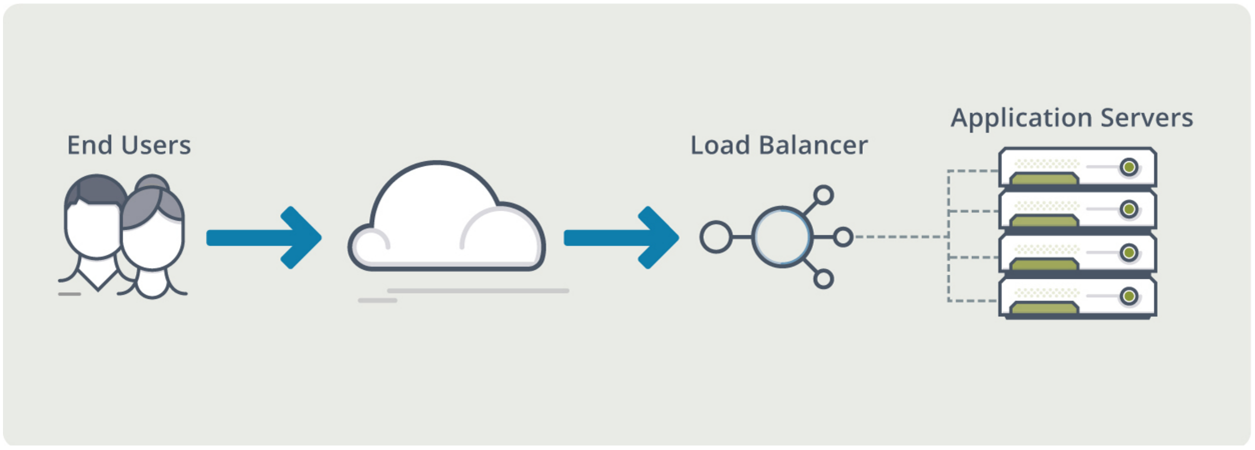
ECE1779: Introduction to Cloud Computing

Fall 2020

Assignment 2

Dynamic Resource Management



Due Date

November 16th, 2020

Objective

This assignment will expose you to the following AWS technologies: EC2, RDS, S3, CloudWatch, and ELB.

Marking Scheme

Component	Requirements		Points
Manager UI	Providing the appropriate menus, bars and buttons for the manager to control the system.		**5**
Basic Cluster Operations	The workers are registered with the the ELB correctly	2	**7**
	HTTP requests to the load balancer should be correctly forwarded to the workers	2	
	Workers should detect the faces with masks in the images	1	
	Saving images on S3	1	
	Saving user data on RDS	1	
Manager Functionality	Display instance details for each worker	1	**8**
	Display performance charts for each worker	1	
	Ability to add workers manually	1	
	Ability to remove workers manually	1	
	Option for deleting all data	1	
	Option for stopping the manager	1	
	Having a page for tuning the auto-scaler policy	2	
AutoScaling	Implementation of autoscaling policy	3	**7**
	Autoscaling policy converges*	4	
	*Your auto scaling policy should converge, meaning if the workload is constant, you shouldn't be adding or removing workers.		

Documentation	Having proper instructions (with screenshots) 1 System architecture (discussion & diagrams) 2 Database schema (discussion & diagrams) 2 Complete results section (discussion & graphs) 3	**8**
Total		/35

Description

In this assignment, your task is to extend the application you developed in assignment 1 into an elastic web application that can resize its ***worker pool*** on demand. The worker pool is the set of EC2 instances that run the user-app. Your application should consist of two parts:

The ***user-app*** *(*shown in the figure as the "Application Servers") is what you already have from Assignment 1 with some changes that will be discussed in the Requirements section. To reiterate, it should have the following features:

1. Login panel. Access to the application should require authentication. Include support for password recovery.
2. User management. The administrator should be able to create additional user accounts. This is not a public application. The only way to create new accounts should be for the administrator to login and create the account. User accounts should have access to all features in the web site, with the exception that they should not be able to create or delete user accounts. All users should have the ability to change their password.
3. Mask detections. Authenticated users should be able to run mask detection on images they specify. A user should be allowed to upload an image by selecting it from their local file system, or by typing a URL for a location on the web from which the image can be downloaded. After an image is supplied, your application should display the number of faces that are detected, the number that are wearing masks, and show a new version of the image with red rectangles drawn around the faces of people who are not wearing masks and green rectangles drawn on the faces of those that are.
4. Upload history. Authenticated users should be able to browse lists of previously uploaded images. Split the history into 4 lists: images with no faces detected, images where all faces are wearing masts, images where all faces are not wearing a mask, and images where only some faces are wearing masks.

The ***manager-app*** used by the *manager* which controls the worker pool. The UI of this application should support the following functionality:

1. Listing workers: Manager should be able to see a list of workers in real time. For each worker, two charts should be displayed in the list:
    a. First chart, showing the total CPU utilization of the worker for the past 30 minutes with the resolution of 1 minute.
    b. Second chart, showing the rate of HTTP requests received by the worker in each minute for the past 30 minutes. The chart has the rate (HTTP requests per minute) on the y-axis and time on the x-axis.
2. A chart on the main page, showing the number of workers for the past 30 minutes.
3. Link to load balanced user-app entry URL: Your manager-app should display the load-balancer (defined later in the document) DNS name.
4. Manually changing worker pool size: There should be two buttons for manually growing the worker pool size by 1 and shrinking the worker pool size by 1.
5. Configuring a simple auto-scaling policy by setting the following parameters:
    a. CPU threshold (**average for all workers** over past 2 minutes) for growing the worker pool
    b. CPU threshold (**average for all workers** over past 2 minutes) for shrinking the worker pool
    c. Ratio by which to expand the worker pool (e.g., a expanding ratio of 2.0 doubles the number of workers).
    d. Ratio by which to shrink the worker pool (e.g., a shrinking ratio of 0.5 shuts down 50% of the current workers).
6. Stopping the manager: A button that terminates all the workers and then stops the manager itself.
7. Deleting all application data: A button to delete application data stored on the database as well as all images stored on S3.

Your application will have two additional components called ***load-balancer*** and ***auto-scaler***:

load-balancer sits in front of the worker pool and routes the incoming http traffic among the workers in the worker pool. There is **no need to implement this yourself** and you should use AWS Application Load Balancer as the load-balancer. The load balancer provides an HTTP endpoint that forwards requests to all the workers in a load balanced fashion.

auto-scaler is a component that automatically resizes the worker pool based on the load. It should monitor the load by watching the average CPU utilization of the pool of workers using the AWS CloudWatch API. When the CPU utilization exceeds (or drops below) the configurable threshold, auto-scaler should use the **AWS Elastic Load Balancing API** to add and remove workers from the load balancing pool as you reconfigure your worker pool in response to load. You can write a python program on the manager instance, that runs in the background and monitors the worker pools, reads the auto-scaling policy info set by the manager interface from the database and re-scales the worker pool based on load.

- There should be no need to manually restart the auto-scaler every time policy is changed.
- Check for the CPU utilization of workers every 1 minute.
- Do **NOT** use the AWS Auto Scaling feature for this assignment.
- **Limit the maximum size of the worker pool set by auto-scaler to 8 and the minimum to 1**

Requirements and Notes for Implementations

1. All photos (processed, unprocessed and thumbnails) should be stored in **S3**.
2. Store information about user accounts and the location of photos owned by a user on **AWS RDS**. Do **not** store the photos themselves in the database. It is up to you to design the database schema, but make sure that you follow design practices and that your database schema is properly normalized. It is advised that you use the smallest possible instance of RDS to save on credit. **Important:** RDS is an expensive service that can cost several dollars per day for larger instances. To ensure that you do not run out of credits, use one of the smaller instance types (e.g., burstable db.t2.small class) and remember to stop your RDS instance when you are not actively using it.
3. You should get the measurements for CPU utilization from AWS CloudWatch. You have to enable **detailed monitoring** to get 1 minute resolution instead of the default 5 minute resolution.
4. You also should use CloudWatch Custom Metrics to create a HTTP request rate chart for the manager-app. Also look at boto3 documentation on how to publish these metrics.
5. If you don't want to copy your AWS credentials to the manager and every worker instance or if you are using temporary credentials, you can use IAM Roles to give permissions to your EC2 instances directly.

   Refer to this parts of AWS documentation:

   - https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_use_switch-role-ec2.html
   - https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html

   In short:

   A. Create an AMI Role for EC2 with permission to use S3
   B. Attach the AMI Role to your EC2 instances
   C. Use EC2 meta-data endpoint from inside the EC2 instance to retrieve temporary access keys (for example by requesting http://169.254.169.254/latest/meta-data/iam/security-credentials/{role-name})

6. Use a separate EC2 instance of type t2.small or any size fits your app to run each worker node. All worker nodes should run the *user-app* functionality.
7. The manager-app should run on a single EC2 instance of type **t2.small** separate from the worker instances.
8. Design the auto scaling algorithm to be smooth and reliable. A badly designed algorithm will make momentary decisions that will not converge.

   An example for an **unreliable algorithm**: When the load goes up, the algorithm correctly decides to increase the size of the worker pool, but before the new workers start and accept requests, it tries multiple times to increase the worker pool size again. When the new workers start accepting requests, the average CPU utilization goes way below the threshold and the algorithm decides to shrink the worker pool, again multiple times and so on. Such an algorithm oscillates between few and too many workers and will never converge.

9. In order to use AWS CloudWatch, you need to enable monitoring of your EC2 instances.
10. It is strongly recommended to use gunicorn or uwsgi to start your Flask web application.
11. This is from assignment 1: Follow basic web application security principles.
    a. All inputs should be validated with reasonable assumptions (for example, do not let the user upload unreasonably big files, do not let the user use a username with more than 100 characters). Also, do not rely on browser-side validation and do the validation on the server side too.
    b. User passwords should not be stored in clear text in the database. Instead, store the hash of the password concatenated with a per-user salt value. Details for how to do this are available here.
    c. Access to stored photos should be restricted to the user that has uploaded them.
12. This is from assignment 1: To simplify testing, your application should **also (in addition to your web interface)** include two URL endpoints as APIs that makes it possible for the TA to automatically register users and upload photos. These endpoints should conform to the following interfaces:

    Register:

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

    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
```

a. These endpoints should generate appropriate HTTP response codes in success and error conditions as well as responses in JSON format
b. Sample forms that conform to the specification above are available here and here.
c. A load generator that conforms to the upload interface is available here (Note that the load generator should be run using python 3.7+). You can use the load generator to test your application. **Warning:** the load generator creates a lot of network traffic. To minimize bandwidth charges, you can use the load generator inside EC2 only.
d. These endpoints should generate responses in the following JSON format:
   In case of failure:

```
        {
                "success": false,
                "error": {
                        "code": servererrorcode,

                                "message": "Error message!"
                        }
        }
```

   In case of success for the ***register*** interface:

```
        {
                "success": true
        }
```

   In case of success for the ***upload*** interface:

```
        {
                "success": true,
                "payload": {
                                "num_faces": number,
                                "num_masked": number,
                                "num_unmasked": number
                }
        }
```

To run the load generator, run the python script using:

```
    python3.7 gen.py upload_url username password
upload_rate(upload_per_second)     image_folder number_of_uploads
```

for example:

```
      python3.7 gen.py http://9.9.9.9:5000/api/upload user pass 1
./my-photos/ 100
```

13. All code should be properly formatted and documented.

Deliverables

We will test your application using your AWS account. For this purpose, you should pre-load the manager-app on an EC2 instance. Please **suspend** the instance when you submit your project to prevent charges from occurring while the TA gets around to grading your submission. Make sure to not restart the instance from the moment you submit your project. Also make sure no other worker instances are running when the manager-app instance is suspended.

You should write a shell or bash script called **start.sh on the manager** that initializes the **manager-app and auto-scaler**. This script should be put in the Desktop folder inside the EC2 instance.

There is no start/restarting the load balancer and there is no need to re-create it in the start.sh script.

Once the manager starts, it should resize the worker pool to 1.

Manager web application should run on port **5000** and be accessible from outside the instance. Make sure that you open this port on your EC2 instance. Documentation about how you can open the port can be found here.

Submit the assignment only once per group. Clearly identify the names and student IDs of group members in the documentation. In your report explain what each member of the group did towards the design and implementation of the project.

You will demo your project to a TA and the TA may ask questions about the parts each member did so having all the members of the group present at the project demo is mandatory.

To submit your project, upload to Quercus a single tar file with the following information:

1. User and developer documentation in a **PDF** file (documentation.pdf). Include a description of how to use your web application as well as the general architecture of your

application. Also include a figure describing your database schema. Click here for tips on how to write documentation. Your documentation will be marked based on how cohesive it is and how well you are able to explain the technical details of your implementation.

2. In the report, include a section called **Results.** This section should show that your auto-scaler component works.

   Design test cases and scenarios that demonstrate the functionality of your auto-scaling algorithm and it's reliability as load increases and decreases. During each scenario, stick to one auto-scaling policy (thresholds and ratios) and try to demonstrate that the size of the worker pool reasonably changes in response to changes in the incoming load. Show that your algorithm converges if load stays at a particular level for a few minutes. Use charts showing the load and worker pool size over time to present your results.

3. In addition to the documentation, put the name and student ID of each student in a separate line in a text file named group.txt.
4. If you are using the AWS Educate account (link provided in the class) with the $100 credit, then we don't need your AWS credentials as we already have access to your AWS console.

   Otherwise, please send us your AWS credentials in a text file (named credentials.txt). We will need these to log into your account. You can create credentials with limited privileges using AWS IAM if you don't want to share your password with the TA; however, make sure that you include permissions to start and stop EC2 instances. Test the credential to make sure they work.

5. Key-pair used to ssh into the EC2 instance (named keypair.pem).

   Do not forget to send the .pem files required for ssh-ing into your VM's.

6. Anything else you think is needed to understand your code and how it works.

Resources

Amazon provides free credits for students to experiment with its cloud infrastructure, including EC2. To apply for an educational account go to Amazon Educate.
The following video tutorials show how to :

- Create an EC2 instance
- Connect to an instance using a VNC Client.
  - using the command: ssh -i key.pem ubuntu@IP -L 5901:localhost:5901 you should first do port forwarding using an ssh tunnel to your instance.
  - If you are using Putty in windows, you can add a tunnel after you are successfully connected to your instance. Right click on the top of PuTTY console window and select Change Settings.... On the left menu, select Connection->SSH->Tunnels.

    Enter "5901" on the Source port and "localhost:5901" on the Destination fields and then click Add. The tunnel should be added to the list. Now click Apply.
  - After creating a tunnel, you can use any VNC viewer to connect to "localhost:5901" using the password "ece1779pass"
- Suspend an instance

To help you get started, an AMI (id: ami-0bf618774e7879c6a) is provided with the following software:
Note: the AMI is only available in the N. Virginia AWS Region

- Python 3.8
- PyCharm IDE
- Firefox
- MySQL Server (root password ece1779pass)
- mysql-workbench
- vncserver (password ece1779pass)
- ImageMagick
- Python libraries pre-installed: flask, boto3, mysql-connector-python, opencv-python, torch, torchvision.
- gunicorn
  This is a high performance server for Python-based applications. For an example of how to run it, look at the run.sh file inside Desktop/Databases
- The following directories in the Desktop:
  - **Databases:** A PyCharm project with all examples from the databases lecture
  - **AWS:** A PyCharm project with all examples from the AWS lecture
  - **FaceMaskDetection:** mask recognition stater code