

CSE 546 - Project Report

Derbique, Austin
aderbiqu@asu.edu

Pappas, Alexander
apappas5@asu.edu

Ruter, Cole
cruter@asu.edu

March 13, 2021

I. PROBLEM STATEMENT

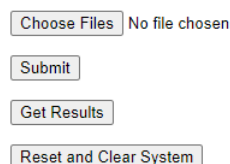
Application services that receive many requests from a single or multiple users should be able to efficiently and effectively process their requests in a meaningful timely manner. With a single server and limited resources these requests can take a long time to process and return the results back to the user. The utilization of IaaS resources from Amazon Web Services, such as EC2 instances, S3 database, and Simple Queue Service enables us to scale out our application service to handle many requests concurrently and return results as quickly as possible without missing any requests.

II. DESIGN AND IMPLEMENTATION

The initial task was to define requirements. While vague, the requirements specified to use a provided Amazon EC2 Machine Image (AMI) support autoscaling, and utilize Infrastructure as a Service (IaaS). It is implied that for the project, we utilize Amazon Web Services as the cloud provider for compute, storage, and networking [6]. With scalability being the primary consideration, the focus was to design this system in such a way that all systems are decoupled. This means separating out the workload into a web tier, app tier, and mechanisms for data storage and delivery. Initially, we tried taking the python classification module provided in the AMI and placing the runtime into a Lambda layer. Unfortunately, this exceeded size requirements. It was decided to use the EC2 instance after all. Using EC2 as the app tier, it made sense to build out from there, using serverless applications where possible, and relying on infrastructure as Little as possible. The reason for this is because a lot of things can go wrong, and infrastructure is sometimes difficult to manage at scale, especially when trying to track down a problem.

The app tier has an Identity Access Management (IAM) role associated with each of the instances allowing the instance to make API calls to other AWS services. These allow the app tier to retrieve and delete Simple Queue Service (SQS) messages, retrieve objects from Simple Storage Service (S3), place result items in DynamoDB, and finally put objects + tags into S3. For actual control, *classifier/main.py* handles running jobs of retrieving images, processing them, and reporting results. Several config options allow for shutdown of the EC2 on end and run indefinitely (always check). A single EC2 instance can process 100 images in about 54 seconds. A multithreaded variant was attempted, but failed to yield a higher performance. Perhaps with some tweaking of pulling more SQS messages at a time, a better performance can be achieved.

File Upload



The screenshot shows a web interface for file upload. At the top, there is a button labeled 'Choose Files' followed by the text 'No file chosen'. Below this is a 'Submit' button. Further down is a 'Get Results' button. At the bottom is a 'Reset and Clear System' button.

Figure 1: Web Tier Frontend

The web tier has an IAM role that grants access to put objects into S3. Instances in the web tier also have the ability to retrieve results from DynamoDB for returning classifications.

Finally, the web tier (ran out of time) has permissions to clear the results out of S3, SQS, and DynamoDB. This in turn resets the system back to a vacant state.

Between the web tier and app tier exist S3 and SQS. Images are uploaded from the web tier via an application route in Flask and put object calls are made placing the images into an S3 bucket. This triggers an SQS message to be created directly from S3. Cloudwatch is configured with various alarms to "Wake" the EC2 instances in the app tier, and scale automatically. This will be covered in more detail in a later section.

Finally, once results are classified from the fleet of app tier instances, results are placed into three locations. The requirements of this assignment forced us to write results to S3. It makes natural sense to update the tags of the objects being classified. As such, each image once classified gets their classification tagged directly to the object itself. S3, however, is an object store, not a database. It is extremely slow and expensive for queries. To increase performance, classification results are written to a serverless database table using DynamoDB. Results can easily be queried by the web tier for superior performance. Finally, out of concern of needing an output bucket for the project requirements, a result label is placed into an output S3 bucket.

In terms of deployment, Infrastructure as Code (IaaS) would have been ideal, but unfortunately time did not allow for this. There was also a plan to utilize Ansible for deployment [1]. Infrastructure was configured using the AWS console instead for a quick means of proof of concept. In the future if this were to be a production graded system, all deployments would be handled using serverless framework, cloudformation, terraform, and Ansible [3][4].

Architecture As seen below, there are many different pieces to this application. In Figure 3, Draw.io is used to illustrate the cloud architecture of our image classification workload [2]. Table II. describes what each of the components are and what they are responsible for.

Resource	Purpose
Amazon Certificate Manager	Provides Amazon publicly trusted SSL certificate cse546-project1.derbique.org certificate. Enables TLS1.2 Encryption on web traffic.
Elastic Load Balancing	Acts as a load balancer for HTTP and HTTPS traffic. All HTTP requests are redirected to HTTPS.
LB Target Group	This holds the web tier EC2 instances and are placed into operation once the /healthcheck test passes.
Web Tier Launch Configuration	Specifies how the web tier EC2 instance should be started. Disk, network, security groups, etc.
Web Tier Autoscaling Group	Controls how many web tier EC2 instances are in operation. Uses the Web tier Launch configuration.
S3 Image Bucket SQS Queue	Location of image uploads. S3 object metadata stored in the messages, created from S3 directly using Simple Notification Service.
Cloudwatch Alarms	Used to trigger alarms when number of messages in the queue reaches a threshold. Alarms are set at 100,500, 1000, and 20. 20 means that most of the images have been processed and scaling in may take place.
Cloudwatch Events	Used to monitor CPU Utilization of Web Tier. Upon reaching 50 percent, more instances will be created.
DynamoDB Table	Classifications table holds image name, classification, and s3 object location
App Tier Launch Configuration	Specifies how an app tier instance should launch.
App Tier Autoscaling Group	Controls how many app tier instances exist. Uses the App tier Launch configuration.
S3 Result Bucket	Image Classification labels are placed here
S3 VPC Endpoint	Used for improving performance inside the VPC

Table 1: Architectural Components

Image Classification using IaaS with Autoscaling

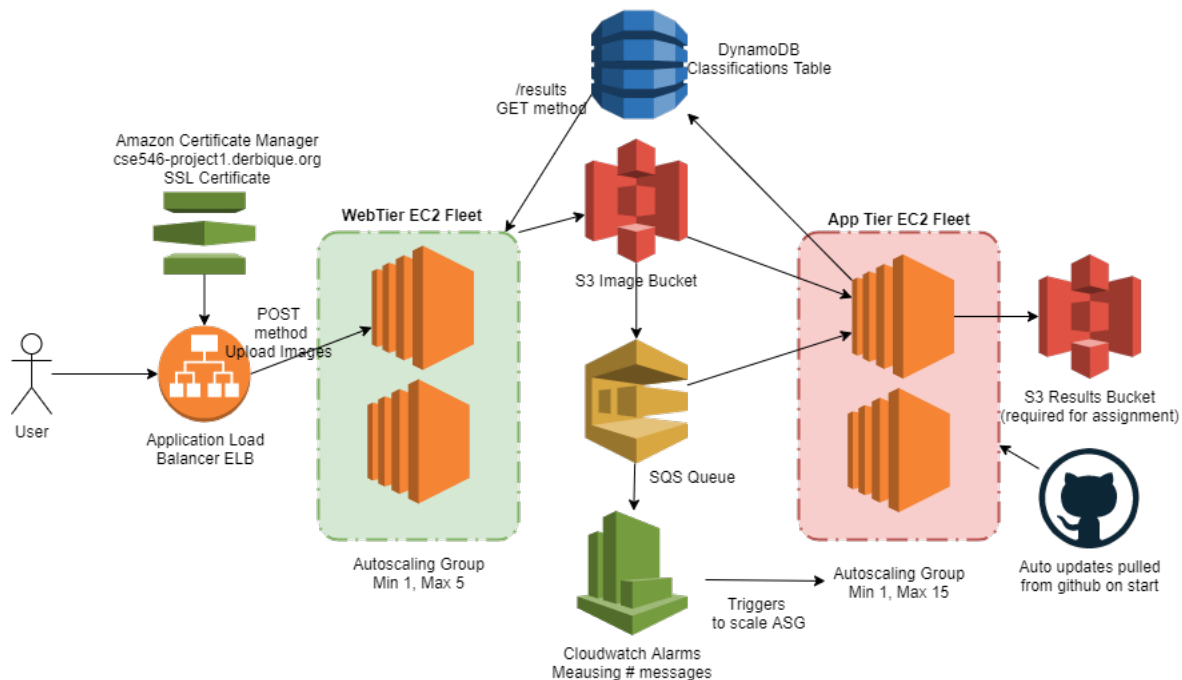


Figure 2: Cloud Architecture Diagram

Autoscaling Autoscaling is achieved using event based triggers. The reason for this is to improve reliability of the system, and ensure that an impaired EC2 instance not affect scaling abilities. For this reason, the EC2 instances are managed using Amazon Auto Scaling Groups (ASGs)

The web tier fleet of EC2 instances is configured to have at a minimum, one instance running all the time. This is unless we turn off the system, in which case desired capacity is set to zero. As load from users is placed on the web server, CPU utilization will increase. The autoscaling group is set to trigger a scaling action when CPU utilization of an instance reaches 50% or more. As additional load is placed, the ASG will continue scaling out until the maximum of five instances is reached. When traffic from users goes down, the CPU utilization will drop as well, triggering a scale in of EC2 instances effectively lowering the number of instances in operation. The web tier is also self healing, where if an EC2 instance fails a health check, the instance will be taken out of operation and replaced by the auto scaling group with a health instance. The failing ec2 instance will then be terminated. Additionally, any updates made to the web server via Github are automatically pulled on start, meaning to update an instance, all you need to do is terminate the instance and a new instance will automatically replace it (because of the ASG) with the updated source code.

The app tier fleet of EC2 instances operates similarly to the web tier in terms of using an auto scaling group, auto updating, and auto healing. However, instead of scaling actions being managed by a Cloudwatch metric of instance CPU utilization, scaling activities are managed by Cloudwatch alarms. Several Cloudwatch alarms are configured, each to raise alarm at set intervals of messages available in the SQS queue. At 100 sQS messages in the queue, autoscaling activities increase the fleet by two capacity units. At 500 messages in the queue, five capacity units are added. Finally, at 1000 messages available in the queue, maximum scaling is enforced by adding 15 capacity units to the ASG. When these scaling events occur, EC2 instances populate the auto scaling group, starting the image classification application on boot. All instances remain in operation until the SQS queue drops to 20 available messages or less. At this point, scaling is set to one instance and all other app tier instance in the ASG are scheduled for termination. When the queue begins to rise again from new image uploads, the process will repeat, scaling out and in as needed. To prevent rapid scale in and out, cool downs are defined such that scaling actions can only occur after a set amount of time has elapsed. A bit of tuning was required to get the desired scaling behavior for this project. In a production environment, time to cooldown would be increased, as running an instance for several minutes is a lot cheaper than terminating and instantiating a new instance every few minutes.

III. TESTING AND EVALUATION

Testing was performed at first by uploading small batches of 100 at a time. It takes about 54 seconds for a single EC2 App Tier instance to process 100 images. Knowing it takes about a minute for an app tier instance to be launched and operational, the tradeoff for scaling intervals was determined. With intervals determined, several more tests were performed, slowly ramping up the image count to 2000+ images. Below is a table of the measured performance of the system. Note that these metrics include autoscaling.

Seconds Surpassed	Messages in Queue	Images/min Processed
0	2136	N/A
30	1998	276
60	1741	514
90	1305	872
120	1000	610
150	812	376
180	509	606
210	275	468

Table 2: Performance Evaluation

As seen, the images processed per minute actually started dropping, despite scaling efforts. This is likely due to API call limits to SQS retrieve message. One possible fix for this would be to obtain multiple messages at a time, such as 10. This would reduce API calls by 90% and effectively allow the app tier to scale without the bottleneck of SQS API service limits.

Image Classification Results

Image	Classification
0_cat.png	shower cap
1_ship.png	ocean liner
2_ship.png	polar bear
3_airplane.png	fireboat
4_frog.png	hyena
5_frog.png	cliff dwelling
6_automobile.png	shower cap
7_frog.png	water snake
8_cat.png	cliff dwelling
9_automobile.png	polar bear

Figure 3: Classification Results

IV. CODE

All code with relevant READMEs can be found at [aderbique/cse546-project1](https://github.com/aderbique/cse546-project1) on Github [5]. The repository is broken down into two main sections. classifier and server. The classifier directory contains all code for app tier. The server directory contains all code for webtier. The application is largely written in python, with some shell scripting used. Finally, the report itself is written in L^AT_EX which some may regard as a form of coding.

The following lists each code file individually and what it does: app.py - Main Flask application file. This file handles starting the Flask server, uploading image files to S3, and getting the results stored in DynamoDB.

get-classifications.py - This Python file has 2 functions regarding the outputted results: get-classifications-dict() which returns a dictionary of images and their classifications from DynamoDB and natsort-dict() which uses the natsort Python library to organize the results

into ascending order of file names.

sqs-receiver.py - This script connects to our SQS queue on AWS, grabs the first available message off the queue, and then outputs the body of the queue message which is the link to an image in our S3 input bucket. The script will also delete the message in the queue after the link to S3 is retrieved.

tag-receiver.py - This Python script simply connects to our S3 input bucket and prints out every object listed in the bucket.

index.html - This HTML file is used to display the front-end web page for the user so they can upload images and then click to see the results

results.html - This web page is displayed after the user clicks the "Get Results" button on the main and it displays the file name they uploaded with the result as a Python dictionary.

lambda-function.py - This Python file has 1 function called lambda-handler() that takes a queue message from SQS and starts an EC2 instance to process an image. It also returns the ID of the new EC2 instance.

How to install and run programs: First, the user must clone the repository available on GitHub. To log into the Web-tier EC2 instance, the user's public IP address must be white-listed in the security group, and only then will they be able to ssh into the instance. To run the code, run the following commands: "tmux", "cd /cse546-project1/server", and "python3 app.py". If in the event that the webserver is already running and needs to be stopped, you can attach yourself to the tmux process to kill the application with "tmux -a". The user will now be able to access the frontend at the web address "localhost:8080". There they can upload the images for processing. After the images are uploaded, the user can go to S3 on AWS and find the bucket "cse546-project1" where all their inputted images would be stored. Then the user is able to go to another bucket "cse546-project1-output" or check EC2 to see if all the images are finished running. Finally, the user can now click "Get Results" on the web server to see their results from the image processing.

INDIVIDUAL CONTRIBUTIONS

A. *Cole Ruter*

My main responsibility for this group project was setting up the front end application that would be ultimately used by the user. There were many different design decisions to go with for the front end. My initial idea was to go with PHP since I have done an upload page before with PHP but Austin informed me about the Boto3 package that python uses that we can utilize to easily send and receive data from our Amazon Web Service components. This tip from Austin pushed me into deciding to go with a flask application that way we can use python for the scripting. Once Austin set up the servers with permissions I connected to the server that would host be our front end and controller code for our application. I was able to install flask onto the server that way we could run our application code. I started the coding by first creating the HTML file that the flask app would load to show to the user. This HTML file, named index.HTML, was a basic form with an upload button that would allow the user to upload many files to the application using a POST request. I then started writing the code for the flask application, named app.py. I made the default route of the application point to the index.HTML file so it would load it to the user when landing on the web page. I created an upload_files() method that would be executed on the POST request from the HTML file that would extract all the files that the user uploaded. I made it so it will accept any file type because I was running into issues with trying to limit it to just ".PNG", ".JPG", and ".JPEG". For each file that it retrieved from the upload I checked to make sure that the file names were not blank and make sure the names were secure. Then one by one it would call a upload_file_to_s3() method that takes in a file and bucket name. This method would utilize that boto3 library that I mentioned earlier and upload the file to the specified bucket name in S3. After the upload is complete it would then retrieve the URL of the S3 uploaded file and create a Simple Queue Service (SQS) message with the

body containing the URL which would later be used by the controller. After the upload processing was complete I had it redirect to the same page the user was already on that way a different web page did not need to be created. In order to run and test the application I had to open the port number 8080 on the EC2 instance to allow users to be able to connect to the web page. I made the port open to everyone that way whoever ended up testing our application would be able to connect to it if necessary. Once this was all set up I was able to run the flask application and test my code. I first started with a simple test of an image upload to see if it accepted images and to see it upload to S3 first. I was able to upload to S3 and in my code once it was uploaded to S3, I retrieved the S3 object and printed the object's URL to console. Once I saw it was printing out the correct URL I then introduce creating the SQS message body with that URL. For this next test I uploaded two images to see the if they were able to be uploaded to S3 and produce a SQS message. In this test case the S3 input bucket successfully had the two images with their original file name and in SQS I had to poll for the messages and it displayed two messages available both having a body with a URL that linked to the S3 items. After this was complete the code was now ready for Austin's controller code. Once Austin started testing his code he ran into an error with uploading over 100 images. I then made coding changes to have no limit on the incoming post request sent by the user, that way everything is retrieved. It was initially limiting the user to about 400 MB of data for upload. I also wrote code that would log the files being uploaded real time. This logging allowed me to see how long the uploading of files would take and it was able to upload 1000 files to S3 in under 2 minutes. After I saw we had no upload issues anymore I removed the logging code since it was no longer needed with everything running smoothly.

B. Alexander Pappas

My responsibility for this project was to implement our team's SQS queue retrieving functionality as well as to implement the part of the frontend application that displays the results of the image processing to the user. Other small contributions I made involved setting up everyone's IAM roles on AWS in order for them to be able to access all of the AWS services we were going to use for the project. Some of this setting up also included help from Austin who knew more about AWS users, security groups, and inbound rules, etc. To implement our SQS (Simplified Queue Service) queue, I first created a basic FIFO queue on AWS that we could connect to with our application in order to send and receive messages with the image data needed to process them and store them in S3. I also created a Python script called "sqs-receiver.py" which connects to the queue via the queue's URL and retrieves the first available message in the queue. The message contents are stored in a variable called "response" and then the script prints out the body of that message which contains the link to the image stored in our S3 input bucket. The purpose of this script is to provide our EC2 instances with the location of each inputted image in S3 so that it can grab the image file and start to process it. We needed to use a queue for this because multiple images would be uploaded at a time so we could create more instances as needed and they could simply take the next image to process in the queue as needed. The "sqs-receiver.py" script also handles deleting the message in the queue after it is received. The second portion of the project that I handled was adding a button in our Flask frontend application to let the user see the results of the image processing model that were outputted into S3. While we do store our image filenames and classification results as a key-value pairs in an S3 bucket labeled "cse546-project1-output", our group figured that querying S3 to retrieve all of these results took longer than it should. To solve this, we implemented our instances to output the key value pairs to another database provided by AWS called DynamoDB. Querying the key-value pairs from DynamoDB was faster and a little simpler to set up and connect to. We created another Python script file called "get-classifications.py" that scanned all entries in the DynamoDB table, then sorted and returned them. My job in this process was to enable this script to be called when the user presses a button in the Flask application. The button is labeled as "Get Results", and once clicked, will take the user to a new page in the application titled "Results" and displays each file name and its corresponding classification in plaintext. Designing the parts of the cloud application that I worked on were not too complicated and just took some time to think about. I mostly just discussed with the other group members about the best strategy to go about implementing my parts. Testing was what took the most time out of this project for me. I knew what I needed to do but

getting it to work correctly was much more challenging. It was especially challenging for me because I had never worked with any AWS cloud services before, so I had to read a lot of documentation about how we could use the services to implement the project. Now that I have actual hands-on experience with AWS, it is much easier to write programs that interact with their services. One of the initial challenges that I took me a while to figure out was how to “ssh” into a running EC2 instance. From the beginning of the project, I was lost on how we would set up the instances to run our code. Essentially a lot of the time I spent on this project was figuring out what to do and how to do them instead of actually doing them. Once I learned how to actually do them, it became much easier for me. The last thing that I tried to implement was a button on the results page (“results.html”) of the frontend that would clear the list of results returned from database. I tried getting the web page to empty the Python dictionary which was holding the output. However, I could not figure out how to set up the button that, when clicked, would clear the entire dictionary without needing to refresh the page. I gave up on that small feature because it was not in the requirements, so it was not needed for the project. Finally, one of the last things I contributed to this project was my AWS account. I already had my account set up when our team got together, so we decided to use my account to host all of the instances, S3 buckets, the SQS queue, and the DynamoDB table which means that I would be charged for any extra fees for passing the free tier limit which we accidentally did for too many S3 requests.

REFERENCES

- [1] Red Hat Ansible. URL: <https://www.ansible.com/>.
- [2] App Diagrams. URL: <https://app.diagrams.net/>.
- [3] Serverless Application Framework. URL: <https://www.serverless.com/>.
- [4] Terraform by Hashicorp. URL: <https://www.terraform.io/>.
- [5] Github Repository. *aderbique/cse546-project1*. URL: <https://github.com/aderbique/cse546-project1>.
- [6] Amazon Web Services. URL: <https://aws.amazon.com/>.