple just post based page. This helped us run testing for the workload_generator scripts provided by the instructor. As part of the web app, I added connections to the AWS SQS. One connection was made to the listener queue that passed the input image request for processing it. Another connection was with the speaker queue that holds the labels for the images that are processed by the app-tier. I also programmed the serializer that converts the input image into a string message that can be transferred via SQS. To achieve this I used PIL, json and numpy libraries. The PIL is used to get the image array from the image file and the numpy and json are used to convert the array to string that is compatible with the queue message type.
- Controller- The entire structure for autoscaling is incorporated into the app_controller.py. I was responsible for coming up with this feature. I implement the protocols for instance monitoring and management such as terminate, create, update, update_stopped. The create is used to launch new instances(app-tier) that consumes messages from the listener queue, the update is used to maintain and update the list of active/running instances, the update_stopped makes sure to update the list of ids for all the instances that are in stop state. Finally the terminate is used to kill all the stopped instances available in the stopped instance list. The terminate protocol is carried in every 3 mins, this means that all the stopped instances are terminated in every 3 mins. The actions of the controller are independent of app tier instances. Thus making it much faster when we scale up and down.
- App Tier- I coded the stop protocol for the app-tier instance. It makes use of the time library and always calculates the difference between the most recently consumed task time with the current time. As soon as the difference hits a threshold of 5 mins the stop protocol is executed and the instance turns off itself. Apart from this I also helped with setting up the deserializer that is used to get back the image from the string message in listener sqs.