Assignment 2: Amazon EC2

Due Nov 21, 2022 by 12p.m. **Points** 26 **Submitting** a file upload
Available Oct 19, 2022 at 12a.m. - Nov 25, 2022 at 12p.m.

This assignment was locked Nov 25, 2022 at 12p.m..

Objective

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

*The following specifications are intentionally left less prescriptive at times, allowing you to make appropriate design decisions that you feel comfortable realizing in the allotted time. You'll be given the opportunity to discuss your design decisions in your report and in the demo. Remember our credo: "**Keep it simple, ...** (https://en.wikipedia.org/wiki/KISS_principle) ", also affectionately known as KISS.*

Note: see Set-Up and Resources (https://q.utoronto.ca/courses/285775/pages/set-up-and-resources) (the same as for Assignment 1).

Description

In this assignment, your task is to extend the web application you developed in Assignment 1 into an **elastic web application** with a memory cache consisting of a pool of storage nodes, where each node is an independent EC2 instance. The memory cache should resize its pool of storage nodes on demand. In this assignment, we do not consider node failures, and as such your system does not need to be fault tolerant. While this would not be acceptable in practice, where server faults are inevitable, we ignore this scenario for simplicity. Your application should consist of the following four components, each of which should be implemented as a separate Flask instance. Think about how to organize your components across multiple EC2 instances; the architecture of the system is left up to you.

1. A **manager-app** that controls the size of the memcache pool. The UI of this application should support the following functionality:
    1. Use charts to show the number of nodes as well as to aggregate statistics for the memcache pool including miss rate, hit rate, number of items in cache, total size of items in cache, number of requests served per minute. The charts should display data for the last 30 minutes at 1-minute granularity.
    2. Configure the capacity and replacement policy used by memcache nodes. **All memcache nodes in the pool will operate with the same configuration.**
    3. Selecting between two mutually-exclusive options for resizing the memcache pool:
        1. **Manual mode.** There should be two buttons for manually growing the pool size by one node and shrinking the pool size by one node. The maximum and minimum sizes should be 1 and

8, respectively.

2. **Automatic**. Configure a simple auto-scaling policy by setting the following parameters (more details on the auto-scaler in component 2 below):
    1. ***Max Miss Rate threshold*** (average for all nodes in the pool over the past 1 minute) for growing the pool.
    2. ***Min Miss Rate threshold*** (average for all nodes in the pool over the past 1 minute) for shrinking the pool.
    3. ***Ratio by which to expand the pool*** (e.g., expand ratio of 2.0, doubles the number of memcache nodes).
    4. ***Ratio by which to shrink the pool*** (e.g., shrink ratio of 0.5, shuts down 50% of the current memcache nodes).
4. Deleting all application data: A button to delete image data stored in RDS as well as all image files stored in S3, and clear the content of all memcache nodes in the pool (see next section on clearing memachce data).
5. Clearing memcache data: A button to clear the content of all memcache nodes in the pool.

2. An **auto-scaler component** that automatically resizes the memcache pool based on configuration values set by the manager-app. It should monitor the miss rate of the mecache pool by getting this information using the AWS CloudWatch API. The auto-scaler should be implemented as a stand-alone process that runs in the background. It should implement the following functionality:
    - There should be no need to manually restart the auto-scaler every time the policy is changed.
    - Check for the cache miss rate every one minute.
    - **Do NOT use the AWS Auto Scaling feature for this assignment.**
    - **Limit the maximum size of the memcache node pool set by auto-scaler to 8 and the minimum to 1.**
3. **The web front-end you developed for A1**. Provide the functionality implemented for A1 with the following modifications:
    1. Remove functionality to configure memcache settings.
    2. Remove functionality that displays memcache statistics.
    3. All image files should be stored in S3.
    4. The mapping between keys and image files should be stored in AWS RDS. **Do not store the images themselves in the RDS database**. 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.*
    5. Route requests to the memcache pool using a consistent hashing approach based on MD5 hashes. For simplicity, assume that the key space is partitioned into 16 equal-size regions which are then allocated to the pool of memcache nodes. Figures 1-3 illustrate how this assignment

changes as the pool size changes from one node to two nodes, to three nodes. More information is provided in the Consistent Hashing section below.

6. [Optional:] Add a feature to the front-end that makes it possible to automatically notify it when the size of the memcache pool changes.

4. **The key-value memory cache developed in A1**. Provide the functionality implemented for A1 with the following modifications:
   - Store memcache statistic every 5 seconds using **CloudWatch Custom Metrics (https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/publishingMetrics.html)** instead of the database as was done in A1. Also, look at **boto3 documentation (https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/cloudwatch.html#Clou** on how to publish these metrics.

Consistent Hashing

We are using a simplified version of consistent hashing that may not necessarily give us the same properties as the original version.

For our simplified consistent hashing, the range of all possible key values is partitioned into 16 equal-sized regions using **MD5 hashing (https://en.wikipedia.org/wiki/MD5)** . The full MD5 hash range extends from 0 to 2^{128} -1 in decimal, or 0 to FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF in hexadecimal. To divide the hash range into 16 equal partitions, you need to determine the range of each partition. Conveniently, using hexadecimal representation, our 16 partitions are as follows:

	Range beginning (hexadecimal)	Range end (hexadecimal)
Partition 1	0	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Partition 2	10000000000000000000000000000000	1FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Partition 3	20000000000000000000000000000000	2FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Partition 4	30000000000000000000000000000000	3FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Partition 5	40000000000000000000000000000000	4FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Partition 6	50000000000000000000000000000000	5FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Partition 7	60000000000000000000000000000000	6FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Partition 8	70000000000000000000000000000000	7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

Partition 9	80000000000000000000000000000000	8FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Partition 10	90000000000000000000000000000000	9FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Partition 11	A0000000000000000000000000000000	AFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Partition 12	B0000000000000000000000000000000	BFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Partition 13	C0000000000000000000000000000000	CFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Partition 14	D0000000000000000000000000000000	DFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Partition 15	E0000000000000000000000000000000	EFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Partition 16	F0000000000000000000000000000000	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

For example:

- Given key "k1", we calculate its MD5 hash: B637B17AF08ACED8850C18CCCDE915DA. This hash would fall into Partition 12.
- Given key "k2", we calculate its MD5 hash: 61620957A1443C946A143CF99A7D24FA. This hash would fall into Partition 6.

Once we have the 16 partitions, we can assign different partitions to different memcache nodes. The following three figures illustrate the assignment of partitions to nodes for 1, 2, and 4 nodes. Take note of the alternating pattern that partitions are assigned to nodes.

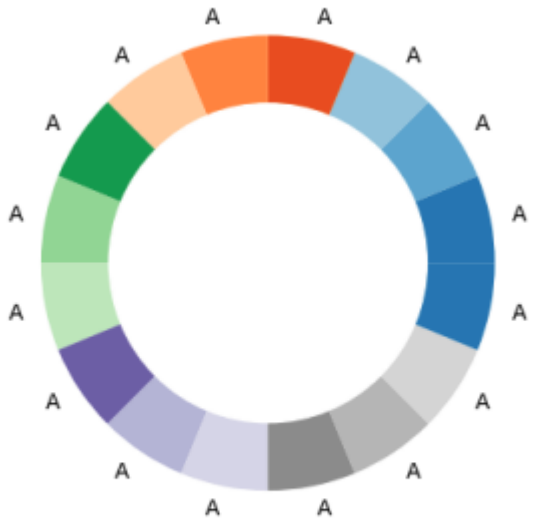

Figure 1

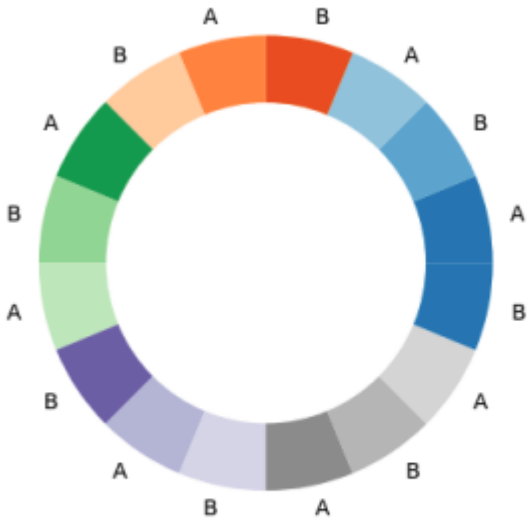

Figure 2

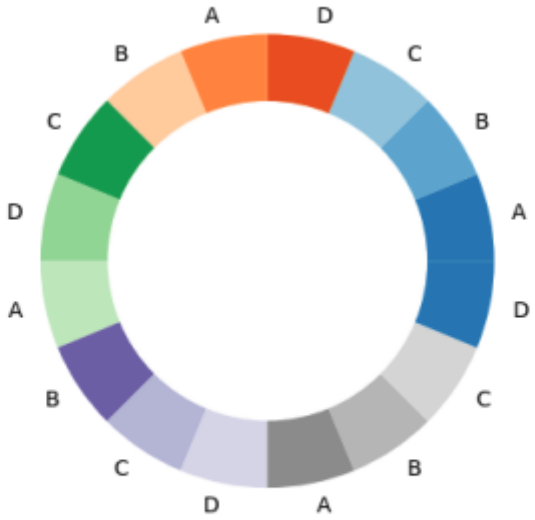

Figure 3

Figure 1: One memcache node exits, A. All of the 16 partitions are assigned to memchache node A, meaning that A is responsible for all possible key values, and all requests are routed to A.

Figure 2: A new memcache node is added, B. Now there are two memcache nodes, A and B. The 16 key partitions are divided between memcache nodes A and B, equally, using an alternating pattern. Now, approximately half of the requests should be routed to A, and half to B, depending on which node the key in the request has been assigned to.

Figure 3: Two new memcache nodes are added, C and D. Now, there are four memcache nodes, A, B, C, D. The 16 partitions are re-divided between all four nodes, such that each node is responsible for one fourth of the possible key values. Approximately one fourth of all requests should be routed to each node.

Notes:

- The observed number of requests routed to each node will depend on the key values at runtime; here, we split the partitions equally among nodes as a simple approach to load balancing.
- The assignment of key values to nodes changes *each time* a new node is added or removed, so ensure that you are updating nodes each time.
- The alternating pattern of partitions should be maintained whenever nodes are added or removed, e.g., for two nodes, the pattern is A, B, A, B...; three nodes A, B, C, A, B, C... ; five nodes A, B, C, D, E, A, B, C, D, E....
- This is a *simplified* version of consistent hashing used for this assignment.

Requirements

- To allow for automatic testing, your application should (in addition to your web interface) include three URL endpoints to automatically upload files and retrieve files. Our automatic testing script will send HTTP requests to your URL endpoints with the parameters in the body. **Your implementation should not accept HTTP requests where parameters have been appended to the URL**. See this link for more information about how HTTP POST requests work: **https://www.w3schools.com/tags/ref_httpmethods.asp. (https://www.w3schools.com/tags/ref_httpmethods.asp)**
- **(https://www.w3schools.com/tags/ref_httpmethods.asp)** The two automatic testing endpoints should conform to the following interfaces:

Upload:

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

Retrieve all keys:

```
relative URL = /api/list_keys
method = POST
```

Retrieve an image associated with a specific key:

```
relative URL = /api/key/<key_value>
method = POST
```

These endpoints should generate responses in the following JSON format: In case of success for the **upload** interface:

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

In case of success for the **retrieve** interface while fetching all keys:

```
{
    "success": "true",
    "keys": [Array of keys(strings)]
}
```

In case of success for the **retrieve** interface while fetching a particular key:

```
{
    "success": "true",
    “content” : file contents
}
```

In case of failure of any of the calls of the **upload** or **retrieve** interface:

```
{
    "success": "false",
    "error": {
        "code": servererrorcode
        "message": errormessage
    }
}
```

Notes:

- Stick to the interface provided above; this is what our automatic testing script during the demo will follow.
- Please prepare at least one test for each endpoint that you can use to demonstrate your endpoints. Ensure that the test request and responses match that outlined in the A2 instructions.
- Here's a sample of a test that may be used for the **upload** endpoint:

```
files = {'file': open('path-to-images/'+filename, 'rb')}
response = requests.post(url+"/upload", files=files, data={'key': 'test'})
```

- Think about possible failure scenarios and provide meaningful error messages as output. Diligently document possibly failure scenarios in your report.

Report

A written report is required alongside the technical requirements of this assignment. The specification and contents of the report are as follows:

- One report per group.
- Group number, assignment number, each member's name and student ID required at the top of page one.
- Include page numbers on the bottom right corner of each page (including the first).
- A page limit of 5 pages for the report (additional pages will result in deducted marks); here, less is more if you are able to succinctly describe everything.
- Good formatting & legibility (Click **here (https://guides.lib.berkeley.edu/how-to-write-good-documentation)** for tips on how to write documentation).
- Include the following sections:
  - **General Architecture** of your application (use figures and diagrams as needed); in particular, carefully describe the interactions among components in your web application. Discuss the architecture, what components are mapped to which machine(s), and how components interact.
  - **Database Schema**: use ER diagrams to describe your schema; explain whether any major changes are required relative to the database schema for A1.
  - **Design Decisions**: what the key decision(s) were in designing your web application, any alternatives you considered, and why you made your choice. Try to articulate at least three decisions.
  - **Results**:
    - **Describe** how your auto-scaler component works.
    - **Graphs for Manual Resizing:**
      - Constant memcache node pool size: choose any pool size greater than 1. Provide one latency graph (time per request(s); gradually increasing the number of requests) and one throughput graph (maximum requests per time; gradually increasing the amount of time).
      - Shrinking memcache node pool size: choose any starting pool size greater than 1 and a smaller ending pool size greater than 1. Repeat the same latency and throughput experiments as above, while shrinking the pool size from the starting size to the ending size. Provide the graphs.
      - Growing memcache node pool size: choose any starting pool size greater than 1 and a larger ending pool size greater than 1. Repeat the same latency and throughput experiments as above while growing the pool size from the starting size to the ending size. Provide the graphs.
      - You should have 6 graphs in total, 2 from each of the cases above. Any unspecified configurations, such as cache, read:write ratio, etc. you may choose yourself, and should indicate in the report. You do not need to show results for all caching strategies, only one of your choice. The above graphs should be derived when your system is operating in manual re-sizing mode.

- **Graphs for Auto-Scalar:**
  - Choose values for the min and max rate threshold and expand and shrink ratios.
    - Design a scenario where your auto-scalar will shrink the number of nodes, and plot the miss rate vs. number of nodes as your system re-sizes. Indicate the min threshold value on your graph.
    - Design a scenario where your auto-scalar will grow the number of nodes, and plot the miss rate vs. number of nodes as your system re-sizes. Indicate the max threshold value on your graph.
  - You should have 2 graphs in total from the above cases. Any unspecified configurations, such as cache, read:write ratio, etc. you may choose yourself, and should indicate in the report. You do not need to show results for all caching strategies, only one of your choice. The above graphs should be derived when your system is operating in automatic re-sizing mode.

Note: in order to evaluate your system, you may need a large dataset of images. Factors like size may impact your results, so you should choose a dataset that you think works best for your system and associated experiments. There are many image datasets available online (e.g., machine learning and neural network training data sets.)

When you're ready to submit, see Submission & Demo Instructions (https://q.utoronto.ca/courses/285775/pages/submission-and-demo-instructions) .

Document Revisions

Here, we manually track major revisions to this document that occurred after the document was released.

Date	Revision
Oct. 31	Sentence dded to Description 3.5.1. that it is okay to have the manager-app handle mapping and routing of keys to memcache nodes.
Nov. 8	Updated hashing figure descriptions to match the number of nodes shown visually.
Nov. 16	Additional notes on testing endpoints added.
Nov. 19	Section 3.6 was somehow removed. It has been added back, but will not count towards marks.

Assignment 2 Rubric

Criteria	Ratings		Pts
Manager UI Providing the appropriate menus, bars and buttons for the manager to control the system and displays performance metrics	**10 pts** **Full Marks**	**0 pts** **No Marks**	10 pts
Manual Pool Resizing	**8 pts** **Full Marks**	**0 pts** **No Marks**	8 pts
Automatic Pool Resizing	**8 pts** **Full Marks**	**0 pts** **No Marks**	8 pts
Report See A2 Report Submission page for more details	**0 pts** **Full Marks**	**0 pts** **No Marks**	0 pts
			Total Points: 26