University of Toronto

Faculty of Engineering

Department of Electrical and Computer Engineering

ECE 1779 A2

Dynamic Resource Management

Group Members

Peiqi Li	pq.li@mail.utoronto.ca
Kaiduo Wang	kaiduo.wang@mail.utoronto.ca
Tianyi Wang	tywing.wang@mail.utoronto.ca

Date Submitted: 2022.3.21

Table of contents

Introduction

This project is to adjust the numbers of memcache built in project 1 in an elastic memory cache. The main functions of this project are:

- Show charts of worker numbers and memcache pool statistics in the last 30 minutes.
- Set the configuration of all memcache nodes in the pool.
- Set the configuration of the memcache pool. Both manual mode and automatic scaler are implemented.
- A button that could delete all images.
- A button that could clear all content in each memcache.

User Manual

How to use

1. Login AWS with the information provided in credentials.txt
2. Start RDS database *ece1779-a2*
3. Start EC2 instance *Manager-Web-App* and check its public ip address
4. Start terminal and connect EC2 instance with the command:

```
ssh -i "path/to/keypair.pem" ubuntu@public_ip
```

5. Run the web first with command:

```
./start_web.sh
```

6. Please wait for 1 minutes for initialization of the system
7. Run the manager app and autoscaler file with command:

```
./start.sh
```

8. Visit the website by url: http://public_ip:5000/ and http://public_ip:5002/. 5000 is the web service, 5002 is the manager service
9. After using the server, terminate them by Ctrl + c
10. If you want to re-run the program, reboot instance and go to step 4

Features Description

Manager-app

Homepage

The homepage of manager-app has five items like figure 1 shows which are “Charts”, “Configuration for Memcache”, “Options for resizing the memcache pool”, “Deleting all application data”, and “Clear memcache data”.

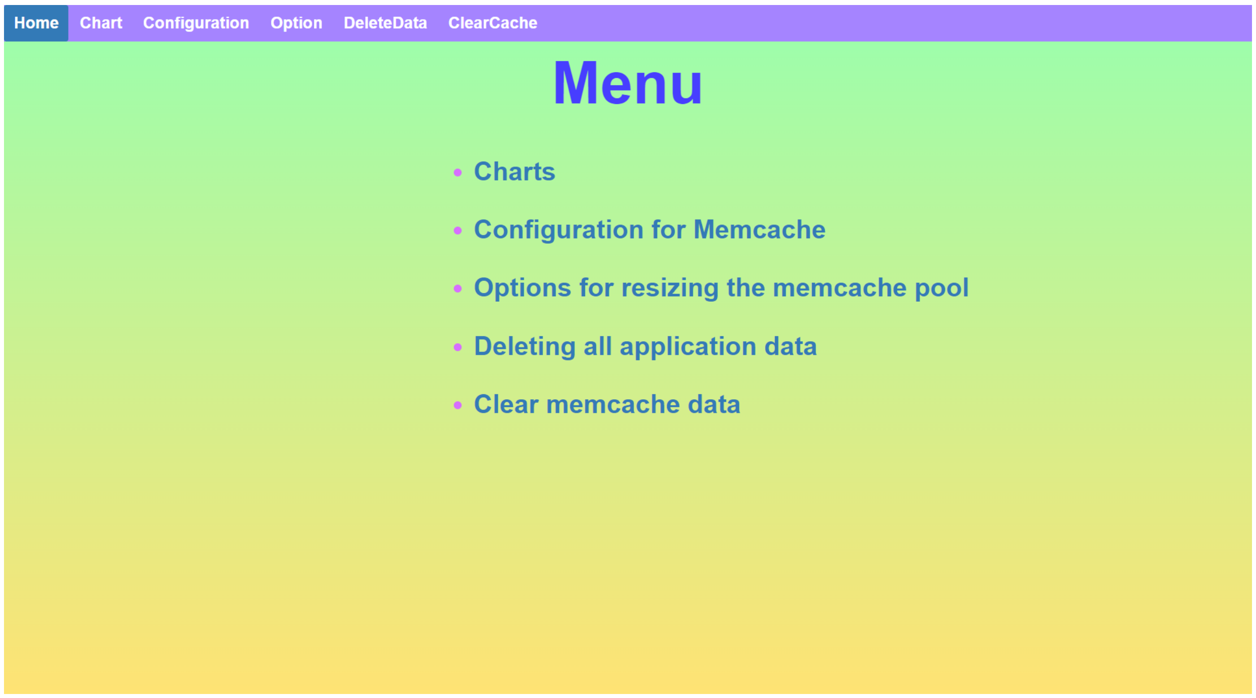

Home
Chart
Configuration
Option
DeleteData
ClearCache
Menu
Charts
Configuration for Memcache
Options for resizing the memcache pool
Deleting all application data
Clear memcache data

Figure 1. Homepage

Charts

The “Charts” page shows five charts as figure 2 and figure 3 show. The first chart is “Worker Numbers”. The second chart is “Miss Rate & Hit Rate” which displays the miss rate and hit rate of the memcache pool. The third chart is “Request Number” used to show the memcache pool’s request in one minute. The fourth chart is “Item Number” which shows the number of memcache’s items. The last chart is “Total Size” used to display the total size of memcache’s item. Each chart has a maximum of 30 data from the last 30 minutes. All the x-axis of these five charts are time and the format is hour and minutes.

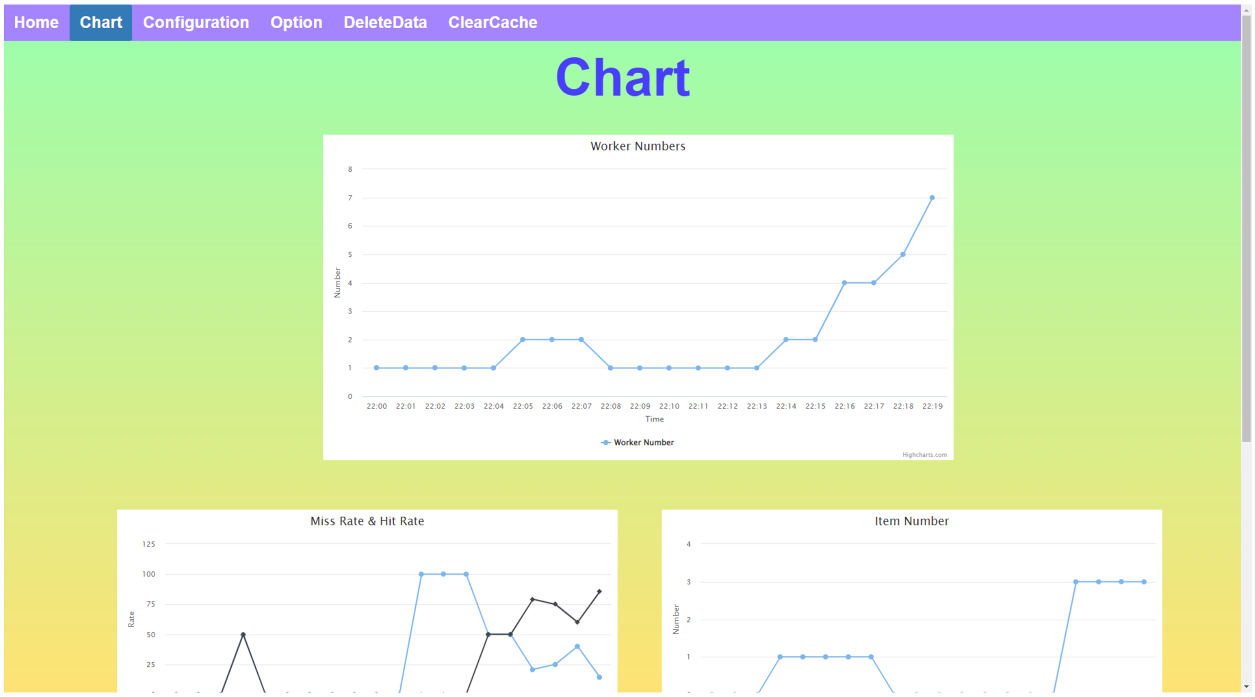


Figure 2. Chart page1

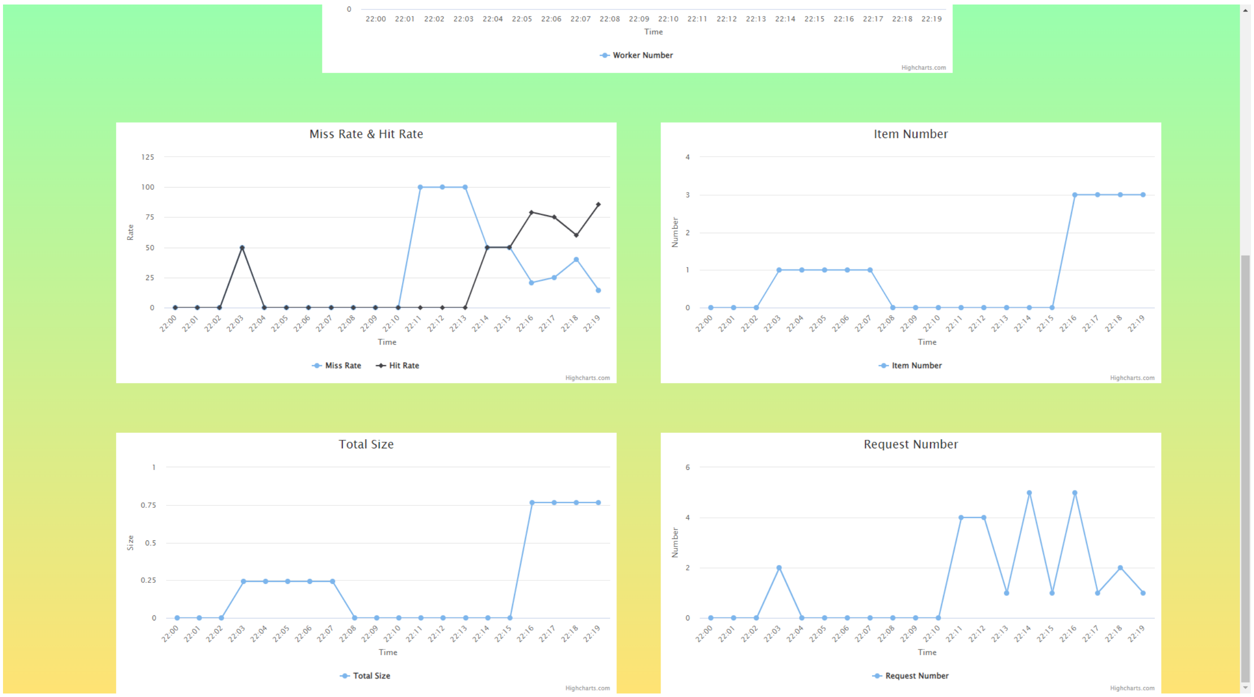


Figure 3. Chart page2

Configuration for Memcache

The “Configuration for Memcache” page shows as Figure 4 has two options for users to set the configuration of the mem-cache. The first option is that users could set the cache capacity between 1MB and 100MB. The second option is that users could choose the replacement policy of men-cache between “Random Replacement” and

“Least Recently Used”. If users configure cache successfully, there will be a return page like figure 5 to show the current memcache configuration.

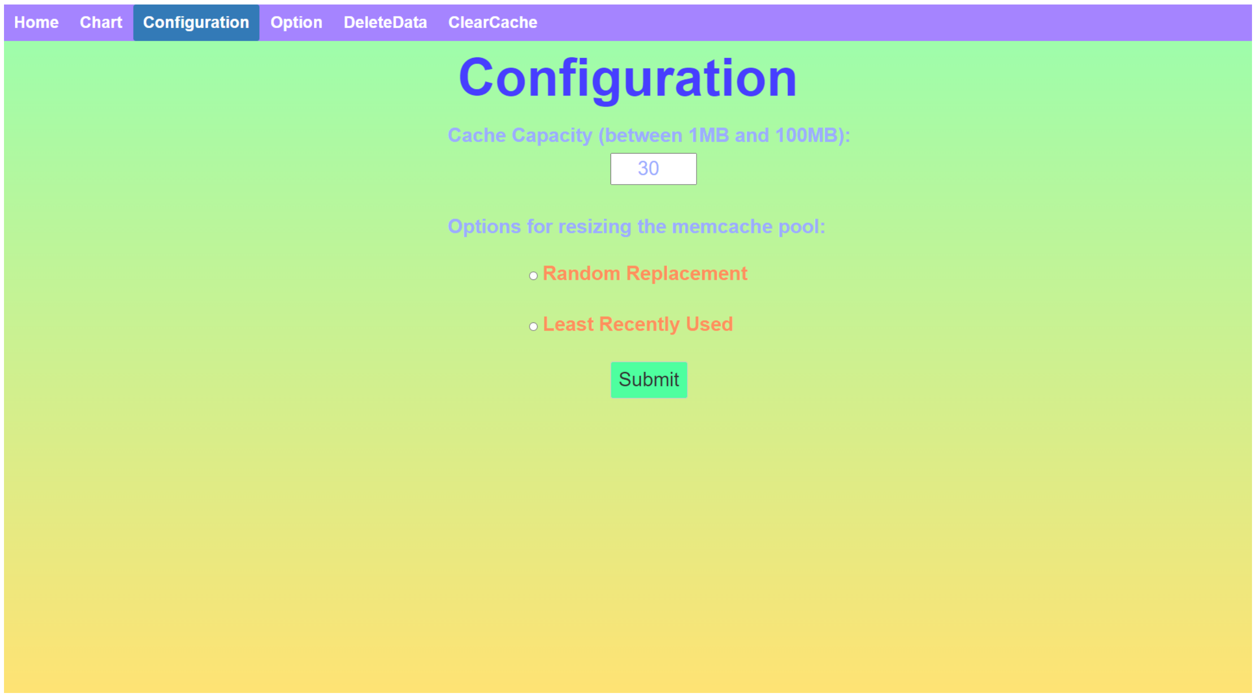
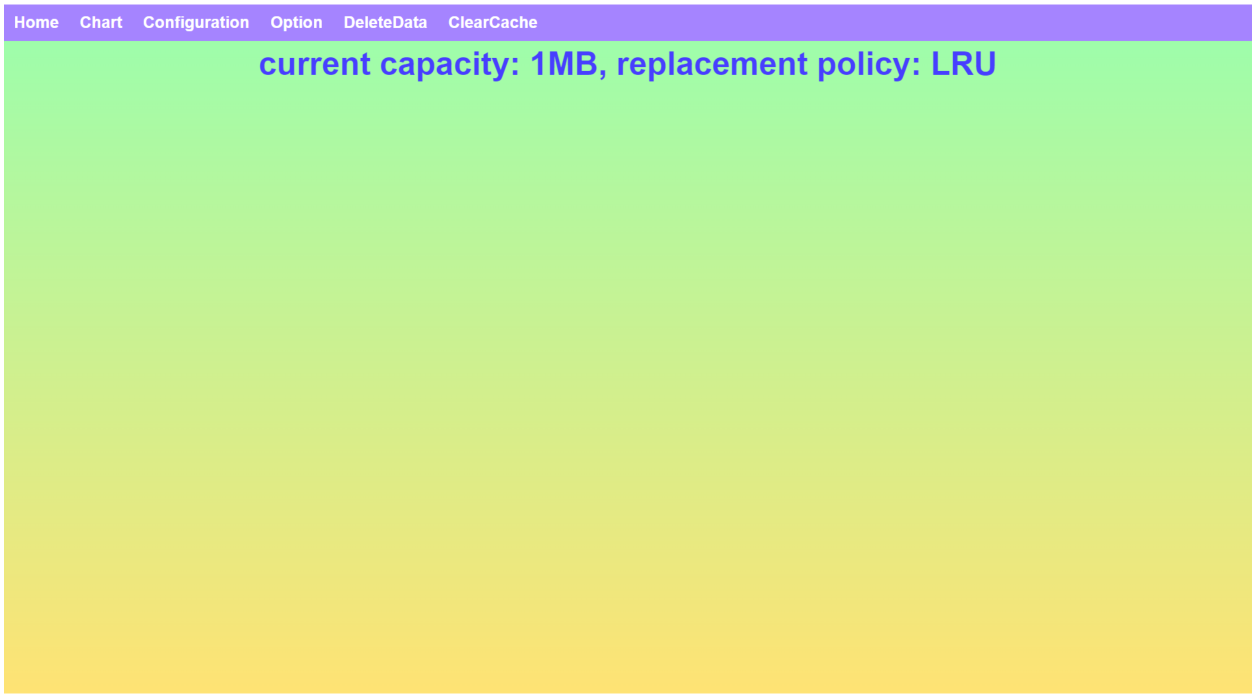


Figure 4. Configuration for memcache

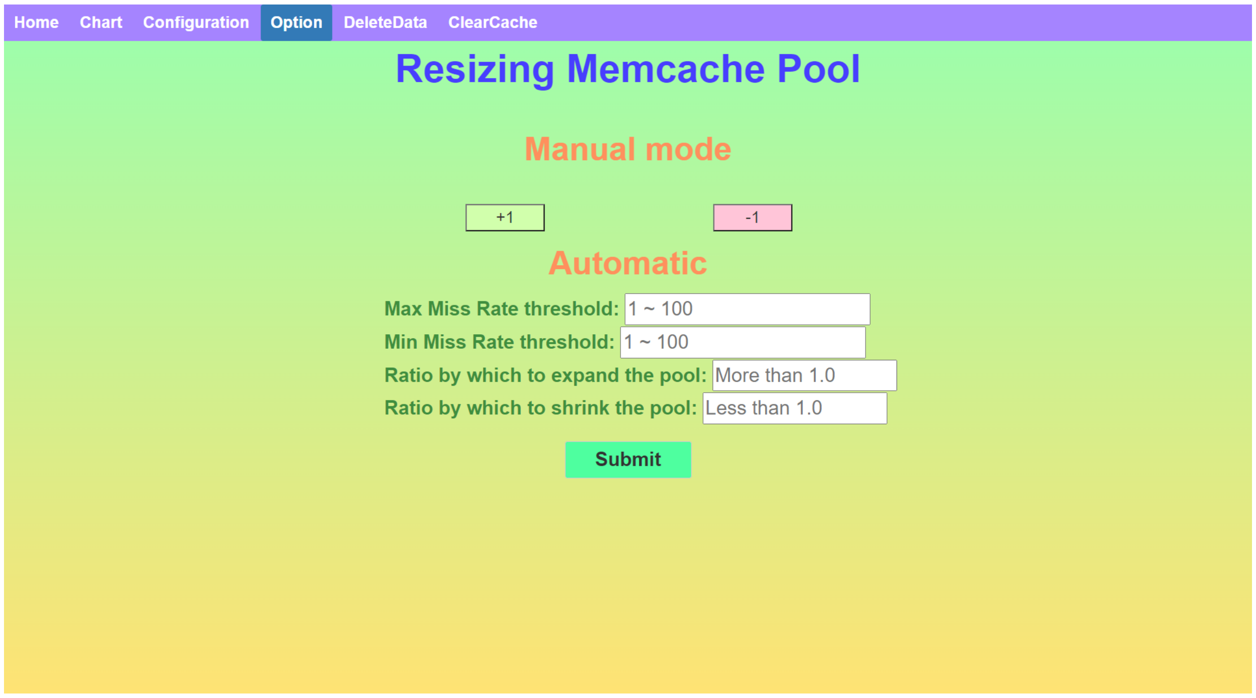


current capacity: 1MB, replacement policy: LRU

Figure 5. Memcache configuration success

Options for resizing the memcache pool

The “Options for resizing the memcache pool” page is used to help users to resize the memcache between two options like figure 6. The first option is “Manual Mode” which has “+1” and “-1” buttons. If users click the “+1” button and the current pool size is less than 8, there will be a return page like figure 7 shows to tell users that one instance has started and the pool size has grown successfully. If users click the “+1” button but the current pool size is bigger than 8, there will be a return page like figure 8 shows to tell users that there should not be more instances in the memcache now. If users click the “-1” button and the current pool size is bigger than 1, there will be a return page like figure 9 shows to tell users that one memcache has stopped and the pool size has shrunk successfully. If users click the “-1” button but the current pool size is less than 1, there will be a return page like figure 10 shows to tell users that there is only one instance in the memcache now. The second option is “Automatic”. Users need to input their max miss rate, min miss rate, expand ratio, and shrink ratio. When users click the “Submit” button, there will be a return page like figure 11.



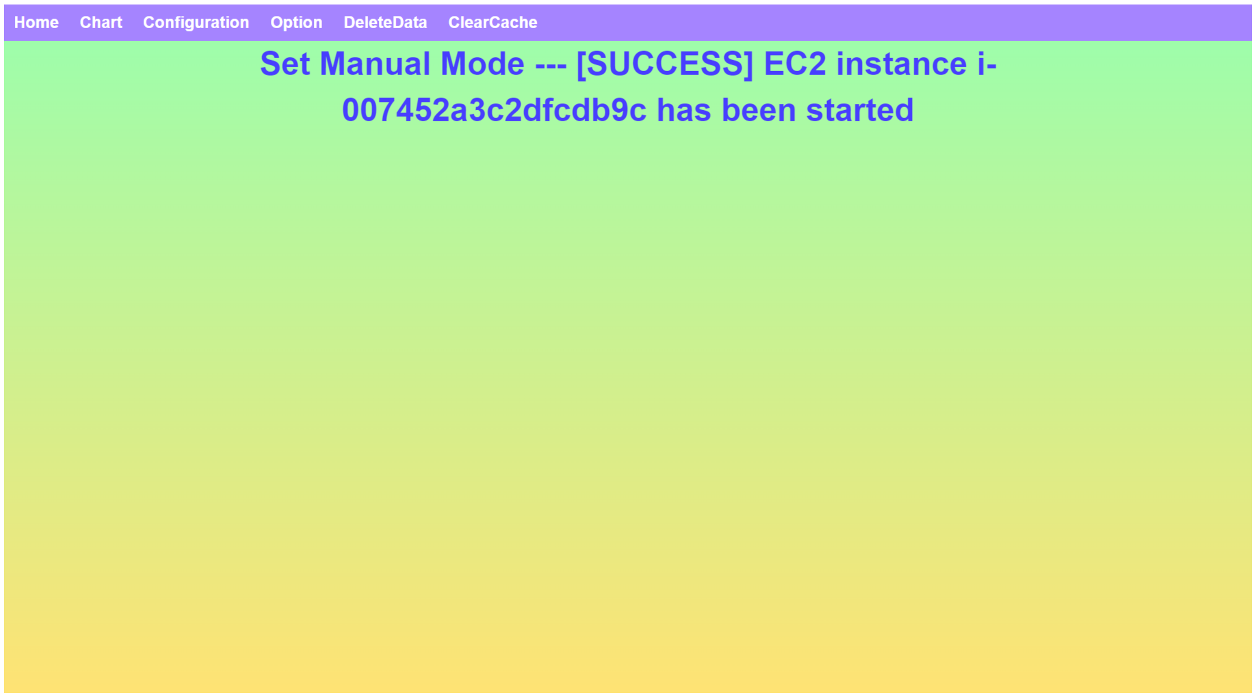
Resizing Memcache Pool

Manual mode

Automatic

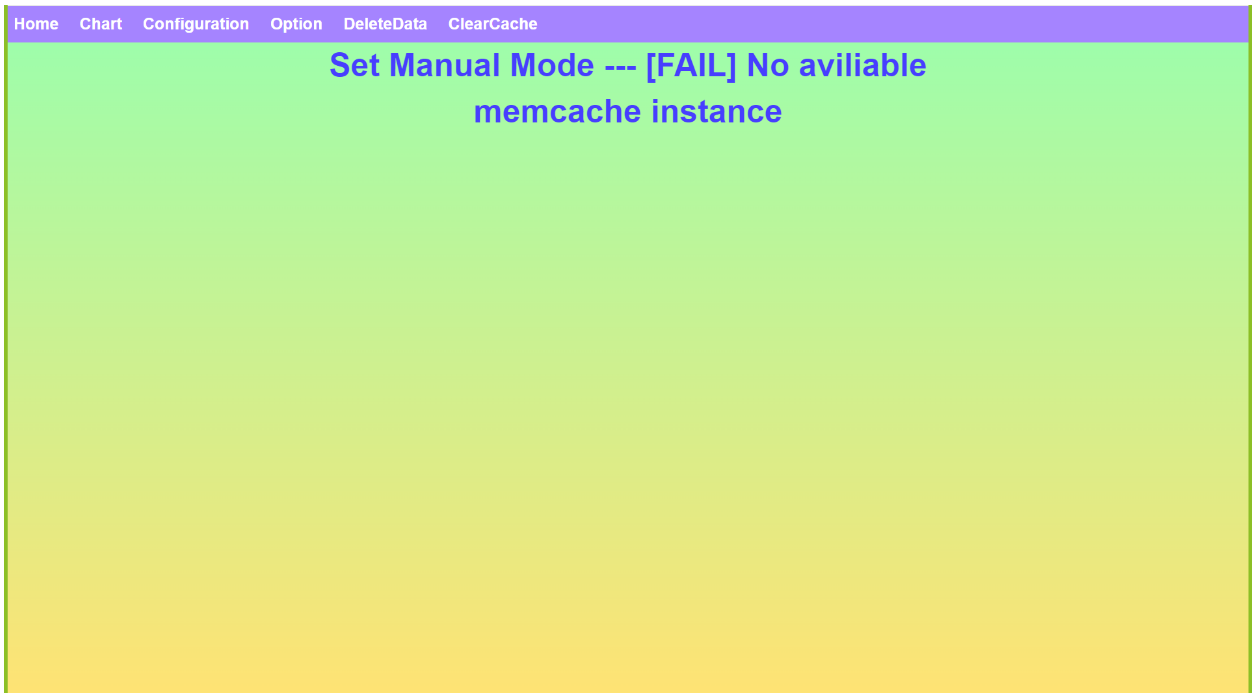
Max Miss Rate threshold:	1 ~ 100
Min Miss Rate threshold:	1 ~ 100
Ratio by which to expand the pool:	More than 1.0
Ratio by which to shrink the pool:	Less than 1.0

Figure 6. Resize Memcache pool



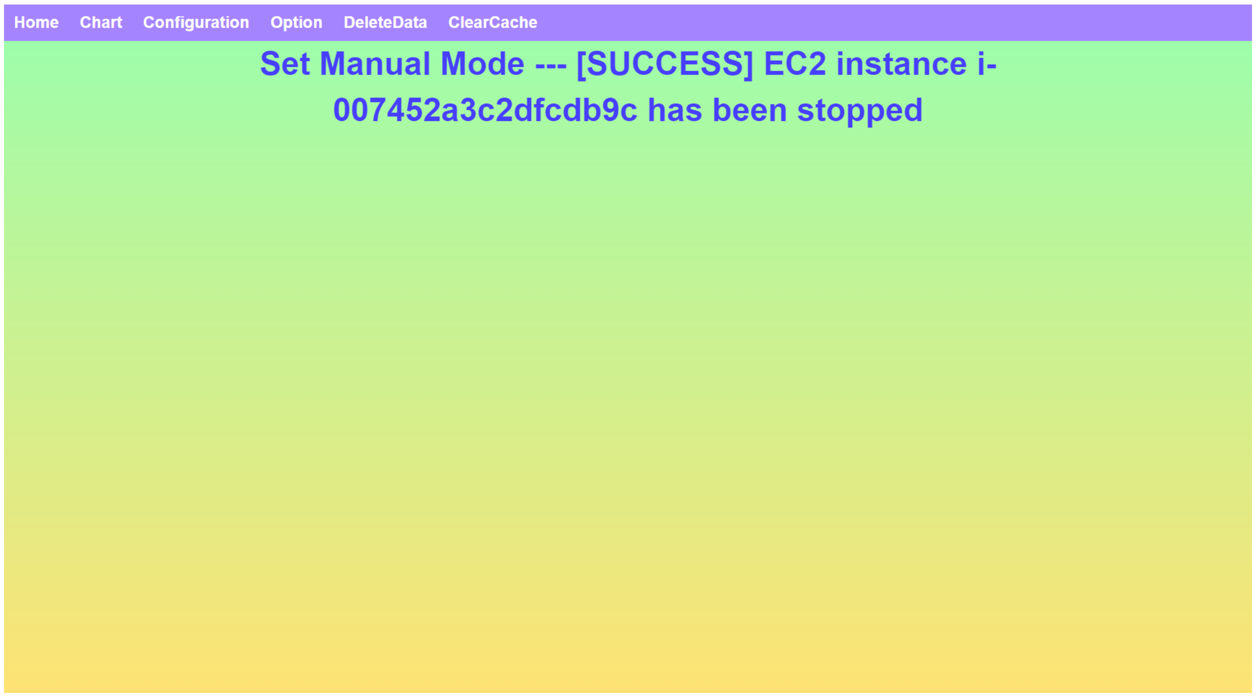
Set Manual Mode --- [SUCCESS] EC2 instance i-007452a3c2dfcdb9c has been started

Figure 7. Grow successfully



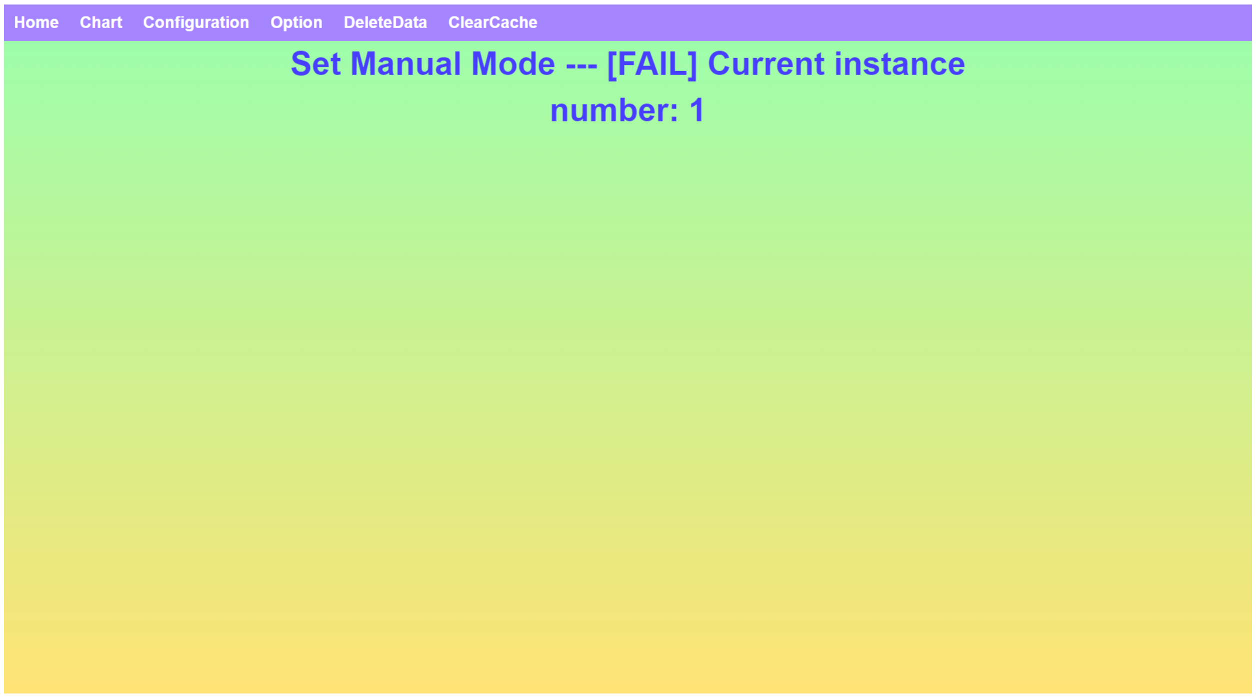
Set Manual Mode --- [FAIL] No aviliable memcache instance

Figure 8. Grow failed



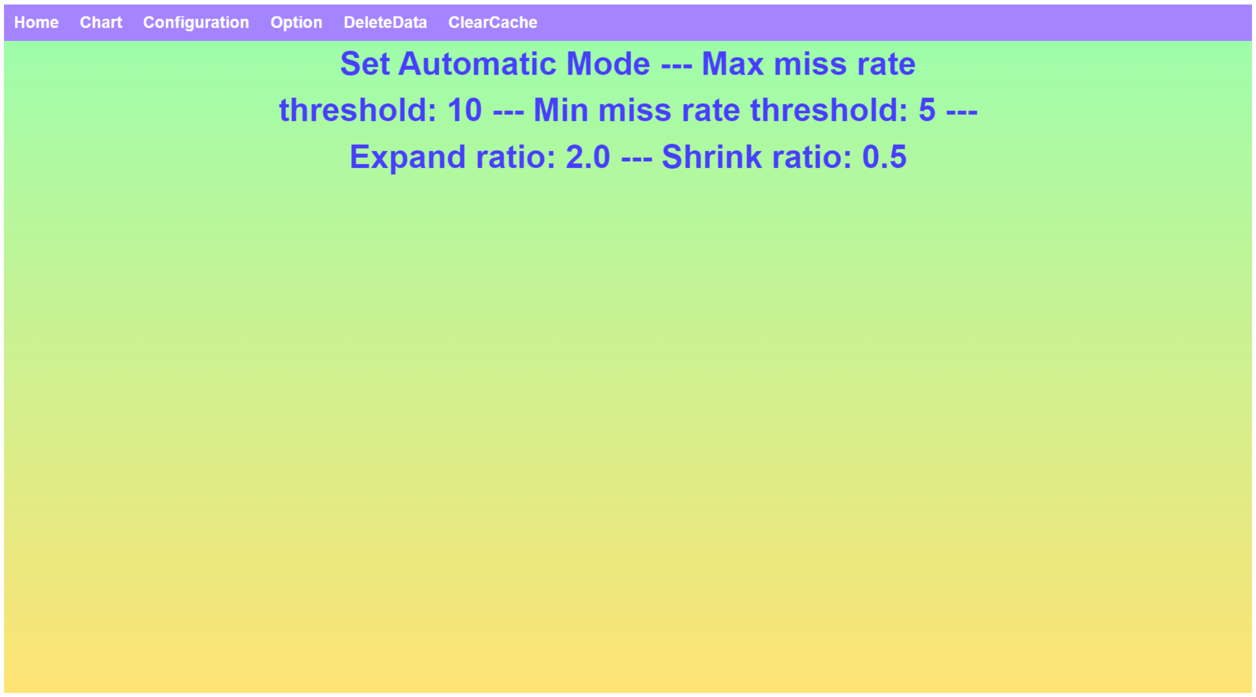
Set Manual Mode --- [SUCCESS] EC2 instance i-007452a3c2dfcdb9c has been stopped

Figure 9. Shrink successfully



Set Manual Mode --- [FAIL] Current instance number: 1

Figure 10. Shrink failed

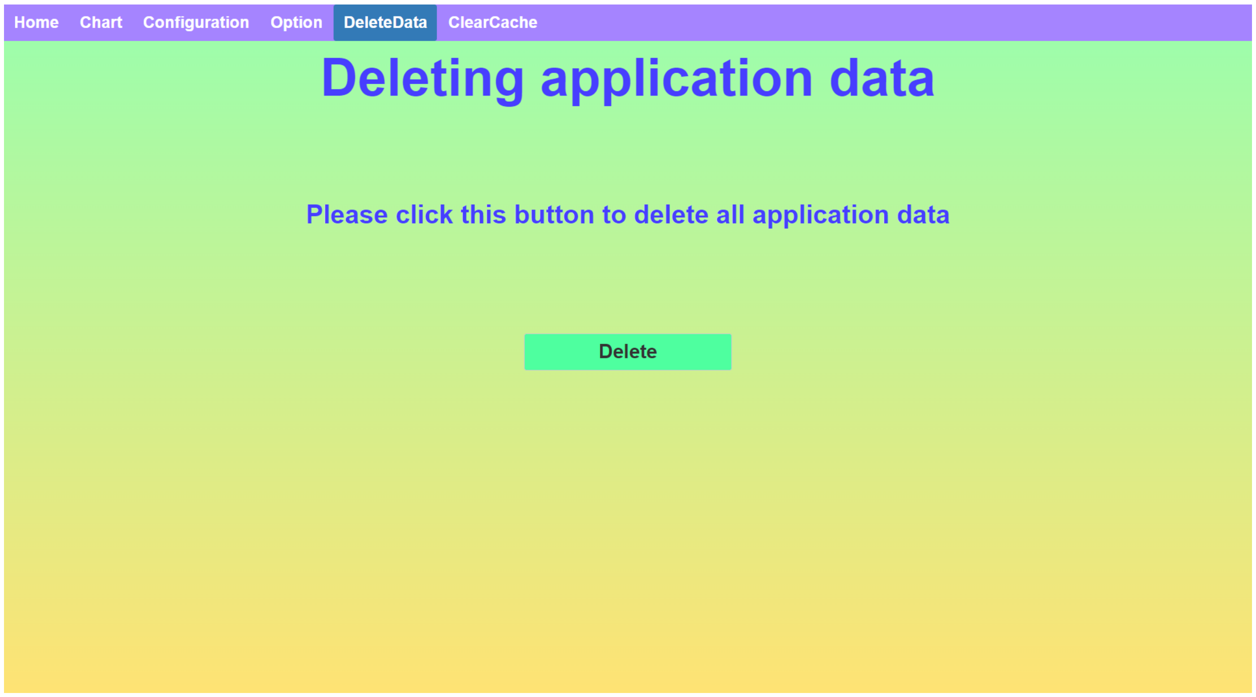


Set Automatic Mode --- Max miss rate threshold: 10 --- Min miss rate threshold: 5 --- Expand ratio: 2.0 --- Shrink ratio: 0.5

Figure 11. Set automatic mode

Deleting all application data

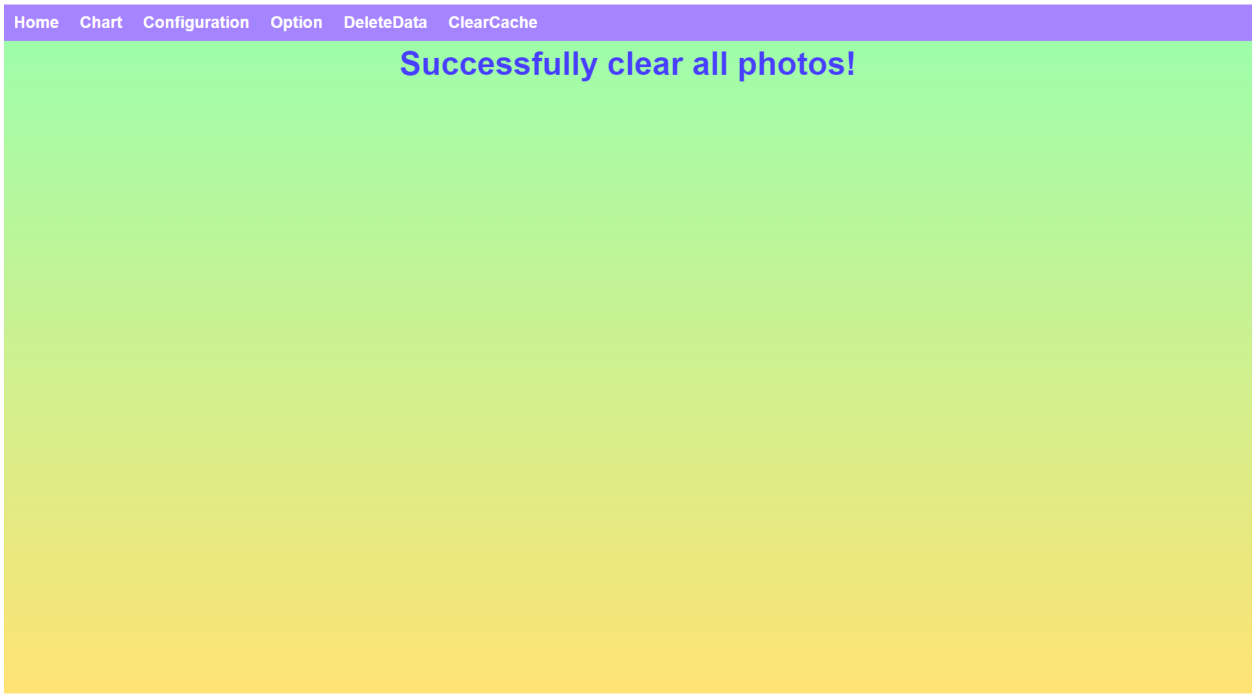
The "Deleting all application data" page only has one "Delete" button for users to delete all data in the database and S3 as figure 12 shows. When there is some data in the application and users click the button, there will be a return page like figure 13 to tell users data has been cleared successfully. However, if there is no data in the application and users click the button, there will be a return page like figure 14 to tell users there are no photos in the system now.



Deleting application data

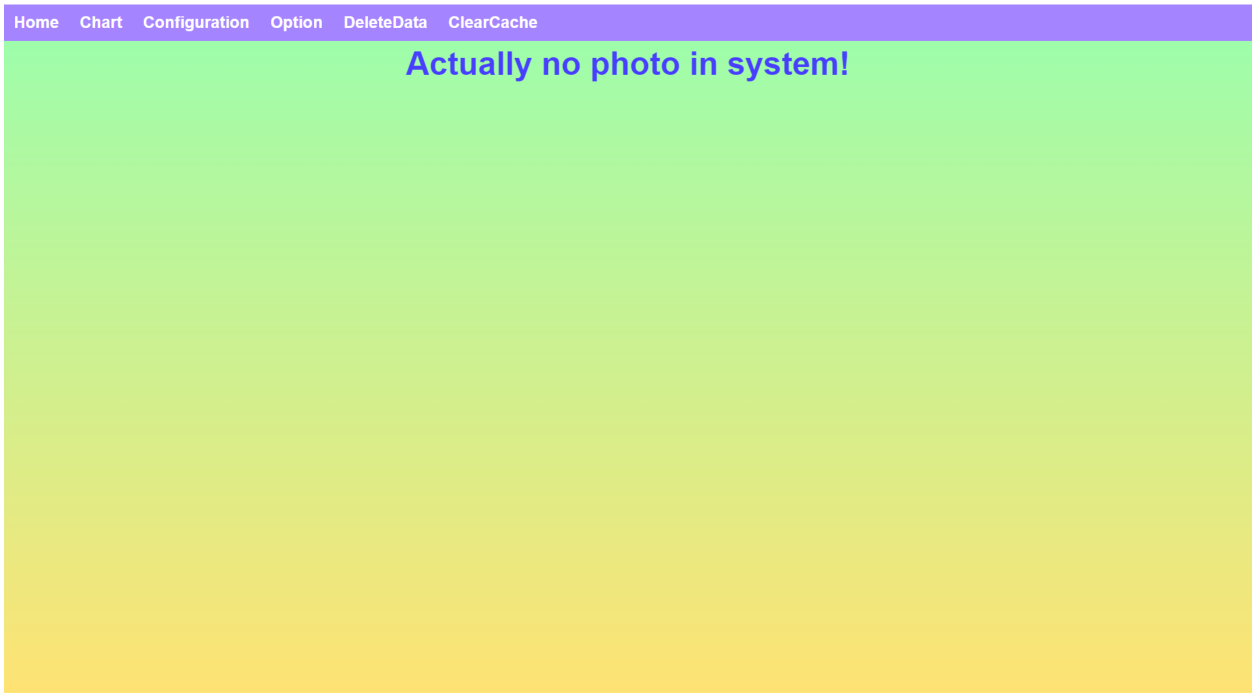
Please click this button to delete all application data

Figure 12. Delete application data



Successfully clear all photos!

Figure 13. Delete data successfully

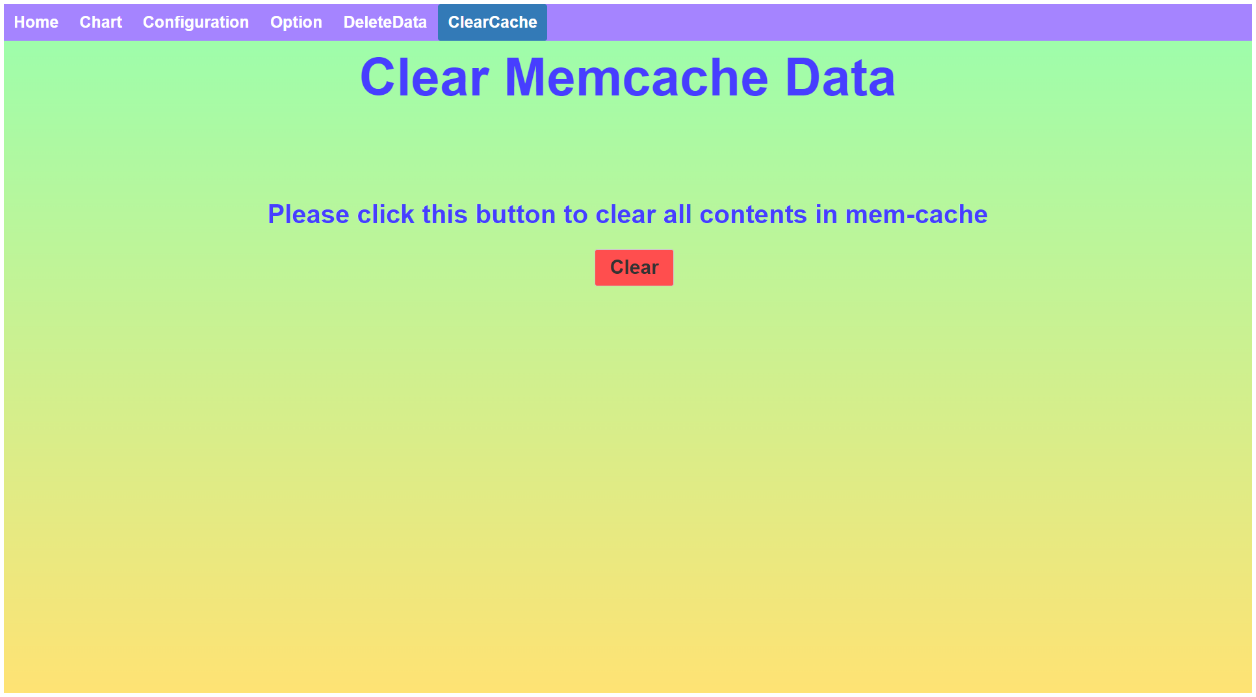


Actually no photo in system!

Figure 14. No photo in system

Clear memcache data

The “Clear memcache data” page only has one “Clear” button to clear memcache node’s content in the pool like figure 15. When users click the button, there will be a page that tells users memcache has been cleared successfully as figure 16 shows.



Clear Memcache Data

Please click this button to clear all contents in mem-cache

Figure 15. Clear memcache data

Memcache has been cleared

Figure 16. Clear memcache successfully

Web front end

Home Page

The homepage of the web front end shows like Figure 17 shows. There are 5 items on this page which are “Image Upload”, “Image Search”, “Display All Image Name”.

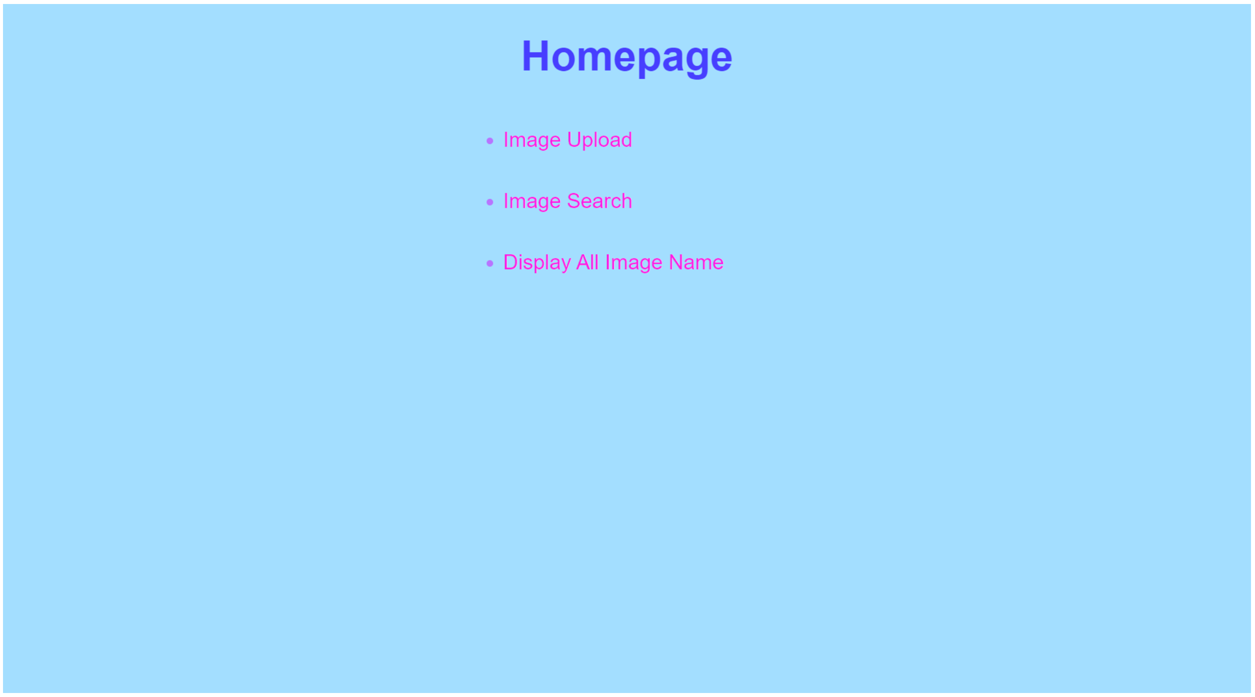


Figure 17. Homepage

Upload a New Pair of Key and Image

The “Image Upload” page like figure 18 shows. Users need to input the key and select an image from the local file. If the image is uploaded successfully, there will be a page to tell the user that “Success upload your photo!” like figure 19.

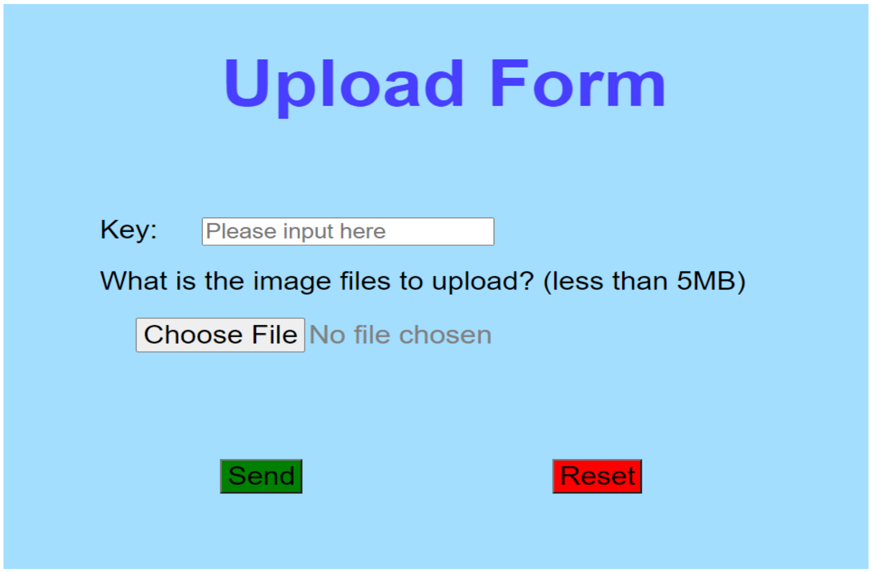


Figure 18. Image Upload

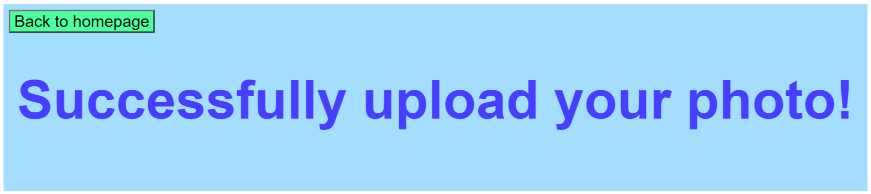


Figure 19. Upload successfully

Image Search

The “Image Search” page as Figure 20 shows. Users just need to input the key of an image and click the “Send” button. If the image is found, then there will be a return page to show the image like figure 21. If the image is not found, there will be a page to tell users “No photo found” as figure 22.

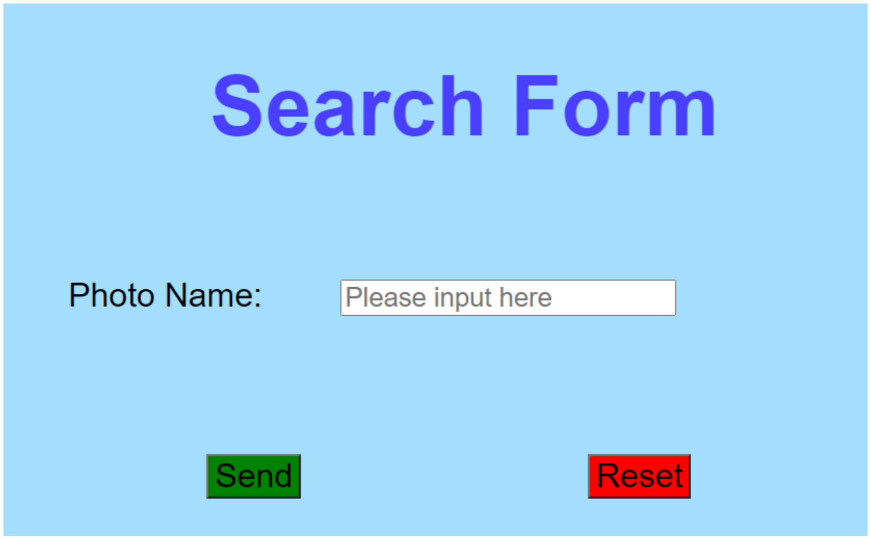


Figure 20. Image Search

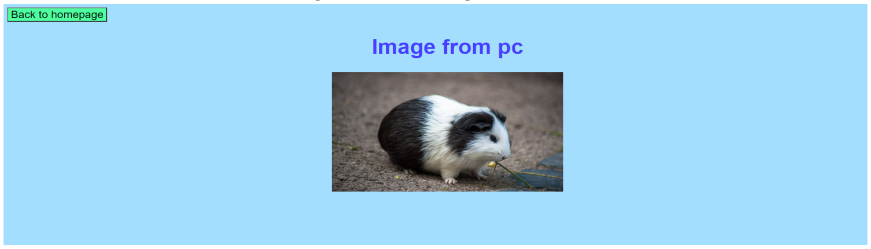
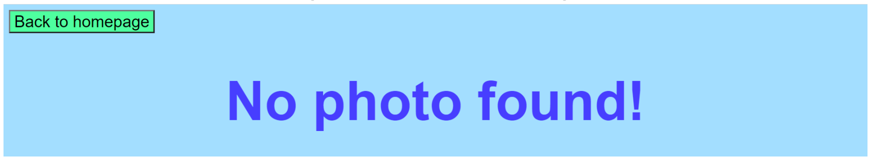


Figure 21. Find the image

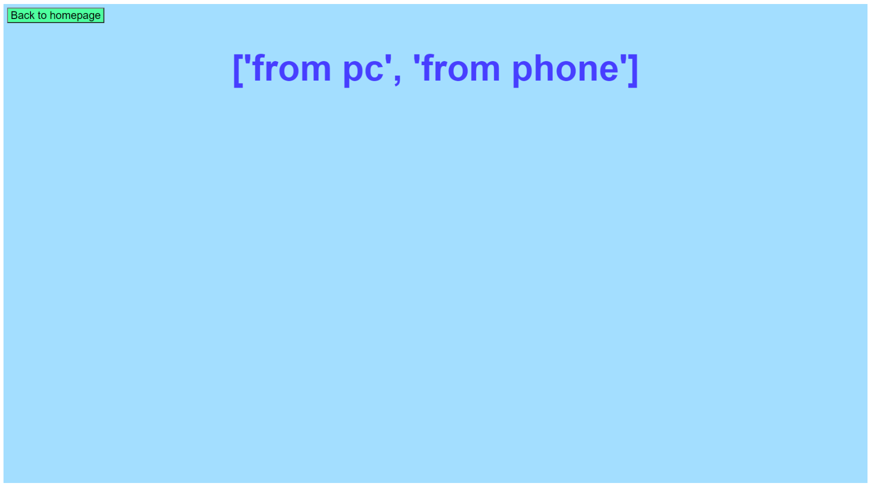


No photo found!

Figure 22. No photo found

Display All Image Name

When users click “Display All Image Name”, it will display all image names on a page like the figure 23.



['from pc', 'from phone']

Figure 23. Display all image name

Development Documentation

New Features

- Show charts of memcache pool
- Configuration of all memcache nodes
- Manual Mode for resizing memcache pool
- Automatically resizing memcache pool
- Delete application data
- Clear memcache data
- Route requests to memcache using consistent hash
- Publish and get CloudWatch Custom Metrics

Dependencies

Our project runs in a python virtual environment:

- Python
- Virtualenv (you can also use Anaconda)

For complete features, we use packages as follow:

- boto3: A AWS SDK for Python
- Flask: A python web framework for development
- Flask-SQLAlchemy: Manipulating MySQL from flask application
- MySQL: RDBMS for data collection
- requests: Send HTTP requests between servers
- uWSGI: A deployment tool for Flask application
- uhashring: A convenient consistent hashing package

```
boto3==1.21.21
Flask==2.0.1
Flask-SQLAlchemy==2.5.1
mysqlclient==2.1.0
requests==2.27.1
```

```
SQLAlchemy==1.4.31
uWSGI==2.0.20
uhashring==2.1
Werkzeug==2.0.1
```

Project Structure

Auto Scaler

It’s an independent program running in background, which checks the missing rate every minute and auto scales the number of memcache nodes when the pool is in automatic mode.

Manager App

This app is responsible for showing charts of memcache pool, config memcache node, config memcache pool and clear data and cache.

```
manager_app
 ├── app
 │    ├── __init__.py  # initialization
 │    ├── aws.py  # Operations related to AWS are encapsulate here
 │    ├── config.py  # basic configuration information
 │    ├── configure.py  # config memcache, config pool and clear data
 │    ├── display.py  # display charts
 │    ├── main.py
 │    ├── metrics.py  # send statistics of pool every 1 minute
 │    ├── models  # Operations related to RDS database
 │    │    ├── create_tables.py
 │    │    └── modify_tables.py
 │    ├── static
 │    │    └── Bootstrap
 │    │        ├── css
 │    │        ├── fonts
 │    │        └── js
 │    └── templates
 │        ├── base.html
 │        ├── cache_config_form.html
 │        ├── clear.html
 │        ├── delete.html
 │        ├── pool_config_form.html
 │       ├── returnPage.html
 │       └── showChart.html
 └── run.py
```

Web server

We use the Model-View-Controller(MVC) design pattern to develop the web server. Operations related to interacting with databases, like creating tables, querying data and modifying data, are handled in Model. View is responsible for presenting data and providing user interface at the front end. All the HTMLs are put in the template folder. Controller is used to handle clients' requests and interact with Model and View. We divided it into photos.py, api.py, config.py and init_cache.py .

```
web_flask
├── app
│   ├── __init__.py
│   ├── api.py  # for auto testing endpoints
│   ├── config.py  # basic configuration information
│   ├── init_cache.py  # initialize cache node when web starts
│   ├── main.py
│   ├── models
│   │   ├── clear_storage.py  # clear old contents when web starts
│   │   ├── create_tables.py  # create tables in database
│   │   └── modify_tables.py  # modify and query info in database
│   ├── photos.py  # photo upload, search and list keys
│   ├── static
│   │   └── style
│   │       ├── config_form_style.css
│   │       ├── mainstyle.css
│   │       ├── returnPage_style.css
│   │       ├── search_form_style.css
│   │       ├── statsstyle.css
│   │       ├── upload_form_style.css
│   │       └── viewstyle.css
│   └── templates
│       ├── base.html
│       ├── config_form.html
│       ├── returnPage.html
│       ├── search_form.html
│       ├── stats.html
│       ├── upload_form.html
│       └── view.html
└── run.py
```

Memory Cache

The mem-cache module serves as a storage buffer of the web server, which would speed up the image lookup process. It supports cache data modification and cache property configuration. Also, it will calculate statistics data periodically and upload it to the database.
The structure of the mem-cache module is as follow:

```
memcache
├── memcache_app
│   ├── __init__.py
│   ├── config.py  # basic configuration information
│   ├── aws.py  # Operations related to AWS are encapsulate here
│   ├── main.py # memcache server api
│   ├── memcache.py # memcache class
│   ├── models # functions to interact with the database
│   │   ├── create_tables.py
│   │   └── modify_tables.py
│   ├── timer.py # periodically publish CloudWatch metrics
│   └── total_size.py # a module to calculate the memcache item size
└── run.py
```

Database Design

Database Schema

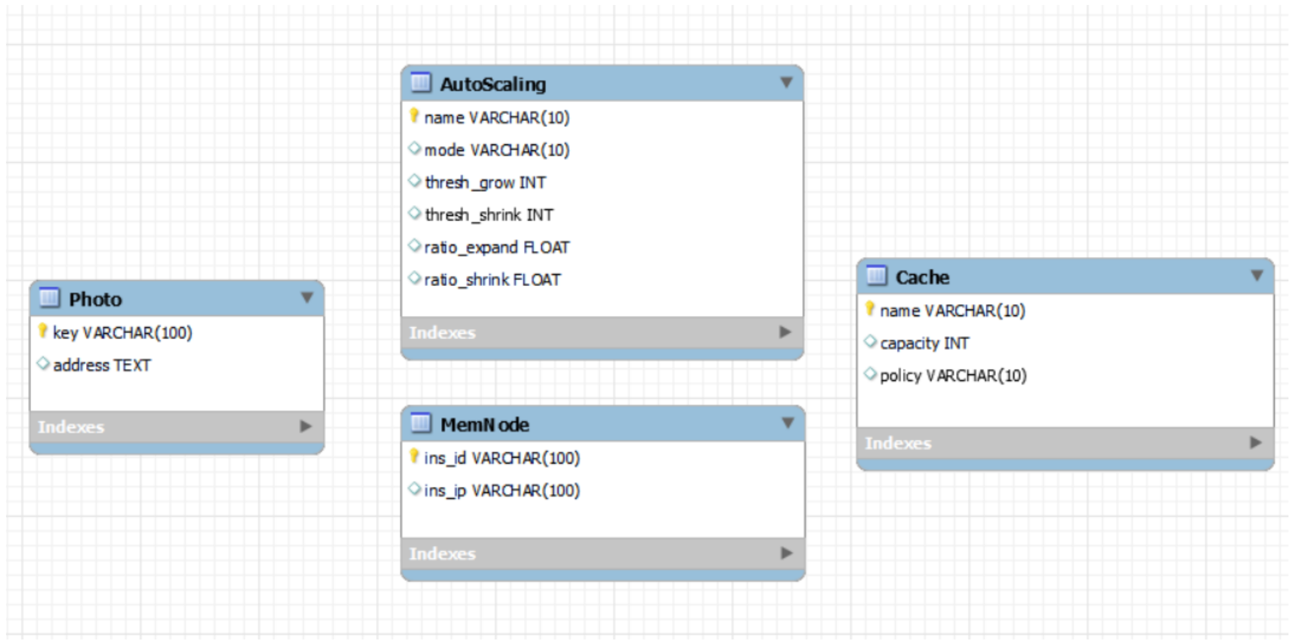


Figure 24

We created fourtables. Table Photo stores key and address of photos. Table Cache stores configuration information of mem-cache. Table MemNode stores the instance id and public ip of all running memcache instances. Table AutoScaling stores the configuration information of the manager pool.

Table Photo

Column	**Data Type**	**Description**
key	String	The key users specified
address	Text	Address of the image in local file system

Table Cache

Column	**Data Type**	**Description**
name(default="local")	String	Name of the mem-cache, in A1 we only have "local"
capacity	Integer	The capacity of mem-cache in MB
policy	String	The replace policy of the mem-cache

Table MemNode

Column	Data Type	Description
ins_id	String	Instance id of running mem-cache node
ins_ip	String	Public ip of corresponding instance

Table AutoScaling

Column	Data Type	Description
name(default="local")	String	Name of the manager pool
mode	String	Manual mode or Automatic mode
thresh_grow	Integer	Max miss rate threshold for growing
thresh_shrink	Integer	Min miss rate threshold for shrinking
ratio_expand	Float	Ratio by which to expand the pool
ratio_shrink	Float	Ratio by which to shrink the pool

Database Manipulation Model

Models.create_tables

Description:
Define the tables in database
Table Class:
Photo(key, address)
Cache(name="local", capacity, policy)
MemNode(ins_id, ins_ip)
AutoScaling(name="local", mode, thresh_grow, thresh_shrink, ratio_expand, ratio_shrink)

Models.modify_tables

Description:
Operations including query and modification on database

Member function

add_photo():
Input: key, absolute address in file system
Description: add 1 row in Photo table in database
change_photo():
Input: photo, new absolute address in file system
Description: modify 1 row in Photo table in database
search_photo():
Input: key
Return: photo address corresponds to the key
Description: query 1 row in Photo table in database
query_all():
Input: An empty key list
Return: key_list stored all the keys
Description: query all rows in Photo table in database
config():
Input: capacity, policy
Description: modify the configuration in Cache table in database
get_config():
Input: None
Return: capacity, policy
Description: get configuration parameters from the database
add_ins():
Input: new instance id and public ip
Description: add new instance's information to table MemNode
remove_ins():
Input: removed instance id and public ip
Description: delete stopped instance's information to table MemNode
clear_photo():
Input: None
Description: delete all the photo data in rds database
config_scaler():
Input: Configuration information of the pool, including maxThresh, minThresh, exRatio,

shRatio
Description: Store the configuration information to AutoScaling table
switch_mode():
Input: None
Description: switch from manual mode to automatic mode
get_ip():
Input: None
Description: get the public ips of all running mem-cache instances
Return: A list of all the public ips

Models.clear_storage

Description:
Clear all the photos and data in the database at the initialization stage

---

Class AWSClient

The class AWSClient is our self-defined class to manipulate AWS services using python boto3 package. In our project, we use AWSClient to accomplish all configurations and manipulations process of EC2 instances and CloudWatch.

Variable

Variable Name	**Type**	**Description**
key	String	AWS access key ID
secret	String	AWS access secret key
region	String	AWS service region
ec2	boto3.resource	handler for AWS ec2
cloudwatch	boto3.client	handler for AWS CloudWatch

Method

AWS CloudWatch manipulation methods:

put_data()
Input: instanceId, attribute name, attribute value, timestamp
Description: Publish a specified statistic of a instance to CloudWatch
put_statistics()
Input: number of items, size of items, number of requests, hit_rate, miss_rate, timestamp
Description: Publish all statistics of the current instance to CloudWatch
get_data()
Input: instanceId, attribute name, attribute value, start_time, end_time
Description: Get a specified statistic of a instance from CloudWatch
get_statistics()
Input: instanceId, start_time, end_time
Description: Get all statistics of a instance from CloudWatch
send_aggregate_metric()
Input: None
Description: Get last minute's statistics of all memcache nodes from CloudWatch, then calculate their aggregate metric and publish to CloudWatch
get_aggregate_metric()
Input: None
Description: Get aggregate metrics during the past 30 minutes

---

AWS EC2 instances manipulation methods:

get_state_instances()
Input: state('running'/ 'stopped')
Description: Get the information of instances that are in specified status and return
get_my_instanceid()
Input: None
Description: Get the instance id of the instance that is calling this instance
grow_by_1()
Input: None
Description: Activate one stopped memcache instance and add it to the memcache pool.
grow_by_ratio()
Input: expand ratio
Description: Activate stopped memcache instances by ratio and add them to the memcache pool.
shrink_by_1()
Input: None

Description: Inactivate one running memcache instance and remove it from the memcache pool.

shrink_by_ratio()

Input: shrink ratio

Description: Inactivate running memcache instances by ratio and remove them from the memcache pool.

API: Manager App

Show Charts

The **metrics.py** file is responsible for getting statistics of all memcache nodes in cloudwatch, calculating required statistics like average missing rate and uploading such statistics to cloudwatch every minute.

When users send http requests to show charts, the **display_charts()** function in display.py will be called.

The **display.py** file is responsible for getting statistics uploaded by metrics.py file and delivering NumberofWorkers, MissRate, HitRate, NumberofItems, TotalSize, NumberofRequests and timestamps to showChart.html.

showChart.html uses javascript and highchart tools to draw those charts and display them.

Configuration of memcache nodes

In our rds **database**. We designed a table called *MemNode*, which stores all the running instances’ id and public ip address.

When users send http requests to config memcache nodes, the **cache_config()** function in configure.py will be called.

That function in **configure.py** first stores the configuration information to rds database table *Cache* and then query data from rds database table *MemNode* to get all the running instances’ public ip addresses. Finally, the function will send clear requests to all the public ip addresses.

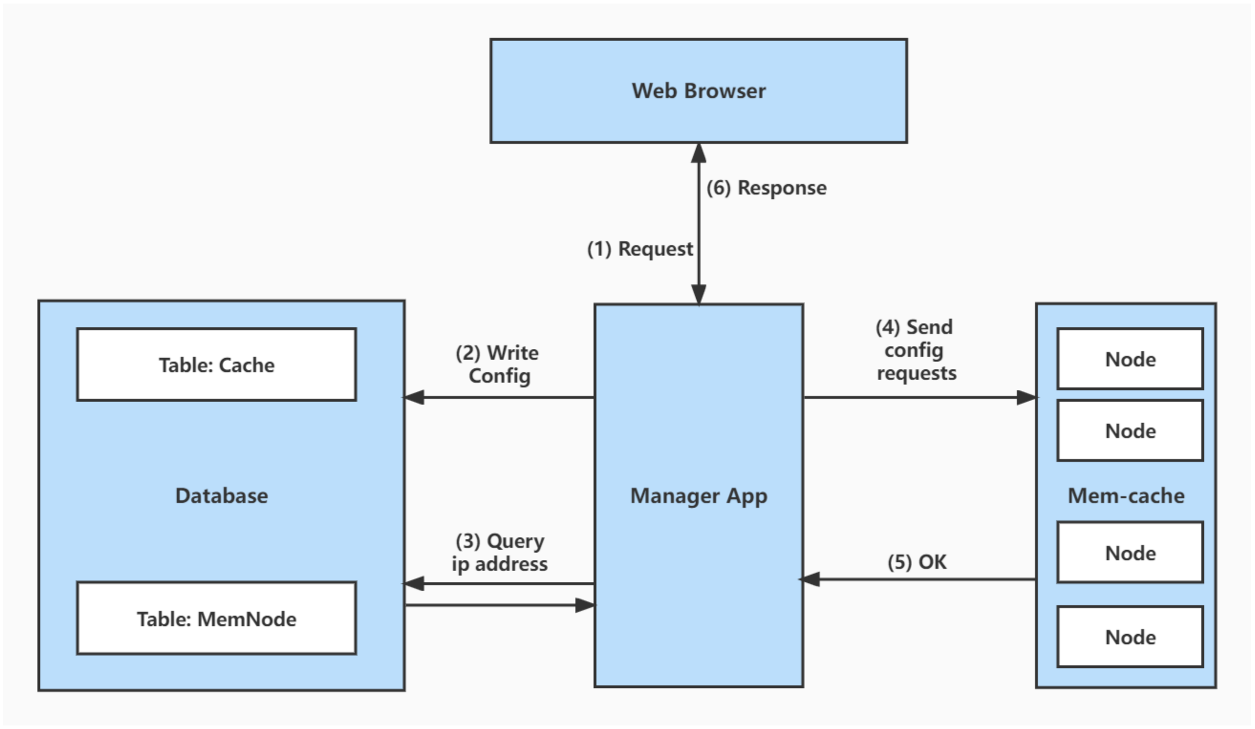


Figure 25

Please note: when a new memcache node is launched, it will first search the configuration information in table *Cache.* If no data exists, then the new memcache will use a set of default parameters and config itself.

Manual Mode for resizing memcache pool

There are two buttons for users to grow or shrink the memcache pool by 1.
Take the grow button for instance, when users click this button, the **pool_grow()** function in the configure.py file will be called.
That function in the **configure.py** file first modifies RDS database table *AutoScaling* to manual mode, then called **grow_by_1()** function in aws.py file.
That function in **aws.py** file will start 1 memcache instance and then send the instance id and public ip to the web server(5000 Port) by http request.
The **add_uhash()** function in the **web app photos.py** file will receive the request, add the new instance information to rds **database** table *MemNode* and change the number of nodes in the hash ring.

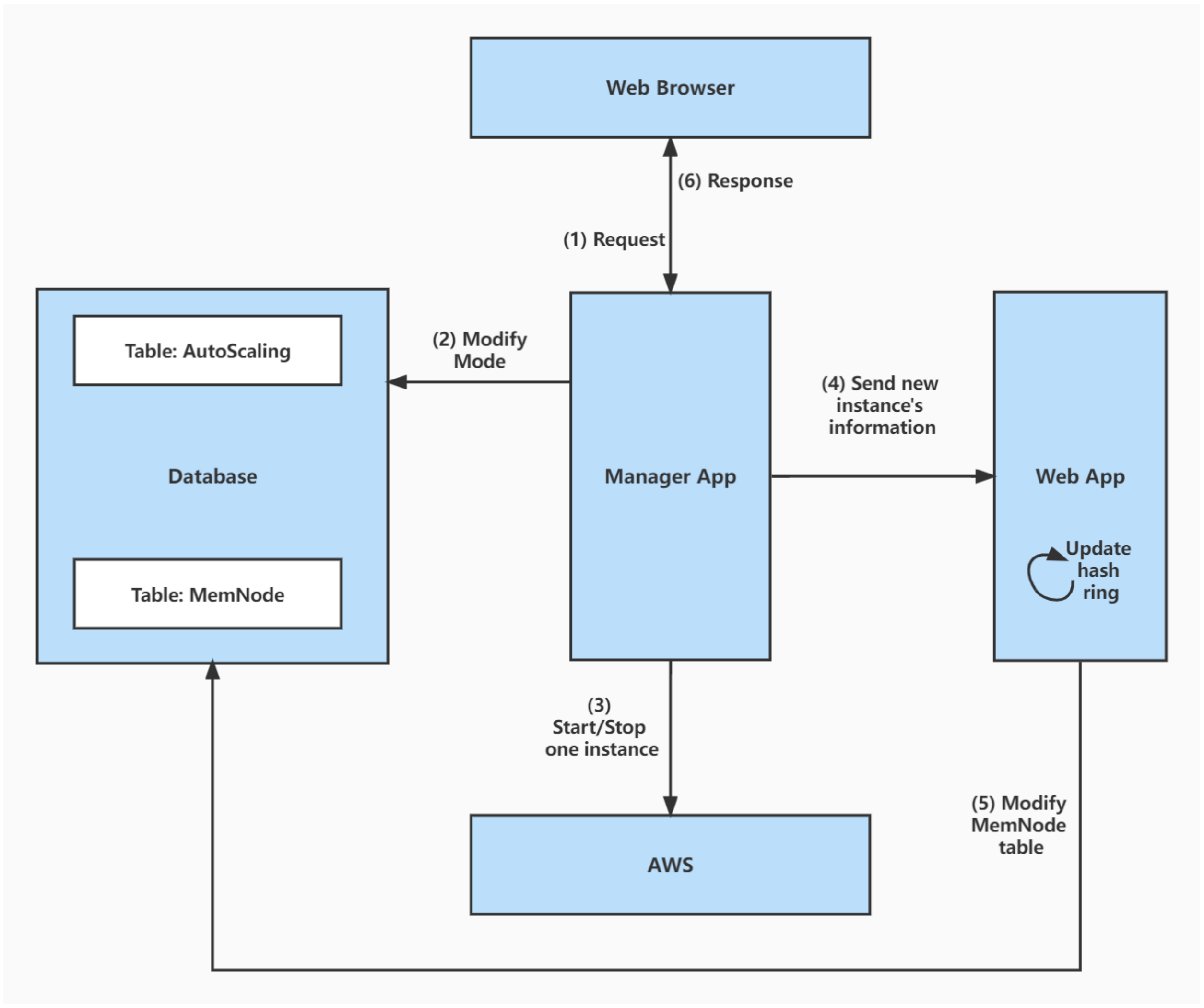


Figure 26

Automatically resizing memcache pool

This feature is implemented independently in the **AutoScaler/auto_scaler.py** file. It will check the pool config mode in RDS **database** table AutoScaling every 5 seconds. If the mode is “auto”, auto-scaler will check the miss rate in **cloudwatch** and the configuration information in the database every 1 minute.
If the miss rate is out of the max threshold or min threshold, the **grow_by_ratio()** or **shrink_by_ratio()** function in **aws.py** file will be called. The implementation logic of the above 2 functions is to iteratively call functions **grow_by_1()** or **shrink_by_1()**

Delete application data

When users send requests at the front end, the **delete_data()** function in the **configure.py** file will be called. This function will clear **photo table** in rds database and delete all the files in **S3 /images** folder

Clear memcache data

When users send requests at the front end, the **clear_cache()** function in the **configure.py** file will be called. This function will first query the **MemNode table** in the rds database and get public ips of all the running memcache instances and send clear requests to all the ips.

API: Web Server

Route requests to memcache using consistent hash

In order to achieve consistent hash, we use the package *uhashring* in python. This package implements a consistent hash ring and some useful functions.
We use **HashRing** data structure and add_node(), remove_node(), and get_node() functions in this package.
The functions **add_node()** and **remove_node()** will be called when the number of memcache node is changed. Their input is the changing memcache instance's public ip address.
The function **get_node()** will be called when invalidate/put/get requests need to be routed to a certain memcache node. This function takes a photo's key as an input and hash it into a 128-bit string by md5 and then return the public ip address of the routed memcache node instance.

S3 and RDS

We store files to S3 by using boto3 s3 client.
We migrate from the local database to the RDS database.

Other implementation logic is the same with Assignment 1.

API: Memcache Server

Publish Cloudwatch Custom Metrics

Different from A1, which uploaded statistics to the database, we now enable the memcache server to store its statistics to AWS CloudWatch periodically.
This function is achieved by our self-defined **AWSClient** class (based on boto3). By using AWSClient object as a handler, we are able to interact with AWS CloudWatch, so as to publish and get CloudWatch metrics efficiently. In our application, the memcache node would call **AWSClient.put_statistics()** every 5 seconds to publish its statistics to the CloudWatch.

Other implementation logic is the same with Assignment 1.

Results

Test Cases Design

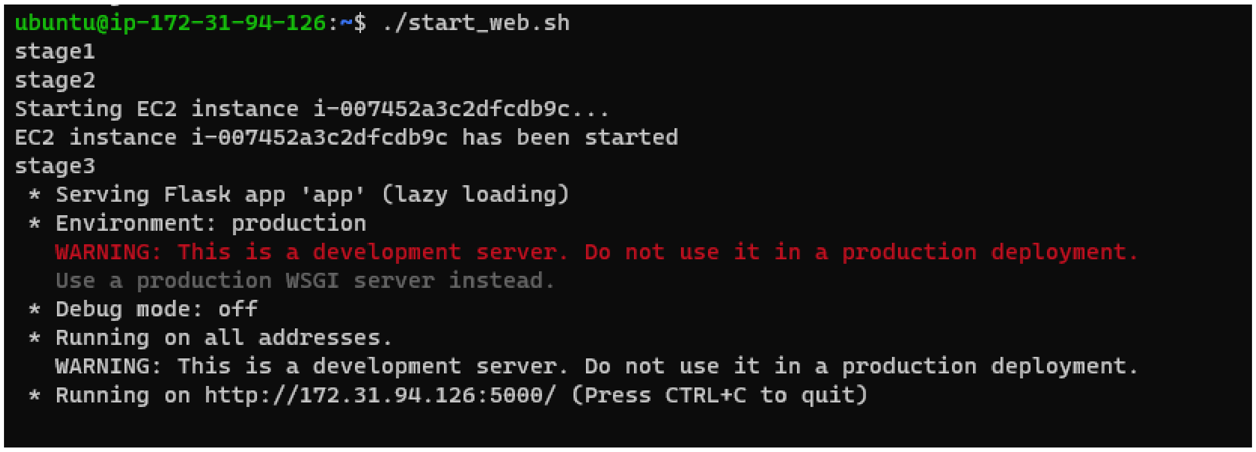
In this part, we will test features of the auto-scaler, which include: parameters configuration, auto-grow, and auto-shrink.
We will first start our application, and perform some tests under different level of load (high/low/balanced) to see the reaction of the auto-scaler.
Here's the definition of the load level:

- High load: the aggregate miss rate exceeds the Max Miss rate threshold
- Low load: the aggregate miss rate is lower than the Min Miss rate threshold
- Balanced load: the aggregate miss rate is between two thresholds

Test Result

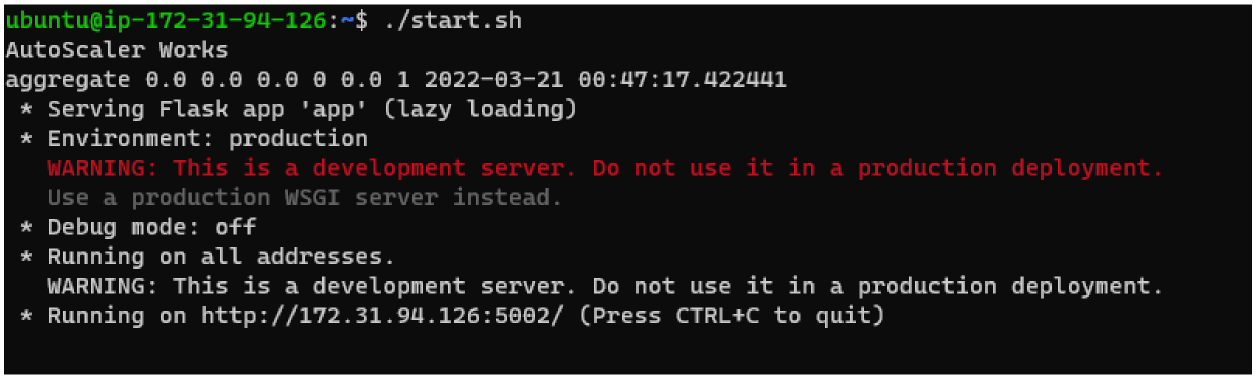
Test Case1

Scenario Description:
The whole system starts from 0. The manager app and web server will be launched, meanwhile, a memcache server automatically start.
Test Data: None
Test Step: Run the launch script of our project.
Expection:
The manager app and web server will start to work, which can be checked in the 5000 and 5002 ports. One memcache instance is launched automatically, and it will be ready in a minute. Then, click the 'Charts' in the manger app web page, you will see there is only 1 worker. Since the no request has been sent, other charts will show 0 value.
Result: test case passed



```
ubuntu@ip-172-31-94-126:~$ ./start_web.sh
stage1
stage2
Starting EC2 instance i-007452a3c2dfcdb9c...
EC2 instance i-007452a3c2dfcdb9c has been started
stage3
 * Serving Flask app 'app' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://172.31.94.126:5000/ (Press CTRL+C to quit)
```

Figure 27



```
ubuntu@ip-172-31-94-126:~$ ./start.sh
AutoScaler Works
aggregate 0.0 0.0 0.0 0 0.0 1 2022-03-21 00:47:17.422441
 * Serving Flask app 'app' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://172.31.94.126:5002/ (Press CTRL+C to quit)
```

Figure 28

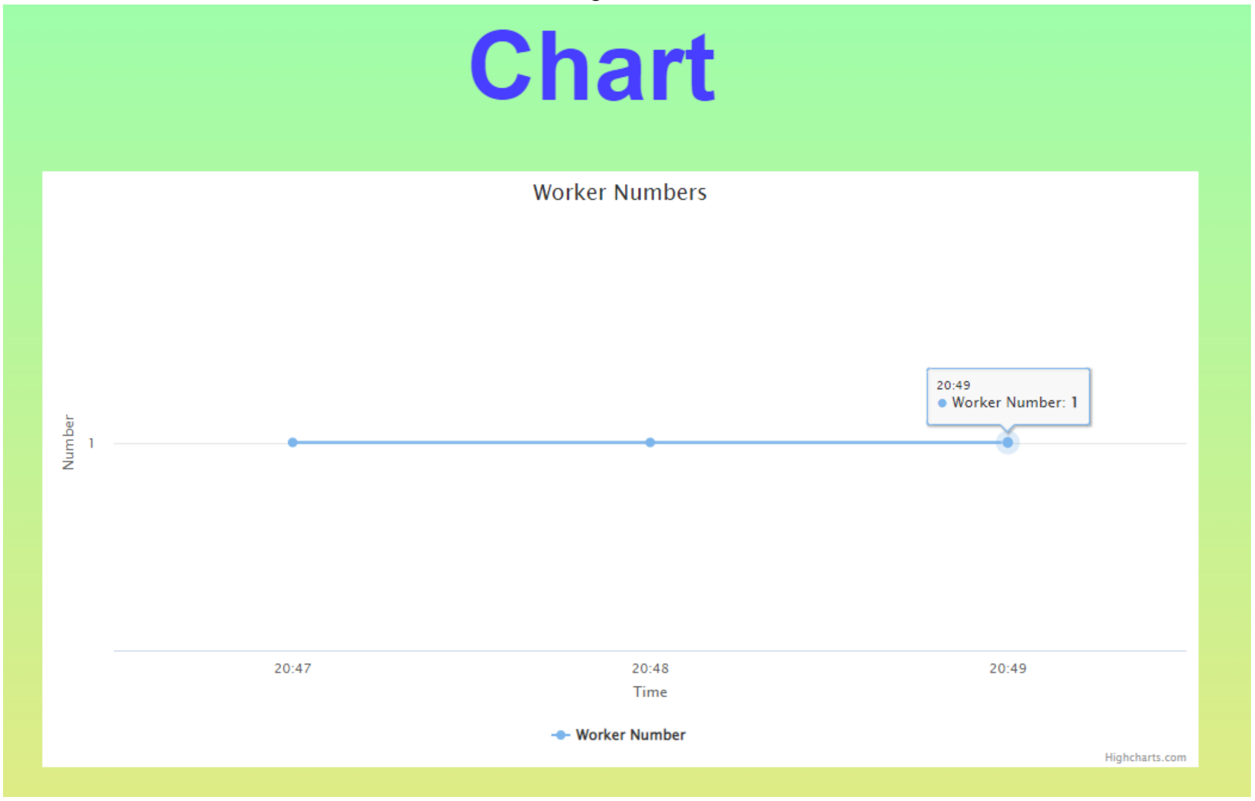


Figure 29

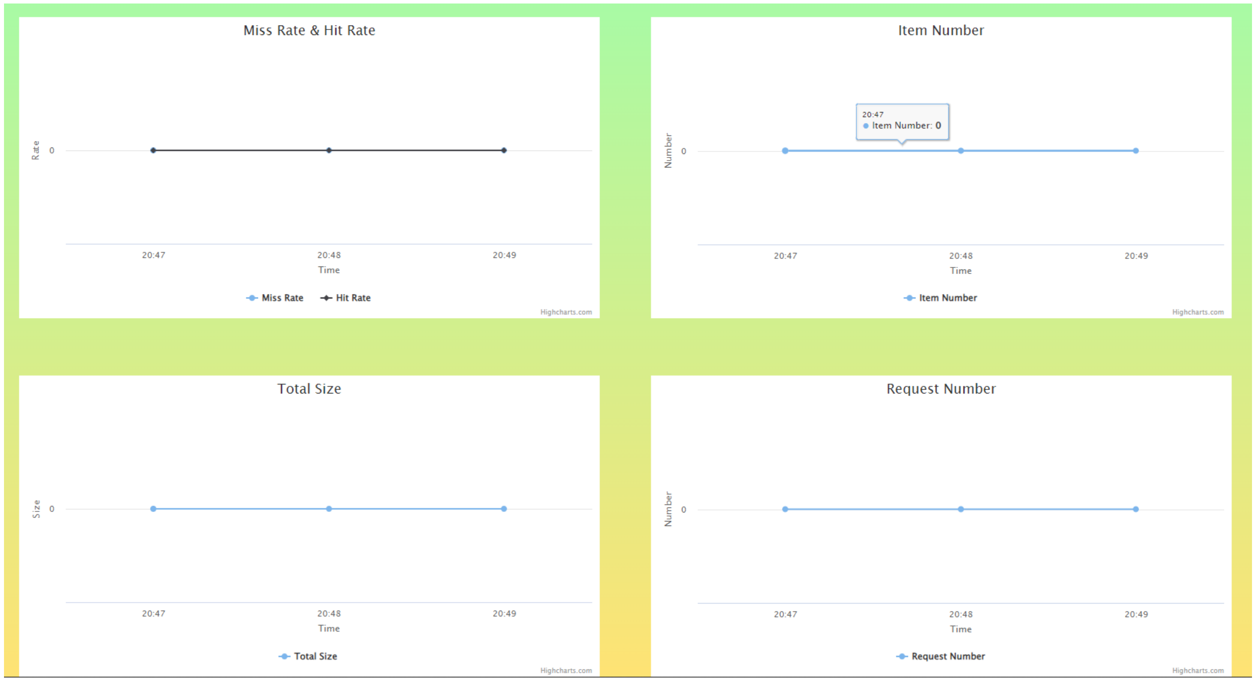


Figure 30

Test Case2

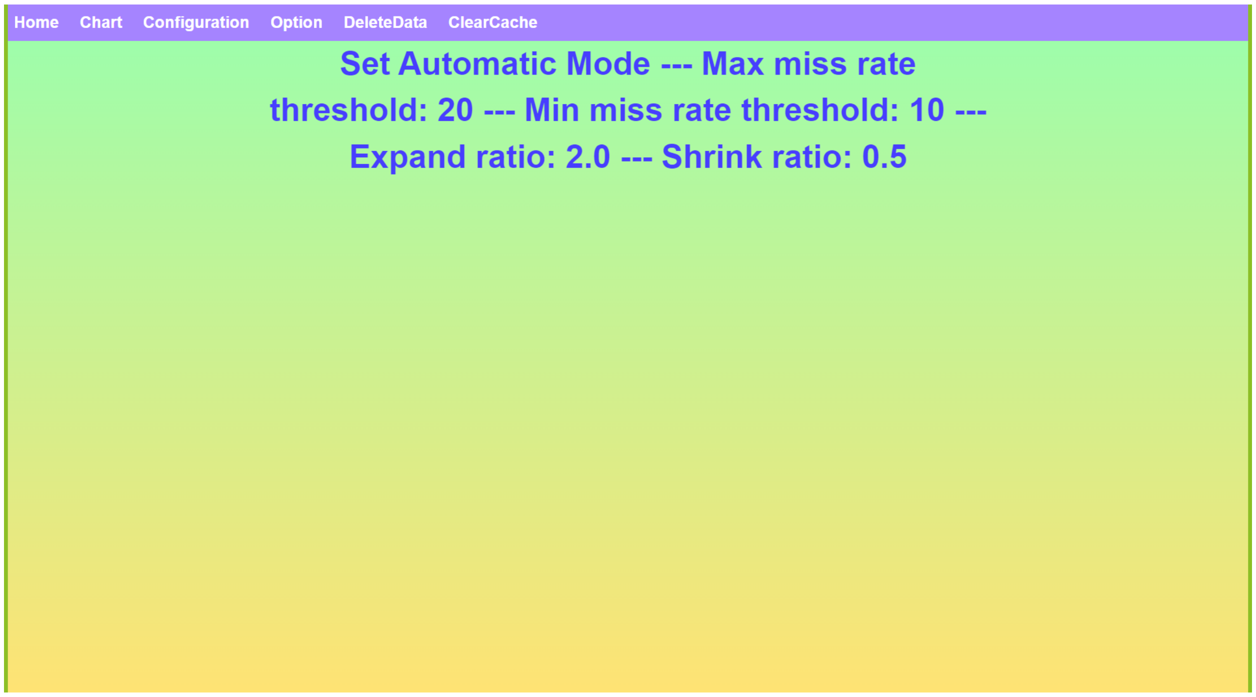
Scenario Description:
Memcache nodes under high load. When the aggregate miss rate exceeds the Max Miss Rate threshold, the auto-scaler will grow the memcache pool to relieve the load.
Test Data:
Auto-scaler config

- Max Miss Rate threshold = 20%
- Min Miss Rate threshold = 10%
- Expand ratio = 2.0
- Shrink ratio = 0.5

Test Step:

- Configure and toggle to automatic mode.
- Keep searching keys that were never uploaded before.
- Monitor the chart to see workers increase.

Expection:
After searching for an unknown key, the aggregate miss rate will increase. Each time the aggregate miss rate exceeds the threshold, the auto-scaler will double the memcache nodes (since the expand ratio is 2.0). We can observe the number of workers growing from 1 to 8 in the chart. Such a process will stop until the aggregate miss rate falls below the threshold or the number of workers reaches the maximum.
Result: test case passed



Set Automatic Mode --- Max miss rate threshold: 20 --- Min miss rate threshold: 10 --- Expand ratio: 2.0 --- Shrink ratio: 0.5

Figure 31

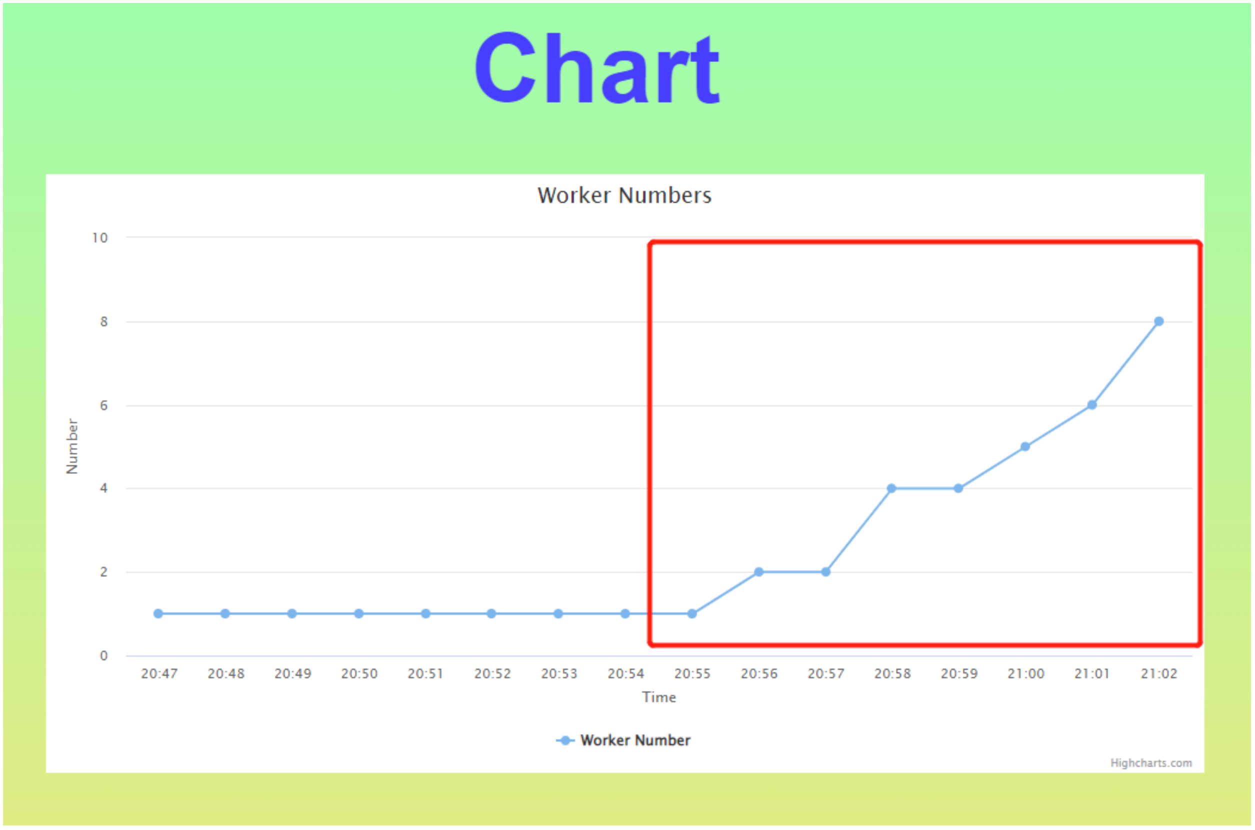


Figure 32

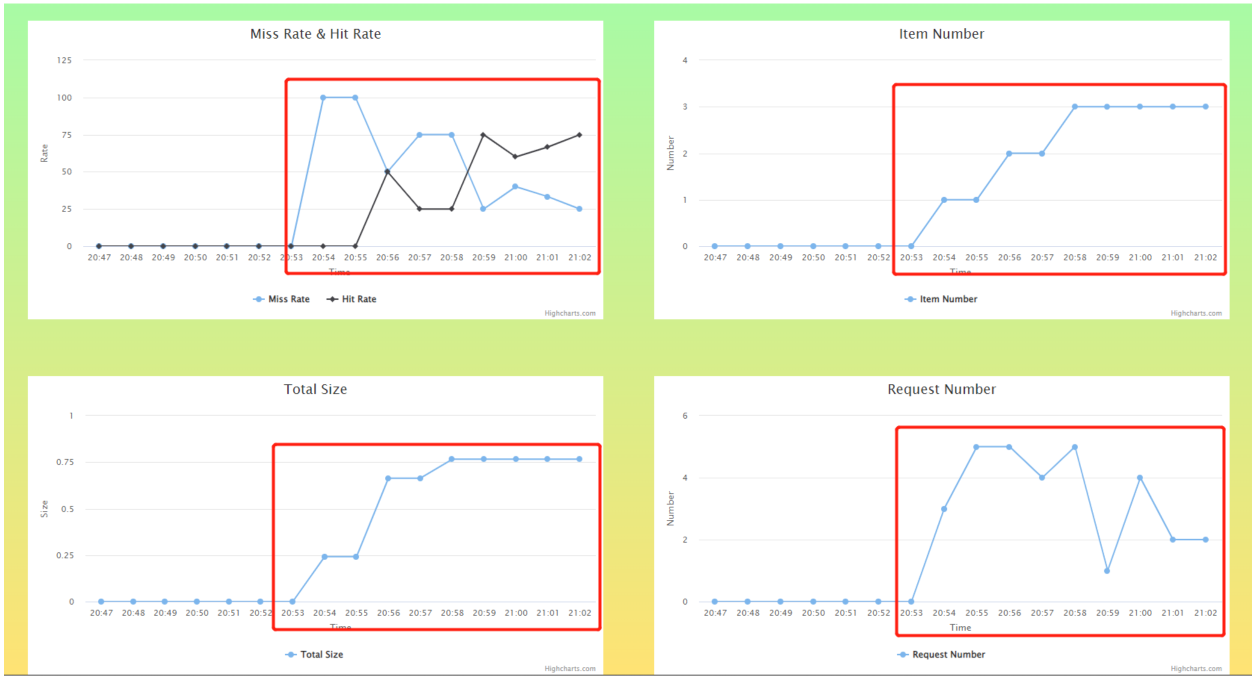


Figure 33

Test Case3

Scenario Description:

Memcache nodes under low load. When the aggregate miss rate is lower than the Min Miss Rate threshold, the auto-scaler will shrink the memcache pool to reduce the resource occupation.

Test Data:

Auto-scaler config

- Max Miss Rate threshold = 90%
- Min Miss Rate threshold = 65%
- Expand ratio = 2.0
- Shrink ratio = 0.5

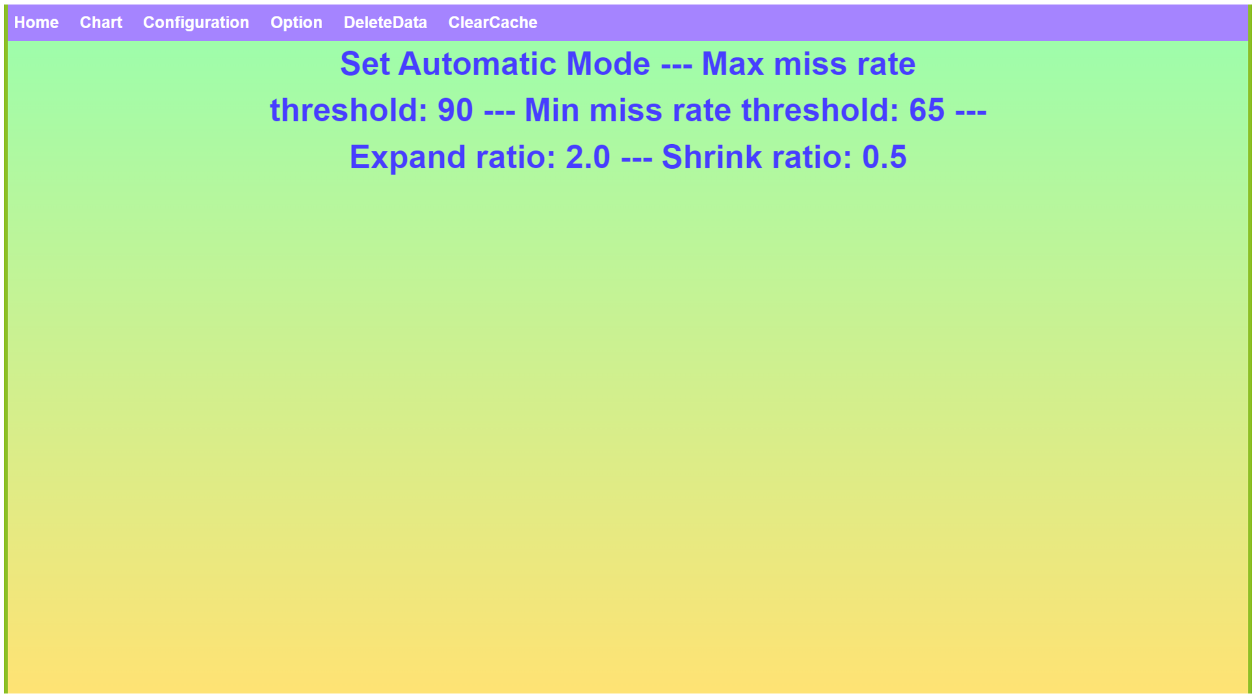
Test Step:

- Use any way you want to increase memcache nodes to 8.
- Configure and toggle to automatic mode.
- Don't search or search for some existing keys to make sure the aggregate miss rate is lower than the threshold.
- Monitor the chart to see workers decrease.

Expection:

When the aggregate miss rate is lower than the Min Miss Rate threshold we can observe that the number of works decrease according the ratio from the chart. Such a process will stop until the aggregate miss rate is larger than the threshold.

Result: test case passed



Set Automatic Mode --- Max miss rate threshold: 90 --- Min miss rate threshold: 65 --- Expand ratio: 2.0 --- Shrink ratio: 0.5

Figure 34

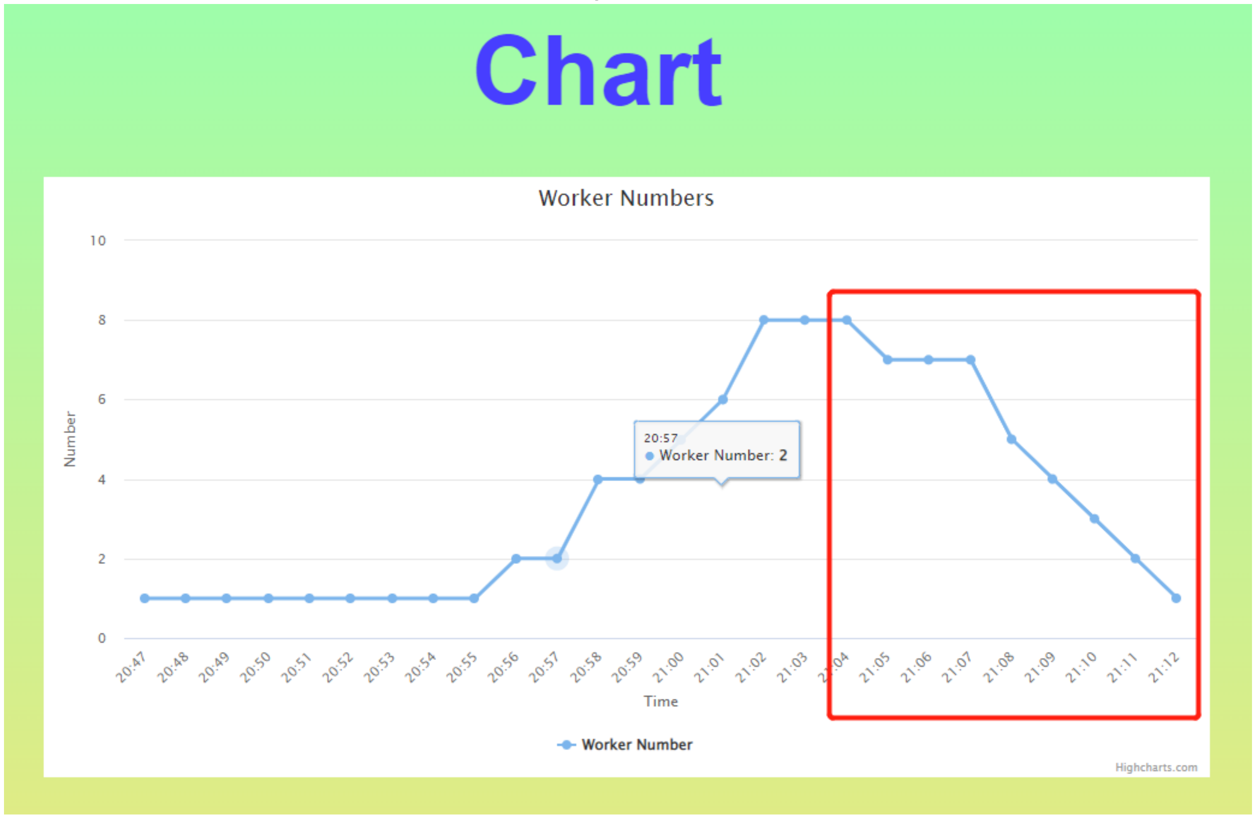


Figure 35

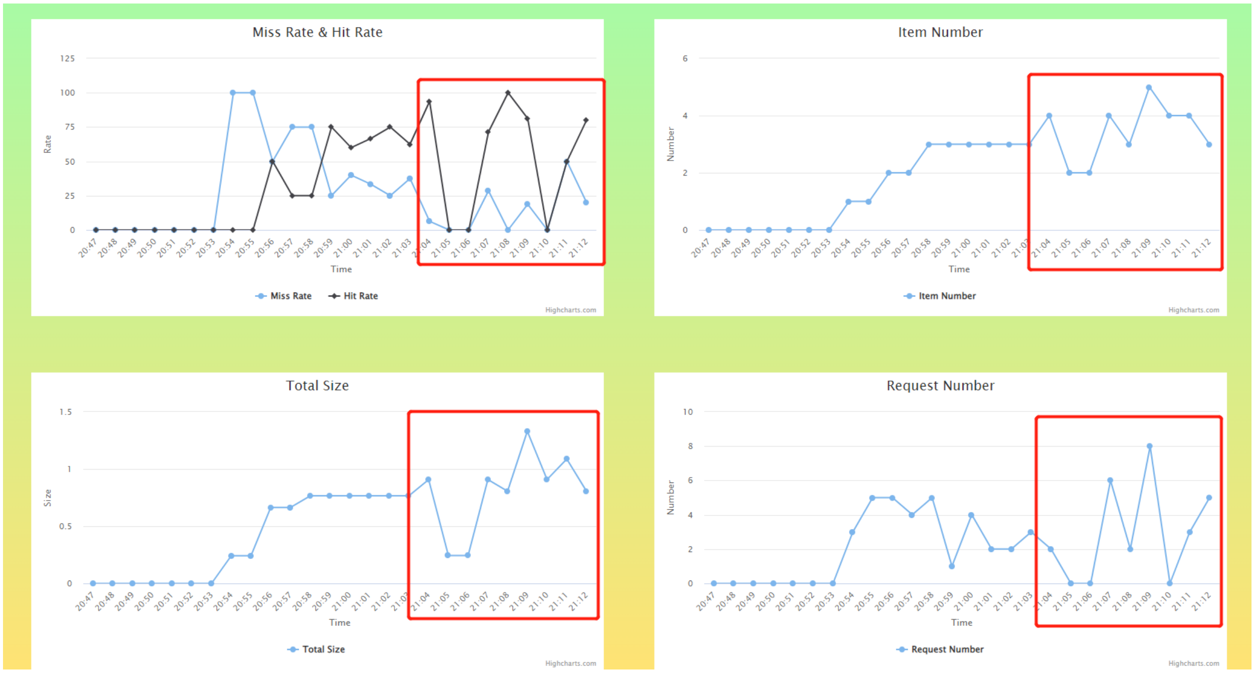


Figure 36

Test Case4

Scenario Description:
Memcache nodes under balanced load. When the aggregate miss rate neither exceeds the Max Miss Rate threshold, nor be lower that Min Miss Rate threshold, the auto-scaler will not change the memcache pool.

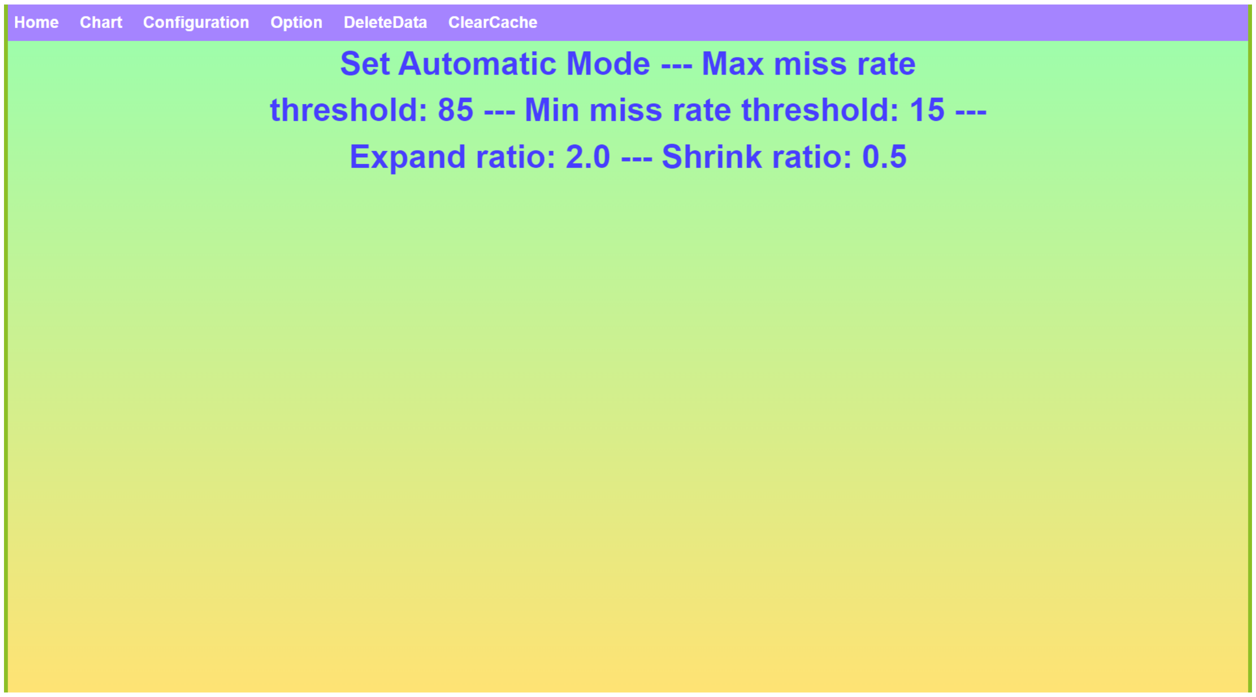
Test Data:
Auto-scaler config

- Max Miss Rate threshold = 85%
- Min Miss Rate threshold = 15%
- Expand ratio = 2.0
- Shrink ratio = 0.5

Test Step:

- Configure and toggle to automatic mode.
- Search for the keys you want, but make sure to keep the miss rate is neither too large nor too small.
- Observe the number of workers in the chart.

Expection:
Since the aggregate miss rate is between the Max threshold and Min threshold, the auto-scaler will maintain the current status, and the number of workers will be stable.
Result: test case passed



Set Automatic Mode --- Max miss rate threshold: 85 --- Min miss rate threshold: 15 --- Expand ratio: 2.0 --- Shrink ratio: 0.5

Figure 37

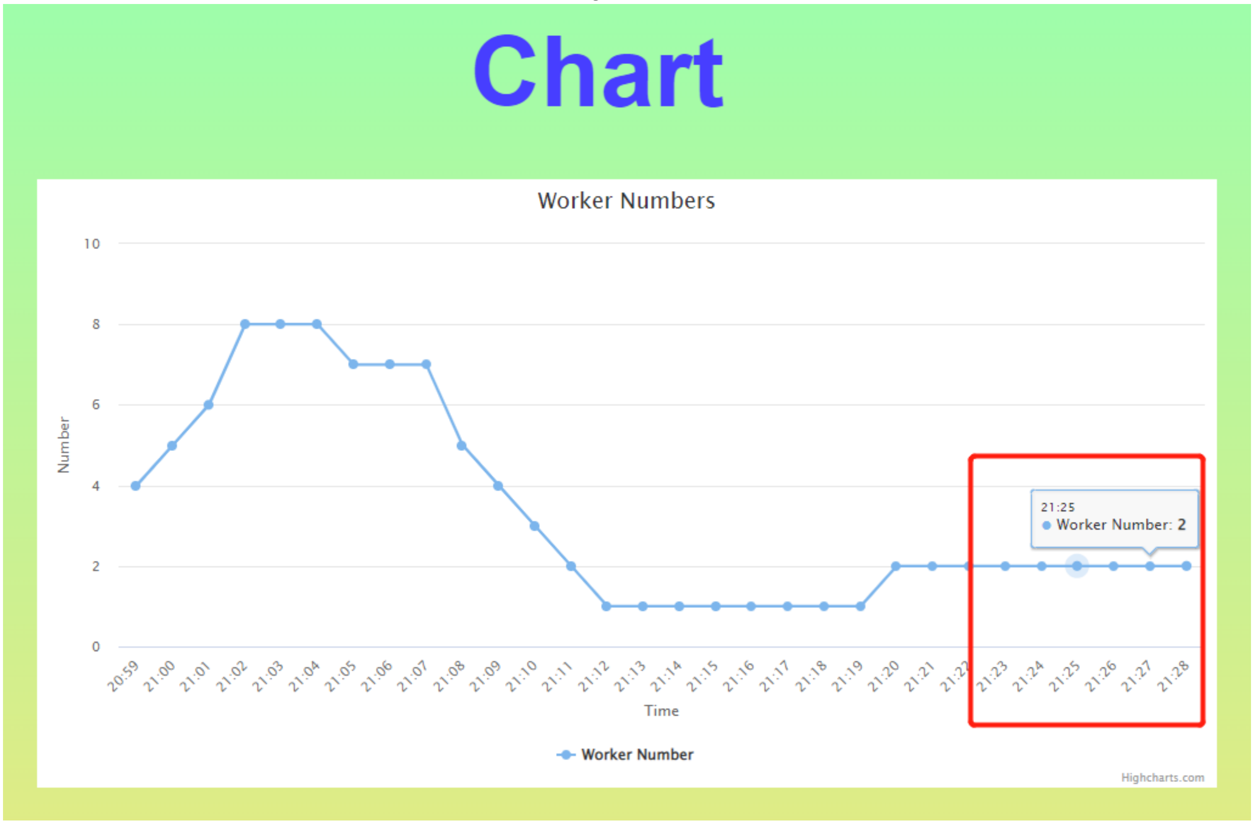
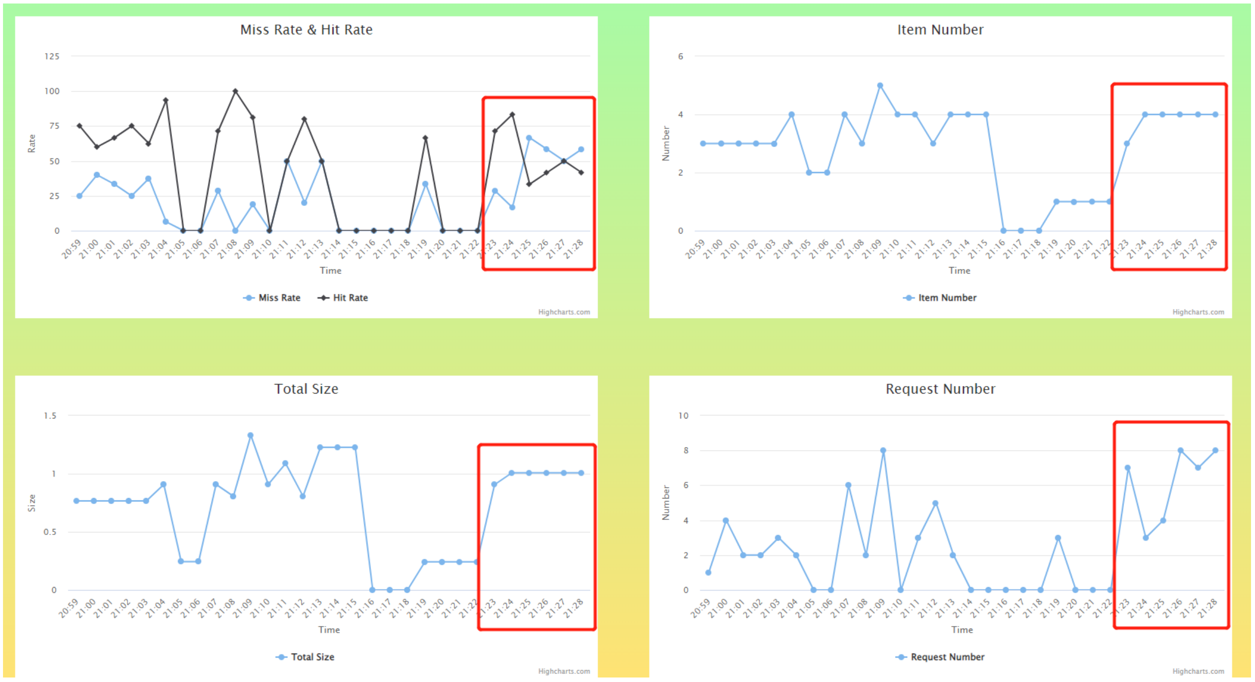


Figure 38



Miss Rate & Hit Rate
Rate
Time
Miss Rate
Hit Rate
Item Number
Number
Time
Item Number
Total Size
Size
Time
Total Size
Request Number
Number
Time
Request Number
Highcharts.com

Figure 39