ECE 1779 Project 2 Documentation

Group 3

Part 1. Project Abstract

In this project, we modified assignment 1 and implemented scaling functionality to our website. Instead of local file system and local database, we utilize AWS s3 and RDS database to store image contents and key-value pairs respectively. Each memcache is implemented as a separate EC2 node. This memcache will scale up and down respectively according to the configurations set in manager-app. The website will automatically distribute the keys-image pair to respective memcache nodes according to MD5 hashing. Figure 1 shows the overview of our system. Detailed workflows will be presented in the part 2.

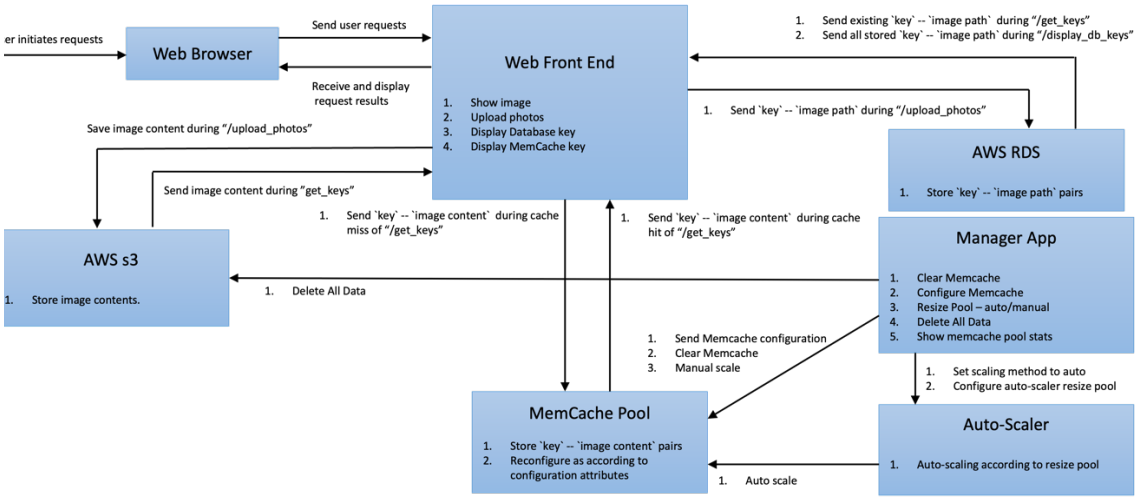


Figure 1. Overall System Architecture

Part 2. Application Components and Functions

1. Front End: Manages user requests
   a. Upload photos: Allows users to upload a photo along with a key. The key-filename pair is stored in the AWS RDS while the file itself, i.e., the image content is stored in AWS S3. Detailed workflows are shown in figure 2.

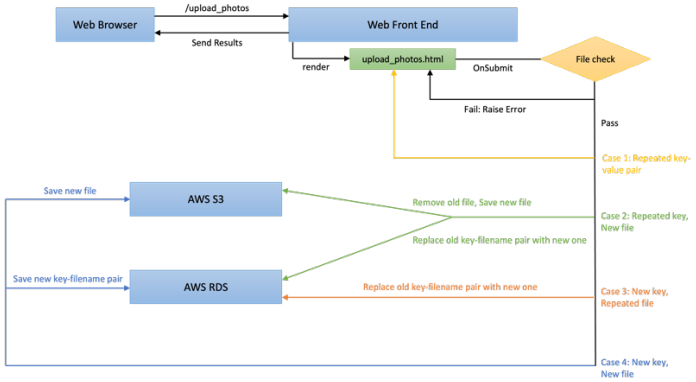


Figure 2. /upload_photo request workflow

   b. Get a photo with key: The full workflow when the front end receives a get photo request is displayed in figure 4. There are three possible cases when a key is an input by the user. Their respective workflows are shown below. Numbers indicate their steps labelled in figure 3.

Cases	Workflow
Key in MemCache	Step 1 → Step 2: Yes → Show image
Key in RDS	Step 1 → Step 2: No → Step 3: Yes → Step 4 → Step 5 → Step 6 → Show image

Key not found	Step 1 → Step 2: No → Step 3: No → "Unknown Key!"

Table 1. /get_photos request workflow

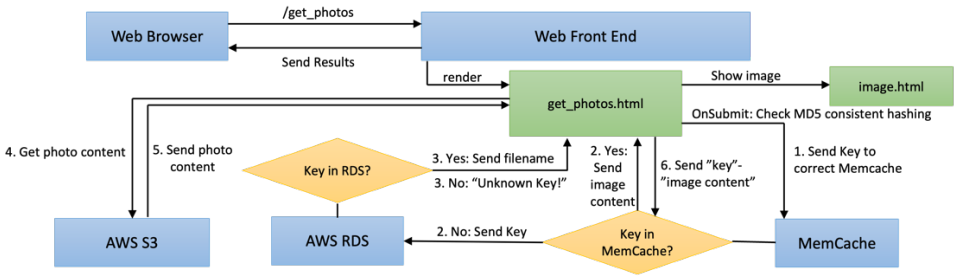


Figure 3. /get_photos request workflow

c. Display current database keys: Front End requests the database for all key-filename pairs stored. There shall be a one-to-one relationship between each of these pairs to the image contents stored in the local file system.
d. Display current Memcache keys: Front End request MemCache for all keys stored in MemCache. The order starting from the top represents the least recently used keys down to the most recently used keys.

2. RDS: Details about the database will be discussed in the **Database Schema** session

3. Manager App:
   a. Delete all data: Delete all image content in AWS S3, all key-filename pair in AWS RDS database, and clear all currently running memcache node.
   b. Clear memcache data: Clear all currently running memcache node.
   c. Configure Memcache attributes: Two attributes can be set, being the capacity of the cache (in MB) and the replacement policy (Random Replacement (*RR*) / Least Recently Used (*LRU*)). The same attributes will be configured for all running memcache nodes. For new memcache nodes due to scaling up, they will all be configured to the most recently set configurations.
   d. Display Memcache stats chart: The following data are shown for the last 30 minutes at 1-minute granularity. 1. Number of nodes 2. Miss rate 3. Hit rate 4. Number of items in cache 5. Total size of items in cache. Since the period of each cloud watch data put by memcache is 5 seconds, the statistics are averaged over 1-minute granularity.
   e. Resize Pool: There are two methods to resize the pool, being manual mode and auto scaling.
      i. Manual mode: Resizing of the memcache pool depends on user set values. Whenever a new node is running, its configuration data is automatically set according to the most recent memcache attributes.
      ii. Auto-scaling mode: Four attributes can be configured as shown in table 2. Whenever the user sets scaling to this mode, the following attributes' values will be posted to auto-scaler for handling all auto-scaling jobs.

Attributes	
Max Miss Rate Threshold	If the average miss rate over the past minute is larger than this attribute, scale up instances according to Ratio by which to expand the pool.
Min Miss Rate Threshold	If the average miss rate over the past minute is smaller than this attribute, scale up instances according to Ratio by which to shrink the pool.
Ratio by which to expand the pool	As mentioned above
Ratio by which to shrink the pool	As mentioned above

Table 2. Auto Scaling Attributes

4. Auto-Scaler: Auto scale MemCache pool according to the attributes in Table 2 set by Manager-App. Each MemCache is a new AWS EC2 node. The miss rate is checked every one minute, and averaged through all nodes.

5. MemCache: Memcache functionality is almost the same as assignment 1. It stores part of the data in a "key"-"image content" form according to locality. It can improve the performance of the code by reducing the number of times to retrieve data from the database. The two attributes are the same as assignment 1, being Capacity and Replacement Policy

   Each running MemCache will send its stats to AWS CloudWatch every five seconds. It will contain the current No_items, the current Total_size, the No_requests received during this five seconds, the Miss_rate, Hit_rate over the last five seconds, which is the same as in Assignment 1.

6. AWS S3: Stores the content of the images. Various workflows relate to this can be seen in Figure 2.

Part 3. Database Schema

In Assignment 2, we only need one table, i.e. the Image table in figure 5 from Assignment 1. It stores all key-filename pairs of all images uploaded. It is used to find a specific image given a key. Note that, we only store the filename as values instead of the image content.

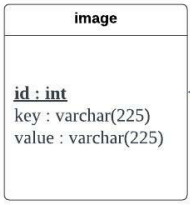


Figure 4. Database ER diagram

Part 5. Design Decisions

- **MemCache EC2 pool**
  Whenever the memcache EC2 pool is scaling up or down, it will block all new requests and wait for the new memcache EC2 pool to be setup. Manager app will make sure a dummy connection to all memcache nodes are successful before starting to rebalance keys. The downside to this method is obviously its damage to performance.
  An alternative method is to allow the system to continue operating, but without special treatment, it may lead to an unstable state. However, to carefully monitor all this state, it may take a lot of time to make sure all components work correctly. It is true that the throughput can be increased, but due to our time limitations we did not choose this method.
  As a result, we chose the blocking method.
- **EC2 creating vs starting**
  Whenever a new memcache EC2 node is needed, the system can either create a new EC2 instance from a set AMI, or start an already created but in "Stopped" state instance.
  Starting from a set AMI can save us the most costs, but it may take more time for it to run. In our experiments, the time from creating an EC2 instance until it successfully runs is around 1.5-2 minutes. On the other hand, just starting a "Stopped" instance takes around 30 seconds, though it may generate slight costs due to its allocation of EBS memory.
  Since in this project, we are making scaling decisions every minute, it is reasonable to choose the latter method as it can guarantee us a workable application. In real world scenarios, the first method may be more useful if scaling in 30 minute or hour scales.
- **Flask instances and EC2 instances**
  We chose to run the front_end, auto_scaler, and manager_app component on the same EC2 instance of port 5000, 5001, 5002 respectively. For every memcache, we use a different EC2 instance.

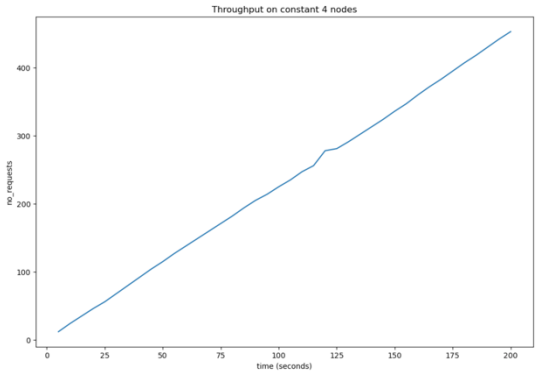
Other design choices include running all components on different EC2 machines, which would be better for fault tolerance or server failure. However, it would quite tedious since we then have to consistently update the locations of all these flask instances on different EC2 instances. Furthermore, the small size of each flask instance means that even running all three components on the same EC2 instance will still work fine without exceeding machine's calculation efficiency.
As a result, we chose to implement all three front_end, auto_scaler, and manager_app components on the same EC2 instance

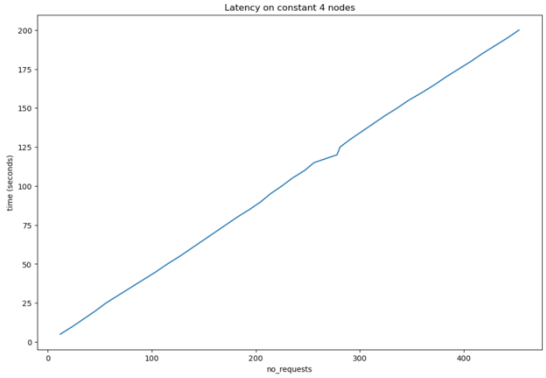
Part 6. Results

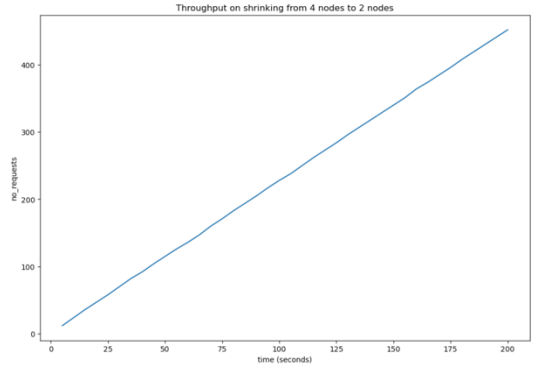
Auto-Scaler Component: Our auto-scaler component is set to 'mode' = 'auto on start. It will always monitor the status of all memcache according to Auto-Scaler Component mentioned in part 2. If the manager_app sends a request of switching to 'manual' mode to auto-scaler, the auto-scaler will stop monitoring and auto-scale the nodes. The monitoring and checking of all memcache stats is run on a different thread, thus making sure that every minute it gets stats from CloudWatch correctly. In addition, this allows the auto-scaler to work correctly without restarting even after changing auto scaling configurations.

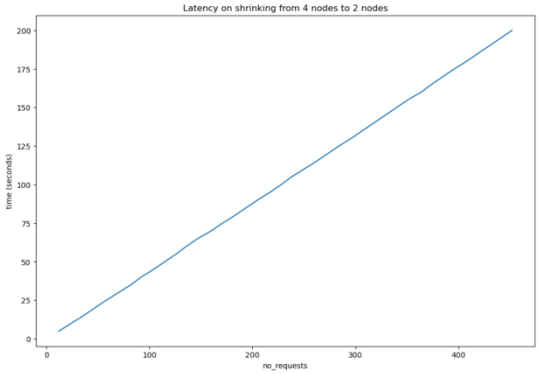
Manual-Scaling Testing Environments :

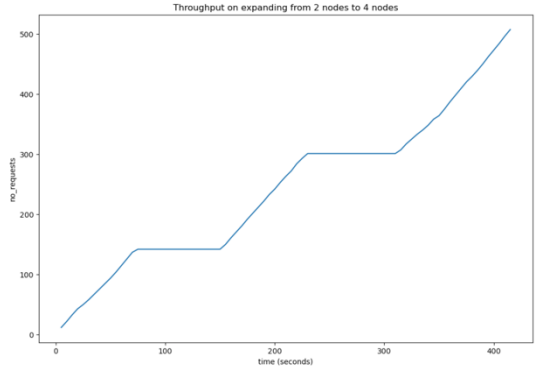
1. Average image size: 100kb per image
2. Memcache Capacity: 1000Mb; Memcache Policy: Random Replacement
3. Read:Write ratio: 50:50

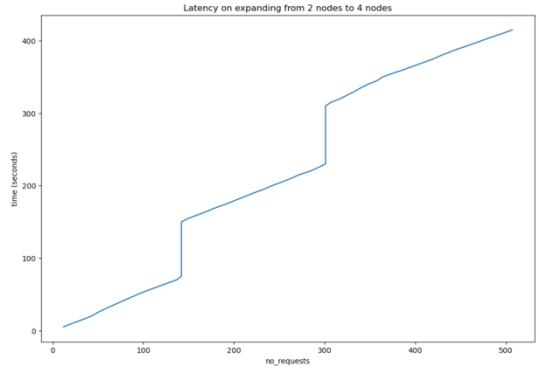






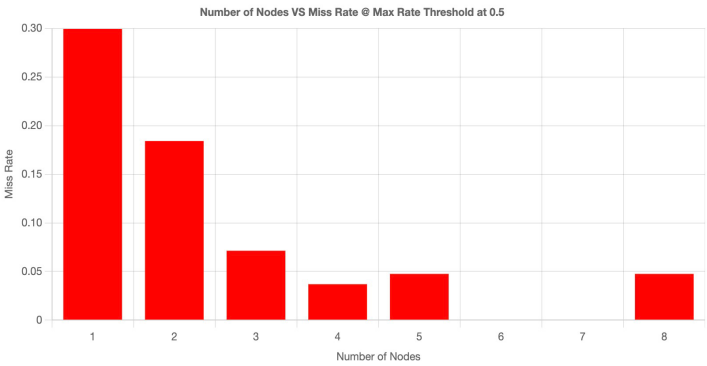




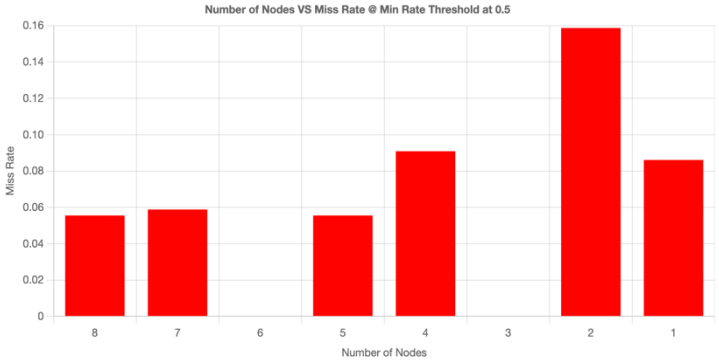


Auto-Scaling Testing Environments :

1. Image size: Randomly scattered between 60kb to 8mb
2. Memcache Capacity: 1000Mb; Memcache Policy: Random Replacement
3. Read:Write ratio: 50:50
4. Max Miss Rate Threshold: 0.5; Ratio to which to expand the pool: 2.0 for expanding pool
5. Min Miss Rate Threshold: 0.5; Ratio to which to shrink the pool: 0.5 for shrinking pool



Expanding



Shrinking

Part 7. Result Discussions

1. For constant 4 nodes, the result is expected that our system serves incoming requests smoothly.
2. For shrinking from 4 to 2 nodes, the result is expected as in our implementation, shrinking nodes does not block any operations. This is because when shrinking nodes, all needed to be done was rebalancing the keys, which shall not take up a significant time since our image size are quite small. We do not need to wait for the “stopping” instance to become “stopped” since in a sense they are no longer needed once scaled down. Therefore, there is minor blocking towards our application, which resulted in the smooth serving of requests.
3. For expanding from 2 to 4 nodes, the result is expected as in our implementation, we need to wait until new Memcache EC2 nodes to be ready to for any requests (described in design decisions in part 5). The testing seems to suggest that it takes around 1 to 1.5 minute to get a new node ready. Therefore, whenever we scale up, the system will stop serving incoming requests and wait until the new memcache node is fully setup.
4. For auto-scaling up, the result is expected as nodes expand, miss rate decreases. This is expected since the more nodes used, the miss_rate shall be lowered it is possible to save more data in memcache. This will also force the auto-scaler to scale down if miss_rate is too low. The data is shown over a 30 minute time duration. Our design test case is making sure that whenever the miss_rate is lowered to below Max Miss Rate Threshold, we put new images in and get them once to increase the miss_rate until it reaches the Max Miss Rate Threshold. This can force the auto-scaler to scale up and thus continue on our experiment.
5. For auto-scaling down, the result is expected as nodes shrinks, miss rate increases. This is expected since the less nodes used, the miss_rate shall be higher, which will then force the auto-scaler to scale up if miss_rate is too high. The data is shown over a 30 minute time duration. Our design test case is making sure that whenever the miss_rate exceeds the Min Miss Rate Threshold, we get a image that we previously saved in memcache to decrease the miss_rate until it reaches the Min Miss Rate Threshold. This can force the auto-scaler to scale down and thus continue on our experiment.
6. For auto-scaling the testing is done through manually uploading and downloading. As a result, the graphs are not as pretty. However, we can still see the trends. In addition, even though we are scaling up/down by 2/0.5 times, it is reasonable that there might be stats for other number of nodes since sometimes uploading stats from node to CloudWatch may not be properly done. Furthermore, when starting or stopping new instances, not all of them are done at once. Therefore there may be a 5 second duration where a instance supposed to be stopped is not stopped yet, thus resulting in a upload of stat to CloudWatch.

Part 8. Contribution Table

Yih CHENG	Finished Front_end implementation and Auto-Scaler
JiaWei MA	Finished Database configuration and Cloudwatch
Ho Wan LUO	Finish Manager-app