CSE 546 — Project 1 Report

Harsh Sanjaykumar Nagoriya, Dhruval Vinod Patel, Kenil Vipulkumar Patel

1. Problem statement

In this global era of modern computing, it is necessary to efficiently provide the solution to the asked problem in a cost-efficient and robust way. Cloud services providers such as AWS, Microsoft Azure, et al provide such facilities to deploy our service (or software) on seemingly infinite resources. At the same time, it is essential to meter these available resources and to continuously monitor them in order to avoid the overuse of resources. If left unmonitored, it is highly likely that it may increase the cost associated with the use of resources. This project is created to handle multiple client requests at the same time with elastically scaling cloud resources that dynamically scale themselves on demand in a cost-efficient manner. The service we implemented using AWS resources will be able to recognise the face image uploaded by the end-user to the application we deployed. It will be able to instantiate and run the multiple EC2 instances concurrently to process a large number of inputs as rapidly as possible. This autoscaling feature we implemented will let the application utilize the capabilities of the cloud to perform thriftily and accurately.

2. Design and implementation

2.1 Architecture

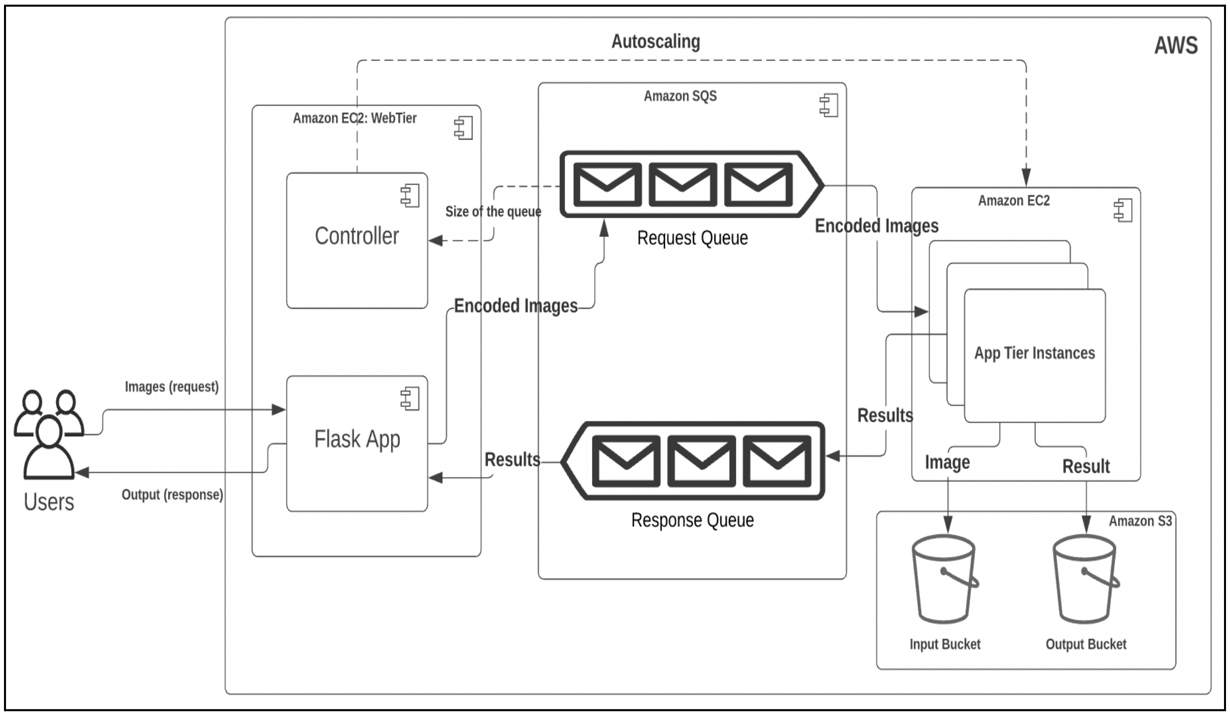

Image 1: System Architecture

2.1.1 Amazon Services

2.1.1.1 Amazon EC2 (Amazon Elastic Compute Cloud)

Amazon EC2 is an IAAS platform that provides virtual computing services over the network. As per the architecture displayed above, the web tier (controller and flask app) and app-tiers were deployed on Amazon EC2 platforms.

2.1.1.2 Amazon SQS (Amazon Simple Queue Service)

Amazon SQS is a distributed messaging queuing service that provides programmatic message handling via different Amazon resources over the internet. Two queues request queue and a response queue were used to pass the information (data) between the web tier and app-tiers. These queues loosely couple the web tier and app-tiers.

2.1.1.3 Amazon S3 (Amazon Simple Storage Service)

Amazon S3 is an object storing service that provides scalable persistent storage over the internet. Two buckets were created for storing the inputs and outputs. The image files are stored in the input bucket and the predicted results in stored in the output bucket as key-value pairs.

2.1.2 Architecture workflow

2.1.2.1 Web tier

As described in the image above, A web tier was implemented on one of the Amazon EC2 instances mainly handling these two tasks. Both of them get invoked automatically through systemctl and cronjob:

- **Flask Application:** The flask deployment application (a framework of python) is implemented to handle the GET and POST HTTP requests. When a user sends/uploads the image file, the flask application renders the relevant GET/POST methods and receives the image sent over the internet. This image gets converted into the stream and gets encoded to base64. Further on, a string message is created consisting of the file name and encoded message and is sent to the SQS request queue. Now, this request is submitted to SQS waits asynchronously to be processed and remains on a waiting period until the respective output of the image file is not received from the app tier. After receiving the respective response from the server, the flask application returns the output to the user and deletes it from the SQS response queue.
  Additionally, several other functions for purging the buckets and queues are also implemented in the flask application to conveniently reset the buckets and queues.

  In this flask application, boto3 SDK external dependency is used in order to interact with the AWS resources.

- **Controller:** The python-based code is used to implement the controller that implements the autoscaling logic. This controller gets invokes as soon as the web tier is started, and continuously checks the statuses of the app-tiers, and queues to perform the autoscaling logic. The main target of this code is to implement autoscaling logic that is explained explicitly below.

2.1.2.2 App tier

The app-tiers are implemented in python language that does the main part of the whole project. In order to get this working, a replica instance of the given AMI was created that consisted of face recognition algorithms provided by the professor. In this instance, boto3, base64 and PIL dependencies were installed. After installing these components, our code was deployed, which works along with the given face_recognition file. The deployed code continuously reads the request queue for the images to be processed. A single app-tier retrieves the first message from the queue and hides it for 30 seconds in order to avoid repetition processing. This retrieved message is split into two parts, file name and encoded message, sent from the web tier. Now this encoded message is decoded and gets converted into an image, in order to make the predictions. This image is saved to the S3 input bucket. Now this image is passed to the face_recognition file to make the predictions. The output from this file is saved to the variable and is appended with the filename. This filename and output are sent to the response queue for the web tier to get the result. Additionally, this filename as a key and output as a value is also stored in the output bucket. After successfully completing the all above-mentioned steps, the request message from the request queue is deleted and the program again continuously checks for the message in the request queue. If a message is received, it again performs the above-mentioned steps. If the queue is empty, then it waits for 5 seconds and checks again for the message. This way, it continuously checks for the message and processes it when received.

In order to run this script automatically without user interaction, a shell script is generated that invokes the python code on every start and every reboot.

For creating 19 similar instances, an AMI of this working code was generated and instance replicas from this AMI were created.

2.2 Autoscaling

As stated above, the controller file does the autoscaling job. It continuously checks the status of the instances and the number of messages in the request queue. Here, for the messages in the queue, it takes the number of all the messages in the queue Visible and NotVisible. Then it checks for the total number of running and stopped instances.

Here one catch is, that it takes a few seconds (more than 30) to start an instance and stop an instance, and in this scenario, an instance goes to the ‘pending’ or ‘stopping’ state which may work inconsistent according to the autoscaling logic. For handling this, a function was implemented that checks whether the instance is not in the ‘pending’ or ‘stopping’ state. If so, the logic implemented does not process that instance and skip that instance for 30 seconds to change its state to an appropriate ‘stopped’ or ‘running’ state.

For the normal scenario, the controller checks if the number of messages is more than the number of active ‘running’ instances or not. If the number of messages is more than the number of active ‘running’ instances, it calculates the difference between them and starts that many instances. For example, if we have a total of 14 messages in the queue and 6 instances running, then, in this case, it will calculate 14 messages - 6 running instances = 8. It will start 8 more instances to get the rest 8 messages processed but here, we have set the limit to 19 total running instances only. So before starting the instances, it will check whether the number of running instances is not exceeding the limit of 19. If so, it will not start any new instance but, only wait for the messages to be processed by app-tiers and get deleted by them. For example, if we have a total of 48 messages in the queue and 17 instances running, then, in this case, it will calculate 48 messages - 17 running instances = 31, here it will find the minimum of (31) and (allowed active instances 19 - running instances 17 = 2), so it will start 2 instances only. In this way, it scales out the instances.

On the other hand, If the number of messages is less than the number of active ‘running’ instances, it calculates the difference between them again and stops that many instances. For example, if we have a total of 4 messages in the queue and 16 instances running, then, in this case, it will calculate 4 messages - 16 running instances = -12 (absolute value 12). It will stop 12 instances. In this way, it scales in the instances.

All the above-mentioned tasks run continuously without any kind of user interaction. The controller program (in which this autoscaling logic is implemented) gets invoked automatically on start and reboot by the systemctl and cronjob.

3. Testing and evaluation

These steps were followed to test the application:

1. The application was tested first on 50 images given in the description and then with 100 images.
2. With the number of images stated above, it was checked whether the application can take multiple images at once during upload time (done using a multi-threaded workload generator).
3. The status of the request queue was checked to verify the number of messages sent to the queue.
4. The status of the input bucket was checked to verify the number of images sent for processing.
5. Autoscaling logic was checked by observing the number of running instances in the EC2 dashboard.
6. The status of the response queue was checked to verify the number of messages sent to the queue.
7. The status of the output bucket was checked to verify the number of images sent for processing.
8. Checked the output of the multi-threaded workload generator for the number of correct responses.

Results were received as:

1. Request queues were getting filled and emptied as soon as messages are received and processed.
2. Input buckets were getting filled as soon as app-tier received the images
3. Request queues were getting filled and emptied as soon as messages are received and processed.
4. Output buckets were getting filled as soon as the app-tier processed the images and generated the output.
5. Finally, the output of the multi-threaded workload generator was as expected and under the given time limit.

4. Code

The detailed functionality of every program is followed.

Explain in detail how to install your programs and how to run them.

- **Files**
  - Web Tier
    - controller.py
      - This file handles the scaling up and down of EC2 instances according to the messages in the request queue.
    - webv2.py
      - It contains the API that receives the multiple images from the front end, sends them to the request queue and gets the processed output from the response queue.
      - It also has an API that navigates users to the home page.
    - autoweb.sh
      - It contains the execution commands for the python file webv2.py
    - auto.sh
      - It contains the execution commands for the python file controller.py

  - App Tier
    - index.py
      - It takes the encoded image from the request SQS, decodes it and sends the image to 'face_recognition.py' for processing and sends the result to the response queue.
      - After receiving the output of the image, the algorithm stores the image in the input bucket and outputs it to the output bucket of S3.

    - face_recognition.py
      - It contains the code that processes the images and classifies them. This file was provided by the professor for this project.
      - It also has an API that navigates users to the home page.

        - auto.sh
            - It contains the execution commands for the python file index.py

- **Execution Flow**
    - Once the web tier instance of EC2 is in running status, the controller.py file is automatically executed. We start the server manually. Now, once the server starts, the user can send multiple images for processing. These images are received by webv2.py which encodes the images, sends them in the request queue (req) and waits for the result.
    - The controller.py continuously checks for the messages in the request queue and starts or stops the EC2 instances as per requirement.
    - Now, the messages from the request queue are received by index.py which decodes the message and sends the decoded image to face_recognition.py for processing.
    - After receiving the output from face_recognition.py, index.py stores input inside the input bucket (ip34) and outputs to the output bucket (op34) of S3. It also, simultaneously, sends the output of the response queue (res) and deletes the message from the request queue.
    - This output is received by webv2.py which extracts the message from the response queue, deletes it from the queue and returns it to the user.

- **Setup and execution:**
    - Copy the above-mentioned files in the proper directory
    - Install the dependencies: boto3, base64, pillow
    - Set the cronjob by writing cronjob -e
        - @reboot sh <filename>.sh

5. **Individual contribution:**

6. **5.1 Harsh Sanjaykumar Nagoriya**

- Design:
  - Coming up with the best possible design that reduces the development efforts and produces correct output was an essential part.
  - My part was to decide the working mechanism of the app tier for multiple request handling. My job was to design a code in a way that the controller should maintain the states of the existing app-tiers. I came up with an idea to check the states of each and every app-tiers continuously so that the autoscaling algorithm gets the status of the instances before starting or stopping an instance.
  - Additionally, my part also included designing app-tiers and a web tier in a way that both do not require manual work orders to perform their tasks. I came up with a logic that automatically invokes the python codes on instance starting.

- Implementation:
  - **AWS access rights:** It remains essential for every module to maintain enough access rights to perform further actions. I set and provided the key access to AWS resources for each and every module we are using.
  - **States of the instances:** For the above-mentioned part, handling the states of the existing app-tiers, I created a logic that checks the number of running, pending, stopping and stopped on a loop that helped the autoscaling code work efficiently within the allowed app-tier instances limit.
  - **Automation scripts:** For the automation of the modules we deployed, I created multiple shell scripts that run on system-level access on every start and reboot we do. I implemented these shell scripts in cronjob tasks and systemctl tasks.
  - **Scale In instances:** For saving on the costs of resources, it is necessary to check if it is in use or not. My above-mentioned logic gives the states of the instances and the queue returns the number of requests to be processed. Combining these two parts, I implemented a scale-in algorithm that stops the instances if we have less number of requests than running instances.
  - **Asynchronous processing:** For multiple requests, it was necessary to implement an algorithm that executes other requests while waiting for a previously sent request and asynchronous task to complete. I implemented python-based await/async function calls that handle this kind of issue and provide an accurate response to each request.

- Testing
  - **Unit Testing:** Each module created by me needed to be tested before merging with other working codes. I performed unit testing on every module/algorithm I created keeping the different scenarios and conditions in mind.
  - **Integration Testing:** After creating multiple algorithm modules, we merged them all togather and checked if they work flawlessly or not. I came up with various conditions of my modules that can cause issues while invoking merged code and tested them all.