CSE 546 — Project Report

Cyriac Biju Narayamparambil

Aksa Elizabeth Sunny

Saran Prasad

1. Problem statement

Build an on-demand service with AWS for image processing that scales linearly with respect to the number of requests. Use IaaS services offered by AWS like EC2 for compute, and other technologies like S3 and SQS for persistent storage and message queueing.

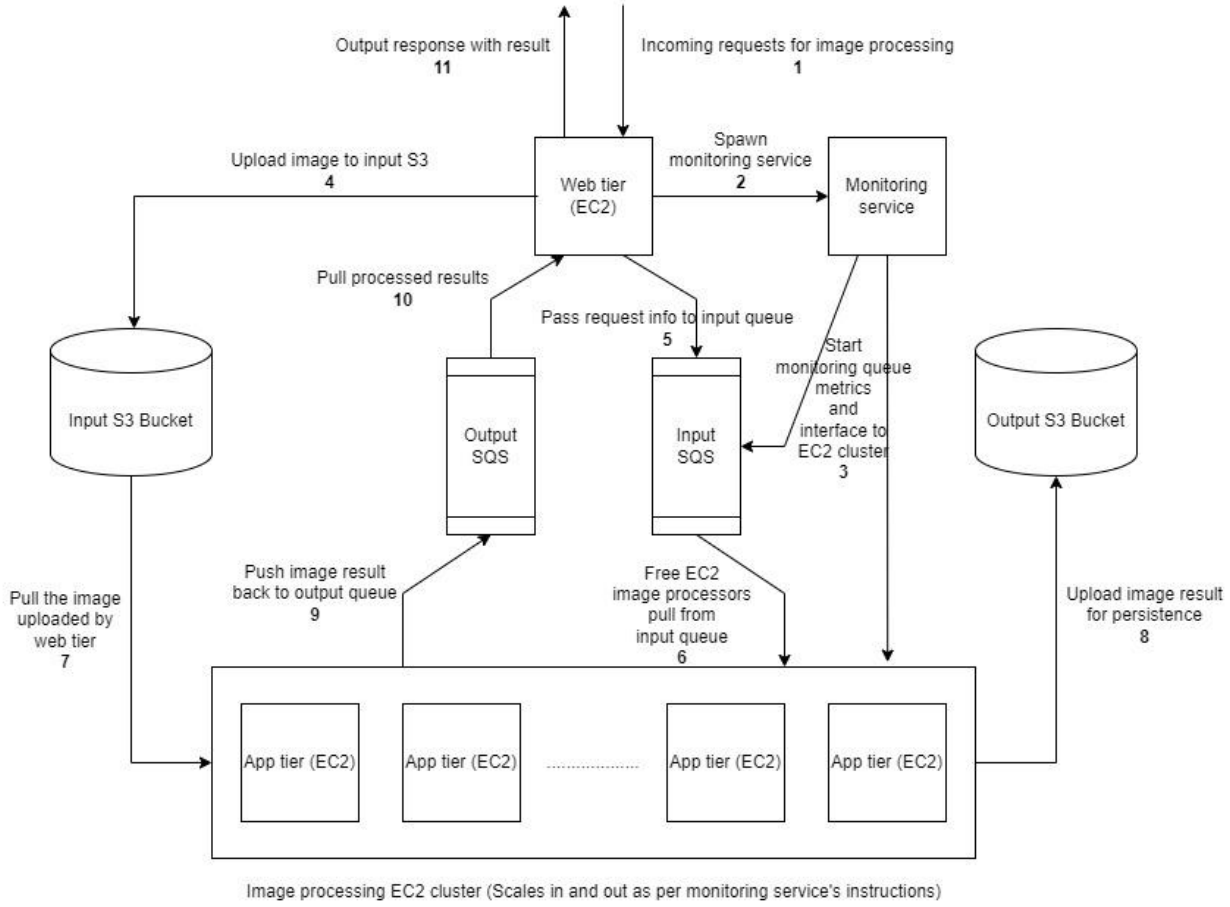
2. Design and implementation

The technical design of the architecture mentioned in section 2.1 has been divided evenly into 3 major categories.

1. **Web request and session handling (Web tier)**
    a. Developed using Node.js technologies like express, aws-sdk and multer
    b. Instantiates HTTP server for receiving web requests.
    c. Spawns monitoring service that runs on the same EC2 instance.
    d. Manages sessions for each request. This is to ensure the polled result from the output queue goes to the correct client.
    e. Deployed in an EC2 instance
2. **Queue monitoring and manual scaling (Monitoring service)**
    a. Developed using Python technologies like boto3.
    b. Monitors input queue metrics periodically to keep track of the number of pending requests.
    c. Uses custom logic for setting expected number of EC2 app-tier servers running in image cluster and leverages this expectation to frame a scaling algorithm.
    d. Calls EC2 APIs directly for scaling.
    e. Deployed in the same web-tier EC2 instance.
3. **EC2 cluster for image processing and persistence (EC2 cluster)**
    a. Developed using Python technologies like boto3 and PyTorch.
    b. Created new private AMIs and launch templates for ease of scaling.
    c. Manages periodic polling of input queue for new requests
    d. Handles new requests by downloading corresponding images from input S3 bucket and running pre-trained PyTorch image classification models.
    e. Uploads results to an output S3 bucket for persistence.

f. Pushes image results back to the output queue for final processing by web-tier.

2.1 Architecture



Each step of the architecture is explained below,

1. Incoming requests are received by the web tier server that specifies an image file in the payload.
2. Spawning of the monitoring service happens as soon as the web-tier server is up. This service periodically monitors the input queue metrics and ensures the required number of EC2 instances are up and running (scale in or out).
3. Starts up the interfaces that monitor the queue metrics and uses EC2 APIs to scale in or out the EC2 image processing cluster.
4. The image obtained from the request is first uploaded into the input S3 bucket.
5. The corresponding image name is passed onto the input queue so that the image processing cluster can fetch the image for classification.
6. The period polling done by EC2 servers in the image processing cluster fetches the new request.
7. With the obtained image name, it is pulled from the input S3 bucket.
8. After image processing (classification), the result is stored to an output S3 bucket for persistence.
9. The result is then pushed into the output queue so that the web tier can respond to the client.

10. Web tier pulls this result from the output queue and matches it with the client that made this request.
11. Web tier responds to the corresponding client with the classified result.

2.2 Autoscaling

Autoscaling is achieved using a formula that takes in the current number of images pending processing in the input queue (**count_of_images**) and the current number of running instances (**count_of_instances**) as follows:

count_of_instances = min((count_of_images) / 4, 12)

The monitoring service automatically creates new instances if current count_of_instances is not sufficient. It also caps the maximum number of instances to 12 to avoid overuse of resources. The monitoring service also takes care of intermediate bursts of requests by quickly turning on stopped instances rather than creating new instances from scratch.

Finally, when the count_of_images in the input queue is empty, the monitoring service automatically terminates all the stopped instances, freeing up resources and reducing wastage.

2.3 Member Tasks

- **Saran Prasad**
    - Developed tasks mentioned in **EC2 cluster for image processing and persistence (EC2 cluster)** in **Design and Implementation** section.
    - Set up AWS services like EC2, private EC2 images, launch templates.
    - Configured privileges and roles for users and resources like EC2 for accessing SQS and S3 services in AWS.
    - Developed image processing wrapper and server side functionalities like result persistence, writing back results to SQS, etc.
- **Cyriac Biju Narayamparambil**
    - Developed **Queue monitoring and manual scaling** monitoring service using Python libraries such as boto3.
    - Connected to SQS input queue and EC2 instances. Monitored input queue metrics periodically to keep track of the number of pending requests.
    - Developed the manual scaling logic to service the pending requests.
    - Found the number of EC2 instances and state of each to write custom logic for setting expected number of EC2 app-tier servers running in image cluster.
    - Setup a launch template for web tier instance
- **Aksa Elizabeth Sunny**
    - Developed the **Web-tier** service using Node.js and technologies such as express, multer, aws-sdk, winston (for logging).
    - Added implementation to spawn the initial app-tier instance as well as the queue monitoring and manual scaling service.

- Developed the session handling logic to map responses from output SQS queue to corresponding client requests.
- Implemented the POST /upload API to send images from requests to input S3 bucket, image names to input SQS queue and get the result from output SQS queue.
- Implemented polling to poll the output SQS queue for results and output queue clean-up after processing.
- Created launch template for web-tier.

3. Testing and evaluation

The code was tested both manually as well as using workload_generator provided.

3.1. Manual testing

Using Postman, local files were uploaded to the web-tier server for image processing. Since this is a sequential task, responses were fast (**under 4s**, even during cold starts). The responses were verified in terms of the classification provided.

3.2 Workload generator

Firstly, the sequential workload generator was tested, the following command was used,

**python3 workload_generator.py **

**--num_request 3 **

**--url 'http://<WebTierEC2PublicIPv4>:3000/upload' **

--image_folder "images/"

All the requests were responded with correct classifications.

Next, the multithreaded workload generator was tested,

**python3 multithreaded_workload_generator.py **

**--num_request 50 **

**--url 'http://<WebTierEC2PublicIPv4>:3000/upload' **

--image_folder "images/"

Up to 50 concurrent requests were fired. According to scaling policy (mentioned above), 8 instances were spawned (ideally 12, but the first instance processed some requests by the time manual scaling kicked in).

4. Code

- **EC2**
    - This directory contains three files (plus the json file containing classification labels)
        - **image_classification.py** - This is the same file provided in project description, just edited to contain a wrapper around the prediction method so that core handler (job.py) would be able to easily call it.
        - **job.py** - This is the main handler file that polls the input SQS queue for new requests. Upon receiving a new request, the corresponding file is searched for in the input S3 bucket after which the image is downloaded and provided to the image classification predict method. Finally, the response is stored in the output S3 bucket and pushed into output SQS.
        - **logger.py** - Program that helps with logging information, debug statements or errors that occur in the core handler.
- **web_controller**
    - This directory contains two components, the web server handling requests and maintaining sessions and queue monitoring/scaling service.
        - **controller.py** - This is the queue monitoring service which controls the manual scaling of EC2 app-tier instances. Policies used to dictate the scale in and out are coded up here. Process also periodically monitors input SQS queue for pending messages metric.
        - **server.js** - Starts up the initial app-tier instance (which stays running even if no requests are in queue). It also runs the http web server for handling requests, maintaining sessions for each request, handles pushing and polling to input/output SQS queues. Also, the queue monitoring (controller.py) is spawned by this process.

Steps to run the service:

1. After logging in to AWS, launch an instance using the web-tier-template (EC2 -> Instances -> Instances -> Launch instance from template). This spawns the web-tier EC2 instance. It already has the required code, python libraries and node modules installed. The server starts automatically on instance launch.
2. Run the **multithreaded_workload_generator.py** script with the URL http://<instance_public_ip>:3000/upload.
3. You should see the image results as responses.