CSE 546 — Project Report

Gaurav Kulkarni, Shreyas Kirtane, Parth Shah

1. Problem statement

A scalable computing system is the one that responds to the changes in user demand and rapidly alters its resources to achieve optimal utilization. The scalability of a particular system is its defining characteristic. In the ubiquitously digitized market, scalability can make or break a brand.

Rapid and automatic scaling is also the overarching theme of modern cloud computing. Several major brands have fully shifted to cloud-based infrastructure to make their systems rapidly scalable on a granular scale. Amazon Web services is a leading on-demand cloud computing provider which elegantly handles the scaling problems with AWS auto scaling.

This project utilizes the Infrastructure as a Service cloud from the AWS tech stack to build an elastic application for image recognition. This application scales out as the user demand rises and scales in accordingly as the demand recedes.

We aim to utilize the diverse cloud services provided by the AWS for this project. We will utilize the AWS Elastic Compute Cloud services to deploy the application over the cloud, the AWS Simple Queue Service to upload and download the API requests and responses and AWS Simple Storage Service for persisting the input images and responses from the Machine learning model. Finally, we will utilize the AWS CloudWatch service to continuously monitor the network flow and automatically scale out and scale in the number of EC2 instances to concurrently handle multiple requests in optimal time.

2. Design and implementation

2.1 Architecture

The system is designed as two layers of web servers, the web tier, and the app tier. Both the servers are deployed using separate EC2 instances. Further we utilize two SQS queues to store classification requests to the app-tier and the responses from the app tier respectively. We also integrate two S3 buckets to store the input images and the classification results respectively.

Docker

Docker was used to simplify deployment of the components of the project, i.e Web tier and App Tier. The major benefit of docker is that we were able to use the same environment for testing and running our code. No extra configuration was required. Other than this, docker also provides isolation and is also adaptable to a container based deployment service rather than EC2.

Web Tier

The web tier consists of a Python Flask server deployed on a single EC2 instance. It receives the API request from the user along with the image that is to be classified. The web tier uploads this image to the input S3 bucket. Here the web server uploads an image file with its name as the unique id. It then enqueues the classification request to the app tier through the requests SQS queue. The web server builds the request using the filename and a unique message Id. Then the web server polls the responses SQS queue for the classification results. If successfully received it deletes the message from the queue and forwards the response back to the user, otherwise it throws an error.

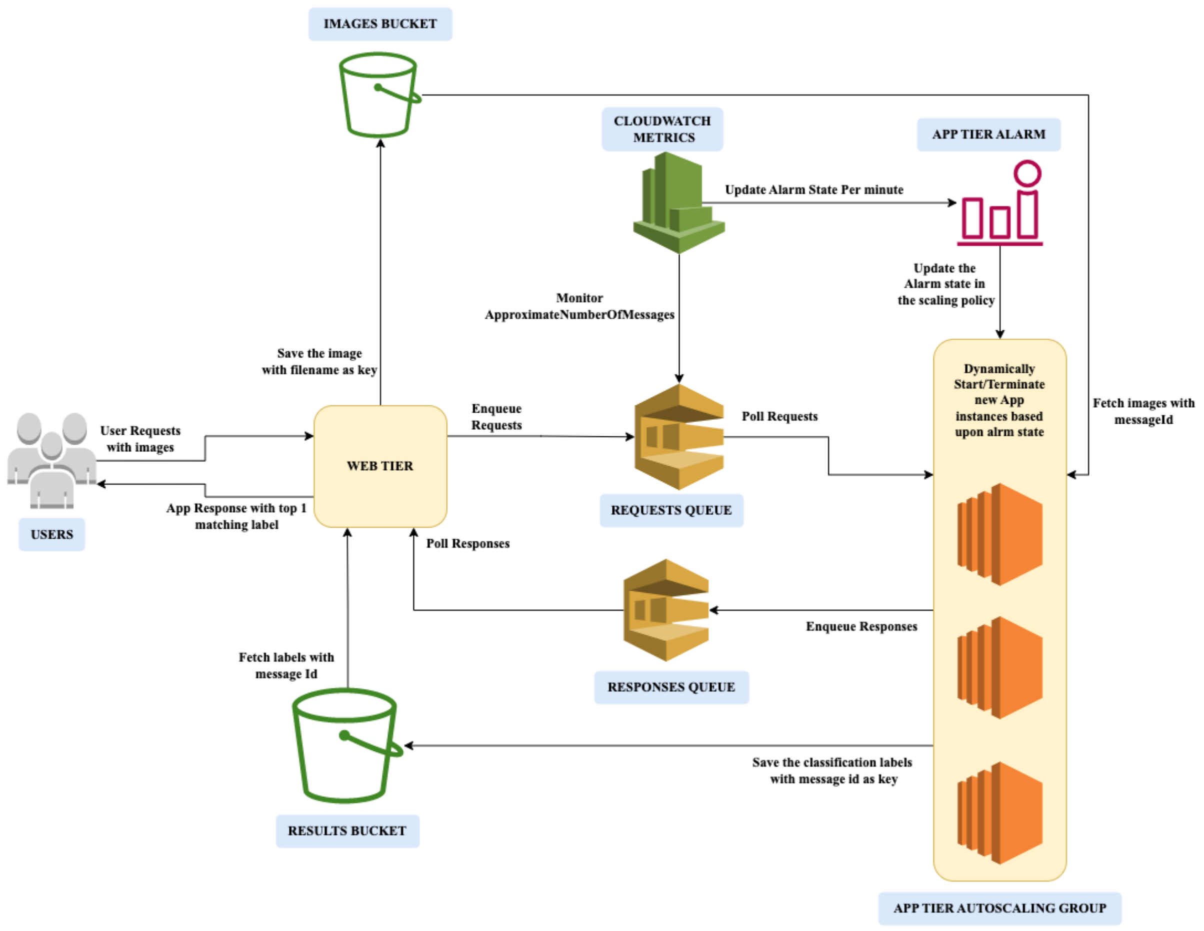

IMAGES BUCKET
CLOUDWATCH METRICS
APP TIER ALARM
Update Alarm State Per minute
Monitor ApproximateNumberOfMessages
Update the Alarm state in the scaling policy
Save the image with filename as key
Dynamically Start/Terminate new App instances based upon alrm state
Fetch images with messageId
User Requests with images
Enqueue Requests
Poll Requests
WEB TIER
USERS
App Response with top 1 matching label
REQUESTS QUEUE
Poll Responses
Enqueue Responses
Fetch labels with message Id
RESPONSES QUEUE
Save the classification labels with message id as key
RESULTS BUCKET
APP TIER AUTOSCALING GROUP

Figure: The Architecture Diagram

S3 Buckets

AWS S3 is a cloud-based object storage service that is highly scalable and reliable and provides high availability and low latency. It organizes the objects into buckets. Further each object is identified by a unique user defined id. The S3 buckets can be managed using the AWS console or the SDK, which is how we have implemented it for this project. We have utilized two S3 buckets named the Images bucket and the Results bucket. When the web tier receives a user request, it stores the image along with its filename as the key into the Input bucket. The app instances can then access this image using the unique message id. After the deep learning model has successfully classified the image, the app instance uploads the classification result along with the same unique id into the Output bucket from where it can be retrieved by the web instance and returned to the user.

SQS Queues

Amazon SQS is a highly scalable messaging service created to mitigate the issues arising out of producer-consumer connectivity and other problems. SQS supports programmatic messaging via web-based applications. In this system, we have utilized two SQS queues named Request queue and Response queue. When the web tier receives a classification request, it uploads a message with the unique message id onto the Request queue. The app instance polls the request queue for new messages. Once it receives the message, it clears it from the request queue and commences the processing. After the image is successfully classified, the app instance loads the response with the same message id onto the Response Queue. From here it's polled by the web instance, which clears the message from the response queue and tunnels the response back to the user.

App Tier

The App instances are responsible for the image classification procedure. It integrates a pre-trained deep learning classification model which returns the topmost label that matches with the given image. The app server polls the Requests queue for new messages. On receiving these messages, it clears the queue and retrieves the image from the Input S3 bucket using the message id as key. If the key is valid, it will execute the ML model and save the results into the Output bucket. Finally, it adds a message to the response queue with the request message id to indicate successful processing. AWS Autoscaling is utilized to scale out and scale in the number of active EC2 instances for the app tier as per user demand. The App uses AWS CloudWatch to monitor the request queue and set an alarm indicating the need to scale out. Once the alarm stops due to decline in user demand, the EC2 instances are terminated and the app scales in. This is covered in detail in section 2.2.

2.2 Autoscaling

Amazon Autoscaling or automatic scaling is the cloud-based service which dynamically allocates additional cloud resources as the user demand increases and accordingly reduces the allocated resources as the demand decreases. It is one of the salient features of AWS which makes its cloud-based infrastructure reliable, durable, and economical.

In this project we utilize the AWS EC2 Autoscaling groups with a step scaling policy, which allocates and deallocates EC2 instances as per user demand. We achieve this through a two-step mechanism. The first process involves a group of EC2 instances known as the Auto scaling groups. The Auto scaling group is a logical set of EC2 instances maintained to achieve the desired capacity of the application through automatic scaling. For each Auto scaling group, you can define the minimum and maximum number of instances to ensure better cost management. Further we can program the automatic scaling by defining the desired capacity or through scaling policies. Scaling policies are applied by the auto scaling group progressively to ensure that the group is at desired capacity as per the user demand. These policies allow the application to define adaptable capacity for its applications and ensure better availability. Finally, the auto scaling group ensures better fault tolerance by performing routine health checks on the EC2 instances, replacing malfunctioning instances with new ones making sure that the application is at full desired capacity.

The second step in auto scaling is the Amazon CloudWatch. It is a monitoring tool that can be utilized to continuously monitor the performance of your application. It can collect and track metrics, which are variables related to the performance of your application and demand. Further we can set alarms to go off based on these metrics. These alarms can be utilized to stop, terminate, reboot, or recover EC2 instances in conjugation with the auto scaling groups. The alarms can be assigned to one or more dynamic scaling policies within an Autoscaling group. The scaling policies can be assigned to make changes based upon the two alarm states: OK, which means that the selected metric is within the requisite limit and ALARM, which means that the given metric has breached its ceiling value.

In this project we have utilized one CloudWatch alarm titled "App Tier Alarm" to monitor the depth of the Request SQS Queue. This queue is used to send the user classification requests from the web tier to the app tier. The Alarm monitors the number of messages in the queue and goes into the ALARM state if the number of messages is greater than or equal to 1 every minute. Further we are also utilizing an Auto scaling group titled "App Tier". This is a collection of EC2 instances running the app server which is responsible for the image classification task. This group has two dynamic auto scaling policies in relation to the alarm – "scale in" which removes up to 20 instances when the alarm state is OK and "step" which linearly increases the number of instances up to 20 based upon the number of messages in the request queue. This ensures that user requests are handled concurrently across multiple servers in an optimal time.

2.3 Member Tasks

All of the members divided the tasks among themselves based on familiarity of the tools used and availability. The contributions were coordinated through github repository and brainstorming sessions were regularly done using zoom. Following are the individual contributions made by each member.

2.3.1 Contributions made by Shreyas Kirtane

All of the team members decided that python was the preferred programming language. So I started looking into available options to run a web server to serve HTTP requests in python. In my research, Django and Flask came up as the most used Frameworks in the python community. There was also the option of running a vanilla webserver. However there were many benefits to using a framework. After weighing the pros and cons of each option I selected Flask and started implementing a basic web server in Flask.

Since this was a big project, First I broke down the requirements into smaller requirements and formulated smaller steps, tasks and goals to reach. I decided to first implement the web server to write the image received from the request to a directory and similarly implement the app tier to read the file from that directory after receiving a message from the web tier. I went through the provided implementation and incorporated that into app tier code as it was more efficient to evaluate it in the app tier rather than calling it as a shell programmatically for each request. I also created a requirements file for each the web tier and app tier to ensure that the required libraries were tracked with correct versions. After that I also created dockerfiles for each of the web tier and app tier. The reasoning behind it was that it enabled us to use the same environment for development as well as testing. This reduced a lot of effort required to set up ec2 instances. By using docker containers we only had to create container images and install docker on ec2 instances. A dockerignore file was also created to exclude built images which were in multiples in sizes of 100MB from getting into docker build context. Furthermore to aid the development process, I created Makefile with very useful phony commands. These commands helped us to build, tag, save, deploy and clean containers/images with a single command. So to install the project, anyone would have to just install docker, make aws credentials available as environment variables, clone the repository and run the make command to deploy the container.

After the deployment process was implemented. I started developing methods to interact with s3 buckets. I implemented the methods in a common utility file so that these can be used in both app tier and web tier programs. I also incorporated this into the current implementation, so now instead of writing to a directory, the images were written to s3 and only the filename was passed as parameter to the app tier program.

After all the members completed their part, the project was thoroughly tested. During this phase I noticed that the web tier was slow in printing its result received from the response queue. I determined that the reason for this was there was a single lock used to synchronize reads and writes in the webtier. So a lot of threads(requests thread equal to the number of requests) and the poll thread to get messages from the response queue. Since we can receive multiple messages in one go from sqs queues. It would be faster if multiple threads could read the result concurrently while poll thread writes once in a while. This was a well known construct in distributed computing called multiple readers and a single writer. However there was no default implementation in python, So I implemented this using condition variables and keeping track of each reader.

Finally I created AMIs for app tier and webtier. Using the app tier AMI, I also created a launch template which can be used in creating an autoscaling group. During the course of this project, I learned about web technologies, AWS services, docker, Makefiles and synchronization constructs.

2.3.2 Contributions made by Parth Shah

My main task was to add integration with SQS to our web tier and app tier servers. I set up the applications to use the official Boto3 SDK for programmatically accessing AWS services from Python. I learnt about the two types of queues AWS offers currently: Standard queues and FIFO queues. I went with the Standard queue since we don’t necessarily need to preserve the order in which the requests are processed. Once I set up the queues on AWS, I added the queue names and URLs to the configuration files for web tier and app tier. For the web tier, I added a method to send the S3 image URL as a message to the request queue and the same method was used to send the classification result to the response queue in the app tier.

The next task was to continuously poll the request queue in the app tier application and do the same for the response queue in the web tier application. I decided to use the AWS API provided parameter MessageId for identifying the message so that we can return the correct classification result. I stored the MessageId at the web tier from the response when pushing a message to the request queue. In the app tier, I sent the MessageId as a Message Attribute to the response queue. Back in the web tier, in the polling thread, I stored all the messages from the results queue to a dictionary with the corresponding request MessageId as the key so that we can efficiently retrieve the correct result from the dictionary.

2.3.3 Contributions made by Gaurav Kulkarni

I worked on implementing the auto scaling of app tier instances. I started by researching Cloudwatch and how to utilize it to automatically scale EC2 instances. Once the web instance was deployed and app tier AMI was created, I created a CLoudWatch Alarm to monitor the SQS metric ApproximateNumberOfMessagesVisible which returned the depth of the requests queue. It indicated the number of messages that are currently waiting in the requests queue. I also created an autoscaling group from the launch template and the app tier AMI image. This group would be utilized to scale out and scale in the app instances.

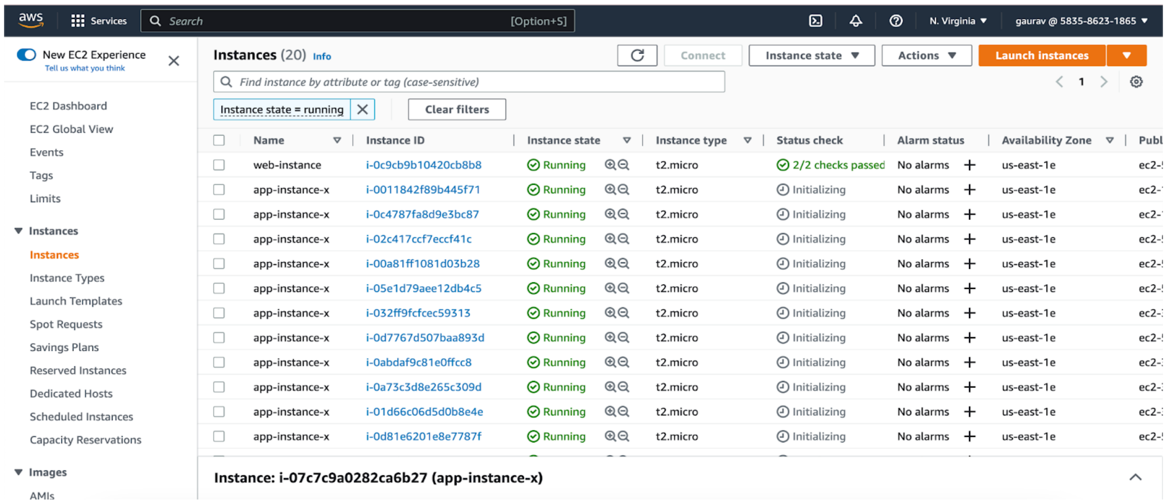
The final step in the project was to implement an optimal scaling policy. The ideal choice was to implement a target tracking policy which implemented metric math by calculating the number of instances active in the autoscaling group and the number of messages in the queue. But since the free tier cloudwatch alarm only changed the state for 3 data points in 3 minutes, it resulted in significant delay in the implementation and failed the evaluation criteria. So I revisited the dynamic scaling policies and the different options available. I utilized a step scaling policy to linearly scale out the app instances each minute based upon the number of requests it receives up to 20 total instances. And another simple scale out policy to terminate up to 20 instances if there are no messages in the requests queue. This successfully achieved the required scaling task within the permissible time. I also worked on the documentation.

With this project, I got to learn about the practical implementation of Amazon EC2, SQS, S3, CloudWatch as well as Docker.

3. Testing and evaluation

We started the testing and evaluation by testing the web tier server locally by uploading an image to make sure that we are getting a successful response. We similarly tested the app server on a physical computer to ensure that images were successfully classified and that the SQS and S3 infrastructure is working successfully. In the next step we tested the code on two EC2 instances for web tier and app tier respectively. For all the three processes our criteria for evaluation were successful upload of the image and accurate classification based upon the deep learning model.

The next step was to test the scaling of the EC2 instances. We started by increasing the number of user requests in the increments of 20 starting from 20 to 100. As the deep learning model is very swift, only one app instance was able to handle up to 100 requests in a few minutes. Therefore, we added a delay to demonstrate the auto scaling mechanism. Post that, we set up a CloudWatch alarm to monitor the request queue depth. Once we ensured that the alarm state changed to ALARM for every minute the queue had one or more messages, we set up the auto scaling group and the dynamic scaling policies. We tried several scaling policies and tested them for a series of iterations of requests i.e.7 requests, 20 requests, 100 requests. The major hindrance to the scaling policy was the fact that CloudWatch Alarm state only changed once every minute. So, the policy had to ensure that the EC2 instances scaled up linearly to the total number of requests or 19 whichever is the highest number. After testing different policies, we achieved an optimum policy.



Instances (20) Info

Instance state = running

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Publ
web-instance	i-0c9cb9b10420cb8b8	Running	t2.micro	2/2 checks passed	No alarms	us-east-1e	ec2-
app-instance-x	i-0011842f89b445f71	Running	t2.micro	Initializing	No alarms	us-east-1e	ec2-
app-instance-x	i-0c4787fa8d9e3bc87	Running	t2.micro	Initializing	No alarms	us-east-1e	ec2-
app-instance-x	i-02c417ccf7eccf41c	Running	t2.micro	Initializing	No alarms	us-east-1e	ec2-
app-instance-x	i-00a81ff1081d03b28	Running	t2.micro	Initializing	No alarms	us-east-1e	ec2-
app-instance-x	i-05e1d79aee12db4c5	Running	t2.micro	Initializing	No alarms	us-east-1e	ec2-
app-instance-x	i-032ff9fcfcec59313	Running	t2.micro	Initializing	No alarms	us-east-1e	ec2-
app-instance-x	i-0d7767d507baa893d	Running	t2.micro	Initializing	No alarms	us-east-1e	ec2-
app-instance-x	i-0abdaf9c81e0ffcc8	Running	t2.micro	Initializing	No alarms	us-east-1e	ec2-
app-instance-x	i-0a73c3d8e265c309d	Running	t2.micro	Initializing	No alarms	us-east-1e	ec2-
app-instance-x	i-01d66c06d5d0b8e4e	Running	t2.micro	Initializing	No alarms	us-east-1e	ec2-
app-instance-x	i-0d81e6201e8e7787f	Running	t2.micro	Initializing	No alarms	us-east-1e	ec2-

Instance: i-07c7c9a0282ca6b27 (app-instance-x)

Figure: 20 Total instances scale out to handle requests concurrently

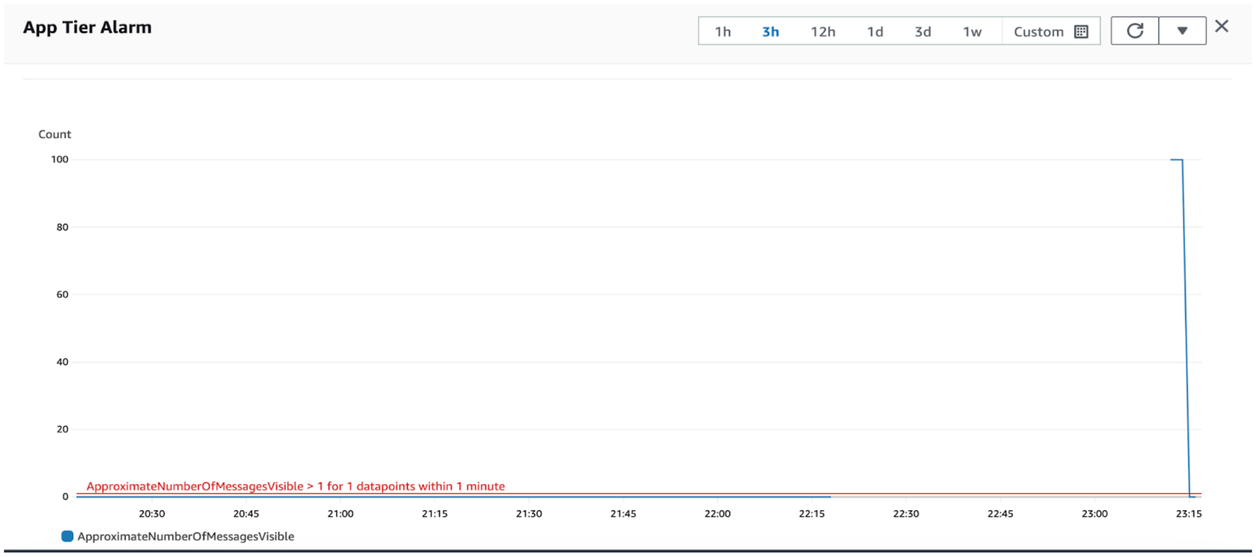


Figure: CloudWatch Alarm scales out swiftly to handle 100 requests

We tested this against the same set of user request iterations. Our criteria for evaluation consisted of three requirements –

- If the requests are less than 20, the auto scaling group should scale out to the same number of instances as the requests
- If the requests are more than 20, the auto scaling group should scale out to 19 app tier instances

- If there are no requests the auto scaling group should scale down accordingly
- The total time taken to scale out, scale in and the processing should be less than 7 minutes

We tested all our scaling policies against these criteria. The initial policies resulted in not scaling out at all even for an ALARM state, not scaling out linearly according to the number of user requests or not scaling in once the processing was complete. Also the time required for the whole process was more than 10 minutes in certain cases. But the final step scaling policy satisfied all the above criteria concluding the testing phase.

4. Code

4.1 Functionality and Purpose of each program

at_server.py

The at_server python file handles all the processing on the app tier EC2 instances. The code to get the classification results is put into the classify function. Other than that, we run this file as soon as the instances are launched. On running this file, we start polling the request queue in the background and fetch the messages from SQS. If there is a new message, the program will send the message data, which is the S3 image URL to the classify function, write the result to the result queue and then delete the message from the request queue.

at_config.ini

This file has all the necessary configuration parameters and variables for the at_server.py file like the name of the S3 buckets and the SQS queue URLs.

at_dockerfile

This dockerfile is used to create an debian image of the at_server with all its requirements so that we can easily run it using Docker

at_requirements.txt

This file has all the python-pip requirements for the service running on app tier

app-tier-key.pem

This is the private key for the app tier EC2 instances

wt_server.py

The wt_server.py file runs a Flask server that has a /classify endpoint. Once a request is sent to this endpoint, the flask server will upload the image to S3 and then push the image URL to the request queue. In another thread, we poll the response queue and save all the available messages to a python dictionary for quicker access. We then filter the response using the SQS MessageId parameter and the flask server returns this result as a response to the workload generator

wt_config.ini

This file has all the necessary configuration parameters and variables for the wt_server.py file like the name of the S3 buckets and the SQS queue URLs.

wt_dockerfile

This dockerfile is used to create an debian image of the wt_server with all its requirements so that we can easily run it using Docker

wt_requirements.txt

This file has all the python-pip requirements for flask server running on web tier

web-tier-key.pem

This is the private key for the web tier EC2 instances

utils.py

This file has the methods to download and upload files from S3 buckets using boto3

RWLock.py

This file has a lock class so that we can ensure data integrity while reading and fetching messages from the results dictionary

.dockerignore

Files that docker should ignore while building the docker images

Makefile

To easily build and run the docker containers

4.2 Installation and deployment of project

a. **AWS**
    1. Create request, response queues
    2. Create input and results buckets
    3. Update config files in the project
b. **Web Tier**
    1. Launch EC2 instance with public ip.
    2. Install docker on these instances
    3. Modify bashrc to provide AWS credentials for the code each time the server reboots.
    4. Clone the git repository
    5. Run the command in git repository: make run_wt
c. **App Tier**
    1. Follow steps 1-4 from Web Tier and run command: make run_at
    2. Save the AMI from the ec2 console using this instance.
    3. Create a Launch template and use the template in AutoScaling group
    4. Create scaling policy for AutoScaling group and Alarm based on messages in the request queue