CSE 546 --— Project Report

Amodini Pathak *acpatha1@asu.edu*

Molife Chaplain *mchapla1@asu.edu*

Rajkumar Elangovan *relango2@asu.edu*

1. Problem statement

In the first project, we will build an elastic application that can automatically scale out and in on demand and cost-effectively by using the IaaS cloud. Specifically, we will build this application using the IaaS resources from Amazon Web Services (AWS). AWS is the most widely used IaaS provider and offers a variety of compute, storage, and message services. Our application will offer a meaningful cloud service to users, and the technologies and techniques that we learn will be useful for us to build many others in future.

2. Design and implementation

The project required the use of AWS services to create a Software as a Service application that does image recognition for the client. The provisioned of SaaS is backed by the availability of infrastructure and for the developers, the access to a platform. With access to the platforms, including runtimes and libraries, we built software where the user has no control over the backend and the only interaction is provided through the web. The idea of scalability meant that the project had to ensure that the components of the system were decoupled and ensured the availability of an elastic service that does not waste resources. For our service, ensuring decoupling, and scalability meant that the web-tier, which focused on client interaction; the app-tier, which focused on running services that were necessary for scaling and functioning the system; and a controller had to be individual components for functionality and scalability. We used an AWS EC2 instance for the web app and the instances are associated with IAM roles.

The language used to implement the project was python, with a Flask server. We chose python as the language of choice because, besides that we were familiar with it, it provides a large number of robust libraries and modules; it also has very extensive, public, easy to understand documentation available. Given that the project required a high throughput and threading, or multiprocessing had to be done, we used Gunicorn as an augmenting service to our Flask server. This was done because, while python enables threads, it suffers from the Global Interpreter Lock problem. Besides the issue with python, Flask, as a standalone, cannot be used in production or deployment and in order to achieve parallel processing and effective multi-threading, we used Gunicorn for this purpose.

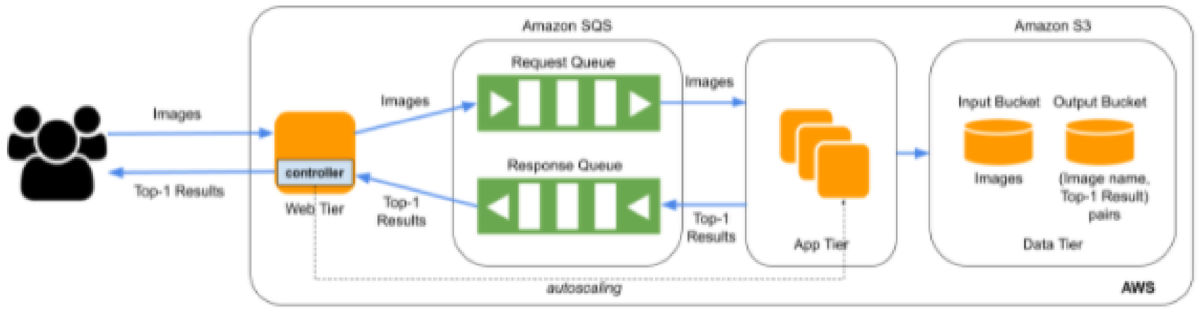
With issues like concurrency and parallelism, including time and memory utilization, we implemented the system using threads and multiprocessing concurrencies as supported by Gunicorn. We also realized that in terms of I/O, sending the message as a message and not the actual image reduced the delay and increased throughput as this meant that the system would simply collect the image from an already stored file in the bucket rather than using more space by increasing the memory needed by the SQS queues and the pipeline between the tiers. Theoretically this approach improves the performance of the system overall.

The implementation also uses threads as a strategy to achieve multiprocessing and as a way of reducing the possibility of bottlenecks in and between the tiers. The importance of multiprocessing and reduction of bottlenecks was evident in the time allotted for the tasks to complete, but also in the fact that the testing heavily focused on ensuring that the results moved through the tiers quickly, including checking the speed of deposit of requests and files.

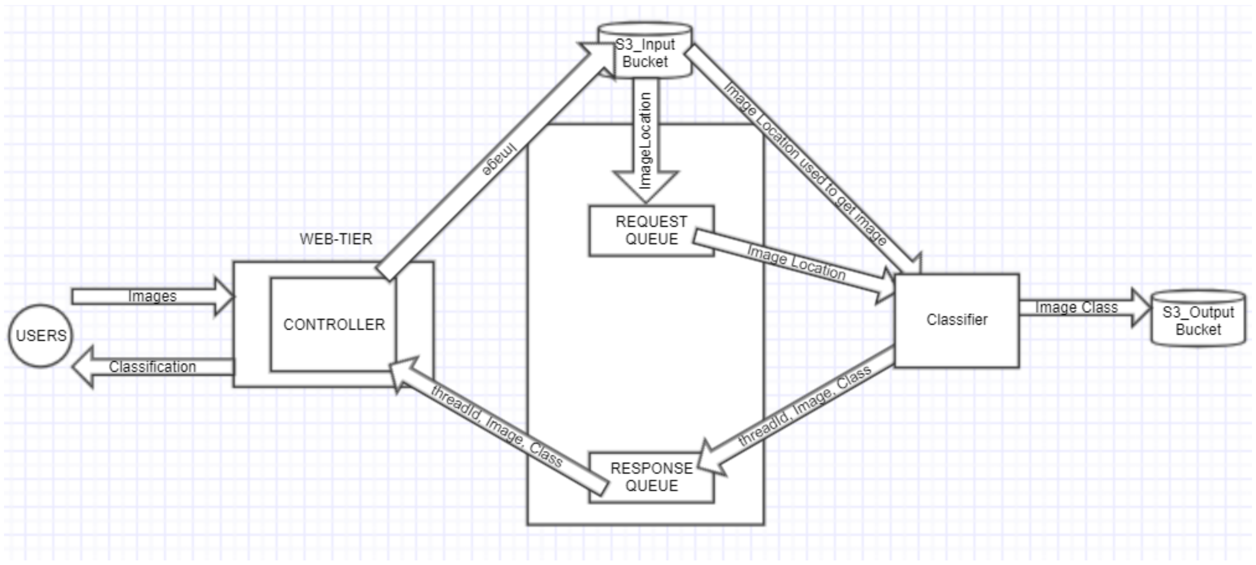
2.1 Architecture

The main point of interaction for the application services are the queues (input, and output) which track instances of the images and responses provided by the web-tier and app-tier respectively. The user can only interact with the web-tier as this is where they send the requests, and it also sends the user the output/response to their request. The elements are decoupled such that they mimic the MVC architecture with the view being the web-tier, the controller being the Controller and the app tier which contains services for manipulation and interaction of the view and logic by the controller being the controller. The image provided below shows the desired architecture. The most important feature being that the different components of the system are loosely coupled but maintain a constant flow of data between them at every instance of the system lifespan.

The server is always listening on a specific port for any user requests which will be sent using the post method of HTML. Once a request is received from any client, the server triggers a few processes. First, using a unique before function feature in flask, the server instantiates a listener/reader thread which helps return thread outputs to their individual threads. Second, it starts the controller and sets its Boolean value to True. While this value is True, the controller will always listen for requests and check the state of the built on SQS architecture. This means that the controller will have access to methods such as getNumberOfMessages in queue which returns an approximate number corresponding to the queue size. On being set to True, the controller triggers the system to start an instance, which will remain on and working as long as the server is up and running.



The diagram below shows our architecture and how the system was implemented in our program:



2.2 Autoscaling

The idea of autoscaling is highly dependent on triggers, and in our system, we used the controller to ensure that the relevant triggers would lead to instances being opened or closed. The controller (which is a Boolean), receives a trigger from the web-tier which triggers it to listen to the input queue, this changes the boolean value to true and maintains the listening until the controller value changes to false. While listening to the queue, the controller keeps track of the instances that are running, sleeping, and or pending. This ensures that the controller retains and ensures control, monitoring and tracking of instances. While there are messages in the input queue, the controller checks how many instances are active, and how many are needed to ensure that the pending workload is processed quickly.

Given the upper threshold of 19 instances, the app tier has access to the 19 instances and the controller ensures that the total number of instances never surpasses 20 (including the web-tier instance). When the instances in use are less than 19, and there are messages in the output queue, the controller triggers the creation of another instance to process the requests.

When the instances reach the threshold, the controller sends a prompt explaining that no more instances can be opened. Once the messages are either less than the number of instances that are open, or the queue is empty, the controller automatically triggers the termination of instances that are no longer in use down to the available requests or to no instance being active. This provides an auto scaling feature to our software, the web tier also uniquely is suited for self scaling as the number of requests directly determines the number of threads it creates and sends across the system. It is also important to note that the threads are terminated once they return the classification of the image/request that instantiated them.

3. Testing and evaluation

Component testing - The services and tiers were individually tested before integration to determine that basic functionality of components was achieved. The web tier was able to deposit images into the S3 bucket and insert the message into the SQS queue, with the ability to wait and anticipate the return of a response. The controller was able to scale as necessary based on server and queue capacity as intended.

System testing

4. Code

We have different code modules to handle the server-side functions and the application functions. Our application uses a flask server. The code is mostly written in Python and the flask server is written in python. There is a substantial amount of scripting language that handles the ssh credentials and other elements. The following are the files and their functions in the program. The program runs a substantial number of modules whose functions are defined below in high level detail. The main purpose of having separate modules was to avoid unscalable implementation using tightly coupled components and hence we separated the components and connected them using modules and threads.

Web-tier.py – main flask server. It handles the uploading of the image to the S3 bucket and the retrieval of the classifications and images from the sqs output queue for return to the user. This also handles the server side and runs/starts the server. It also initiates threads for the requests and invokes the controller to start listening for requests from clients.

controller.py – controls most of the workings of the system. This module is invoked from the web-tier where it proceeds to create an instance of a machine. The controller also listens to the queue to ensure that autoscaling occurs in the direction it is needed. This ensures that resources are not wasted.

sqsService.py – takes care of the interaction between the other services and the sqs queues. This involves providing a method for inserting and retrieving messages into and from the input and output queues, respectively.

S3Service.py – provides the method for storing the image into the input and output buckets, but also the method to get the image from the bucket using the url provided by the input queue message and getting the message classification from the classifier.

Ec2Service.py – connects to a given ec2 instance and provides the prediction method which allows the image to be passed to the classifier for a result to be obtained.

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.