ECE1779 Assignment II Documentation

Hao Jin (1000303405)
Louis Chen (1000303502)

Table of Contents

1.0 High-Level Description

This documentation serves as a general guideline for ECE1779 assignment 2. End deliverable of this assignment is a web server which provides online photo storage service. The web server is preloaded on Amazon Web Services' (AWS) Elastic Compute 2 (EC2) server. It provides features such as new account registration, login authentication, photo uploading, and face detection in the uploaded photos. This web server consists of two applications: *user-app* and *manager-app*.

The *user-app* is the application with web user interface (UI) for end users to register, login, and upload photos. Using Amazon's Elastic Load Balancer (ELB), the web server provides abstraction of load balancing to end users, such that users always retain sufficient bandwidth of the web service, and need not worry about high service traffic.

The *manager-app* is the application with web UI which allows web server administrator to monitor CPU utilization and HTTP request rate on each spawned web server in the worker pool (each server runs on an Amazon EC2 instance in the worker pool). It also allows administrator to manually add or remove an EC2 instance and deletes all users and photos stored on the cloud (Amazon S3).

Meanwhile, there is a script (*auto_scaling.py*) which automatically scales the worker pool size, according to current load of the pool. Given some scaling rules and factors predefined by the manager-app, the script scales the pool size accordingly to ensure web server's accessibility and load balancing.

Please refer to subsequent sections for *Accessing the Web Server*, *Server Architecture for User-App*, *Server Architecture for Manager-App*, *HTTP Request Monitor*, *Database Architecture*, *APIs*, *Results*, *Individual Contributions*, and *Revision Control*.

2.0 Accessing the Web Server

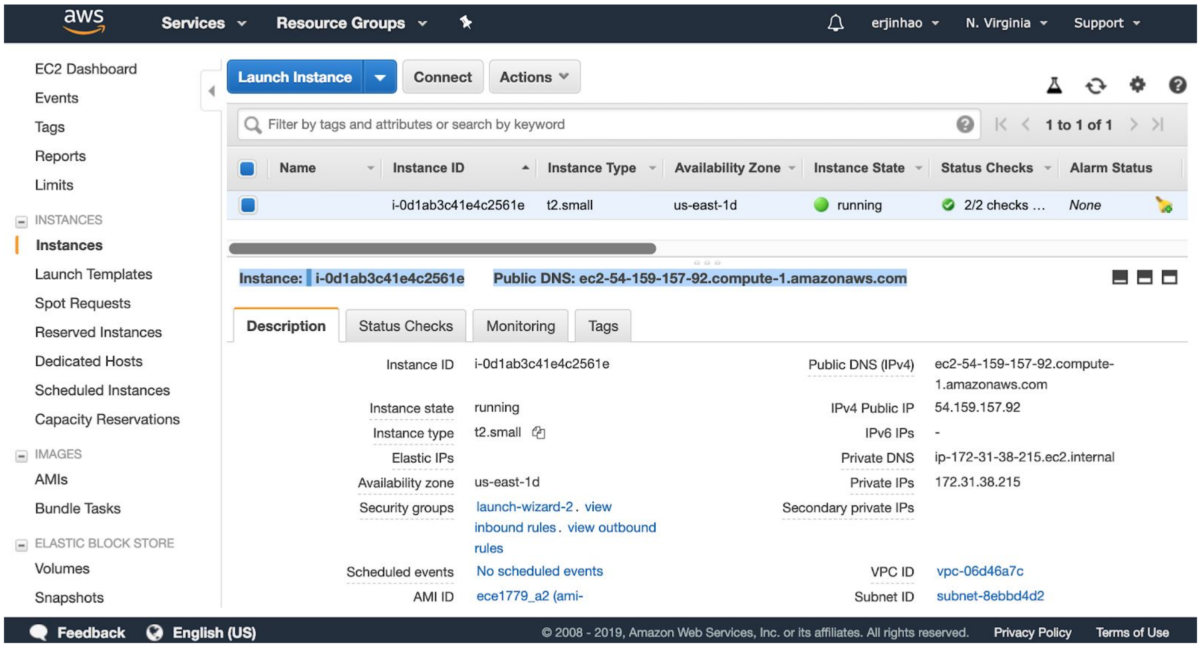
This section shows the step-by-step procedures for launching the web service. By default, the user-app is activated with an ELB, and therefore the only things user needs to activate are the manager-app in order to allow back-end monitoring, and the auto_scaling script in order to avoid high load congestion on any worker instance.

The following shows the URL for accessing the user-app.
http://loadbalance-1666451346.us-east-1.elb.amazonaws.com:5000

The following shows steps for launching the manager-app.

1. Open a browser, navigate to https://aws.amazon.com/.
2. Click "My Account" on the top navigation bar, and choose "AWS Management Console".
3. Sign in using the email and password provided in the file "credentials.txt" (provided in the zipped package).
4. Under "Services" on the navigation bar, click "EC2".

5. Under “EC2 Dashboard” on the left, click “Instances”.
6. You should see an EC2 instance, named *Manager*, is present and shown status as *stopped*. If instance state shows *running*, go to step 8). Now select the instance, click “Actions” button on the top, choose “Instance State”, and click “Start”. When prompted, click “Yes, Start”.
7. Wait for the “Instance State” to turn green (running). Once it turns green, the EC2 instance has been launched and ready to run the web server.
8. Locate the “keypair.pem” file provided in the zipped package. Open a terminal, type “chmod 400 <path_to_keypair.pem>” to set the correct access permissions.
9. Now press “Connect” on the top menu. You should see an example command to ssh into the EC2 instance. In the terminal, type the command in. The command should look similar to the following: “ssh -i <path_to_keypair.pem> ubuntu@ec2-54-159-92.compute-1.amazonaws.com”.
    a. Make sure to specify the correct path to the pem file “keypair.pem” (provided in the zipped package).
    b. Make sure to change “root” to “ubuntu”.
10. Now your terminal should be directed to manager-app’s terminal. Type the following command to launch the manager-app and auto_scaling script: “~/Desktop/start.sh”.
11. The manager-app is now running. On your local computer, open a browser and type the following url to connect to the manager-app: “http://ec2-3-84-217-120.compute-1.amazonaws.com:5000”.
    a. Make sure you use the correct DNS, as it changes once the instance is rebooted. Correct DNS can be found in the main EC2 console, shown as follows.
    b. Make sure to stop the web server and the EC2 instance once you are done. In the terminal, type the following command to stop the manager-app: “~/Desktop/stop.sh”. To stop the EC2 instance, you may take step 6) and select “Stop” instead of “Start”.



Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status
	i-0d1ab3c41e4c2561e	t2.small	us-east-1d	running	2/2 checks ...	None

Instance: i-0d1ab3c41e4c2561e Public DNS: ec2-54-159-157-92.compute-1.amazonaws.com

Instance ID	i-0d1ab3c41e4c2561e	Public DNS (IPv4)	ec2-54-159-157-92.compute-1.amazonaws.com
Instance state	running	IPv4 Public IP	54.159.157.92
Instance type	t2.small	IPv6 IPs	-
Elastic IPs		Private DNS	ip-172-31-38-215.ec2.internal
Availability zone	us-east-1d	Private IPs	172.31.38.215
Security groups	launch-wizard-2. view inbound rules. view outbound rules	Secondary private IPs	
Scheduled events	No scheduled events	VPC ID	vpc-06d46a7c
AMI ID	ece1779_a2 (ami-	Subnet ID	subnet-8ebbd4d2

3.0 Server Architecture for User-App

The User app is as same as the one in assignment one except for the following changes:

- Requirements for assignment two:
    - Photos are now stored in AWS S3
    - All information (user credentials, photo references, HTTP request log, etc) is stored on AWS RDS
- Feedbacks from assignment one
    - Separate profile page and upload page
    - Remove background image and add a navigation bar

The following is the architecture for the web server. * means new changes made in the assignment two

- ❏ app/
    - ❏ static/
        - ❏ css/
        - ❏ users/
    - ❏ templates/
        - ❏ image.html
        - ❏ login.html
        - ❏ profile.html
        - ❏ register.html
        - ❏ upload.html*
    - ❏ train_file/
        - ❏ eye.xml
        - ❏ face.xml
    - ❏ __init__.py
    - ❏ api.py
    - ❏ db.py
    - ❏ image.py
    - ❏ login.py
    - ❏ macro.py
    - ❏ main.py
    - ❏ profile.py
    - ❏ register.py
    - ❏ upload.py*
- ❏ run.py

Static folder contains all static resources for the web server categorized into two folders which are css and users.

- css
    - This folder has all the CSS files used to decorate the HTML files in the templates. Login.css is linked in login and register pages and profile.css is linked in the profile page.

    Besides customized CSS file, pages are also linked with bootstrap CSS and font awesome CSS for designing the front-end UI.

- users
    - This folder contains all the images uploaded by the users and corresponding face-detected images. All uploaded images are stored and categorized into specific folders named after the username of uploaders.

Templates folder contains five HTML templates, which are login.html, register.html, profile.html, upload.html, and image.html. Following describes what users are able to do in these pages.

- login.html
    - This template renders a page for users to log in with their credentials if they have registered before. It has two inputs. One is the username. Another one is the password. If users never registered, there is a button linked to the register page. If users input wrong username or password, a proper message will be displayed in red color.
- register.html
    - This template renders a page for users to register a new account. It has three inputs which are username, password, confirm password. If users input a username that has been registered or mistyped the password, an error message will be displayed. There is also a button below these inputs to redirect the URL to the login page.
- profile.html
    - This template renders a profile page that displays thumbnails of the images uploaded by users. It starts with a welcome sentence including username. Users can hover on the thumbnail to see the face-detected image. Users can also click on the thumbnail to redirect to the image.html to see images with full resolution.
- upload.html *
    - This template renders a profile page that enables users to upload images and display the latest uploaded images. There is a “choose file” button for users to choose images to upload. After users click on the button, a system window will pop up to prompt to select a image file to upload. After being selected, the file will be automatically uploaded to the server. A feedback message will also be displayed on the page telling users if the file is uploaded or not. The file will be processed on the server and generated a new file with rectangles drawn around faces.
- image.html
    - This template renders a page for displaying images in full resolution. After a user clicks on the thumbnail of an image in the profile page, he can view the details in the image page. This template displays two images. One is the original image. Another one is the new image around faces detected by the application. This template also has a button that enables users to the profile page.

Train_file folder contains two training files for face and eye detection, provided by OpenCV.

- eye.xml
    - This training file trains the OpenCV model to detect human eyes.
    - Due to assignment requirements, this file is not used.

- face.xml
  - This training file trains the OpenCV model to detect human faces.

Python files that contribute to the main functionality of the web server are listed below.

- __init__.py
  - Initializes the web server.
  - Imports python files.
- api.py
  - Handles API requests (refer to section 6.0 for API details).
- db.py
  - Contains database configuration parameters (user, password, host, database name).
  - Contains functions to connect/disconnect to database.
  - Contains function to generate a random salt for account registration.
- image.py
  - Handles image URL and receives image paths through URL
  - Renders image.html to show images in full resolution
- login.py
  - Handles user login POST form. It first validates the input strings to check if they have an improper character to avoid SQL input hack. It then queries the database to check if the username is used and if the password is correct. It returns error messages if there is any problem.
  - Redirects to profile page if the user successfully logged in.
- macro.py
  - Stores secret key for Flask session as *secret_key*.
  - Stores valid characters for username as *username_char*
  - Stores valid characters for password as *password_char*.
  - Stores valid characters for salt as *salt_char*.
- main.py
  - Renders the main page of the web server. It checks whether user has previously logged in using Flask session. If so, it renders the page to profile.html. If not, it renders the page to login.html to prompt user to log in.
- profile.py
  - Contains function to retrieve all photo paths associated with currently logged in user.
  - Handles user logout POST form. It removes username from the Flask session and redirects the browser to main login page.
- upload.py*
  - Handles photo upload POST form. It first validates all parameters (username, password, file). If they are not valid, the function will return error message with proper HTTP status code. It then retrieves the photo, performs face detection on it, and makes a copy of the photo with face detection marked in blue rectangles. Finally, it stores the original photo and the photo marked with face detection onto EC2 instance's local disk, under the relative directory "app/static/users/". The photos are named after timestamp (in the format of yymmdd_hhmmss).

- register.py
    - Contains function to render the account registration form (register.html).
    - Handles account registration POST form. It first checks if username, password, and confirm_password contain valid characters as listed in macro.py. It then checks if the username has been registered. If there is no error, it will redirect the user to the profile page. Otherwise, it will display the corresponding error on the webpage in red.
- run.py
    - Serves as top-level python file for the user to launch the web server.

4.0 Server Architecture for Manager-App

The following is the architecture for the web server. * means new changes made in the assignment two

- ❏ app/
    - ❏ static/
        - ❏ css/
    - ❏ templates/
        - ❏ ec2_list.html
        - ❏ ec2_details.html
        - ❏ s3_list.html
        - ❏ scale_params.html
    - ❏ __init__.py
    - ❏ aws.py
    - ❏ config.py
    - ❏ db.py
    - ❏ main.py
- ❏ auto_scaling.py
- ❏ run.py

Static folder contains all static resources for the web server. It now contains css folder

- css
    - This folder has all the CSS files used to decorate the HTML files in the templates. Manager.css is linked in all templates. Besides customized CSS file, pages are also linked with bootstrap CSS for designing the front-end UI.

Templates folder contains four HTML templates, which are ec2_list.html, ec2_details.html, s3_list.html and scale_params.html. Following describes what users are able to do in these pages.

- ec2_list.html
    - This template renders a page for users to see a list of instances in the worker pool. It contains a button to create a new instance and register the instance into the worker pool. The table shows the details of the instances including instance id, name, type, Availability Zone, status. There are two buttons beside each instance to see more details and delete the instance from the worker pool.
- ec2_details.html

    - This template renders a page for users to see the details of an instance. It contains a table showing the instance info including instance id, ami id, key-pair name, public IP address, and state. It also displays two charts which are CPU utilization and HTTP request rate. CPU utilization chart shows the CPU utilization of the instance in 60 minutes. Http request rate chart shows the number of HTTP request per minute in the instance
- s3_list.html
    - This template renders a page that displays a list of images stored in S3 and a button to delete all the data stored in RDS and images in S3.
- scale_params.html
    - This template renders a page that enables users to set the parameters to configure the auto-scaling policy including CPU max threshold to grow the worker pool, CPU min threshold to shrink the worker pool, the increase ratio and the decrease ratio.

Python files that contribute to the main functionality of the web server are listed below.

- __init__.py
    - Initializes the web server.
    - Imports python files including main.py, aws.py and db.py.
- aws.py
    - Contains a function to create a new instance and create a new thread to register it into load balance group
        - Use boto3 to connect to aws ec2 and elastic load balancing
        - Use ec2.create_instances() to create new instance
        - Use elbv2.register_targets() to register the instance into the load-balance group and wait for it until running
    - Contains a function to delete an instance with a specific id.
        - Given the instance id, call the ec2.instances.filter(InstanceIds=ids).terminate () to delete the instance
        - Delete corresponding HTTP request information in RDS
    - Contains a function to display a list of instances in the worker pool
        - Use ec2.instances.filter() to get all running instances and display them
    - Contains a function to show the details of an instance
        - Use boto3 to connect to ec2 and cloudwatch
        - Use ec2.Instance(id) to display instance information
        - Call cloudwatch.get_metric_statistics(params) to get the CPU Utilization Information and display the chart
        - Use the HTTP request information stored in RDS to display the chart
    - Contains a function to display a list of images stored in S3
        - Use boto3 to connect to S3
        - Use s3.Bucket() to get and display all the image objects
    - Contains a function to delete all the data in S3 and RDS
        - Use boto3 to connect to S3 and call s3.Object().delete() to delete all images
        - Query RDS to delete all information in users and photos

    - Contains a function to display the setting scaling parameter form and store the parameters in the RDS.
        - Get method to display the form to set the scaling parameters
        - POST method to store parameters in the RDS
- db.py
    - Contains functions to connect/disconnect to database.
    - Contains a function to delete outdated HTTP request stored in RDS
    - Contains a function to retrieve the HTTP request rate
- config.py
    - Contains database configuration parameters (user, password, host, database name).
    - Contains ami id used to create new instance
    - Contains target group for instances to register in
    - Contains S3 bucket name
- main.py
    - Renders the main page showing the worker pool
- auto_scaling.py
    - Uses pymysql to connect to the database
    - Gets scaling parametes including max threshold, min threshold, grow ratio and shrink ratio
    - Uses boto3 to connect to ec2 and cloudwatch
    - Calls ec2.instances() to get all running instances
    - Calls cloudwatch.get_metric_statistics(params) to get CPU Utilization statistics for all instances
    - Compares the average of the all CPU Utilization with the threshold to decide if there is a need to expand and shrink the worker pool.
    - Grow the worker pool with grow ratio by calling ec2.create_instances() to create new instances and elbv2.register_targets() to register instances into load-balance group
    - Shrink the worker pool with shrink ratio by calling ec2.instances.filter(InstanceIds=[id]).terminate() to delete instances
- run.py
    - Serves as top-level python file for the user to launch the web server.

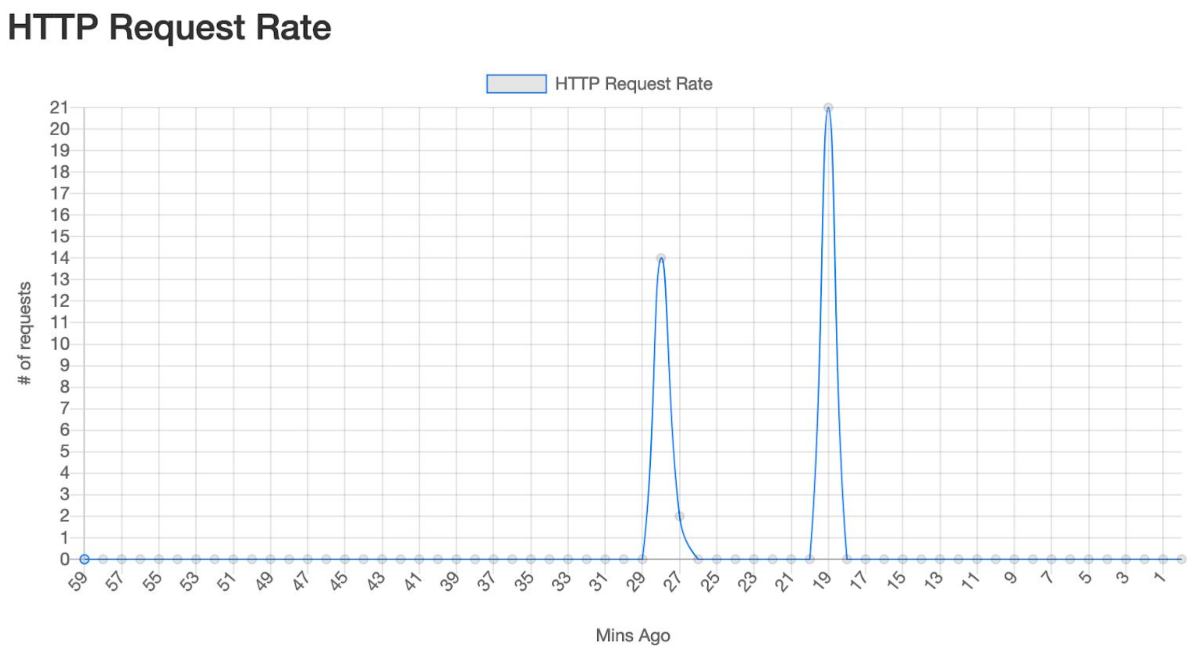
5.0 HTTP Request Monitor

Not only does the manager-app provide information on CPU utilization on every worker, it also provides a graph of HTTP request rate per minute over the past 30 minutes on every worker. HTTP request rate per minute is defined to be the sum of all HTTP requests that have happened over the corresponding minute. For example, suppose URL /main is requested once at 14:35:00 (hour:minute:second) and once at 14:35:39, and URL /api/upload is requested once at 14:35:59 and once at 14:36:00. Then the HTTP request rate at 14:35 (hour:minute) will be 3 (2 /main and 1 /api/upload requests).

In order to calculate this, the user-app writes a log into its database whenever there is an HTTP URL requested. These URLs include /, /index, /main, /login, /register, /upload, /api/register, /api/upload, etc.

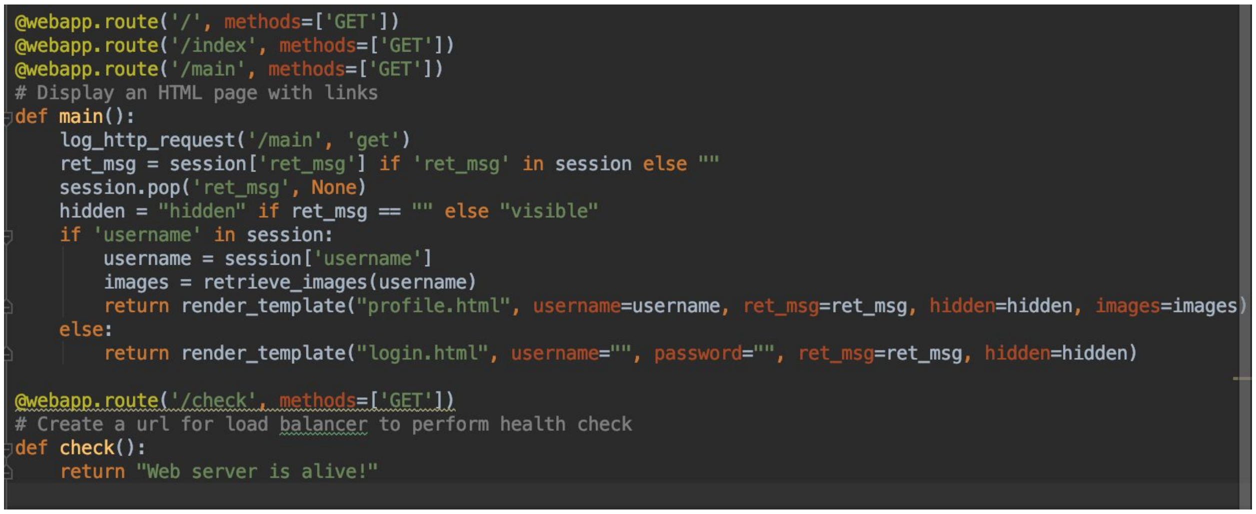
More specifically, user-app writes a row of content into its database immediately after any of these URL handler functions is called, right before the actual code executes. The row of content includes the instance ID in which the request is handled, timestamp, URL (e.g. /main), and its type (e.g. GET, POST).

When plotting HTTP request rate graph in manager-app, it first selects rows of contents with corresponding instance ID, sums and bins the requests into per-minute chunks, and displays them on the UI. In the background, manager-app spawns a thread which deletes HTTP request log that should be obsolete (older than 30 minutes as predefined) to avoid excessive log storage. A screenshot of the HTTP request rate graph is shown as follows.



As one can tell, this particular worker instance has handled 14 HTTP requests during the last 28th minute, and 21 HTTP requests during the last 19th minute, measured from the time when this screenshot was taken. Please disregard the number of HTTP requests handled, as auto scaling was disabled when this screenshot was taken.

This HTTP request monitor does not take ELB's health check requests into account. Instead, the team sets the health check URL to /check, and the corresponding URL handler (def check()) does not write a log into database. The following is the code snippet of URLs /main and /check handlers. In main(), log_http_request() is called to write the log into database, but not in def check().



```
@webapp.route('/', methods=['GET'])
@webapp.route('/index', methods=['GET'])
@webapp.route('/main', methods=['GET'])
# Display an HTML page with links
def main():
    log_http_request('/main', 'get')
    ret_msg = session['ret_msg'] if 'ret_msg' in session else ""
    session.pop('ret_msg', None)
    hidden = "hidden" if ret_msg == "" else "visible"
    if 'username' in session:
        username = session['username']
        images = retrieve_images(username)
        return render_template("profile.html", username=username, ret_msg=ret_msg, hidden=hidden, images=images)
    else:
        return render_template("login.html", username="", password="", ret_msg=ret_msg, hidden=hidden)

@webapp.route('/check', methods=['GET'])
# Create a url for load balancer to perform health check
def check():
    return "Web server is alive!"
```

For more information regarding HTTP log database structure, please refer to the next section.

6.0 Database Architecture

The web server stores all users' username, salt, hashed password (in table *users*), and their photo paths (in table *photos*) on AWS RDS. Other than those, it stores HTTP request log (in table *requests*) and auto scaling parameters (in table *scale_params*) on RDS as well. The AWS RDS structure is configured to be MySQL type. The following figure shows the relationship between tables *users* and *photos*.

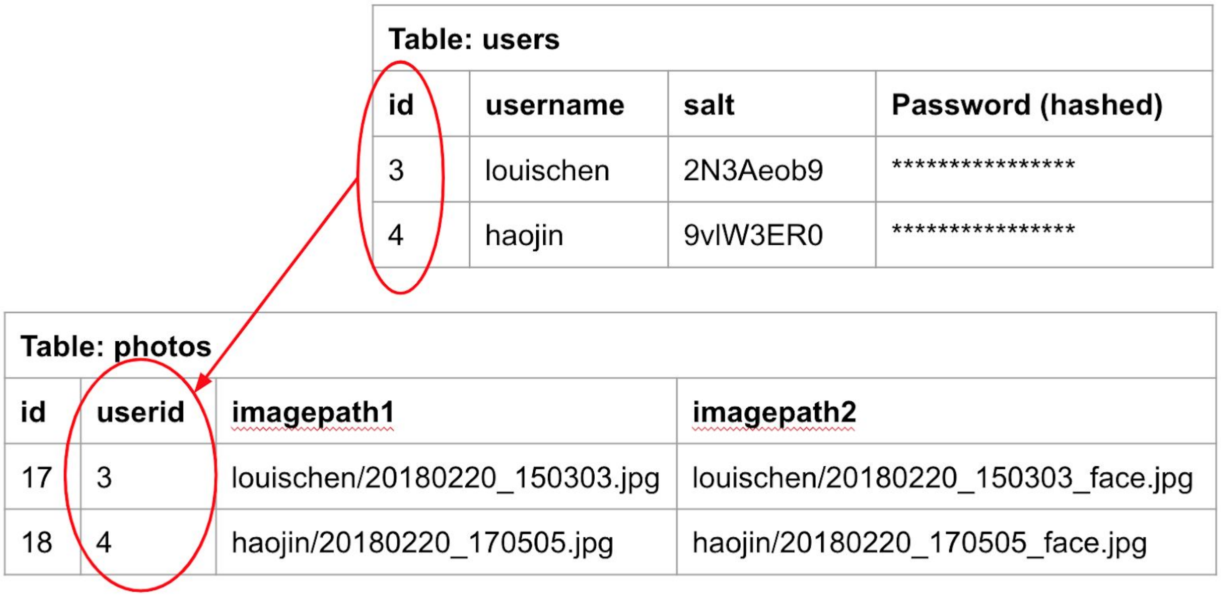


Table: users

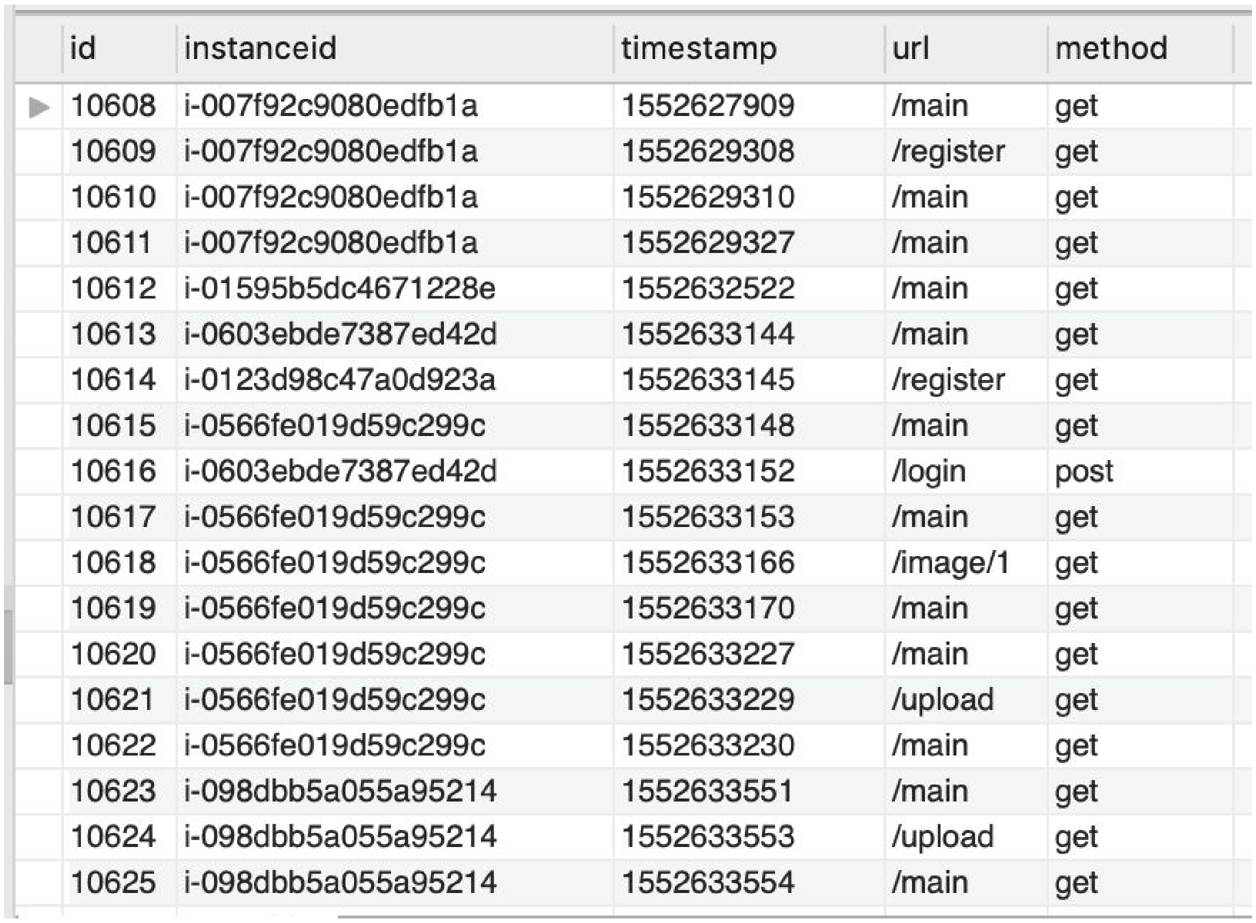
id	username	salt	Password (hashed)
3	louischen	2N3Aeob9	****************
4	haojin	9vIW3ER0	****************

Table: photos

id	userid	imagepath1	imagepath2
17	3	louischen/20180220_150303.jpg	louischen/20180220_150303_face.jpg
18	4	haojin/20180220_170505.jpg	haojin/20180220_170505_face.jpg

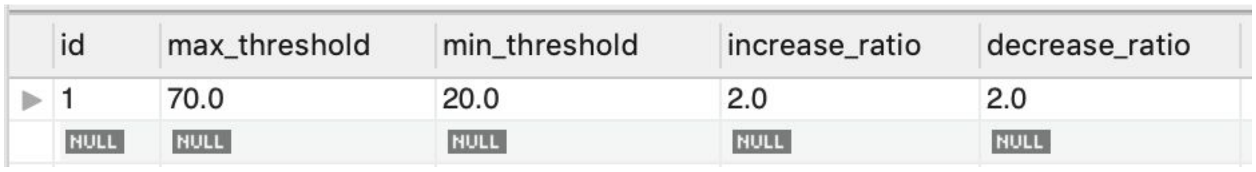
Table *users* stores information about user login credentials. It has columns *id* (primary key), *username*, *salt*, and *password* (SHA hashed value of password concatenated with salt). Table *photos* stores paths of the photos uploaded by the user. The paths are reference names for the user-app to retrieve actual photos from AWS S3. Table *photos* consists of columns *id* (primary key), *userid* (foreign key), *imagepath1* (path to original photo), and *imagepath2* (path to photo with face detection). The relationality of the database takes place in between table *users*'s primary *id* and table *photos*'s *userid*.

In order to calculate HTTP request rate on every active worker (EC2 instance which runs user-app), the web server needs to record a log whenever an HTTP request URL comes in. Therefore, all logs are recorded in table *requests*. For every incoming HTTP request, the user-app stores 4 pieces of information in table *requests*: worker instance ID in which the HTTP request comes, timestamp, type of HTTP request URL, the method of HTTP request URL. A snippet of table *requests* is shown as below.



id	instanceid	timestamp	url	method
10608	i-007f92c9080edfb1a	1552627909	/main	get
10609	i-007f92c9080edfb1a	1552629308	/register	get
10610	i-007f92c9080edfb1a	1552629310	/main	get
10611	i-007f92c9080edfb1a	1552629327	/main	get
10612	i-01595b5dc4671228e	1552632522	/main	get
10613	i-0603ebde7387ed42d	1552633144	/main	get
10614	i-0123d98c47a0d923a	1552633145	/register	get
10615	i-0566fe019d59c299c	1552633148	/main	get
10616	i-0603ebde7387ed42d	1552633152	/login	post
10617	i-0566fe019d59c299c	1552633153	/main	get
10618	i-0566fe019d59c299c	1552633166	/image/1	get
10619	i-0566fe019d59c299c	1552633170	/main	get
10620	i-0566fe019d59c299c	1552633227	/main	get
10621	i-0566fe019d59c299c	1552633229	/upload	get
10622	i-0566fe019d59c299c	1552633230	/main	get
10623	i-098dbb5a055a95214	1552633551	/main	get
10624	i-098dbb5a055a95214	1552633553	/upload	get
10625	i-098dbb5a055a95214	1552633554	/main	get

As mentioned in previous section, the web server provides configurability of auto scaling parameters. These parameters are also stored on AWS RDS in table *scale_params*. The table always has only one row of content, storing the maximum threshold, minimum threshold, increase ratio, and decrease ratio. A screenshot of this table is shown as follows.



	id	max_threshold	min_threshold	increase_ratio	decrease_ratio
▶	1	70.0	20.0	2.0	2.0
	NULL	NULL	NULL	NULL	NULL

7.0 APIs

Despite of direct access to the web server via browser, the web server also provides APIs to enable direct URL access. Two features that are accessible via APIs are account registration and photo upload.

The following figure shows the URL and the required parameters for account registration.

```
relative URL = /api/register
method = POST
POST parameter: name = username, type = string
POST parameter: name = password, type = string
```

The following figure shows the URL and the required parameters for photo upload.

```
relative URL = /api/upload
enctype = multipart/form-data
method = POST
POST parameter: name = username, type = string
POST parameter: name = password, type = string
POST parameter: name = file, type = file
```

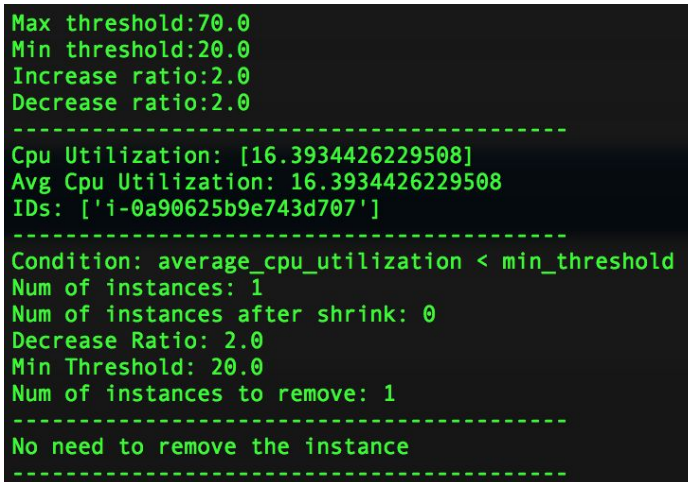
Users may use the command-line tool, curl, to post API requests. Example curl commands for posting account registration and photo upload are as follows.

```
curl -F “username=<username>” -F “password=<password>” http://127.0.0.1:5000/api/register
curl -F “username=<username>” -F “password=<password>” -F “file=@<path_to_file>” \
        http://127.0.0.1:5000/api/upload
```

If the API requests are handled successfully, users should receive a return message “Success”. If not, users will receive “Error: ” followed by a reason of failure.

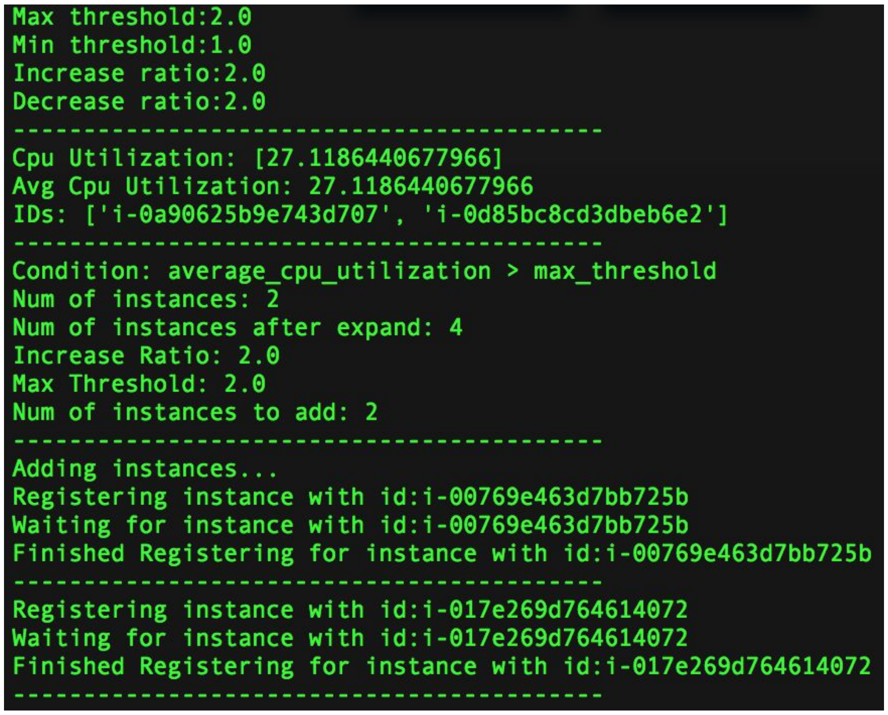
8.0 Results

Scenario One: There is only one instance running in the worker pool and its CPU Utilization is below the min threshold. No need to delete the instance and do nothing.



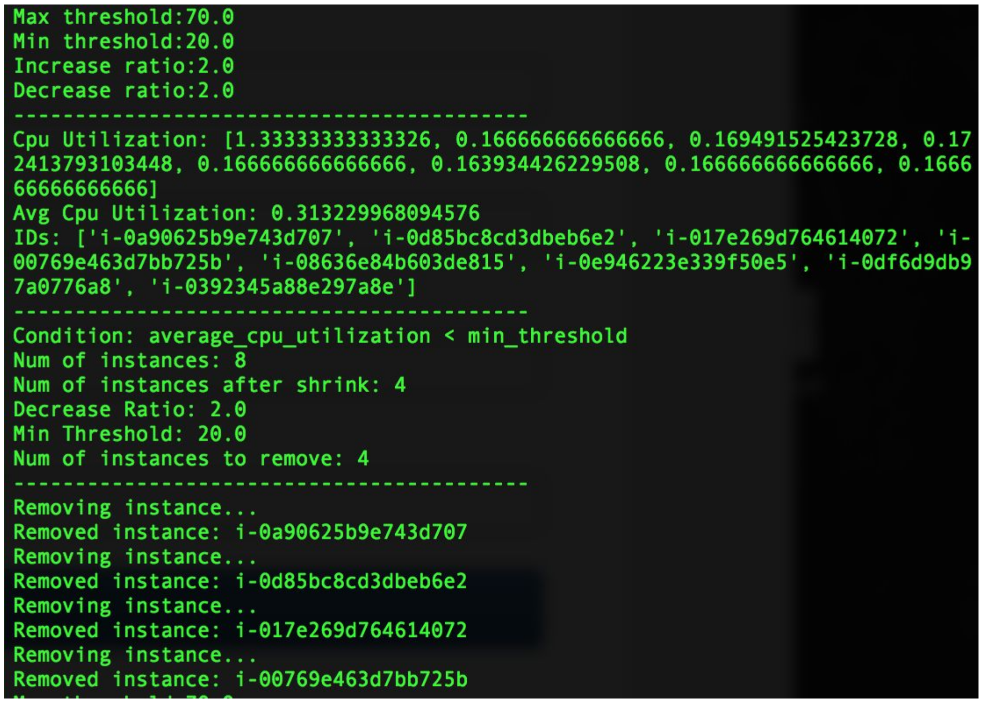
```
Max threshold:70.0
Min threshold:20.0
Increase ratio:2.0
Decrease ratio:2.0
------------------------------------------
Cpu Utilization: [16.3934426229508]
Avg Cpu Utilization: 16.3934426229508
IDs: ['i-0a90625b9e743d707']
------------------------------------------
Condition: average_cpu_utilization < min_threshold
Num of instances: 1
Num of instances after shrink: 0
Decrease Ratio: 2.0
Min Threshold: 20.0
Num of instances to remove: 1
------------------------------------------
No need to remove the instance
------------------------------------------
```

Scenario Two: The average CPU Utilization is greater than the max threshold, thus the auto-scaling program creates instances. The number of instances to add is determined by the grow ratio(increase ratio). For example, in the below screenshot, the current number of instances is two and the increase ratio is two. Thus, the program will double the worker pool if the condition is met. In this case, The program adds two instances and registers them in the load-balance group.



```
Max threshold:2.0
Min threshold:1.0
Increase ratio:2.0
Decrease ratio:2.0
------------------------------------------
Cpu Utilization: [27.1186440677966]
Avg Cpu Utilization: 27.1186440677966
IDs: ['i-0a90625b9e743d707', 'i-0d85bc8cd3dbeb6e2']
------------------------------------------
Condition: average_cpu_utilization > max_threshold
Num of instances: 2
Num of instances after expand: 4
Increase Ratio: 2.0
Max Threshold: 2.0
Num of instances to add: 2
------------------------------------------
Adding instances...
Registering instance with id:i-00769e463d7bb725b
Waiting for instance with id:i-00769e463d7bb725b
Finished Registering for instance with id:i-00769e463d7bb725b
------------------------------------------
Registering instance with id:i-017e269d764614072
Waiting for instance with id:i-017e269d764614072
Finished Registering for instance with id:i-017e269d764614072
------------------------------------------
```

Scenario Three: The average CPU Utilization is smaller than the min threshold, thus the auto-scaling program removes instances. The number of instances to remove is determined by the shrink ratio(decrease ratio).
For example, in the below screenshot, the current number of instances is eight and the decrease ratio is two. Thus, the program will shrink the worker pool by four instances if the condition is met. In this case, The program removes four instances.



```
Max threshold:70.0
Min threshold:20.0
Increase ratio:2.0
Decrease ratio:2.0
------------------------------------------
Cpu Utilization: [1.33333333333326, 0.166666666666666, 0.169491525423728, 0.17
2413793103448, 0.166666666666666, 0.163934426229508, 0.166666666666666, 0.1666
66666666666]
Avg Cpu Utilization: 0.313229968094576
IDs: ['i-0a90625b9e743d707', 'i-0d85bc8cd3dbeb6e2', 'i-017e269d764614072', 'i-
00769e463d7bb725b', 'i-08636e84b603de815', 'i-0e946223e339f50e5', 'i-0df6d9db9
7a0776a8', 'i-0392345a88e297a8e']
------------------------------------------
Condition: average_cpu_utilization < min_threshold
Num of instances: 8
Num of instances after shrink: 4
Decrease Ratio: 2.0
Min Threshold: 20.0
Num of instances to remove: 4
------------------------------------------
Removing instance...
Removed instance: i-0a90625b9e743d707
Removing instance...
Removed instance: i-0d85bc8cd3dbeb6e2
Removing instance...
Removed instance: i-017e269d764614072
Removing instance...
Removed instance: i-00769e463d7bb725b
```

9.0 Individual Contribution

This section shows the workload breakdown in the process of web server deployment. The following table breaks workload into several areas of implementation, with each area showing the primary and secondary contributors.

Area of Implementation	**Primary Contributor**	**Secondary Contributor**
Database Migration onto RDS	Hao Jin	Louis Chen
Storage Migration onto S3	Louis Chen	Hao Jin
Manager-App Front End	Hao Jin	Louis Chen
Manager-App Back End	Louis Chen	Hao Jin
HTTP Request Monitor	Louis Chen	Hao Jin

Worker Pool Auto Scaler	Hao Jin	Louis Chen
Documentation	Hao Jin & Louis Chen	

10.0 Revision Control

The web server has been revision controlled in GitHub. The following table outlines the information of contributors who have co-worked during the development process.

Contributor	**Email**	**GitHub ID**
Hao Jin	hao.jin@mail.utoronto.ca	erjin
Louis Chen	louis.chen@mail.utoronto.ca	louistjchen

The repository is currently set in private mode. However, it will be accessible to the public at https://github.com/louistjchen/Photo-Web-Server sooner or later.