Group number 26
Group members:
Shiyu Xiu, 1004724872
Yuangan Zou, 1004761778
Qian Tang, 1005228656
Assignment 2

Application Architecture

The general structure of our application consists of 4 parts running on 4 different Flask instances: Manager App, Auto scaler, Web FrontEnd as well as a backend (memcache). The web front end as well as the manager app will interact with users directly. Specifically, the web front end will deal with ‘upload’ requests made by user, display all the keys as well as allow the user to retrieve an image with a key provided. The Manager app offers the following functionalities: 1. show the statistics for the memcache pool. 2. configure capacity and replacement policy for all nodes 3.allow user to choose between manual and auto-scaling as well as setting auto scaler parameters. 4. Delete data in S3, memcache, RDS as well as memcache nodes. The auto scaler and backEnd will run in the background and are not visible to our users. The backend contains all the logic for modifying memcache such as clear, invalidate, config, etc. The webfront end will post requests to backend for modifying the memcache. The auto scaler will read data from cloud watch and compare them with the data set in Manager app in order to scale the number of nodes in memcache pool.

Different from Assignment1, we will store our images in S3 instead of a local file system. Also, we are using RDS instead of MySQL.

The workflow of a user uploading an image:

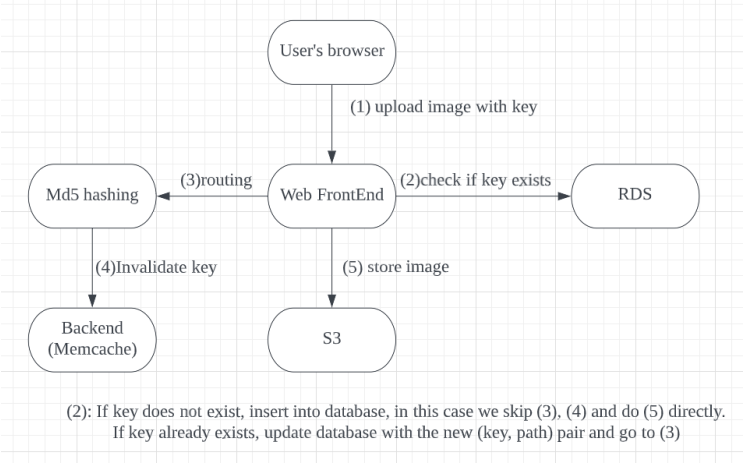


Figure 1

The workflow of a user retrieving an image (cache hit):

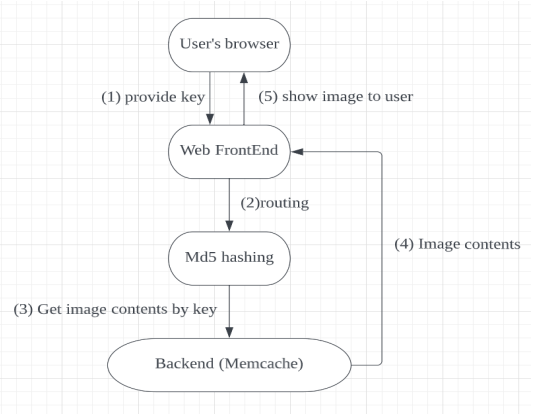


Figure 2

The workflow of a user retrieving an image(cache miss):

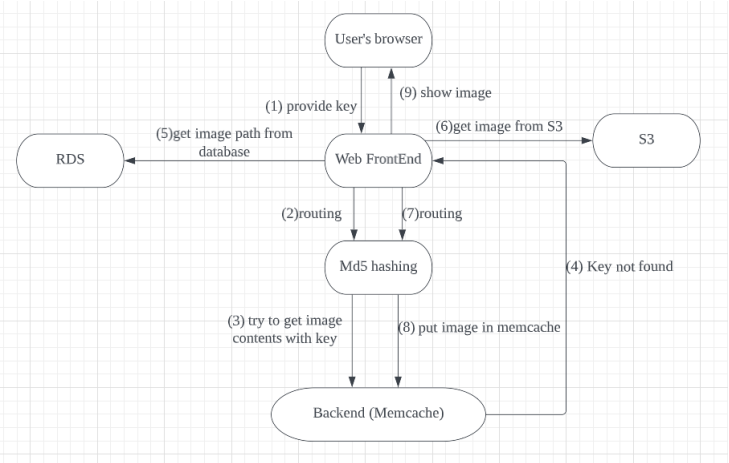


Figure 3

Database Schema

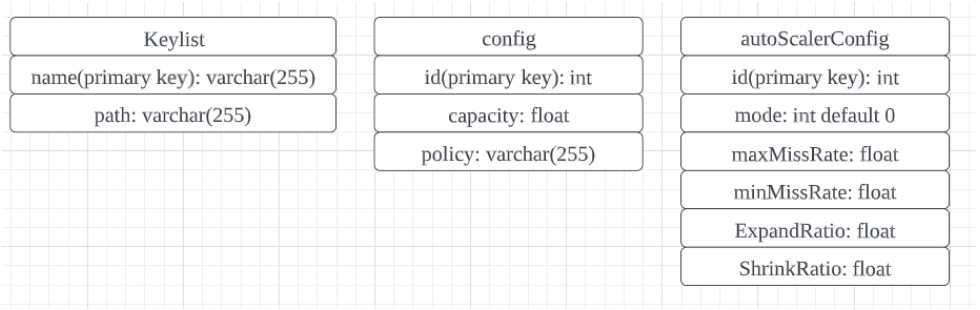


Figure 4

We designed our database so that it has 3 tables: a table “Keylist” to store the values of keys as well as their corresponding paths to S3, this table will be accessed by frontend when retrieving the image; a table “config” to store the configuration data of all memcache nodes in the pool, this table is used by backend in order to do the configuration; a table “autoScalerConfig” to store the configuration data of the auto scaler, this table will be updated by the manager app and read by the auto scaler to scale the number of nodes in the memcache pool.

Design Decisions

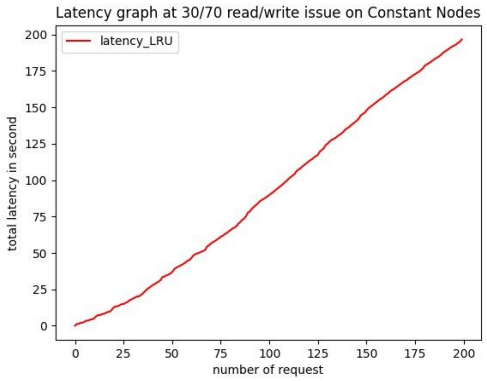
Arrangement of EC2 and Flask instances: The general structure of our system includes 8 EC2 instances running 8 memcahce nodes as well as an extra EC2 instance running the manager app and auto scaler. Inside each of the 8 EC2 instances running memcache, there are a frontend and backend. Frontend serves as the web frontend and backend is just the memcache. When frontend needs to send a request to memcache(backend), it goes to manager app to get all available ip addresses then goes through hashing with key to get the correct address to send its request. The manager app and auto scaler run on a separate EC2 instance. The manager app will interact with user and updates the database.
Auto-scaler will read database for configuration and read cloudwatch for miss rate in order to scale the number of nodes in memcache pool. An alternative is to let the 8 EC2 instances only run backend(memcache), an extra EC2 instance running Web Frontend and an extra EC2 instance running manager app and auto-scaler. However, since we have included some APIs as well as memcache logics in the frontend as well, then we need to include the frontend in the 8 EC2 instances running memcache.

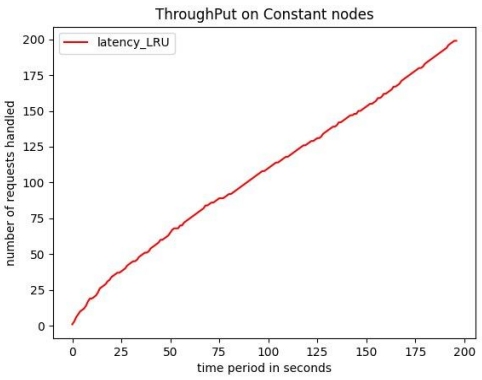
user_data: We use the UserDara parameter when creating a new EC2 instance so that when being created, it automatically runs a script to set up memcache. The memcache in each node is similar to what we designed in Assignment1 but with minor changes. An alternative is to manually create 8 EC2 instances in AWS console and deploy the code for memcache on each of them. However, this approach requires all eight EC2 instances not being terminated during implementation as well as to restore the states of memcache when they are restarted. Therefore, we decide to use boto3 to create new EC2 instances with UserData defined.

Sleep: Each time we increase or decrease the number of nodes in memcache pool, we let the program sleep for 60s in order to make sure the newly created node has entered running state and is assigned a valid public ip address.

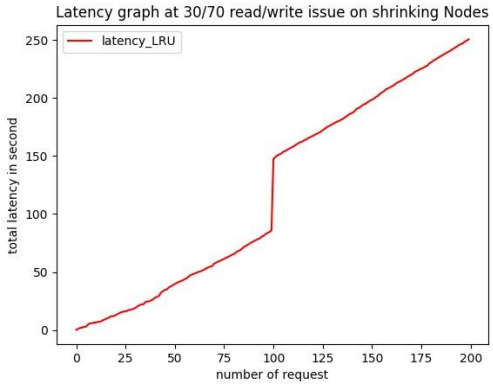
Redistribute Memcache Contents: Each time we increase or decrease the number of nodes in memcache pool, we redistribute the contents in current memcache pool to adapt to the change. Specifically, when we create a new memcache node, we first wait for it to be running. We will then go to each memcache node separately to iterate through the keys in the memcache to do the hashing. Based on the hashing result, we redistribute the contents stored in all memcache nodes. When we terminate a memcache node, we will first pop off the IP address of the node being terminated, then go to the node that is going to be terminated, iterate through all keys in memcache to do the hashing. Based on the hashing result, we redistribute the contents of memcache to the remaining nodes. We will then go to the remaining memcahce nodes to redistribute memcache contents as well in order to keep the hashing consistent across all nodes.
An alternative is to let the newly created memcache node run with no content and drop all contents in the node when it is being terminated. However, this approach leads to loss in memcache contents and it does not guarantee consistent hashing across all nodes. Our design is able to maintain the contents as much as possible when deleting a node as well as maintaining a consistent hashing across all memcache nodes.

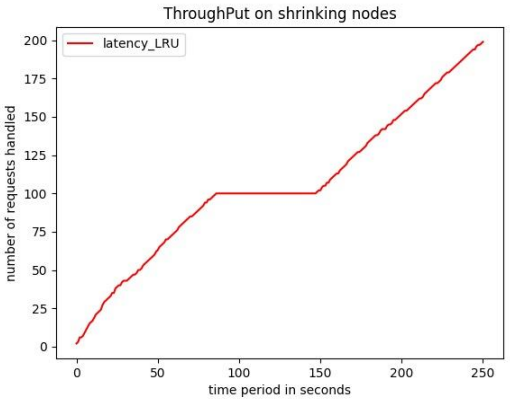
Result



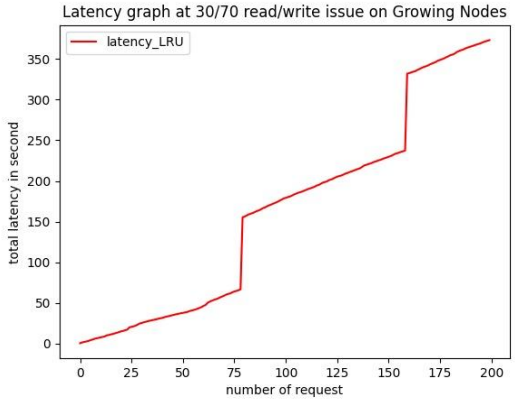


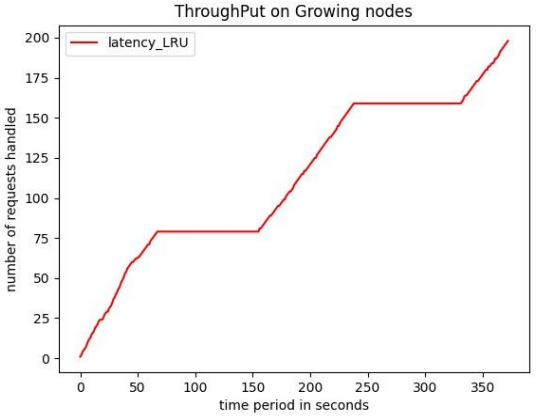
Constant memcache node pool size: latency graph and throughput graph.
Pool size = 2





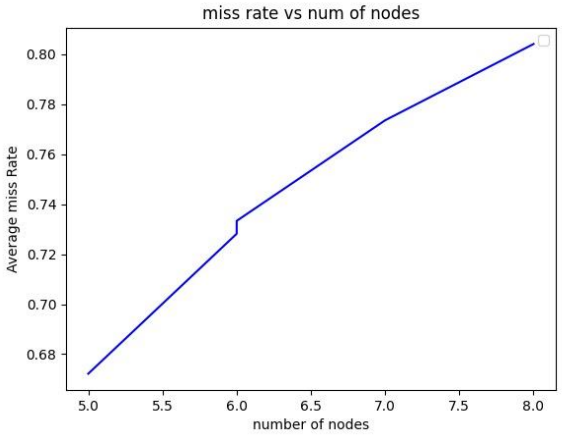
Shrinking memcache node pool size: latency graph and throughput graph.
Starting pool size = 3, Ending pool size =2





Growing memcache node pool size: latency graph and throughput graph.
Starting pool size = 1, Ending pool size = 3

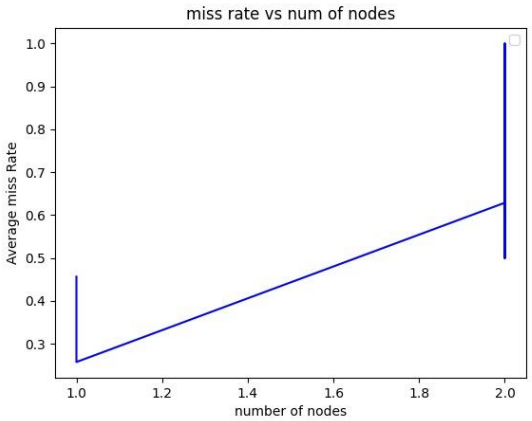
Automatic Auto-Scalar Analysis
Parameters used are:
max_missrate = 0.4
min_missrate = 0.1
expand_ratio =2
shrink_ratio = 0.5



Auto-scaler growing nodes

We can see that if average miss rate > max miss rate, then auto scaler keeps on growing the number of nodes at an expand ratio of 2. The original graph is discontinuous along the x-axis and we are connecting the dots in order to show the trend of increasing nodes.

We change min_missrate = 0.7, below graph is generated:



Auto scaler shrinking nodes

We can see that if average miss rate < min miss rate, then auto scaler keeps on shrinking the number of nodes at a shrink ratio of 0.5. The original graph is discontinuous along the x-axis and we are connecting the dots in order to show the trend of decreasing nodes. Since the number of nodes are consistently switching between 1 and 2, we see a line between them.

Attribution table

Group member	Milestone	Signature
Yuangan Zou	Auto-scaler configuration in Manager App, Delete data, connect to EC2, EC2 instance auto-running script, construct report	Bill Zou
Shiyu Xiu	deploy on AWS, build auto scaler and tested auto-running script on EC2, created database connections in RDS	Shiyu Xiu
Qian Tang	Make modifications for frontend and backend, show statistics page, debugging and generate testing graphs for the system, making design decisions	Qian Tang