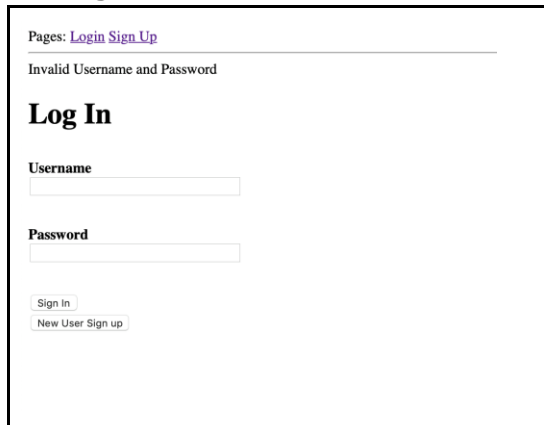


AWS Dynamic Resource Management on Web Applications Documentation

1.0 User-App

Below is snips of our webpages for the user-app. We make use of inherited templates via jinga2 and always maintain a navigation bar at the top of the page for easy access

1.1 Login/Index



Pages: [Login](#) [Sign Up](#)

Invalid Username and Password

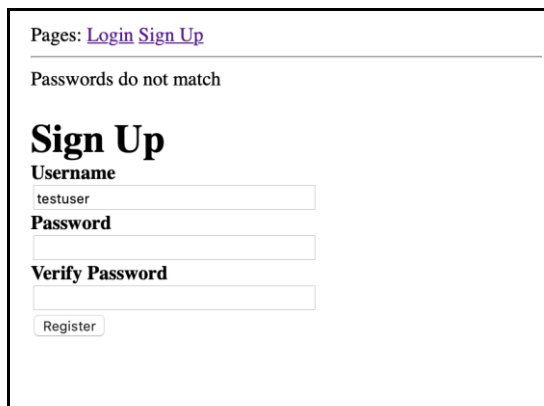
Log In

Username

Password

1.2 New User Sign up

When a new user registers they are prompted to provide a username and password. After verifying the input is valid the password is hashed using `generate_password_hash` from `werkzeug`. This hashes the password with a salt produced by `generate_password_hash` function and returns both the salt used in the hash and the hash value itself as a string which is stored in our sql database on RDS. When a user subsequently logs in this value is retrieved from the database and provided to the `check_password_hash` function along with the user submitted password for validation.



Pages: [Login](#) [Sign Up](#)

Passwords do not match

Sign Up

Username

Password

Verify Password

1.3 Upload Image

The upload image page allows users to upload images to the database. The file extensions are restricted to typical image format (.png, .jpg, .jpeg) to ensure that only images are uploaded. Once file type is validated, the file used to produce a thumbnail produced by resizing the image using the pillow module and text detected version of the image. The original, thumbnail and text detected version are saved to S3 using appropriate prefixes to distinguish between the 3 files. The image name is stored within the upload table for subsequent retrieval.

[Pages: Browse Upload Logout](#)

Welcome testuser

Upload

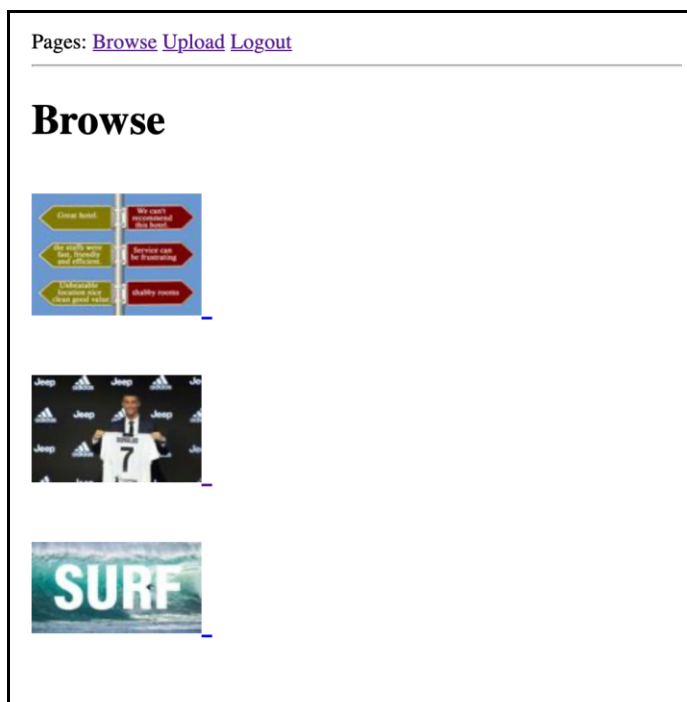
Choose File

no file selected

Submit

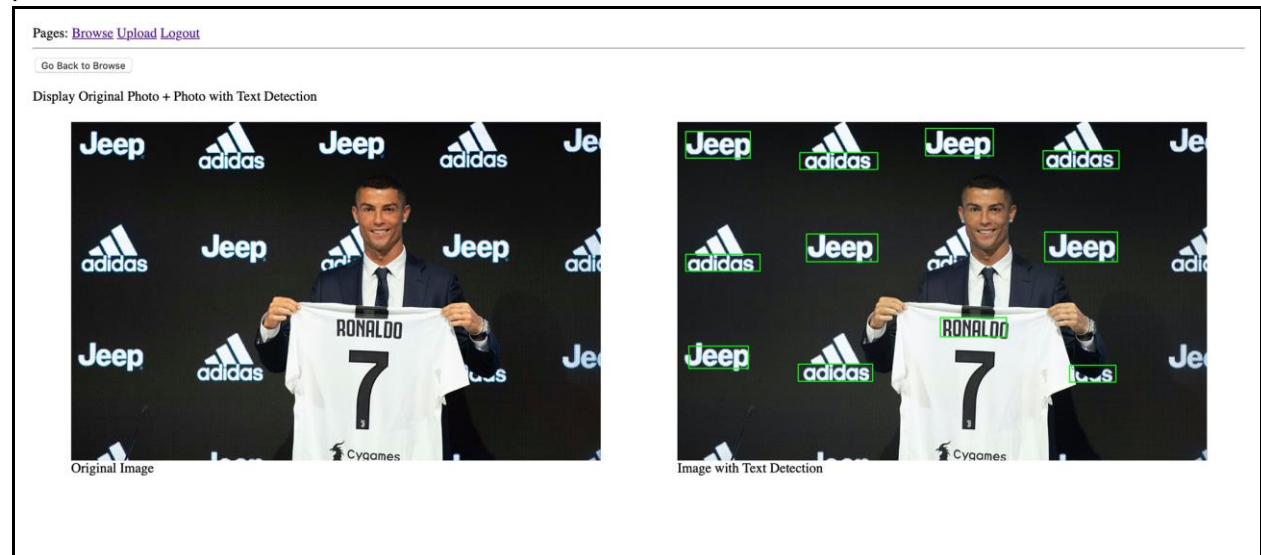
1.4 Browse Images

Users can see the images they have uploaded here as thumbnails. Clicking one of the thumbnails displays the full size image alongside an image with any text detected in green boxes as described in section 1.5.



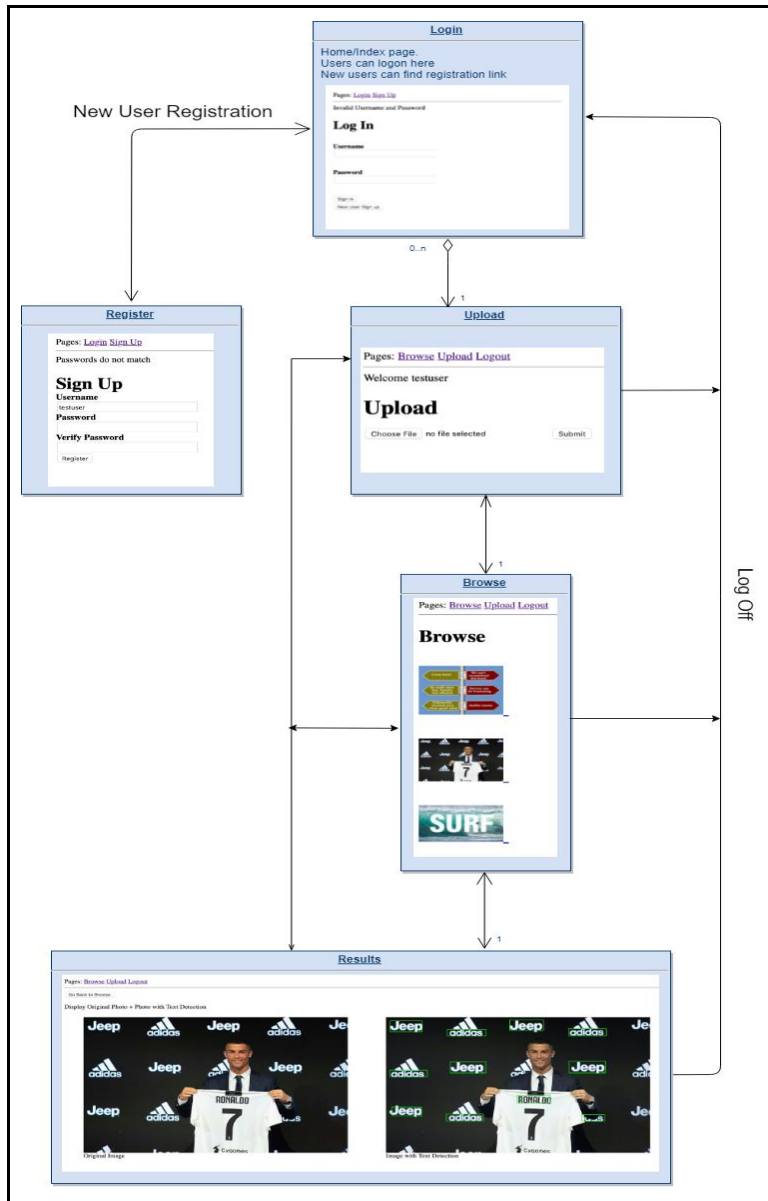
1.5 Results

This page displays the results of the text detection and displays the before/after side by side. The text detection is performed using code provided by pyimagesearch which utilizes opencv to perform text detection.



1.6 Webpage Flow

Below is the general flow of the webpages for each instance of our userapp



2.0 Database Schema

Our database in MySQL now 3 tables represented below in MySQL diagram which is handled by AWS RDS.

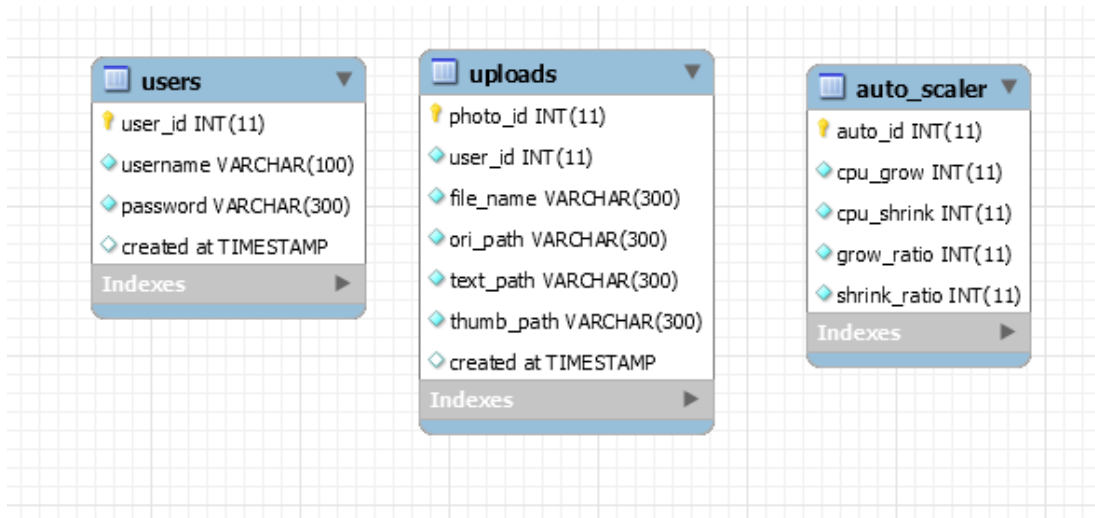


Table: Users

Column Name	Data Type
user_id	int
username	Variable char
password	Variable char
Created at	timestamp

Table: autoscaler

Column Name	Data Type
auto_id	int
cpu_grow	int
cpu_shrink	int
grow_ratio	int
shrink_ratio	int

Table: Uploads

Column Name	Data Type
photo_id	int
user_id	int
file_name	Variable char
ori_path	Variable char
text_path	Variable char
thumb_path	Variable char
Created at	timestamp

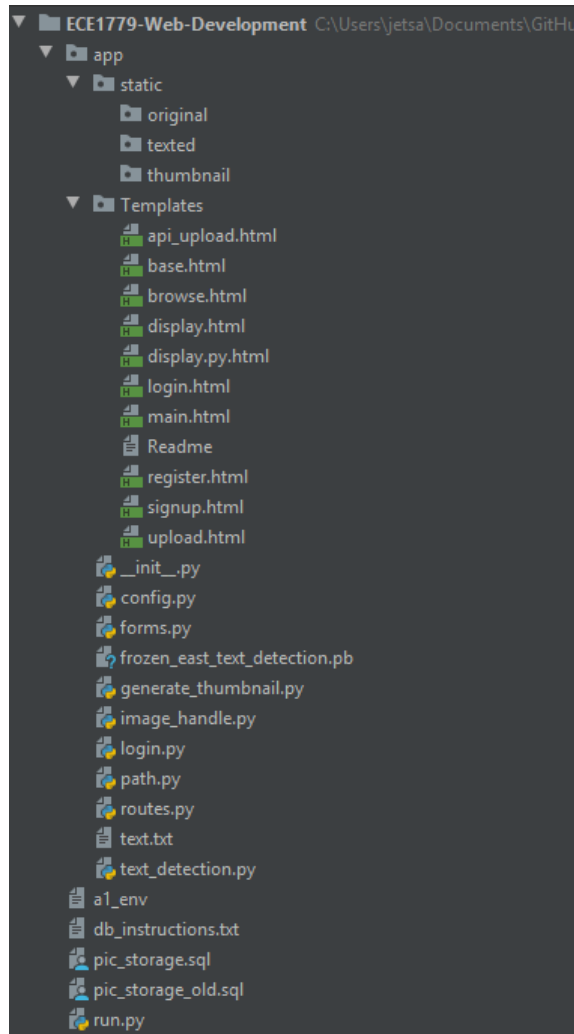
3.0 Source Code File Structure

Our UserApp source code file structure.

Templates : contains all html templates

App : contains all python codes

Static: contains all saved photos



routes.py : Contains all the url endpoints used by Flask to generate responses for http requests

config.py: Contains configuration parameters for mysql database on RDS

foms.py: Contains flask_wtf form objects for registration and login

test_detection.py: Contains functions used for generating image with text detected

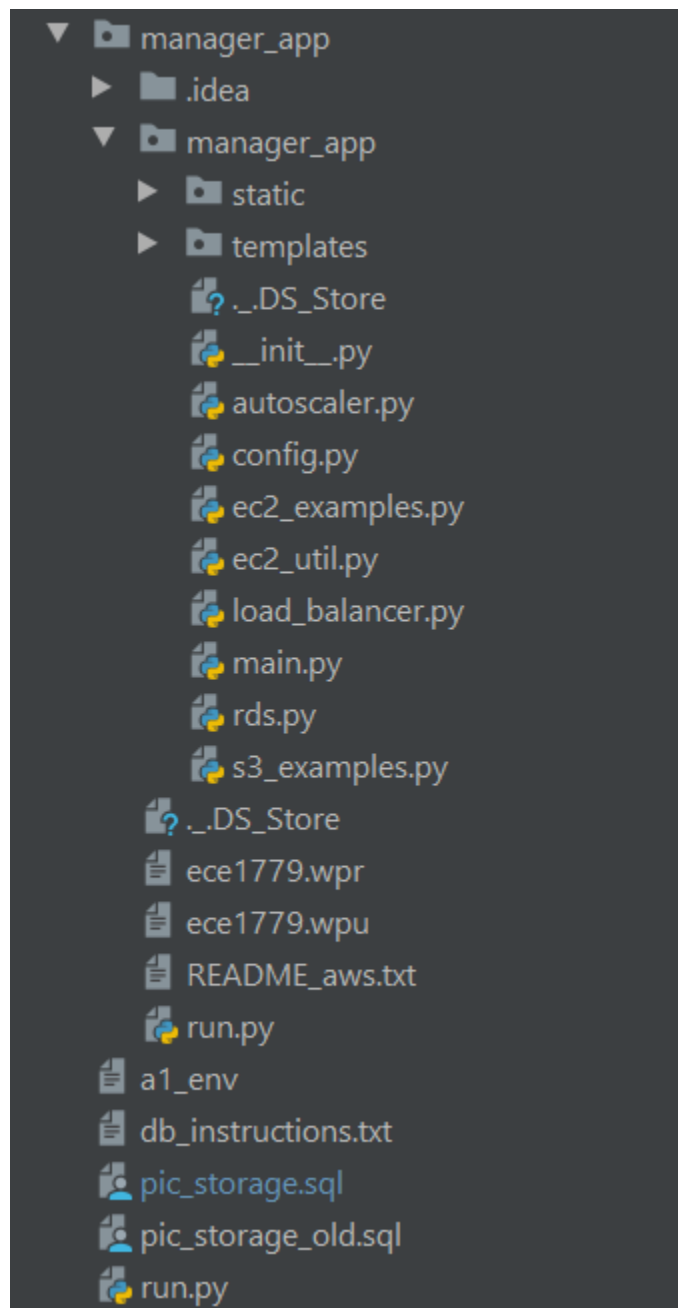
login.py: Contains functions for authenticating and registering new users

path.py: Contains functions for determining file paths for saving images on the local machine

generate_thumbnail.py: Contains functions used for generating thumbnails for images

image_handle.py: Contains functions that save files to an S3 bucket

Our Manager App source code:



autoscaler.py : Contains autoscaler code

config.py: Contains configuration parameters for mysql database on RDS, ec2 AMI parameters, s3 parameters

ec2_examples: Contains main routes for manager app

ec2_utils.py: Contains utility functions for creating, removing and monitoring instances

load_balancer.py: Contains functions for adding and removes instances to load balancer

rds.py: Contain functions for interacting with rds

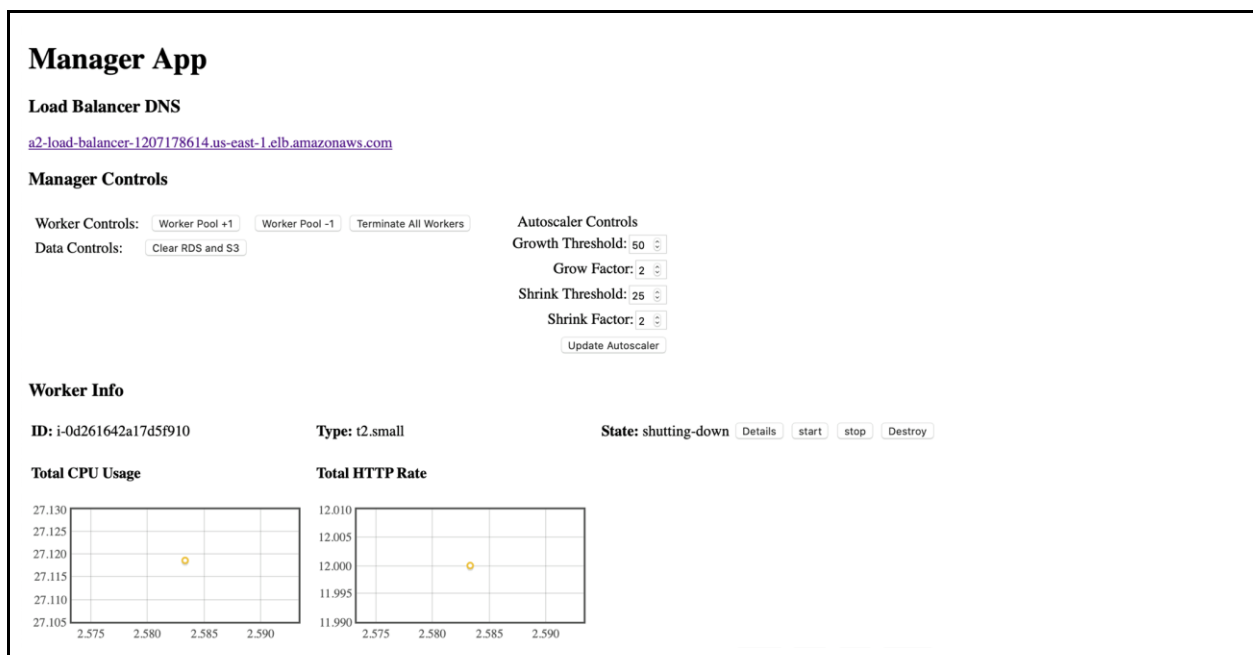
s3_examples.py: Contains functions for interacting with s3

4.0 Manager App

The Manager App is a separate instance that starts the autoscaler function and provides an interface to monitor and control instances running the user-app. It does the following things:

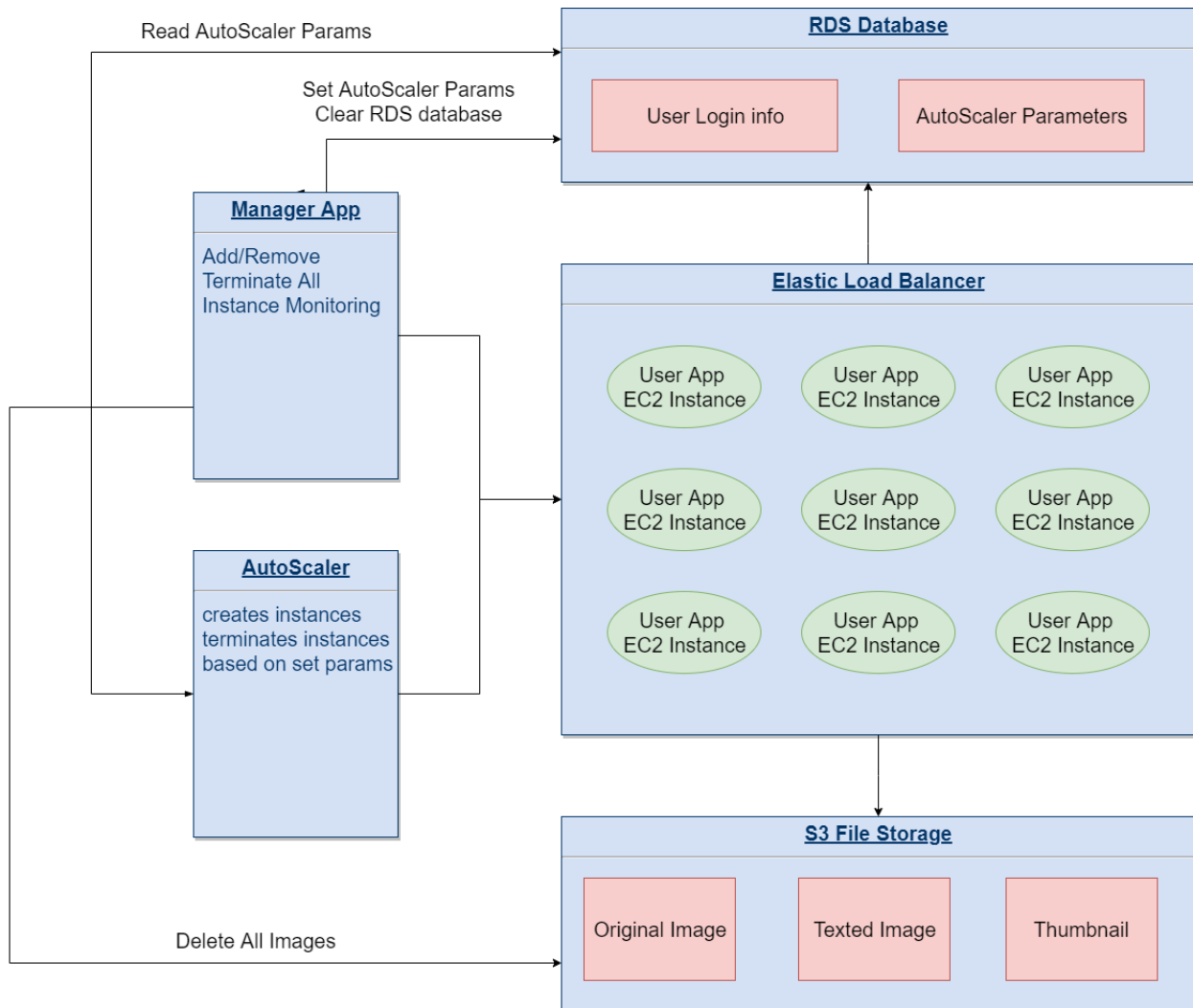
1. General worker control
 - a. Add worker
 - b. Remove worker
 - c. Terminate all workers
2. Specific worker info/monitoring/control
 - a. Worker id/type/status
 - b. Monitor CPU usage and http rate
 - c. start/stop/destroy instances
3. Clear RDS and S3 data
4. Set Auto Scaler parameters
5. Stop Manager app

All controls are placed on one page to allow user of the manager easy access to stats and controls



4.1 Flow diagram for Scaling Control

The diagram below illustrates how the manager app interacts with different components of the overall app.



5.0 Autoscaler

5.1 Autoscaler Algorithm

The autoscaler operates as a separate thread from the manager app. The function runs a loop which continuously monitors instance cpu usage health and adjusts the number of active workers based on the autoscaling parameters specified by the manager app. Initially the autoscaler creates one worker. On each subsequent iteration the autoscaler will measure the average cpu usage of all healthy instances and based on the thresholds set by the manager app scale the worker pool accordingly. In the case there are no healthy workers, one will be created.

As a heuristic to prevent overscaling, the autoscaler makes sure that all healthy instances have valid cpu statistic (i.e. at least one cpu measurement) before changing a pool size. This ensures that each healthy instance has been initializing and is taking some amount of the load, so that a valid average measurement is read before scaling again. As an added check the autoscaler also does a check to ensure that scaling only performed when newly added instances take on load by comparing differences in the average cpu usage between loop iterations.

5.2 Autoscaler Results

5.2.1 Test Case 1

This test has the following parameters

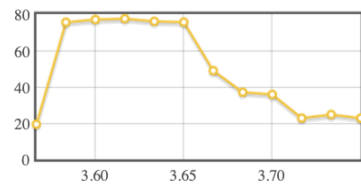
Grow Factor = 2

Shrink Factor = 2

Grow Threshold = 60

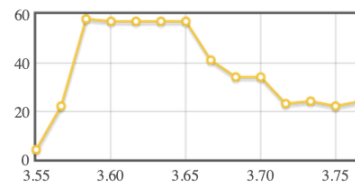
Shrink Threshold = 10

Total CPU Usage



ID: i-0bf8dfc0facfb4863

Total HTTP Rate

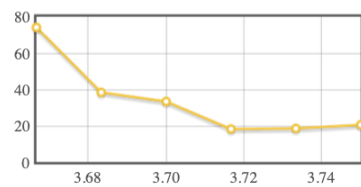


Type: t2.micro

State:
running

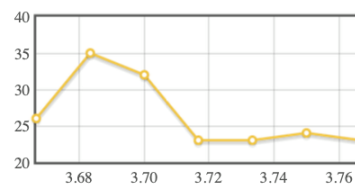
[Details](#) [start](#) [stop](#) [Destroy](#)

Total CPU Usage



ID: i-098c1a254ed31a9ae

Total HTTP Rate

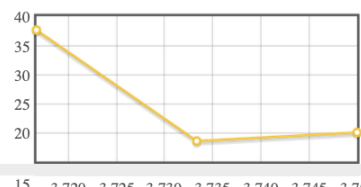


Type: t2.small

State:
running

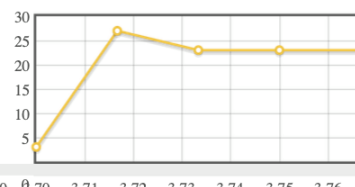
[Details](#) [start](#) [stop](#) [Destroy](#)

Total CPU Usage



ID: i-0e4d973323a3ea875

Total HTTP Rate

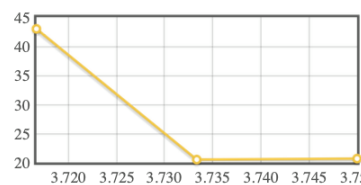


Type: t2.small

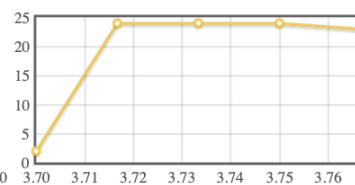
State:
running

[Details](#) [start](#) [stop](#) [Destroy](#)

Total CPU Usage



Total HTTP Rate



5.2.2 Test Case 2

This test has the following parameters

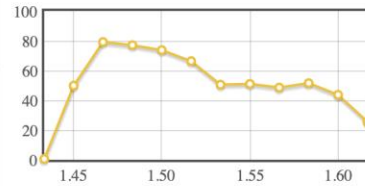
Grow Factor = 2

Shrink Factor = 2

Grow Threshold = 80

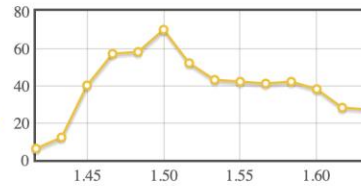
Shrink Threshold = 20

Total CPU Usage



ID: i-01d4d8fe51eb592fd

Total HTTP Rate

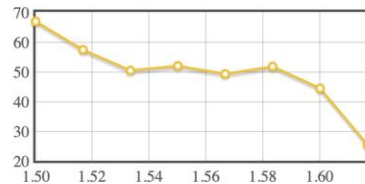


Type: t2.small

State: running

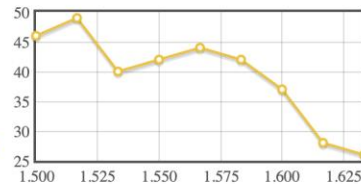
[Details](#) [start](#) [stop](#) [Destroy](#)

Total CPU Usage



ID: i-056a42cbdae3decf1

Total HTTP Rate

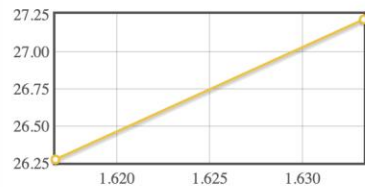


Type: t2.small

State: running

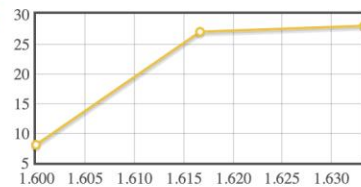
[Details](#) [start](#) [stop](#) [Destroy](#)

Total CPU Usage



ID: i-0de5013eb9f93aafd

Total HTTP Rate



Type: t2.small

State: running

[Details](#) [start](#) [stop](#) [Destroy](#)

Both tests were done using the provided load generator. Total CPU usage is always high during the beginning when new instances are being launched leading to an initial upsurge in the number of EC2 instances created. Eventually, in both bases, they will converge to a steady value

