CSE 546 — Project Report

Aayush Hemalbhai Shah (1214024859)
Prerak Kishor Raja (1219453586)
Varad Mahesh Bhogayata (1219972780)

1. Problem statement

The project aims at building an elastic web application that can automatically scale out and scale in on-demand and cost-effectively by using cloud resources. The resources used were from Amazon Web Services. It is an image recognition application exposed as a Rest Service to the clients to access. The application takes the images and returns the predicted output by the deep learning model by using the AWS resources for computation, transportation, and storage. So the tasks involved designing the architecture, implementing RESTful Web Services, a load balancer that scales in and scales out EC2 instances at App Tier according to the demand of the user.

2. Design and implementation

2.1 Architecture

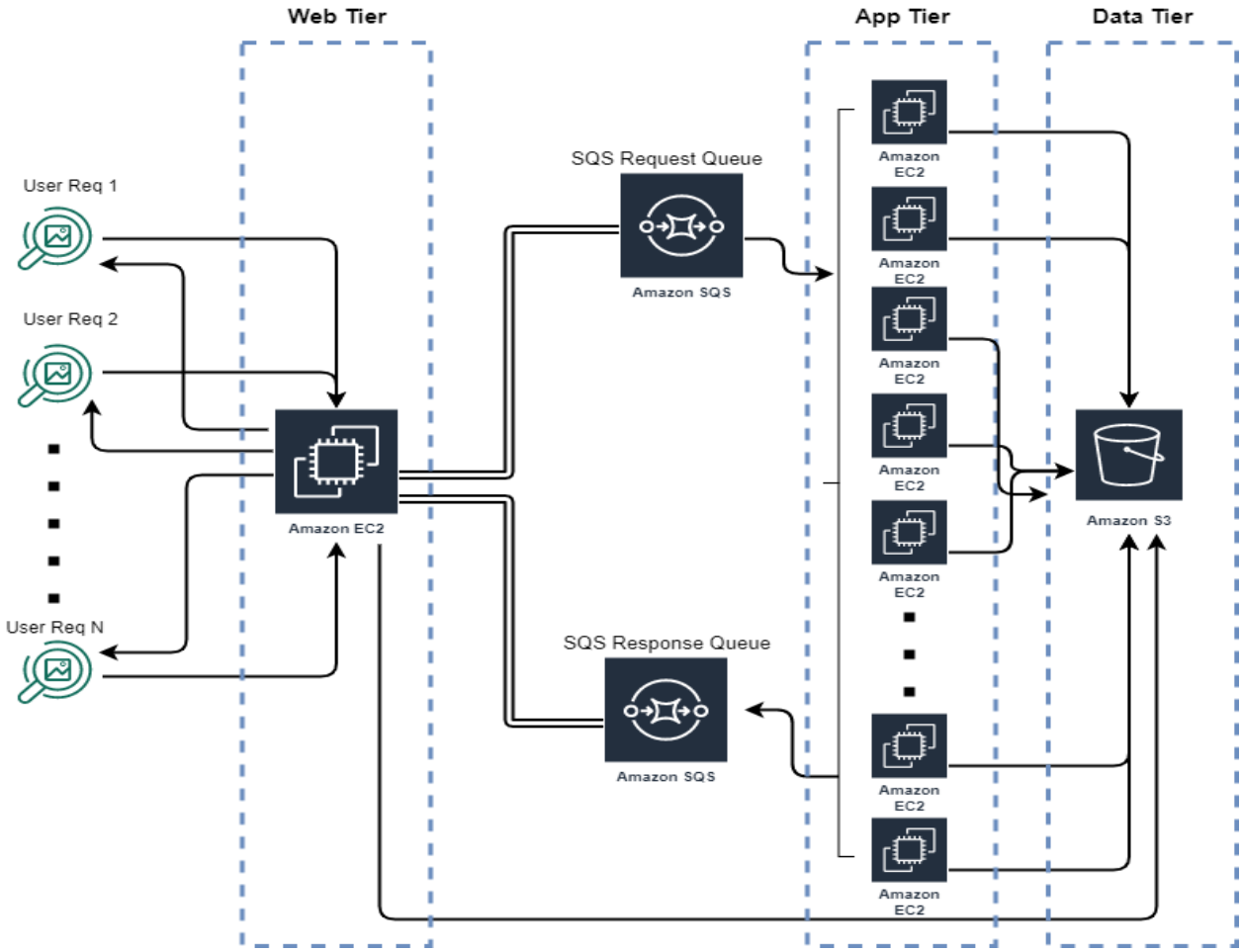


Figure 1.1 Architecture

- **AWS Services**
  - AWS EC2
    - Using EC2 we created our virtual machine in the Amazon cloud for running our application. In our project, we created EC2 instances (Linux images with a deep learning environment pre-installed) for implementing our application at Web Tier and App Tier
  - AWS SQS
    - In our project, we are using standard SQS by maintaining a request queue and response queue, where in the request queue stores the requests from the Web Tier, and the response queue stores the requests from the App Tier. SQS is used to make Web Tier and App Tier loosely coupled.
  - AWS S3
    - We are using S3 in our project to store results from Web Tier and App Tier for persistency. In our project, we are storing the images received from the Web Tier through user requests and the output received from the App Tier after the classification of the image is completed in the S3 bucket in form of (key, value) object where the key is the image name and value is the predicted output.

- **Architecture Description**

As observed in figure 1.1 our architecture consists of 1 EC2 instance of Web Tier that takes multiple requests from a user in the system. These requests will be pipelined into the SQS request queue which will be fed into App Tier EC2 instances (1 min, 19 max) as well as these image requests will be stored in the S3 bucket for persistence. Web Tier also acts as a controller for scaling in and scaling out through a load balancing algorithm that decides to scale-out app tier EC2 instances on the basis of the demand in the SQS request queue. A maximum of 19 app tier EC2 instances can be scaled out due to limitations of AWS(free eligible) usage. App Tier instances run a deep learning algorithm to predict output labels for the user-requested images by taking the requests from the SQS request queue. The predicted output label from the App Tier is stored in the S3 bucket as a (key, value) object for persistence and the output is sent to the SQS response queue. We have designed a dictionary in the Web Tier so that the output from the SQS response queue can be quickly pushed to the dictionary and when the user requests for the results, the output gets deleted from the dictionary which improves the overall efficiency of the architecture and simplifies the design.

Initially, we have 1 Web Tier EC2 instance running continuously for accepting the user requests and then the Load Balancer starts the App Tier instances according to the demand. Scaling in is handled by App Tier instances themselves as they terminate when they are unable to find any new requests in the SQS request queue.

Our architecture is also capable of handling the fault tolerance on App Tier. Web Tier keeps the image URL in the SQS request queue from where the App Tier picks it up. Due to the visibility timeout feature of SQS other App instances are unable to read the image when one App instance is processing it. Visibility timeout has been set taking into consideration the maximum time taken by the deep learning algorithm to produce the result. So by any chance the current App Tier EC2 instance that is processing the image terminates, after the visibility timeout the message in the request queue of SQS will be visible to other App instances. If the current App instance is able to successfully produce the result the message will be deleted from the SQS. So, visibility timeout not only provides fault tolerance but also contention avoidance. At web tier if the number of requests increases beyond a certain limit we can create a new Web Tier instance but due to limitations of free tier AWS usage we have implemented only one continuously running Web Tier which provides response to each user requests. SQS also provides a feature called "Receive Message Wait Time" so that the number of API requests to the SQS can be limited. Receive Message Wait Time provides long polling so that SQS can wait for a message to show up before returning an empty response. Long Polling also known as anticipation in a system is good as if the overhead of spinning off a new request is significantly high it makes sense to wait a bit long before terminating an app instance in anticipation of a new request coming really soon.

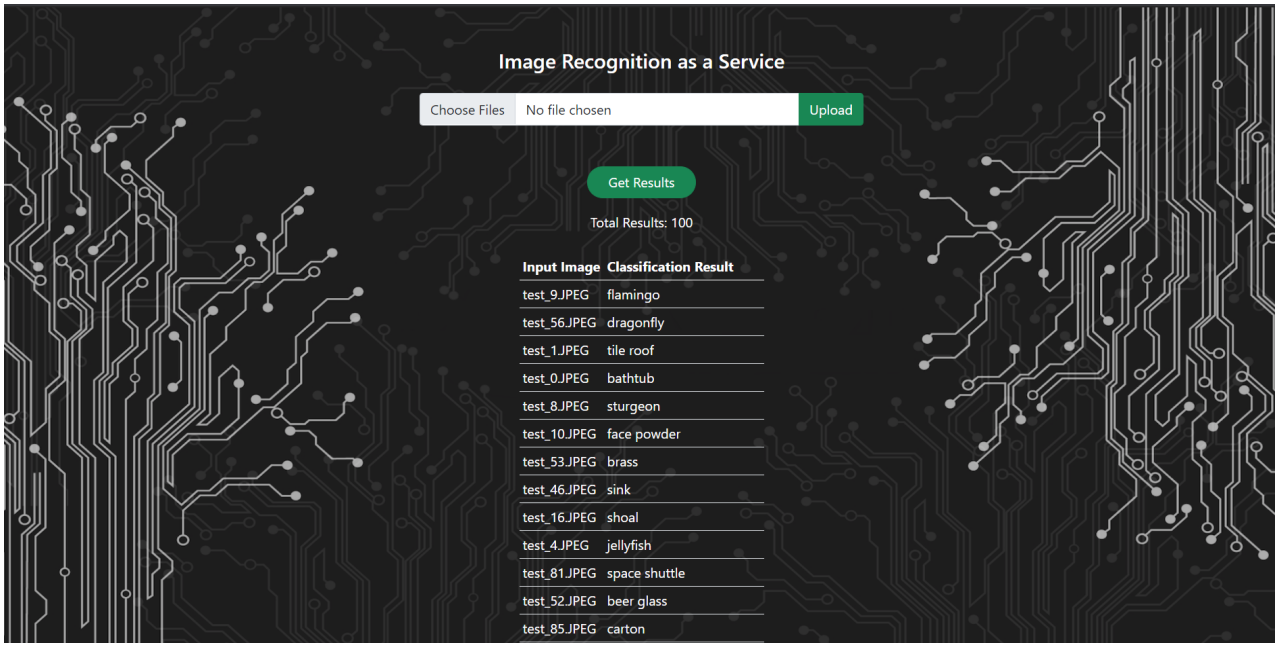
2.2 Autoscaling

Scaling out and scaling in takes place at Web Tier and App Tier respectively. For the purpose of scaling out we have implemented a greedy approach. For this approach we are using two inputs: number of running instances and number of visible messages in the SQS request queue. We have set the maximum limit for the number of app instances to be created is 19. So the maximum number of running instances at any point of time will be 20 (1 Web instance and 19 App instances). If the number of visible messages are less than 19 we will create app instances according to the number of visible messages present in the queue. If the number of messages at any point of time exceeds 19 our controller will not allow any more instances to start. This approach provides users the service as fast as possible. We have created a custom AMI which has the capability to start the load balancer using shell script every time it boots up.

Scaling in takes place at the App tier. After an app instance runs the deep learning algorithm and predicts the output, the output is stored in the S3 bucket and the result is sent to the SQS response queue. After that the app instance looks for any message present in the SQS request queue for 10 seconds. We have used the feature of Wait Time of SQS services so the app instance will wait for 10 seconds for any new request in the request queue of SQS and if it does not find any new request it will terminate itself.

3. Testing and Evaluation

- We followed these steps to test our application:
  - Tested our application on 100 images given by TA.
  - Ensured whether our application can take multiple images at once during upload time.
  - Checked the status of the input queue to verify the number of messages sent to the SQS queue.
  - Verified the number of uploaded input images in the S3 input bucket.
  - Tested the implementation of auto-scaling and load balancing by observing the number of running instances in the EC2 dashboard.
- Testing and Evaluation results are as follows:
  - Stored the result generated by classification on S3 for persistent storage.
  - Sent the result to the SQS response queue through which the server fetches the result to display it on the frontend.



4. Code

4.1 Files

- Web Tier
  - LoadBalancer
    - LoadBalancer.java
      - Module to handle scale-in and scale-out functionality according to a number of the input queue.
  - Server
    - app.js
      - API for uploading multiple images from the frontend and storing them in the S3 bucket and input the SQS queue.
      - API for showing results received from the response queue and deleting messages from the response queue.
- App Tier
  - Classifier
    - image_classification.py
      - Contains image classification code which is already provided. We have done some modifications in the code to get the output result in the desired format so that we can store it in S3 and SQS.
    - imagenet-labels.json
      - JSON file through which image classification code will provide labels to each image.
  - Controller
    - sqs_receive_app_tier.js
      - Receive message present in SQS input queue.
      - Fetch image URL from the received message.
      - Download image from S3 locally.
    - sqs_send_app_tier.js
      - Upload output from classification to S3 in text file format.
      - Send output in the form of a message to the SQS response queue.
    - terminate_instance.js
      - Fetch instance id of the current instance.
      - Terminate the instance.

4.2 Flow

- Web Tier
    - Take multiple images from the user (website) and upload them to input S3 Bucket and input SQS queue.
    - Show the classification results to the user, which internally receives messages from the SQS response queue as well as deleted the messages as read.
    - As the user uploads multiple images from the client website (using the upload button), the server stores all the images temporarily in the folder called “uploads” and then all images are then stored to input S3 Bucket and SQS input queue.
    - After the App-tier performs classification the results get stored into the response queue. So the Web-tier will fetch the results from the response queue and store it in a dictionary. Then the output is sent to the frontend using ejs templating language.

- App Tier
    - As soon as we get a request in the web tier, app instances start getting created. App instances are created from custom AMI created by us. This AMI contains all the code files mentioned in the code section. Moreover, we have set up a cron job that gets called when the app instance reboots. In the cron job, we have called up a bash script that runs all the codes in sequential order.
    - The instance gets created.
    - Bash file app_tier.sh(performs code execution) gets called up because cron job is set.
    - sqs_receive_app_tier.js called to receive messages from SQS and download images from s3.
    - image_classification.py called to perform image classification on the downloaded image.
    - sqs_send_app_tier.js called to send output response to SQS and upload text output to S3. Then, we check the number of messages present in SQS. After this there are 2 main decision this code takes which are as follows:
    - If message present in the queue:
        - Repeat the whole process again by calling the bash script of app_tier.sh. We do this process because we don’t want to terminate the instance if messages are still present in the queue.
    - If message not present in the queue:
        - Call bash script terminate_app_tier.sh which will invoke terminate_instance.js code and instance will be terminated.

4.3 Setup and Execution

- Setup of Web Tier Instance
    - Setup
        - Start web instance from the Ubuntu AMI
        - Send source code to web instance via scp
        - Create a JAR file of LoadBalancer source code and send it to Web Instance via scp
        - Run web_tier_installtion.sh file to install required dependencies
            - bash web_tier_installation.sh
    - Execution
        - Run the backend server code
            - pm2 start app.js
        - Run load balancer code
            - java -jar “jar file”
- App Tier
    - Setup
        - Create app instance from Ubuntu AMI
        - Send source code to app instance via scp
        - Run app_tier_installation.sh file
            - bash app_tier_installation.sh
        - Setup Cron job of app tier using app_tier.sh
        - Create an AMI after doing this process from the AWS ec2 dashboard. This AMI will be used by load balancer code for creating app instances.

5. Individual contributions

5.1 Aayush Hemalbhai Shah (1214024859)

Design

- After brainstorming through various architectures and designs we as a team came up with the final design of the project. This phase involved which Amazon Web Services to use and how to incorporate them in our project was the major task, how the app-tier and web-tier can be made loosely coupled, and so on.
- My task was to create an algorithm of the load balancer and how we will integrate it with the Web Tier itself. I decided how this Java code will run separately and communicate with other modules during the integration phase of the project. The main task of scaling out the EC2 instances on user's demand that makes the whole service elastic was developed by me.

Implementation

- My job was to implement the load balancer in the Web Tier and also the creation of EC2 instances using Java.
- I performed the following tasks in the Web Tier module of the Service:
    - Creating an EC2 instance for the Web Tier.
    - Implementing the load balancer algorithm.
    - Find the number of visible messages in the Simple Queue Service.
    - Find the number of running instances in the App Tier.
    - Evaluating and Using correct values for SQS parameters like Receive Message Wait Time for long polling to SQS

Testing

- After implementing the above 5 tasks in Java, my job was to conduct unit, integration and end-to-end testing of our Image Recognition Service.
    - **Unit Testing:** This part was particularly important as I developed the load balancer logic and so I ensured that the load balancer code invoked the correct number of EC2 instances on the App Tier for the maximum utilization of the resources provided.
    - **Integration Testing**: After uploading all the modules i.e EC2 Web instance creation and Load Balancer code on the GitHub, we as a team tested the whole Web Tier module to check if the Load Balancer works fine or not.
    - **End to End Testing**: As a whole team we were all involved with the end-to-end testing of our built service to ensure that the message passing through SQS from web tier to app tier worked perfectly as well as the responses from the App tier were correct or not.

5.2 Prerak Kishor Raja (1219453586)

Design

- The design phase included many things to ponder upon such as which services to use. What technologies to use in the backend and frontend, which codes will reside in the web tier and app tier, how all the codes will be invoked and how we will evaluate our project. We sat in a team to create the design of the whole architecture.
- My main contribution towards the design of this project was of the app tier. I created the architecture of the app tier. I decided what all steps will be required to implement the app tier, how internally the app tier will communicate with each module and how efficiently we can design the app tier so that we can individually test separate modules and integrate them in the later stage.

Implementation

- I handled the implementation part of the app tier. I implemented all the codes in NodeJS. I performed the following tasks in app-tier:
    - Create a bash script that can install dependencies required in the app tier and invoke all codes on startup.
    - Receive a message from the queue and delete it as soon as we receive it.
    - Download image from S3 bucket.
    - Perform classification and store output in the format that can be used in later parts.
    - Send a message to the response queue.
    - Upload output to S3 output bucket.
    - Calculate the number of messages present in the input SQS queue to decide whether to terminate the instance or not
    - Terminate instance after retrieving instance-id.

Testing

- There were 3 main testing phase during the testing of app tier which are as follows:
    - **Unit Testing**: I created each and every code module and tested them individually by keeping in different parameters and scenarios in mind.
    - **Integration Testing:** I created the GitHub repository and uploaded all the required modules of the app tier. I tested the flow of the app tier after setting up all the code modules on the app tier. I manually invoke bash script to test the flow of the app tier.
    - **End-to-End Testing:** We tested the whole flow of the web tier and app tier in the team. We tried different parameters and uploaded different numbers of images to verify the correctness of our code.

5.3 Varad Mahesh Bhogayata (1219972780)

Design

- I collaborated with my teammates in designing the architecture of the system. We brainstormed various architectures and then finally came up with the final architecture. I also worked on finalizing the design of the web-tier and configuration of Load Balancer code in the web-tier. Furthermore, I designed the structure of individual modules in a web-tier that involves the frontend and backend.
- Primarily, I contributed to designing, implementing, and integrating web-tier. I designed a flow where I mapped the inputs given by users (images) and the outputs (to store data in S3, SQS) by web-tier to app-tier instances. Furthermore, I have contributed to creating custom AMI of app-tier, in which we require some extra dependencies upon the given AMI by TAs.

Implementation

- Set up of required AWS Resource required for the project:
    - security groups in EC2, IAM users, key-value pair, a group in IAM, users in IAM.
- Implemented the RESTful Web Service (Backend) for web-tier in node.js. The backend mainly performs four operations:
    - Storing all uploaded input images in S3 Bucket which are received from a frontend (User interface).
    - Storing the uploaded input images into input SQS Queue
    - Receive the {image, result} message in output from response SQS Queue.
    - Delete the received message {image, result} from the response SQS Queue and store the result in server code.
- Developed User Interface using HTML, CSS, JavaScript.
    - To upload multiple images and see the output result.

Testing

- I tested frontend and backend extensively on my local machine as well as in web-tier. I conducted testing in three phases: unit testing, integration testing, and end-to-end testing.
    - **Unit Testing:** I tested each module independently (local server) to ensure the correctness of the functionality. For example, tested functions like uploadFile (upload to S3), sendMessage (send images to input SQS queue), and receiveMessage separately.
    - **Integration Testing:** I uploaded the source code to GitHub and tested each functionality on the web-tier to ensure the server is not getting errors while making requests to various APIs.
    - **End-to-End Testing:** We tested end-to-end flow by checking outputs in SQS queues and S3 Buckets as well as the scale-in and scale-out functionality. We tested the project with different parameters to check the correctness.