ICT3102 D2

Assignment 2 – Performance Requirements & Testing

Team 6

Student Name	**Student ID**
Raynold Tan Yong Ren	1902632
Pak Shao Kai	1902698
Muhammad Haikal Bin Othman	1801649

*Text in red are the updates based on D1 feedback

Updated Estimated Workload - Safe Distancing Ambassador (SDA)

Note: data for SDA deployment is not available, the team will use justification to estimate the number of SDA in each scenario

Category	**Value/Working**	**Justification**
Number of Levels in SIT@NYP	**7 Levels**	Source: https://www.singaporetech.edu.sg/sites/default/files/2020-10/Annual_Report_2020.pdf
Number of SDA(s) **(Peak)**	2 SDA for each Level = 2 x 7 = **14 SDA**	Assumption: Peak meaning the highest number of possible SDA within SIT@NYP Building
Number of SDA(s) **(Non-Peak)**	**2 SDA** for Entire SIT@NYP Building	
Rate of calling insert beacon API calls for 1 SDA for 1 Level from mobile to flask server	**1 Call/4 Seconds** Or 15 Calls/Minute	Assumption: 1. SDA is walking from Lobby A to Lobby B for each level 2. Based on our average pacing between the ends of SIT NYP campus, it takes about an average of 4 seconds to cross between rooms with beacons.
Rate of calling insert beacon API Calls (Peak) from mobile to flask server	15 Calls/Minute X 2 SDA X 7 Levels = 210 Calls/Minute **= <u>3.5 Calls/Second</u>**	2 SDA Patrolling Each Level X 7 Levels in SIT@NYP Minimal required throughput of the flask server's insert operations being performed in one second.
Rate of calling insert beacon API Calls from mobile to flask server (Non-Peak)	15 Calls/Minute X 2 SDA = 30 Calls/Minute = **0.5 Calls/Second**	2 SDA Patrolling the Whole Building

Updated Estimated Workload – Admin Monitoring

*Text in red are the updates based on D1 feedback

Note: data for usage of monitoring page are estimates

Category	**Value/Working**	**Justification**
Number of HAWCS Server	1 Server including 14 SDA Staff (Mock HAWCS Server will call only 2 staff every 10 second. Staff 0 and staff 2)	Scaled down mock HAWCS Server stated in assignment package
NYP Opening Hours	7.30am – 7.30pm Total opening hours per day **12 hours**	https://www.nyp.edu.sg/student-life/visit-nyp.html
SIT Admin Staff Working Hours	9.00am – 6.00pm Total Working hours per admin staff = **8 hours**	https://www.mom.gov.sg/employment-practices/hours-of-work-overtime-and-rest-days Assuming the staff follow the 8 hours a day or 44 hours a week stated by MOM
Rate of retrieve API calls made from refreshing of monitoring page	~2 Calls/Minute or **0.0333 Calls/Second**	Assuming the monitoring page is refreshed once every 30 seconds (Manually and automatically)
Rate of retrieve API calls made from HACS server for all SDA staff	For 1 SDA staff = 6 Request/minute or **0.1 Calls/Second** For 14 SDA Staff = 84 Request/minute or **1.4 Calls/Second**	The automatic update request from HACS Server to Flask server is every **10 second**.
Total Rate of calling the retrieve API service in flask server	Monitoring Page Rate + HACS rate 0.0333 + 1.4 = **1.433 Calls/Second**	Minimal required throughput of the flask server's retrieve operations being performed in one second.

Performance Test Playbook

Req No.	PERF-TP-1
Title:	Throughput for inserting detected beacon information from mobile to flask server
Description	In 99% of all cases, the operation shall have a maximal transaction rate of **3.5 Calls/Second** (Peak) and **0.5 Calls/Second** (Non-peak) for uploading information for detected beacon
Test Purpose	To ensure that the system can handle the estimated peak throughput of **3.5 Calls/Second**.
Test Environment	AWS EC2 Instance (Flask Server)
Testing Tool	Jmeter
Remarks	Operational efficacy of monitoring page and HACS load have been included into the test

Load Testing – Test Plan 1

A. Initialise

A1 [AWS CLI] Log into AWS VM CLI
A2 [AWS CLI] Start Docker engine – service start docker
A3 [AWS CLI] Copy project folder into AWS VM using scp
scp -i <<your private key file>> -r ICT3102Team6 ec2-user@<< AWS EC2 >>.ap-southeast-1.compute.amazonaws.com:/home/ec2-user
A4 [AWS CLI] cd into ICT3102Team6 folder and run the following command for server setup:
Bash ICT3102Team6New.sh or ICT3102Team6Old.sh
A5 [Local HACS] Start up HACS server locally and change IP to the AWS IP as well as point the endpoint to port 80
A6 [Monitoring Page] Open browser and key in
New Architecture:
<<AWS IP>>:4000/monitoring to start monitoring page
Old Architecture:
<<AWS IP>>/
A7 [Jmeter] Start Jmeter

B. Execute

B1 [Jmeter] Create a test plan
B2 [Jmeter] Under the test plan, create a thread group
B3 [Jmeter] Under the thread group, create a HTTP request with the following parameters. Server name as <<AWS IP ADDRESS>>, method as "GET", path as "newbeacondetect"
B4 [Jmeter] HTTP request, add get fields for: user_address = 0,
beacon_address = C2A628384B08
rssi =-20
B5 [Jmeter] Specify thread properties to have **210** threads, with **60** second's ramp-up, loop count of **1**
B6 [Jmeter] Duplicate the thread and specify thread properties to have **2100** threads, with **60** second's ramp-up, loop count of **1**
B7 [Jmeter] Save the test plan as PERF-TP-1.jmx
B8 [Jmeter] jmeter -n -l test-results-perf-tp-1.csv -e -o report-folder-perf-tp-1 -t PERF-TP-1.jmx

Variation Note:
TP-1-1: If Testing Sharding (Divide the total amount of threads and ramp up into 5 and insert 5 different staff_id evenly)

C. Record

C1 [Jmeter] Navigate to report-folder-perf-tp-1 and open index to view Jmeter Dashboard
C2 [Jmeter] Navigate to test-results-perf-tp-1.csv to view summary report

D. Clean-up

D1 [AWS CLI] Stop, delete and clear all container data and volumes
docker kill $(docker ps -q)
docker rm $(docker ps -a -q)
docker volume prune --force
D2 [Jmeter] Delete the output Jmeter files and folders

Performance Test Playbook

Req No.	PERF-TP-2
Title:	Throughput for extracting all user locations between flask server to HACS server.
Description	In 99% of all cases, the operation shall have an average transaction rate of **1.433 Calls/Second** to view the location history of a staff
Test Purpose	To ensure that the system can handle the estimated peak throughput of **1.433 Calls/Second** for the retrieve operation
Test Environment	AWS EC2 Instance (Flask Server)
Testing Tool	Jmeter
Remarks	Operational efficacy of the load for mobile insert API calls are not included in the test. Monitoring page load has been included

Load Testing – Test Plan 2

A. Initialise

A1 [AWS CLI] Log into AWS VM CLI
A2 [AWS CLI] Start Docker engine – service start docker
A3 [AWS CLI] Copy project folder into AWS VM using scp
scp -i <<your private key file>> -r ICT3102Team6 ec2-user@<< AWS EC2 >>.ap-southeast-1.compute.amazonaws.com:/home/ec2-user
A4 [AWS CLI] cd into ICT3102Team6 folder and run the following command for server setup:
Bash ICT3102Team6New.sh or ICT3102Team6Old.sh

A5 [Monitoring Page] Open browser and key in
New Architecture:
<<AWS IP>>:4000/monitoring to start monitoring page
Old Architecture:
<<AWS IP>>/
A6 [Jmeter] Start Jmeter

B. Execute

B1 [Jmeter] Create a test plan
B2 [Jmeter] Under the test plan, create a thread group
B3 [Jmeter] Under the thread group, create a HTTP request with the following parameters. Server name as <<AWS IP ADDRESS>>, method as “GET”, path as “extractbeacon”
B4 [Jmeter] HTTP request, add get fields for:
- staff_id = 0,
- start_time = 0
- end_time = 1700000000

B5 [Jmeter] Specify thread properties to have **86** threads, with **60** second's ramp-up, loop count of **1**
B6 [Jmeter] Save the test plan as PERF-TP-2.jmx
B7 [Jmeter] jmeter -n -l test-results-perf-tp-2.csv -e -o report-folder-perf-tp-2 -t PERF-TP-2.jmx

C. Record

C1 [Jmeter] Navigate to report-folder-perf-tp-2 and open index to view Jmeter Dashboard
C2 [Jmeter] Navigate to test-results-perf-tp-2.csv to view summary report

D. Clean-up

D1 [AWS CLI] Stop, delete and clear all container data and volumes
```
docker kill $(docker ps -q)
docker rm $(docker ps -a -q)
docker volume prune --force
```
D2 [Jmeter] Delete the output Jmeter files and folders

Performance Test Playbook

Req No.	PERF-RT-1
Title:	Response time for inserting detected beacon information from mobile to flask server
Description	In 99% of all cases, the operation shall have a mean response time of no more than **1 Seconds** to upload the information of the detected beacon
Test Purpose	A stress test will be conducted to determine the **performance bottleneck of the insert operation and what issues arise** when the system crashes.
Test Environment	AWS EC2 Instance (Flask Server)
Testing Tool	Jmeter
Remarks	Operational efficacy of monitoring page and HACS load have been included into the test

Stress Testing – Test Plan 3

A. Initialise

A1 [AWS CLI] Log into AWS VM CLI
A2 [AWS CLI] Start Docker engine – service start docker
A3 [AWS CLI] Copy project folder into AWS VM using scp
scp -i <<your private key file>> -r ICT3102Team6 ec2-user@<< AWS EC2 >>.ap-southeast-1.compute.amazonaws.com:/home/ec2-user
A4 [AWS CLI] cd into ICT3102Team6 folder and run the following command for server setup:
Bash ICT3102Team6New.sh or ICT3102Team6Old.sh
A5 [Local HACS] Start up HACS server locally and change IP to the AWS IP as well as point the endpoint to port 80
A6 [Monitoring Page] Open browser and key in
New Architecture:
<<AWS IP>>:4000/monitoring to start monitoring page
Old Architecture:
<<AWS IP>>/
A7 [Jmeter] Start Jmeter

B. Execute

B1 [Jmeter] Create a test plan
B2 [Jmeter] Under the test plan, create a thread group
B3 [Jmeter] Under the thread group, create a HTTP request with the following parameters. Server name as <<AWS IP ADDRESS>>, method as "GET", path as "newbeacondetect"
B4 [Jmeter] HTTP request, add get fields for: user_address = 0,
beacon_address = C2A628384B08
rssi =-20
B5 [Jmeter] Specify thread properties to have **2100** threads, with **60** second's ramp-up, loop count of **1**
B6 [Jmeter] Save the test plan as PERF-RT-1.jmx
B7 [Jmeter] jmeter -n -l test-results-perf-rt-1.csv -e -o report-folder-perf-tp-1 -t PERF-RT-1.jmx
B8 [Jmeter] Gradually increase the user threads until an error or an unfavourable response time is observed

Variation Note:
RT-1-1: If Testing Sharding (Change the staff ID per round)

C. Record

C1 [Jmeter] Navigate to report-folder-perf-rt-1 and open index to view Jmeter Dashboard
C2 [Jmeter] Navigate to test-results-perf-rt-1.csv to view summary report
C3 [Jmeter] Check if average response time is below 1000ms

D. Clean-up

D1 [AWS CLI] Stop, delete and clear all container data and volumes
```
docker kill $(docker ps -q)
docker rm $(docker ps -a -q)
docker volume prune --force
```
D2 [Jmeter] Delete the output Jmeter files and folders

Performance Test Playbook

Req No.	PERF-RT-2
Title:	Response time for extracting all user locations between flask server to HACS server.
Description	In 99% of all cases, the operation shall have a mean response time of no more than **6 Second** to viewing location history of a user
Test Purpose	A stress test will be conducted to determine the **maximum data load of the retrieve operation and what issues arise** when the system crashes.
Test Environment	AWS EC2 Instance (Flask Server)
Testing Tool	Jmeter
Remarks	Operational efficacy of the load for mobile insert API calls are not included in the test. Monitoring page load has been included

Stress Testing – Test Plan 4

A. Initialise

A1 [AWS CLI] Log into AWS VM CLI
A2 [AWS CLI] Start Docker engine – service start docker
A3 [AWS CLI] Copy project folder into AWS VM using SCP
scp -i <<your private key file>> -r ICT3102Team6 ec2-user@<< AWS EC2 >>.ap-southeast-1.compute.amazonaws.com:/home/ec2-user
A4 [AWS CLI] cd into ICT3102Team6 folder and run the following command for server setup:
Bash ICT3102Team6New.sh or ICT3102Team6Old.sh

A5 [Monitoring Page] Open browser and key in
A6 [Monitoring Page] Open browser and key in
New Architecture:
<<AWS IP>>:4000/monitoring to start monitoring page
Old Architecture:
<<AWS IP>>/
A7 [Jmeter] Start Jmeter

B. Execute

B1 [Jmeter] Create a test plan
B2 [Jmeter] Under the test plan, create a thread group
B3 [Jmeter] Under the thread group, create a HTTP request with the following parameters. Server name as <<AWS IP ADDRESS>>, method as "GET", path as "newbeacondetect"
B4 [Jmeter] HTTP request, add get fields for: staff_id = 0,
start_time = 0
end_time = 1700000000
B5 [Jmeter] Specify thread properties to have **860** threads, with **60** second's ramp-up, loop count of **1**

B6 [Jmeter] Save the test plan as PERF-RT-2.jmx
B7 [MongoDB] Add **600** location data for staff 0
B8 [Jmeter] jmeter -n -l test-results-perf-rt-2.csv -e -o report-folder-perf-tp-2 -t PERF-RT-2.jmx
B9 [Jmeter] Gradually increase the number of location data in the staff collection by **600** and **change the staff_id** depending in test objective.
Variation Note:
RT-2: Testing Circuit Breaker (insert 600 staff 0 each round)
RT-2-1" Testing Sharding (Insert 600 data and change the staff_id per round)

C. Record

C1 [Jmeter] Navigate to report-folder-perf-rt-2 and open index to view Jmeter Dashboard
C2 [Jmeter] Navigate to test-results-perf-rt-2.csv to view summary report
C3 [Jmeter] Check if average response time is below 6000ms

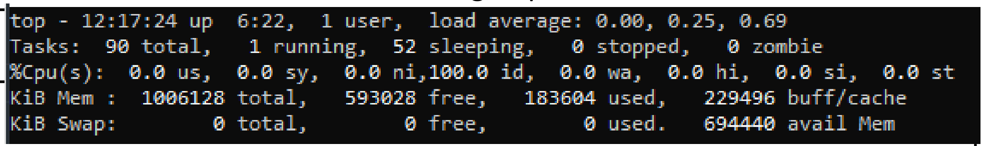
D. Clean-up

D1 [AWS CLI] Stop, delete and clear all container data and volumes
```
docker kill $(docker ps -q)
docker rm $(docker ps -a -q)
docker volume prune --force
```
D2 [Jmeter] Delete the output Jmeter files and folders

Updated Test Environment

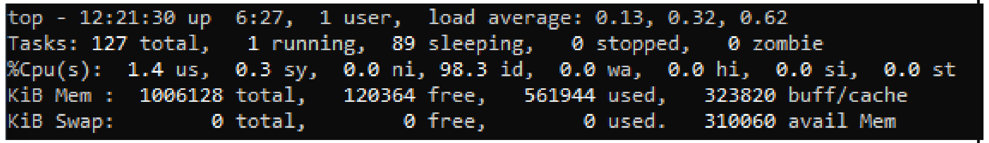
Flask Server Environment	System Description
AWS EC2 Instance (t2.micro) Amazon Linux 2 AMI (HVM), SSD Volume Type - ami-024221a59c9020e72 (64-bit x86) / ami-0a2263b7710a89837 (64-bit Arm) Operating system: Linux CPU: Intel® Single Core Scalable Processor 3.3Ghz RAM: 1GiB	As free memory is very limited, the strategy that the team will be implementing is to limit the maximum memory usage of each docker container, providing more priority for MongoDB containers as their recommended min RAM is 256MB. This is due to the WiredTigerCache setup used by MongoDB. Container Memory Limit Optimization 9 Docker Containers (425MB average usage as shown below)

AWS t2.micro, no containers running, top command. ~ 590MB free RAM

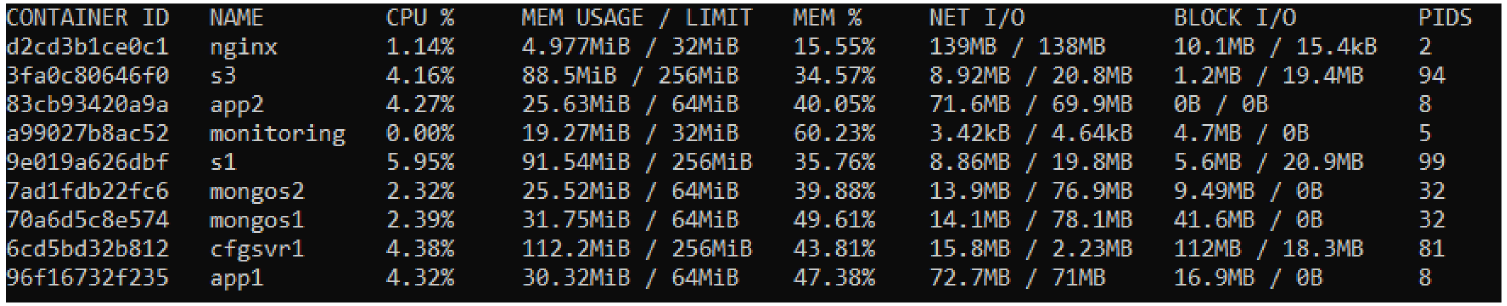


```
top - 12:17:24 up  6:22,  1 user,  load average: 0.00, 0.25, 0.69
Tasks:  90 total,   1 running,  52 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1006128 total,   593028 free,   183604 used,   229496 buff/cache
KiB Swap:        0 total,        0 free,        0 used.   694440 avail Mem
```

AWS t2.micro, 9 containers running, top command. ~ 120MB free RAM

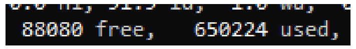


```
top - 12:21:30 up  6:27,  1 user,  load average: 0.13, 0.32, 0.62
Tasks: 127 total,   1 running,  89 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1.4 us,  0.3 sy,  0.0 ni, 98.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1006128 total,   120364 free,   561944 used,   323820 buff/cache
KiB Swap:        0 total,        0 free,        0 used.   310060 avail Mem
```



```
CONTAINER ID   NAME         CPU %    MEM USAGE / LIMIT     MEM %     NET I/O           BLOCK I/O         PIDS
d2cd3b1ce0c1   nginx        1.14%    4.977MiB / 32MiB      15.55%    139MB / 138MB     10.1MB / 15.4kB   2
3fa0c80646f0   s3           4.16%    88.5MiB / 256MiB      34.57%    8.92MB / 20.8MB   1.2MB / 19.4MB    94
83cb93420a9a   app2         4.27%    25.63MiB / 64MiB      40.05%    71.6MB / 69.9MB   0B / 0B           8
a99027b8ac52   monitoring   0.00%    19.27MiB / 32MiB      60.23%    3.42kB / 4.64kB   4.7MB / 0B        5
9e019a626dbf   s1           5.95%    91.54MiB / 256MiB     35.76%    8.86MB / 19.8MB   5.6MB / 20.9MB    99
7ad1fdb22fc6   mongos2      2.32%    25.52MiB / 64MiB      39.88%    13.9MB / 76.9MB   9.49MB / 0B       32
70a6d5c8e574   mongos1      2.39%    31.75MiB / 64MiB      49.61%    14.1MB / 78.1MB   41.6MB / 0B       32
6cd5bd32b812   cfgsvr1      4.38%    112.2MiB / 256MiB     43.81%    15.8MB / 2.23MB   112MB / 18.3MB    81
96f16732f235   app1         4.32%    30.32MiB / 64MiB      47.38%    72.7MB / 71MB     16.9MB / 0B       8
```

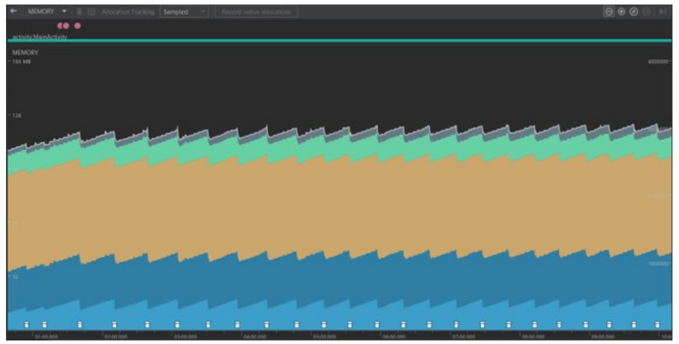
Average free memory
From 60mb to 80mb



```
 88080 free,    650224 used,
```

We have noted that the total memory limits set surpass the available free memory and could be a possible issue once the 3 MongoDB containers start to consume their maximum memory capacity (768MB) before clearing their wiredTigerCache. We have also made additional settings to the mongo instance to cap the cache size to 250MB (minimum setting) as the default instead takes the total system memory of 1GiB before resetting

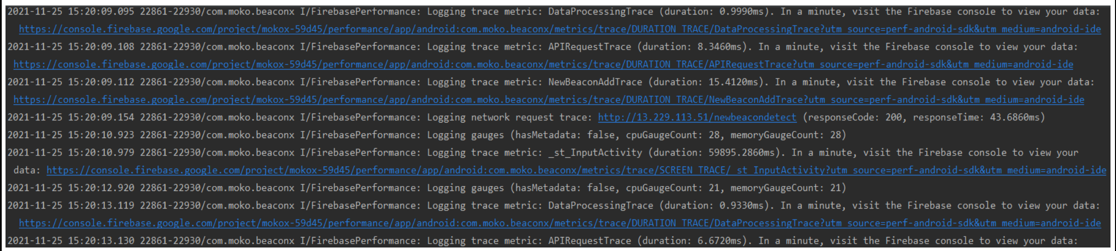
Docker Compose command: mongod --configsvr --replSet cfgrs --port 27017 --dbpath /data/db --wiredTigerCacheSizeGB 0.25

Test Mobile Device	System Description	
Google Pixel 2 XL	OS: Android 10 Chipset: Qualcomm MSM8998 Snapdragon 835 (10 nm) CPU: Octa-core (4x2.35 GHz Kryo & 4x1.9 GHz Kryo) GPU: Adreno 540 WLAN: Wi-Fi 802.11 a/b/g/n/ac Bluetooth: 5.0, A2DP, LE, aptX HD	When the phone is scanning for the Moko Beacons, Google Pixel 2 XL's memory usage hover around 100MB usage while running the optimized application. 

Mobile Optimization Testing: API Call Tests

Throughput (calls/second)

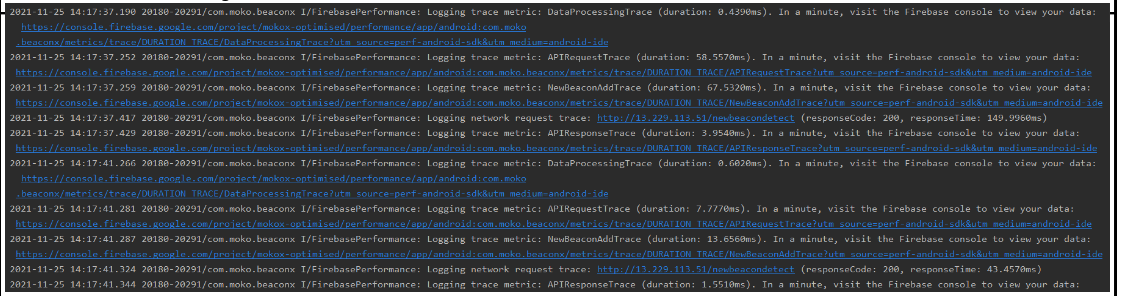
Before Optimisation



```
2021-11-25 15:20:09.095 22861-22930/com.moko.beaconx I/FirebasePerformance: Logging trace metric: DataProcessingTrace (duration: 0.9990ms). In a minute, visit the Firebase console to view your data:
  https://console.firebase.google.com/project/mokox-59d45/performance/app/android:com.moko.beaconx/metrics/trace/DURATION_TRACE/DataProcessingTrace?utm_source=perf-android-sdk&utm_medium=android-ide
2021-11-25 15:20:09.108 22861-22930/com.moko.beaconx I/FirebasePerformance: Logging trace metric: APIRequestTrace (duration: 8.3460ms). In a minute, visit the Firebase console to view your data:
 https://console.firebase.google.com/project/mokox-59d45/performance/app/android:com.moko.beaconx/metrics/trace/DURATION_TRACE/APIRequestTrace?utm_source=perf-android-sdk&utm_medium=android-ide
2021-11-25 15:20:09.112 22861-22930/com.moko.beaconx I/FirebasePerformance: Logging trace metric: NewBeaconAddTrace (duration: 15.4120ms). In a minute, visit the Firebase console to view your data:
 https://console.firebase.google.com/project/mokox-59d45/performance/app/android:com.moko.beaconx/metrics/trace/DURATION_TRACE/NewBeaconAddTrace?utm_source=perf-android-sdk&utm_medium=android-ide
2021-11-25 15:20:09.154 22861-22930/com.moko.beaconx I/FirebasePerformance: Logging network request trace: http://13.229.113.51/newbeacondetect (responseCode: 200, responseTime: 43.6860ms)
2021-11-25 15:20:10.923 22861-22930/com.moko.beaconx I/FirebasePerformance: Logging gauges (hasMetadata: false, cpuGaugeCount: 28, memoryGaugeCount: 28)
2021-11-25 15:20:10.979 22861-22930/com.moko.beaconx I/FirebasePerformance: Logging trace metric: _st_InputActivity (duration: 59895.2860ms). In a minute, visit the Firebase console to view your
 data: https://console.firebase.google.com/project/mokox-59d45/performance/app/android:com.moko.beaconx/metrics/trace/SCREEN_TRACE/_st_InputActivity?utm_source=perf-android-sdk&utm_medium=android-ide
2021-11-25 15:20:12.920 22861-22930/com.moko.beaconx I/FirebasePerformance: Logging gauges (hasMetadata: false, cpuGaugeCount: 21, memoryGaugeCount: 21)
2021-11-25 15:20:13.119 22861-22930/com.moko.beaconx I/FirebasePerformance: Logging trace metric: DataProcessingTrace (duration: 0.9330ms). In a minute, visit the Firebase console to view your data:
  https://console.firebase.google.com/project/mokox-59d45/performance/app/android:com.moko.beaconx/metrics/trace/DURATION_TRACE/DataProcessingTrace?utm_source=perf-android-sdk&utm_medium=android-ide
2021-11-25 15:20:13.130 22861-22930/com.moko.beaconx I/FirebasePerformance: Logging trace metric: APIRequestTrace (duration: 6.6720ms). In a minute, visit the Firebase console to view your data:
```

In this test, Firebase Performance Monitoring was used to get the throughput of the API that was developed before any optimization was done. To give context, a single API call for testing here would consist of detecting and sorting beacon data and sending the request through the API. Firebase Performance Monitoring was used to trace the timing and displaying it as a log entry in logcat. The initial targeted throughput for the application was 1 call/4 seconds. As shown above, the expected throughput was achieved from timestamp 15:20:09 to 15:20:12. This was observed throughout the duration of the test.

After Optimisation



```
2021-11-25 14:17:37.190 20180-20291/com.moko.beaconx I/FirebasePerformance: Logging trace metric: DataProcessingTrace (duration: 0.4390ms). In a minute, visit the Firebase console to view your data:
  https://console.firebase.google.com/project/mokox-optimised/performance/app/android:com.moko
 .beaconx/metrics/trace/DURATION_TRACE/DataProcessingTrace?utm_source=perf-android-sdk&utm_medium=android-ide
2021-11-25 14:17:37.252 20180-20291/com.moko.beaconx I/FirebasePerformance: Logging trace metric: APIRequestTrace (duration: 58.5570ms). In a minute, visit the Firebase console to view your data:
 https://console.firebase.google.com/project/mokox-optimised/performance/app/android:com.moko.beaconx/metrics/trace/DURATION_TRACE/APIRequestTrace?utm_source=perf-android-sdk&utm_medium=android-ide
2021-11-25 14:17:37.259 20180-20291/com.moko.beaconx I/FirebasePerformance: Logging trace metric: NewBeaconAddTrace (duration: 67.5320ms). In a minute, visit the Firebase console to view your data:
 https://console.firebase.google.com/project/mokox-optimised/performance/app/android:com.moko.beaconx/metrics/trace/DURATION_TRACE/NewBeaconAddTrace?utm_source=perf-android-sdk&utm_medium=android-ide
2021-11-25 14:17:37.417 20180-20291/com.moko.beaconx I/FirebasePerformance: Logging network request trace: http://13.229.113.51/newbeacondetect (responseCode: 200, responseTime: 149.9960ms)
2021-11-25 14:17:37.429 20180-20291/com.moko.beaconx I/FirebasePerformance: Logging trace metric: APIResponseTrace (duration: 3.9540ms). In a minute, visit the Firebase console to view your data:
 https://console.firebase.google.com/project/mokox-optimised/performance/app/android:com.moko.beaconx/metrics/trace/DURATION_TRACE/APIResponseTrace?utm_source=perf-android-sdk&utm_medium=android-ide
2021-11-25 14:17:41.266 20180-20291/com.moko.beaconx I/FirebasePerformance: Logging trace metric: DataProcessingTrace (duration: 0.6020ms). In a minute, visit the Firebase console to view your data:
  https://console.firebase.google.com/project/mokox-optimised/performance/app/android:com.moko
 .beaconx/metrics/trace/DURATION_TRACE/DataProcessingTrace?utm_source=perf-android-sdk&utm_medium=android-ide
2021-11-25 14:17:41.281 20180-20291/com.moko.beaconx I/FirebasePerformance: Logging trace metric: APIRequestTrace (duration: 7.7770ms). In a minute, visit the Firebase console to view your data:
 https://console.firebase.google.com/project/mokox-optimised/performance/app/android:com.moko.beaconx/metrics/trace/DURATION_TRACE/APIRequestTrace?utm_source=perf-android-sdk&utm_medium=android-ide
2021-11-25 14:17:41.287 20180-20291/com.moko.beaconx I/FirebasePerformance: Logging trace metric: NewBeaconAddTrace (duration: 13.6560ms). In a minute, visit the Firebase console to view your data:
 https://console.firebase.google.com/project/mokox-optimised/performance/app/android:com.moko.beaconx/metrics/trace/DURATION_TRACE/NewBeaconAddTrace?utm_source=perf-android-sdk&utm_medium=android-ide
2021-11-25 14:17:41.324 20180-20291/com.moko.beaconx I/FirebasePerformance: Logging network request trace: http://13.229.113.51/newbeacondetect (responseCode: 200, responseTime: 43.4570ms)
2021-11-25 14:17:41.344 20180-20291/com.moko.beaconx I/FirebasePerformance: Logging trace metric: APIResponseTrace (duration: 1.5510ms). In a minute, visit the Firebase console to view your data:
```

In a similar test, after some optimization and changes made to the code, the throughput for the application was still about 1 call/4 seconds. Firebase Performance Monitoring was also used here to trace the API call and the timing, displaying them in the logcat. As shown above, the expected throughput was achieved from timestamp 14:17:37 to 14:17:41. This was observed throughout the duration of the test.

Response Time (ms)

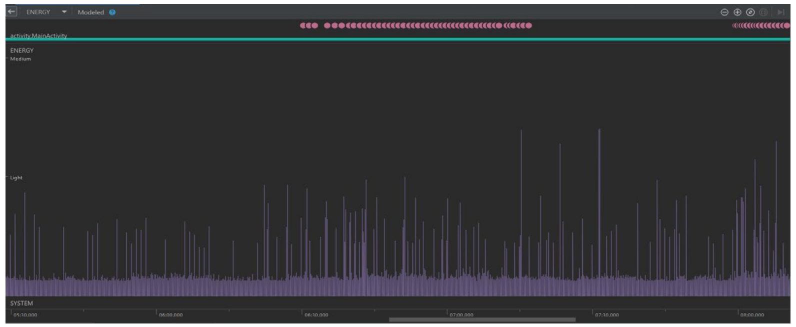
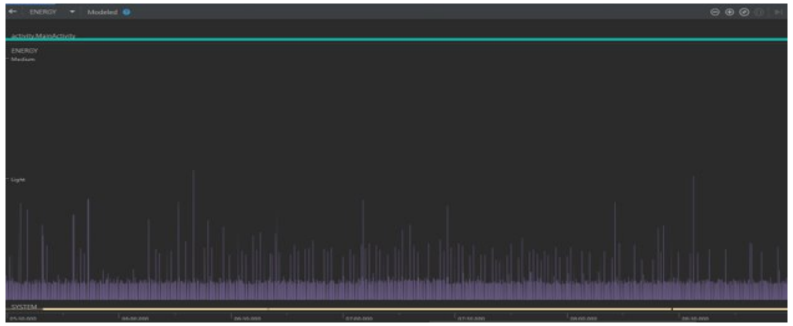
Calls	Before Optimisation	After Optimisation
1	0.6490 + 7.0530 = 7.702	0.6160 + 7.6330 = 8.249
2	0.3570 + 8.8310 = 9.188	0.1360 + 2.4290 = 2.565
3	0.3160 + 8.1340 = 8.45	0.4020 + 6.6390 = 7.041
4	0.7010 + 8.2870 = 8.988	0.7190 + 6.6850 = 7.404
5	0.7070 + 7.2500 = 7.957	0.3930 + 7.9940 = 8.387
6	0.4830 + 8.6970 = 9.18	0.4450 + 8.6290 = 9.074
7	0.3530 + 10.4490 = 10.802	0.4750 + 9.4450 = 9.92
8	0.4950 + 10.5070 = 11.002	0.4450 + 9.2860 = 13.731
9	0.4280 + 10.3650 = 10.793	0.4730 + 7.9510 = 8.424
10	0.4200 + 7.1620 = 7.582	0.4100 + 6.7430 = 7.153
Average	9.1644	8.1948

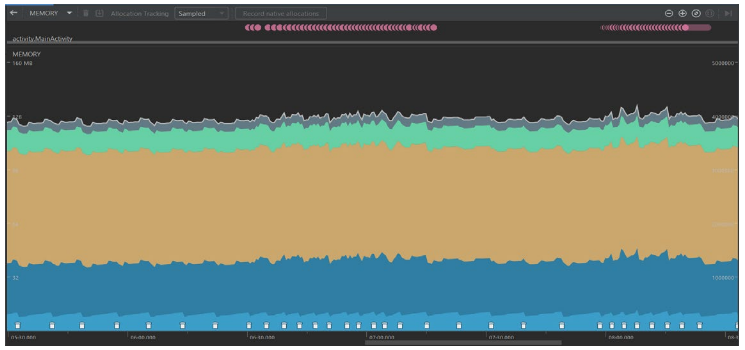
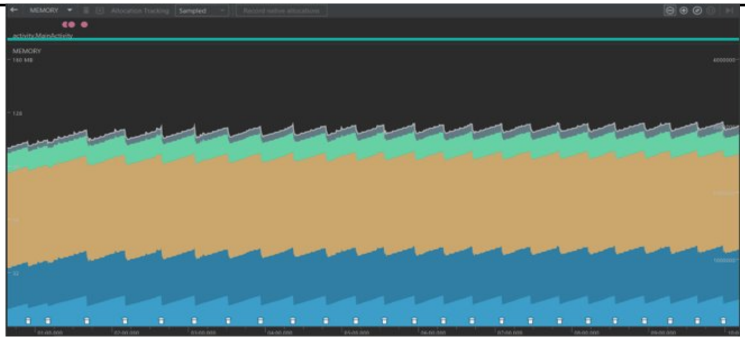
In this test, the total response time was captured for the API call. To give context, an API call would consist of detecting and sorting beacon data and sending the request through the API.

Firebase Performance Monitoring was used to trace at which the API was called and in capturing the response times. 10 calls were observed, and their responses were recorded. Then, the average out of their total response times would be taken to compare. As observed, the response time for the optimised application had a slight improvements.

On hindsight, both before and after optimised application is able to handle the required 1 Call every 4 Seconds as stated in the Estimated Workload.

Mobile Optimization Testing: Energy and Memory Usage

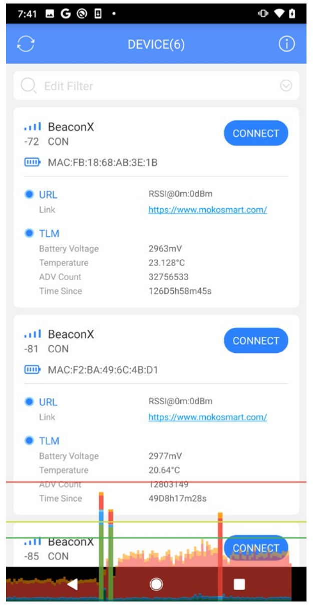
Energy Usage	
Before Optimisation	 In this test, the battery usage was monitored while using the application. The app was made to run for 10 minutes and the Android Studio Profiler was used to monitor the energy levels of the application throughout the duration. As observed, the energy usage was at the higher end of the "light" classification for most of the 10 minutes.
After Optimisation	 In a similar test, the application was optimized such that irrelevant information to the user was removed entirely from view. The application was also made to run for 10 minutes while the energy levels were monitored. As observed above, there was a slight improvement to the energy levels as the energy usage was more on the lower end of the "light" classification. We observed that, having less elements to load and display had reduced the amount of energy required for the application.

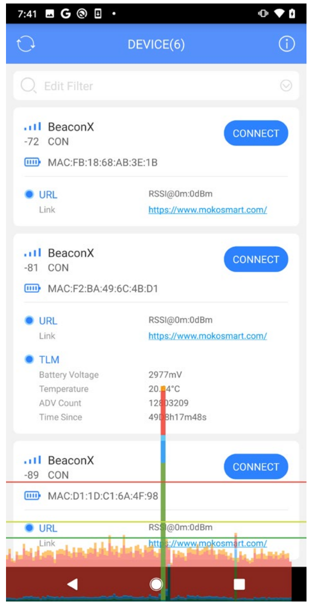
Memory Usage	
Before Optimisation	 In this test, the memory usage was monitored while using the application. The application was made to run for 10 minutes and the Android Studio Profiler was used to monitor the memory usage while using the application. As observed, the application used an average of 125.2MB of memory for the application.
After Optimisation	 In a similar test, the application was optimized and made to run for another 10 minutes. The Android Studio Profiler was used to monitor the memory usage of the application during the 10 minutes it was running. As observed, the application used an average about 118.3MB of memory which was a slight improvement to the one observed before optimization. It is observed that removing redundant code and irrelevant elements to load allowed for lesser usage in memory, resulting in the given improvement.

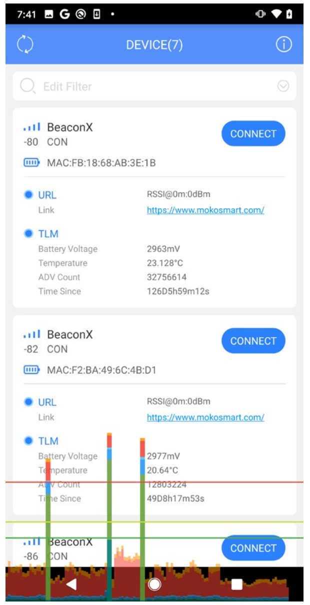
Mobile Optimization Testing: GPU Rendering Profile

GPU Rendering Profile

Before Optimisation


7:41
DEVICE(6)
Edit Filter
BeaconX -72 CON CONNECT
MAC:FB:18:68:AB:3E:1B
URL RSSI@0m:0dBm
Link https://www.mokosmart.com/
TLM
Battery Voltage 2963mV
Temperature 23.128°C
ADV Count 32756533
Time Since 126D5h58m45s
BeaconX -81 CON CONNECT
MAC:F2:BA:49:6C:4B:D1
URL RSSI@0m:0dBm
Link https://www.mokosmart.com/
TLM
Battery Voltage 2977mV
Temperature 20.64°C
ADV Count 12803149
Time Since 49D8h17m28s
BeaconX -85 CON CONNECT

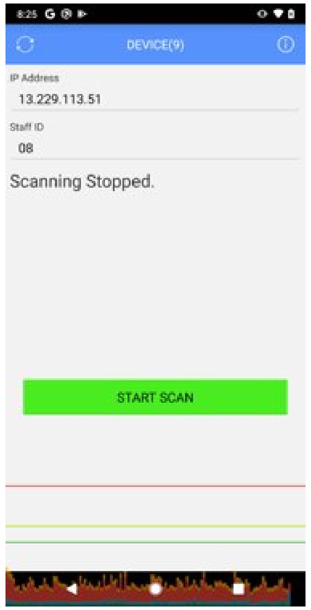

7:41
DEVICE(6)
Edit Filter
BeaconX -72 CON CONNECT
MAC:FB:18:68:AB:3E:1B
URL RSSI@0m:0dBm
Link https://www.mokosmart.com/
BeaconX -81 CON CONNECT
MAC:F2:BA:49:6C:4B:D1
URL RSSI@0m:0dBm
Link https://www.mokosmart.com/
TLM
Battery Voltage 2977mV
Temperature 20.64°C
ADV Count 12803209
Time Since 49D8h17m48s
BeaconX -89 CON CONNECT
MAC:D1:1D:C1:6A:4F:98
URL RSSI@0m:0dBm
Link https://www.mokosmart.com/

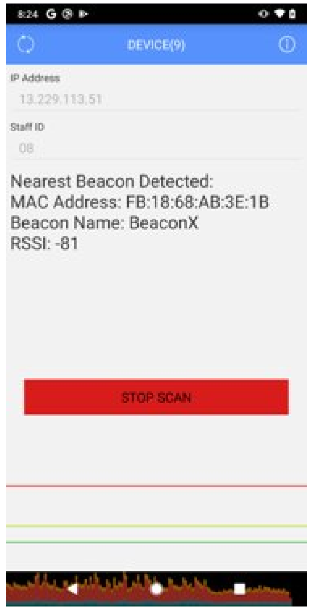

7:41
DEVICE(7)
Edit Filter
BeaconX -80 CON CONNECT
MAC:FB:18:68:AB:3E:1B
URL RSSI@0m:0dBm
Link https://www.mokosmart.com/
TLM
Battery Voltage 2963mV
Temperature 23.128°C
ADV Count 32756614
Time Since 126D5h59m12s
BeaconX -82 CON CONNECT
MAC:F2:BA:49:6C:4B:D1
URL RSSI@0m:0dBm
Link https://www.mokosmart.com/
TLM
Battery Voltage 2977mV
Temperature 20.64°C
ADV Count 12803224
Time Since 49D8h17m53s
BeaconX -86 CON CONNECT

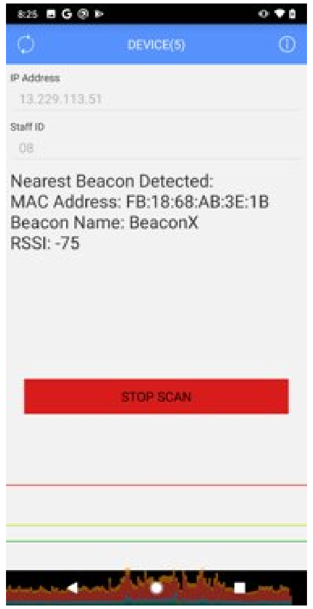
In this test, the GPU Rendering profile is tested and analysed using the phone's HWUI Profiler in the Pixel 2 XL. The profiler identifies if there any pauses and stutters in the animation from running the application. The application is made to run for 10 minutes and the line graphs on the screen are produced. If the graph passes the green bar (16ms), it means that there will be stutters in animation.

As shown, throughout the application's runtime, there were multiple spikes in the render time which is above the green bar, thus, resulting in slight skips in animation. Although not frequent, it is observed that this happens when a card of beacon information is loaded and displayed. Furthermore, the tall line graph shows that a lot of time is spent on drawing the layouts and processing the view hierarchy. Not only that, but it is also shown that a view binder, the recycler view, is taking a long time to process.

After Optimisation


8:25
DEVICE(9)
IP Address
13.229.113.51
Staff ID
08
Scanning Stopped.
START SCAN

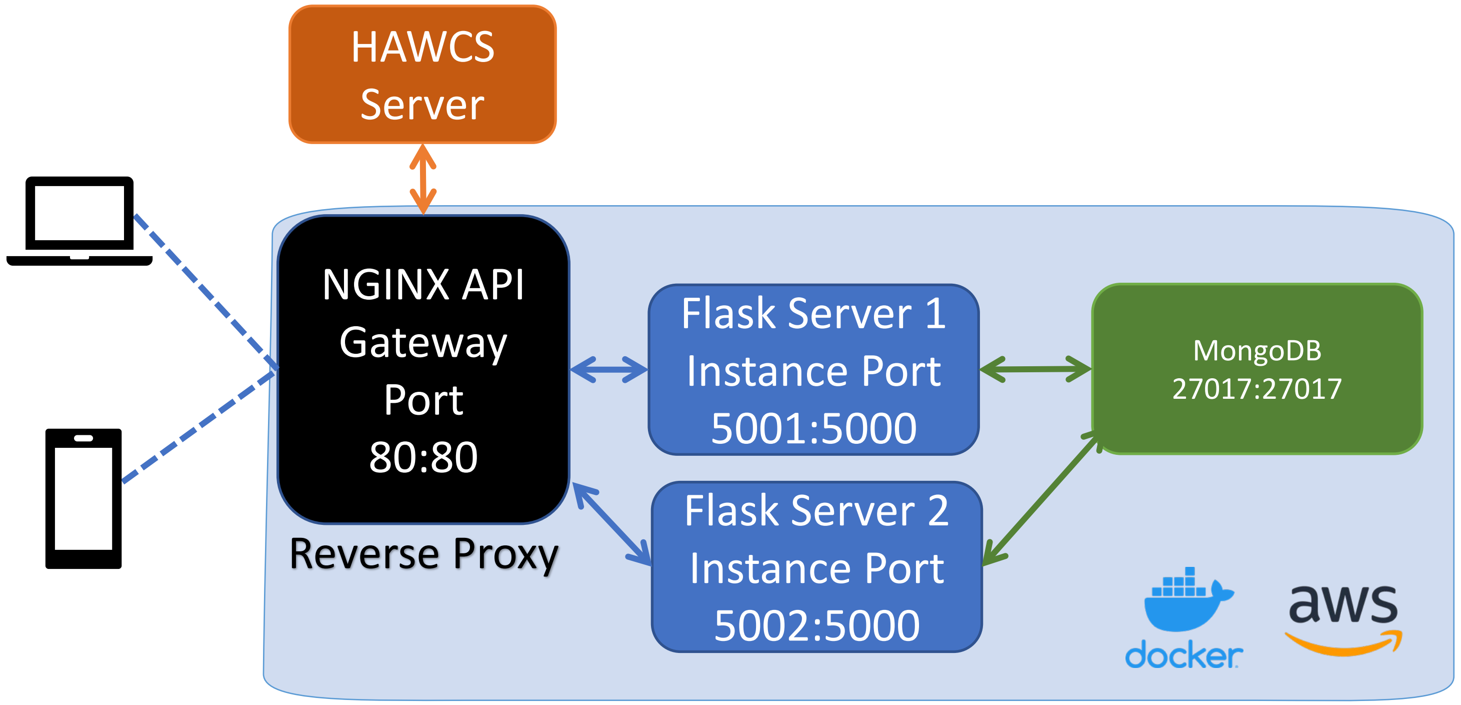

8:24
DEVICE(9)
IP Address
13.229.113.51
Staff ID
08
Nearest Beacon Detected:
MAC Address: FB:18:68:AB:3E:1B
Beacon Name: BeaconX
RSSI: -81
STOP SCAN


8:25
DEVICE(5)
IP Address
13.229.113.51
Staff ID
08
Nearest Beacon Detected:
MAC Address: FB:18:68:AB:3E:1B
Beacon Name: BeaconX
RSSI: -75
STOP SCAN

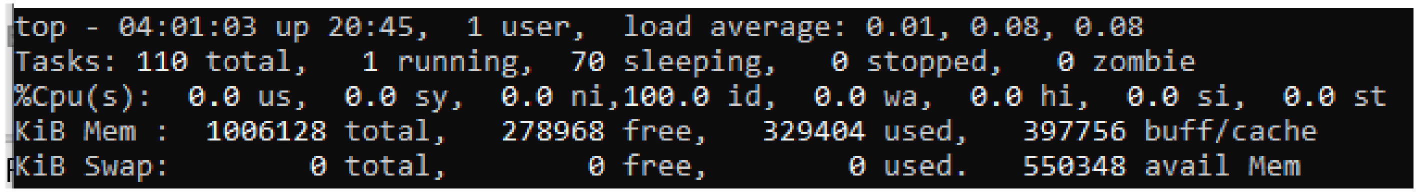
In a similar test, the GPU Rendering profile is tested using the phone's HWUI Profiler. The application is optimized by removing the irrelevant information that were loaded in the form of a recycler view. The view is simplified to only minimal changing elements. In addition, the application was made to run for 10 minutes, and the line graphs were observed in that time.

As observed, there was minimal to almost no spikes in the graph. The graph did not pass the green bar, thus, achieving a consistent 60 frames per second. Furthermore, there graphs showed that by removing irrelevant elements in the UI, minimal processing was required for the views, the view hierarchy. In addition, by removing the need for a recycler view, it minimised the slowdown that a view binder causes.

D1 Flask Server Architecture Before Optimisation

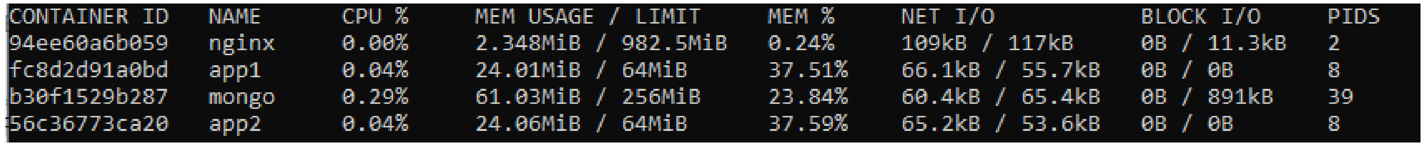

HAWCS Server
NGINX API Gateway Port 80:80
Reverse Proxy
Flask Server 1 Instance Port 5001:5000
Flask Server 2 Instance Port 5002:5000
MongoDB 27017:27017
docker
aws

AWS t2.micro, 4 containers running, top command. ~ 270MB free RAM



```
top - 04:01:03 up 20:45,  1 user,  load average: 0.01, 0.08, 0.08
Tasks: 110 total,   1 running,  70 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1006128 total,   278968 free,   329404 used,   397756 buff/cache
KiB Swap:        0 total,        0 free,        0 used.   550348 avail Mem
```

4 Docker Containers (111MB average usage as shown below)



```
CONTAINER ID   NAME     CPU %    MEM USAGE / LIMIT     MEM %    NET I/O           BLOCK I/O      PIDS
94ee60a6b059   nginx    0.00%    2.348MiB / 982.5MiB   0.24%    109kB / 117kB     0B / 11.3kB    2
fc8d2d91a0bd   app1     0.04%    24.01MiB / 64MiB      37.51%   66.1kB / 55.7kB   0B / 0B        8
b30f1529b287   mongo    0.29%    61.03MiB / 256MiB     23.84%   60.4kB / 65.4kB   0B / 891kB     39
56c36773ca20   app2     0.04%    24.06MiB / 64MiB      37.59%   65.2kB / 53.6kB   0B / 0B        8
```

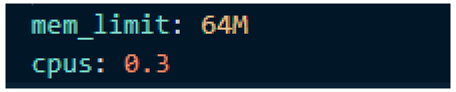
Old Implementation to be tested against

Flask Servers

The original setup combines the monitoring page and the read/write operations in the flask server. We will be testing the difference of separating the services into microservices in the following tests.

To ensure that tests are fairly performed with accurate comparisons, both setups will have the same limitations on the flask servers (**Memory limit of 64MB, CPU of 0.3**).

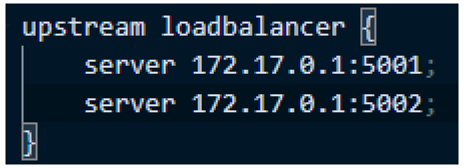
Docker compose file settings in Docker Compose file.



```
mem_limit: 64M
cpus: 0.3
```

Existing API Gateway

Furthermore, the D1 implementation has already implemented the API gateway using NGINX. The original setup uses the basic boiler plate code for load balancing which by default uses the round robin algorithm to allocate load to the servers.



```
upstream loadbalancer {
    server 172.17.0.1:5001;
    server 172.17.0.1:5002;
}
```

Database

In the original design, a single MongoDB database will be used which will hold the entire application's collections and communicate directly to both flask servers. This possibly puts a lot of stress onto the single container instance. Previous stress testing in D1 shows that this could be a possible bottleneck area for the application.

The following setup will be tested against the optimised architecture to showcase the optimisations to the architecture if it makes a difference in the application's performance. The goal of the optimisation will include mitigating the database bottleneck as well as implementing patterns such as the circuit breaker to improve overall response time of the flask server.

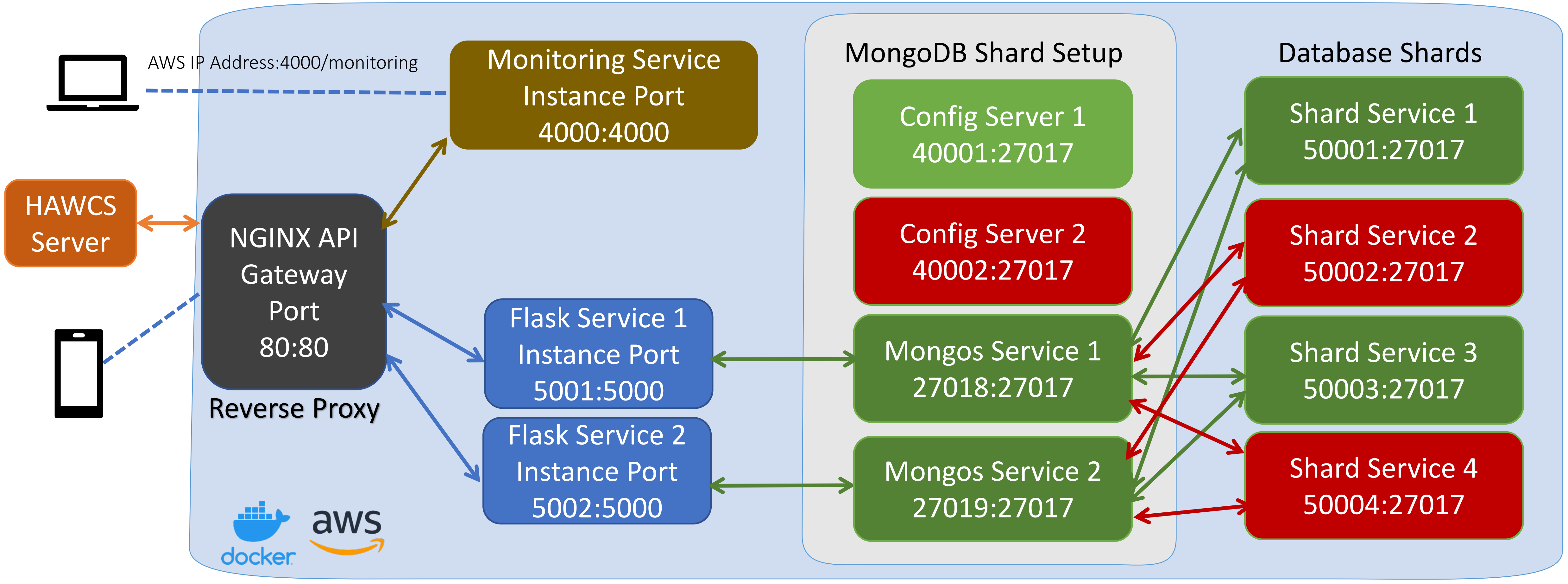
Updated Flask Server Architecture Optimisation Overview

API Gateway
Circuit breaker / GZIP / least connection load balancing

Microservices
Breaking down services into smaller components. One service for monitoring and the two other services for read/write operations.

DB Service Implementation
MongoDB database routing using Mongos and config server replica sets storing shard distribution. Database uses hashed indexing to store data.

DB Shards Implementation
Records hash indexed and distributed into respective database shards, Shards can be horizontally scaled for better performance.



AWS IP Address:4000/monitoring
Monitoring Service Instance Port 4000:4000
HAWCS Server
NGINX API Gateway Port 80:80
Reverse Proxy
Flask Service 1 Instance Port 5001:5000
Flask Service 2 Instance Port 5002:5000
MongoDB Shard Setup
Config Server 1 40001:27017
Config Server 2 40002:27017
Mongos Service 1 27018:27017
Mongos Service 2 27019:27017
Database Shards
Shard Service 1 50001:27017
Shard Service 2 50002:27017
Shard Service 3 50003:27017
Shard Service 4 50004:27017
docker
aws

Each component service above is contained in its own Docker Container deployed at **Amazon Web Services** EC2 T2.Micro

Note: 🟥 Replica Set

As a proof of concept, due to 1GB memory limitations of t2.micro, replica set for **Config Server 2**, **Shard 2** and **Shard 4** have not been implemented but should be ideally implemented for availability of the database

Flask Server Optimization 1: MongoDB Database Sharding

The team wanted to explore the concept of database sharding to helping **improve the read/write performance throughput** of our MongoDB database queries. Sharding is MongoDB's method to distribute data across multiple machines enabling the possibility of horizontal scaling. A typical sharding setup consists of 3 key components:

Mongos Router

Mongos is a query router for our flask servers handling both read/write operations. They do not hold any information of the database at this point and instead gets information of where the data is stored through the configuration servers. It is also recommended for **each service to have their own dedicated mongos route**r, like **each microservice having their own database**, just that they share the same group of database shards.

Config Servers

The configuration servers store all the metadata of the sharded cluster which includes the state and organisation of where each data is stored. Each database shards use a concept of hash sharding which provides even distribution of data between database shards based on a shard key (Single Field Hashed Index). This shard key information is stored in the configuration servers.

Database Shards

sharded databases which are recommended to have replica sets for availability of the shard.

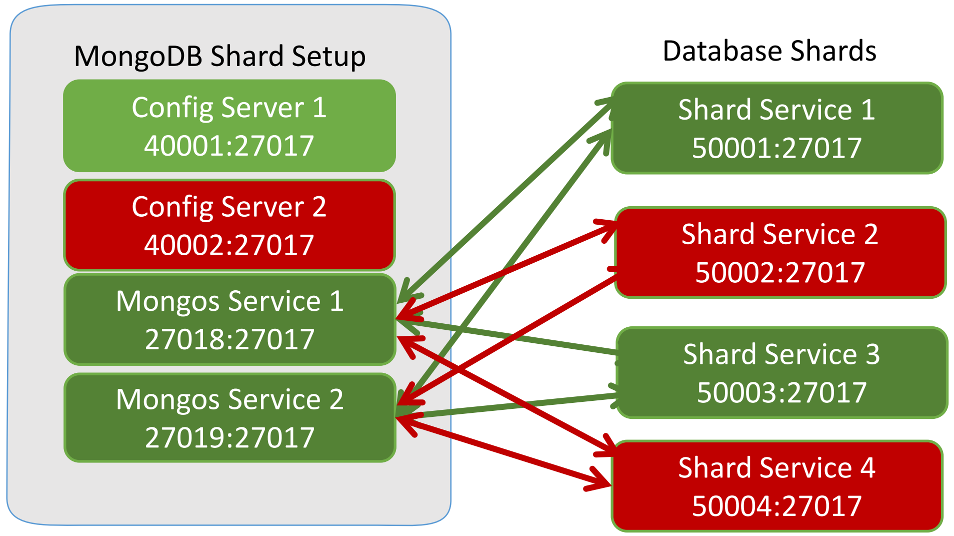
Implementation Strategy

To setup sharding, a few additional settings must be added to the mongos containers.

```
Sh.enableSharding("ICT3102") // Enabling sharding on the collection
sh.shardCollection("ICT3102.beacons", {"level":"hashed"})
sh.shardCollection("ICT3102.staff", {“staff_id":"hashed"})
```

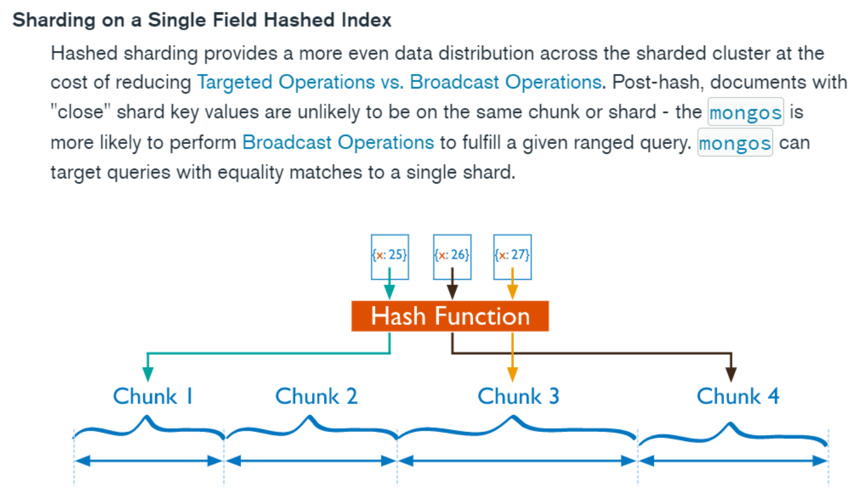
The strategy here is to shard the beacons by level (i.e level 1 to 4 beacons will be found in shard 1, and level 5 to 7 beacons will be found in shard 2). With this setup, retrieving beacon location from the collection should be more efficient with the hashed index method.

As for the heavy load of the incoming inserts of staff location, we will be sharding them based on staff_id. (i.e staff 0 will be found in shard 1 while staff 2 might be found in shard 2). As the database size grows, retrieving a single staff from the DB will get slower. By using the hash index method, staff_locations will be evenly distributed between shards by staff_id, meaning each shard will have lesser records to perform a search operation on. Thus, retrieval of such data will be more efficient with the hashed index method as well. More will be explained on the next slide.



MongoDB Shard Setup
Config Server 1 40001:27017
Config Server 2 40002:27017
Mongos Service 1 27018:27017
Mongos Service 2 27019:27017
Database Shards
Shard Service 1 50001:27017
Shard Service 2 50002:27017
Shard Service 3 50003:27017
Shard Service 4 50004:27017

Implementing this strategy was challenging due to the memory limitations of the t2.micro server. As mentioned previously, each MongoDB instance requires minimally 256MB of RAM to be stable in the long run and for our setup, we had to tone down the number of containers and replica sets in order to get a baseline to work. Thus the image above is the ideal setup which will only work well with a system with about 2GB of RAM.

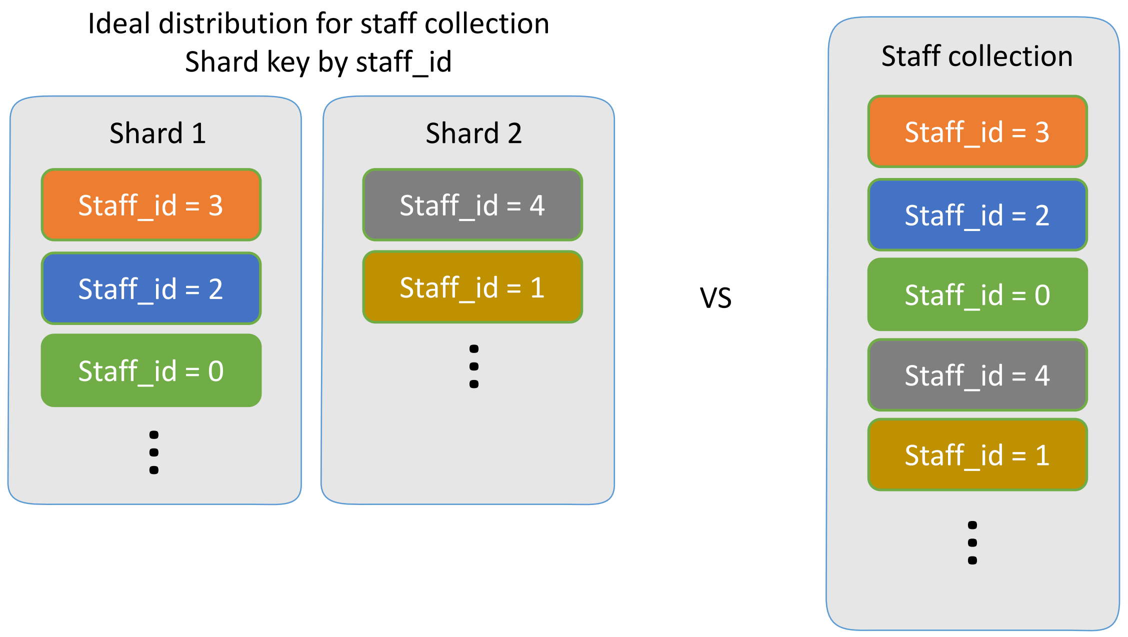


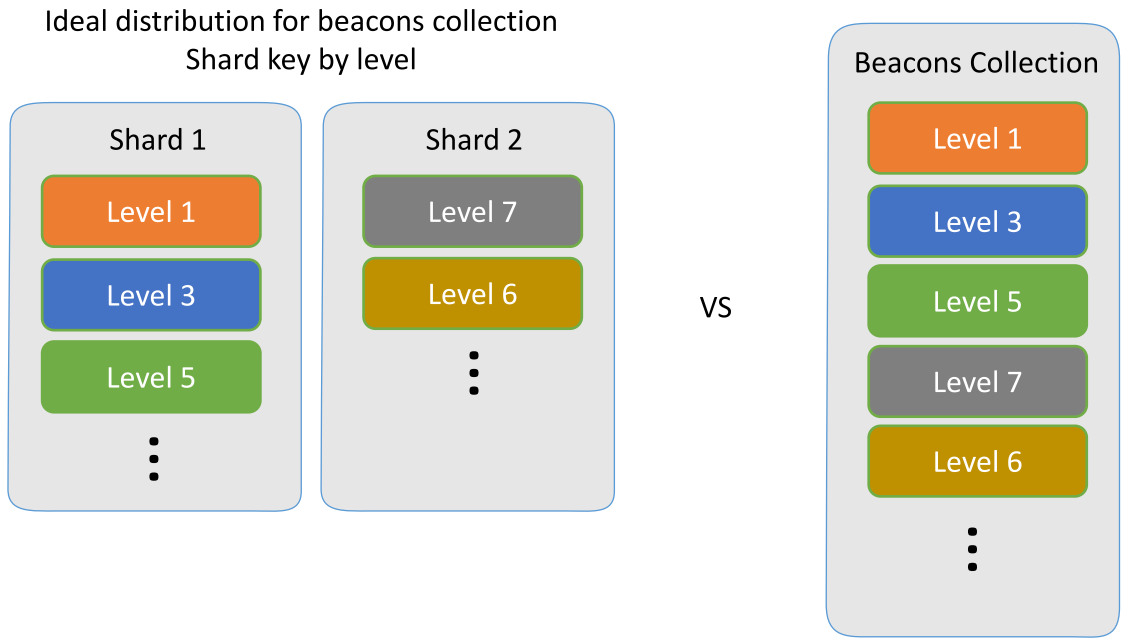
Sharding on a Single Field Hashed Index

Hashed sharding provides a more even data distribution across the sharded cluster at the cost of reducing Targeted Operations vs. Broadcast Operations. Post-hash, documents with "close" shard key values are unlikely to be on the same chunk or shard - the `mongos` is more likely to perform Broadcast Operations to fulfill a given ranged query. `mongos` can target queries with equality matches to a single shard.

https://docs.mongodb.com/manual/core/hashed-sharding/#std-label-sharding-hashed-sharding

Flask Server Optimization 1: MongoDB Database Sharding (cont)





<u>Shard Distribution Strategy (Cont)</u>

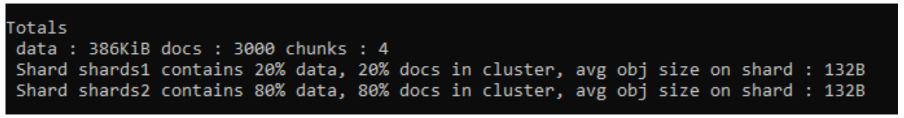
Sharding documents by a shard key helps to distribute the chunks to available shards evenly. With a good shard key, the database can be divided and indexed to their respective shards. As shown in the diagrams on the left, ideally we want a shard key that can place document chunks in a consistent way where it is easily searchable and retrievable.

As mentioned earlier we have selected the staff_id for the staff collection, which stores all the staff locations, to be the shard key. On every insert operation, the documents with staff id 0 will be hashed indexed into shard 1. This means that all staff 0's location details will only be found in shard 1. Compared to searching the entire mongo collection, this will effectively reduce the amount of documents the search operation will be have to go through.

Similarly, the beacons collection storing all the details of each beacon provided, have a shard key of level. This means that beacon locations with the level of 1 will be stored in only shard 1. When the application needs to retrieve a beacon from level one, it won't have to access shard 2 at all, reducing the number of operations the search has to go through.

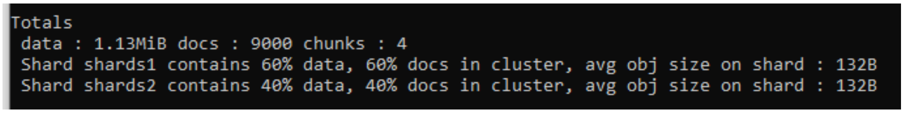
Both shard key information will be stored in the configuration mongo servers which both mongos routers will be attached to, essentially providing the information of where each document is to the mongos.

There is also a possibility that the shard key strategy hashes documents unevenly which will cause an uneven shard distribution. This might affect the performance of the shard server and the only way to fix this is to pick the shard key very carefully.



```
Totals
 data : 386KiB docs : 3000 chunks : 4
 Shard shards1 contains 20% data, 20% docs in cluster, avg obj size on shard : 132B
 Shard shards2 contains 80% data, 80% docs in cluster, avg obj size on shard : 132B
```

Bad Distribution



```
Totals
 data : 1.13MiB docs : 9000 chunks : 4
 Shard shards1 contains 60% data, 60% docs in cluster, avg obj size on shard : 132B
 Shard shards2 contains 40% data, 40% docs in cluster, avg obj size on shard : 132B
```

Ok Distribution

Performance Optimization Testing: MongoDB Database Sharding (Write First Pass)

Before Optimisation (Inserting different staff id)

PERF-RT-1-1: Response time for inserting detected beacon information from mobile to flask server				
Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads=2100 Ramp up = 60	34.58	12	248	17.76
Threads=2520 Ramp up = 60	41.67	12	109	16.58
Threads=2940 Ramp up = 60	50.01	12	477	20.56
Threads=3360 Ramp up = 60	55.53	12	1020	19.88
Threads=3780 Ramp up = 60	62.50	11	233	18.55

After Optimisation (Inserting different staff id)

PERF-RT-1-1: Response time for inserting detected beacon information from mobile to flask server				
Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads=2100 Ramp up = 60	34.57	16	1022	**21.49**
Threads=2520 Ramp up = 60	41.67	15	122	**19.64**
Threads=2940 Ramp up = 60	50.00	15	772	**31.61**
Threads=3360 Ramp up = 60	55.56	15	135	**20.60**
Threads=3780 Ramp up = 60	62.50	14	294	**25.59**

Result Analysis

In this test, we stress tested the limits of the write operation once again. However, we used the variation of RT-1 where instead of inserting staff_id 0 all the way, we changed up the staff_id inserted per round. **The staff_id to be specified in this test would be 0, 1, 2, 3, 4**. This will leverage on the distribution of insert operations to the respective shard servers based on the staff_id (shard key). Due to the limitations of the t2.micro instance, we were not able to create more than 2 shards. In concept, the more shards created, the better the write operation will be as mongos routers will be able to distribute the load efficiently between all shard servers.

In the above test, we noticed that the average response time for the optimised setup has slightly longer response times compared to the one without the shard setup. With some investigation we have might have found a possible reason. In the setup of hashing staff_id 0 to 4, the worst case scenario happened where 20% of the test distribution went to shard 1 while the remaining 80% went to shard 2, giving shard 2 an uneven load distribution. Another possible reason for the increase in write times would be the overhead from using the mongos routers. A possible improvement for this test would change the staff_id per insert to evenly distribute the load.

We will perform a second pass for this test using a different set of staff ids to confirm if sharding distribution affecting the performance or just the overhead from using additional mongos in the next slide.

Performance Optimization Testing: MongoDB Database Sharding (Write Second Pass)

Before Optimisation (Inserting different staff id)

PERF-RT-1-1:
Response time for inserting detected beacon information from mobile to flask server

Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads=2100 Ramp up = 60	34.58	12	248	17.76
Threads=2520 Ramp up = 60	41.67	12	109	16.58
Threads=2940 Ramp up = 60	50.01	12	477	20.56
Threads=3360 Ramp up = 60	55.53	12	1020	19.88
Threads=3780 Ramp up = 60	62.50	11	233	18.55

After Optimisation (Inserting different staff id)

PERF-RT-1-1:
Response time for inserting detected beacon information from mobile to flask server

Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads=2100 Ramp up = 60	34.56	16	292	**22.35**
Threads=2520 Ramp up = 60	41.67	16	59	**19.78**
Threads=2940 Ramp up = 60	50.00	16	247	**21.97**
Threads=3360 Ramp up = 60	55.55	16	212	**22.50**
Threads=3780 Ramp up = 60	62.50	16	1036	**27.80**

Result Analysis

In this the second pass, we used a different set of staff ids to test if the sharding distribution affecting the performance or just the overhead of the mongos setup. **The staff ids used in this test are 0,11,12,30,40** which after checking gives us a distribution of 60% of records in shard 1 and 40% of records in shard 2. The results are very similar to the previous test with 3780 insert operations having an average response time of 25 – 27ms. This shows that the shard distribution does not really affect the performance and the most possible cause for the increase in response times would be from the overhead of using mongos setup (each API call requires to go through an additional router step to the required database).

In conclusion, the overall test may not be the most accurate way to showcase the write operation's improvement and ideally we want to test with more than 2 shards to show a convincing improvement. Ideally we want to stress test the limits of the insert operation further to see the difference between a sharded and non sharded database. However the tests do show that both flask server implementations can handle more than 10x of our workload model plus 80% more for this insert operation between mobile API request to the flask server.

Performance Optimization Testing: MongoDB Database Sharding (Retrieve First Pass)

Before Optimisation (Inserting different staff id)

PERF-RT-2-1: Response time for extracting all user locations between flask server to HACW server.				
Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads = 860 Ramp up = 60 DB Data = 600	14.29	41	107	51.95
Threads = 860 Ramp up = 60 DB Data = 1200	14.29	41	162	51.83
Threads = 860 Ramp up = 60 DB Data = 1800	14.29	42	200	**53.73**
Threads = 860 Ramp up = 60 DB Data = 2400	14.29	43	502	**57.78**
Threads = 860 Ramp up = 60 DB Data = 3000	14.29	45	312	**55.06**

After Optimisation (Inserting different staff id)

PERF-RT-2-1: Response time for extracting all user locations between flask server to HACW server.				
Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads = 860 Ramp up = 60 DB Data = 600	14.30	41	80	49.23
Threads = 860 Ramp up = 60 DB Data = 1200	14.29	42	484	58.41
Threads = 860 Ramp up = 60 DB Data = 1800	14.29	41	89	**50.63**
Threads = 860 Ramp up = 60 DB Data = 2400	14.29	40	113	**51.03**
Threads = 860 Ramp up = 60 DB Data = 3000	14.29	39	85	**52.01**

Result Analysis

Similar to the previous test, we will be using the variation of RT-2 where we insert a different staff_id for every 600 records added. **The staff id's include 0,1,2,3,4**. The test will only **retrieve all records of staff_id 0**, aiming to test the effectiveness of the retrieve operation. Ideally, the optimised setup should be a lot quicker when retrieving a single group of records as the database compared to the original will be less bloated, making retrieve operations much faster.

In this test, there is not much improvement in performance that can be seen. The reason why this test is a false negative is because we are still retrieving an optimal amount of staff 0 records (only 600). This has not passed the stress point of the retrieve operation. A following test of RT-2 shows that the limit before the circuit breaker is triggered is when the system performs 860 threads of the retrieve operation with 2400 staff 0 records. Thus more passes of this test must be performed until we can see a visible improvement in performance. We will continue a second pass of the same test, adding more data into both databases, in attempt to bloat the databases.

Performance Optimization Testing: MongoDB Database Sharding (Retrieve Second Pass)

Before Optimisation (Inserting different staff id)

PERF-RT-2-1: Response time for extracting all user locations between flask server to HACW server.				
Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads = 860 Ramp up = 60 DB Data = 3600	14.28	**68**	**545**	**116.97**
Threads = 860 Ramp up = 60 DB Data = 4200	14.28	**69**	**679**	**120.96**
Threads = 860 Ramp up = 60 DB Data = 4800	14.27	**72**	**809**	**125.87**
Threads = 860 Ramp up = 60 DB Data = 5400	14.27	**68**	**701**	**118.02**
Threads = 860 Ramp up = 60 DB Data = 6000	14.28	**71**	**1104**	**162.67**

After Optimisation (Inserting different staff id)

PERF-RT-2-1: Response time for extracting all user locations between flask server to HACW server.				
Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads = 860 Ramp up = 60 DB Data = 3600	14.29	**61**	**1209**	**84.50**
Threads = 860 Ramp up = 60 DB Data = 4200	14.28	**66**	**895**	**93.64**
Threads = 860 Ramp up = 60 DB Data = 4800	14.29	**68**	**306**	**82.94**
Threads = 860 Ramp up = 60 DB Data = 5400	14.28	**69**	**854**	**91.12**
Threads = 860 Ramp up = 60 DB Data = 6000	14.28	**65**	**229**	**81.18**

Result Analysis

In the second pass of the test, we re-ran the same jmeter test without clearing the database. Essentially adding another 3000 records into the database with the **same staff id allocation as earlier**. Finally we can see a slightly bigger improvement in performance of the retrieve operation. Before optimisation, it can be seen that as more data is added into the database, the average response time will gradually increase. This confirms that it does take a longer duration to perform retrieve operations when more data is present in the database. In the optimised setup, the response times to **retrieve the 1200 of staff 0** records are consistent throughout the 5 rounds. This is expected as it could be that other staff's data are divided into both shards, and the mongos routers know which shard to route the requests to.

This shows that our database sharding strategy is working, however after further investigation of the shard distribution, 20% of the inserted data went into shard 1 and 80% went into shard 2 (Using staff id 0, 1, 2, 3, 4). This could mean that staff 0's data is the only data in shard 1, skewing the results to look like the performance is better when in actual fact if I retrieve staff 1, the results could be a lot slower since all the data is in shard 2. To show that the data is being properly sharded, we have to run another test with a fairer distribution.

Performance Optimization Testing: MongoDB Database Sharding (Retrieve Third pass)

Before Optimisation (Inserting different staff id)

PERF-RT-2-1:
Response time for extracting all user locations between flask server to HACW server.

Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads = 860 Ramp up = 60 DB Data = 6600	12.40	**131**	**9527**	**4739.55**
Threads = 860 Ramp up = 60 DB Data = 7200	12.24	**208**	**10502**	**4904.86**
Threads = 860 Ramp up = 60 DB Data = 7800	12.28	**181**	**10387**	**4915.27**
Threads = 860 Ramp up = 60 DB Data = 8400	12.29	**158**	**10039**	**5183.70**
Threads = 860 Ramp up = 60 DB Data = 9000	12.28	**130**	**10033**	**4843.03**

After Optimisation (Inserting different staff id)

PERF-RT-2-1:
Response time for extracting all user locations between flask server to HACW server.

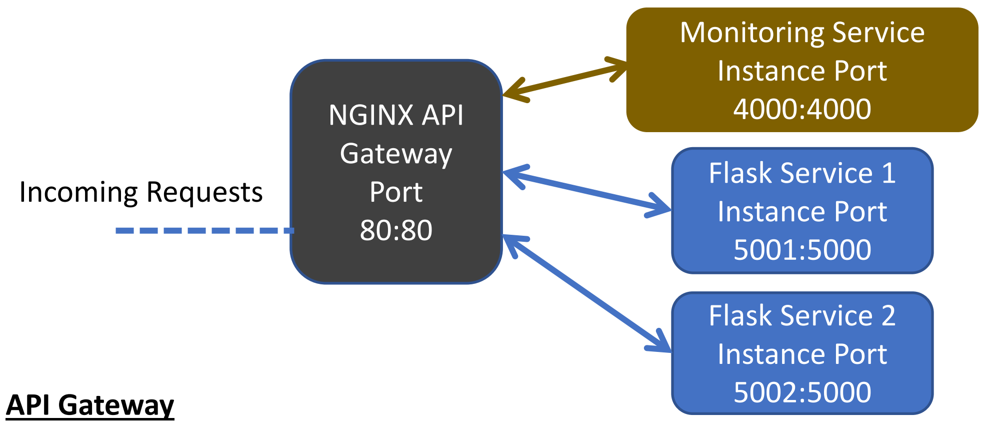
Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads = 860 Ramp up = 60 DB Data = 6600	14.29	**76**	**346**	**103.25**
Threads = 860 Ramp up = 60 DB Data = 7200	14.28	**76**	**376**	**104.83**
Threads = 860 Ramp up = 60 DB Data = 7800	14.27	**74**	**1131**	**121.75**
Threads = 860 Ramp up = 60 DB Data = 8400	14.28	**74**	**387**	**101.96**
Threads = 860 Ramp up = 60 DB Data = 9000	14.28	**73**	**318**	**100.14**

Result Analysis

In this test, **we inserted Staff_id 0, 11, 12, 30, 40** instead, giving us a shard distribution of 60% in shard 1 and 40% in shard 2 after **re-running all 3 passes again each with the new set of staff ids.** Now with this results, we can clearly see that a bloated database will eventually prevent the original setup from performing retrieve operations efficiently. **Retrieving 1800 staff 0 documents** in the original setup, with all the other inserted staff data, will eventually show a gradual increase in response time going up to 5 seconds. On the other hand, the sharded database hashed index method with the shard key of staff id is working effectively with constant response times between 100ms to 120ms, as long as the shard distribution is evenly distributed.

This final test confirms that our sharding strategy is working effectively and is not a false positive test as both shards have a fairly even amount of record distribution. Sharding would be more effective if there are more shards as well. Currently we are only hashing records into 2 databases and the chances of not evenly distributing the documents are quite high, like in the earlier test using staff id 0,1,2,3,4. To improve the results of this test, we will need to change up the testing environment to be able to handle more MongoDB shard instances.

Flask Server Optimization 2:
API Gateway and Circuit Breaker Pattern


Incoming Requests
NGINX API Gateway Port 80:80
Monitoring Service Instance Port 4000:4000
Flask Service 1 Instance Port 5001:5000
Flask Service 2 Instance Port 5002:5000

API Gateway

To combat the Chatty Microservices anti-pattern, the team has implemented an API Gateway pattern using NGINX. The gateway listens on port 80 for incoming requests and routes them to flask servers running on port 5001 and 5002. It performs **load balancing** based on **least number of connections** on the respective services. It also routes to the monitoring service on port 4000 which makes API calls through the API gateway and not directly from the flask services to ensure a single point of communication is adhered too.

Circuit Breaker Pattern

In the NGINX configuration, we have setup passive health checks on both flask services. Upon failing to respond 5 times, the API gateway will close off the connection to that server for 20 seconds and attempts to direct the load to another available server. If no servers are available, it will set the circuit state to open. NGINX will determine if a server has failed based on either errors or connect/send/read timeouts which are configured to 5s, 10s, 10s respectively. When the circuit is open, the backup server will be the monitoring service which will return a quick response, reducing overall error respond time. NGINX will constantly check for the "half open" state every 20s until either one of the flask server has recovered, which will then return that recovered server to the closed state.

GZIP and GUNZIP

Since the application API server transmits mostly JSON data, the team has implemented gzip compression level 6 in the NGINX gateway onto JSON request to further improve response times being made.

```
proxy_next_upstream error timeout invalid_header http_502 http_503
http_504;
```

Least Connection load balancing

Maximum fails before server considered down.
Service will timeout for 20s if service fails = 5

```
upstream loadbalancer {
    least_conn;
    server 172.17.0.1:5001 max_fails=5 fail_timeout=20s;
    server 172.17.0.1:5002 max_fails=5 fail_timeout=20s;
    server 172.17.0.1:4000  backup;
}
```

Backup health monitoring server

```
server {
    listen 80;
```

GZIP configurations

```
    gzip on;
    gzip_types      text/plain application/xml application/json;
    gzip_proxied    no-cache no-store private expired auth;
    gzip_min_length 1000;
    gzip_disable    "msie6";
    gzip_comp_level 6;
    gunzip on;

    location / {
```

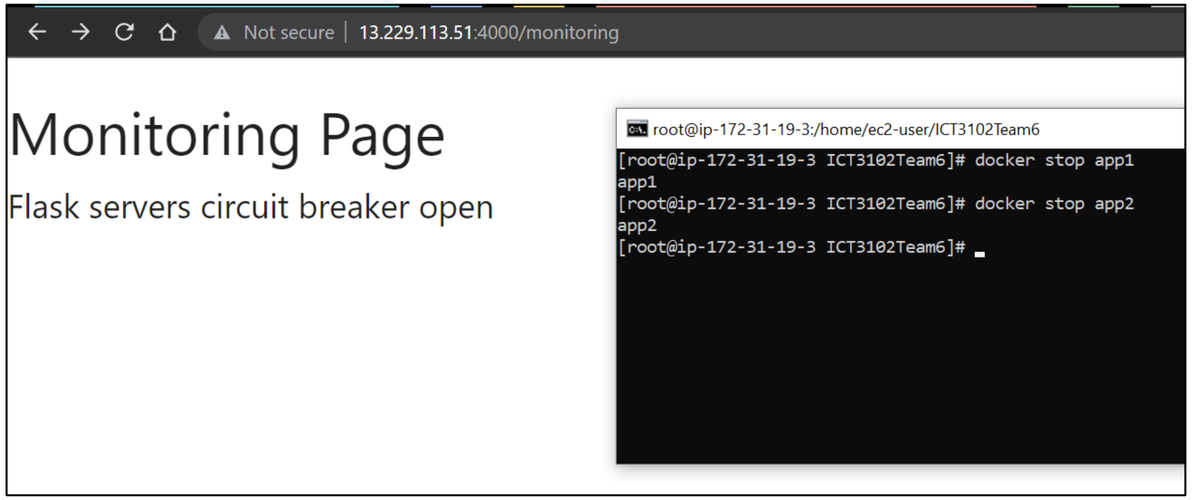
Timeout For Circuit breaker.

```
        proxy_set_header Proxy                      "";
        proxy_connect_timeout                       5s;
        proxy_send_timeout                          10s;
        proxy_read_timeout                          10s;

        proxy_redirect                              off;
        proxy_buffering                             off;

        proxy_http_version                          1.1;
        proxy_pass http://loadbalancer;
    }
}
```

Code Snippet of Nginx.config implementation

Flask Server Optimization 2:
API Gateway and Circuit Breaker Pattern (Cont)

Monitoring page when circuit breaker is in the open/half-open state



Not secure | 13.229.113.51:4000/monitoring

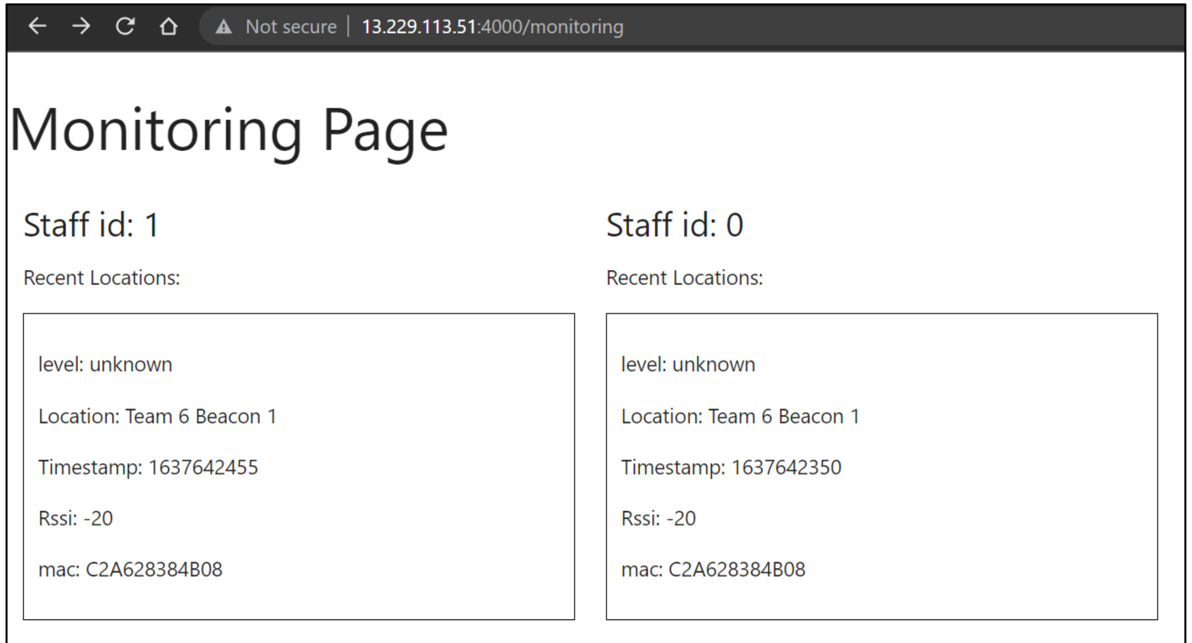
Monitoring Page

Flask servers circuit breaker open

root@ip-172-31-19-3:/home/ec2-user/ICT3102Team6

```
[root@ip-172-31-19-3 ICT3102Team6]# docker stop app1
app1
[root@ip-172-31-19-3 ICT3102Team6]# docker stop app2
app2
[root@ip-172-31-19-3 ICT3102Team6]# _
```

Monitoring page when circuit breaker is in the closed state

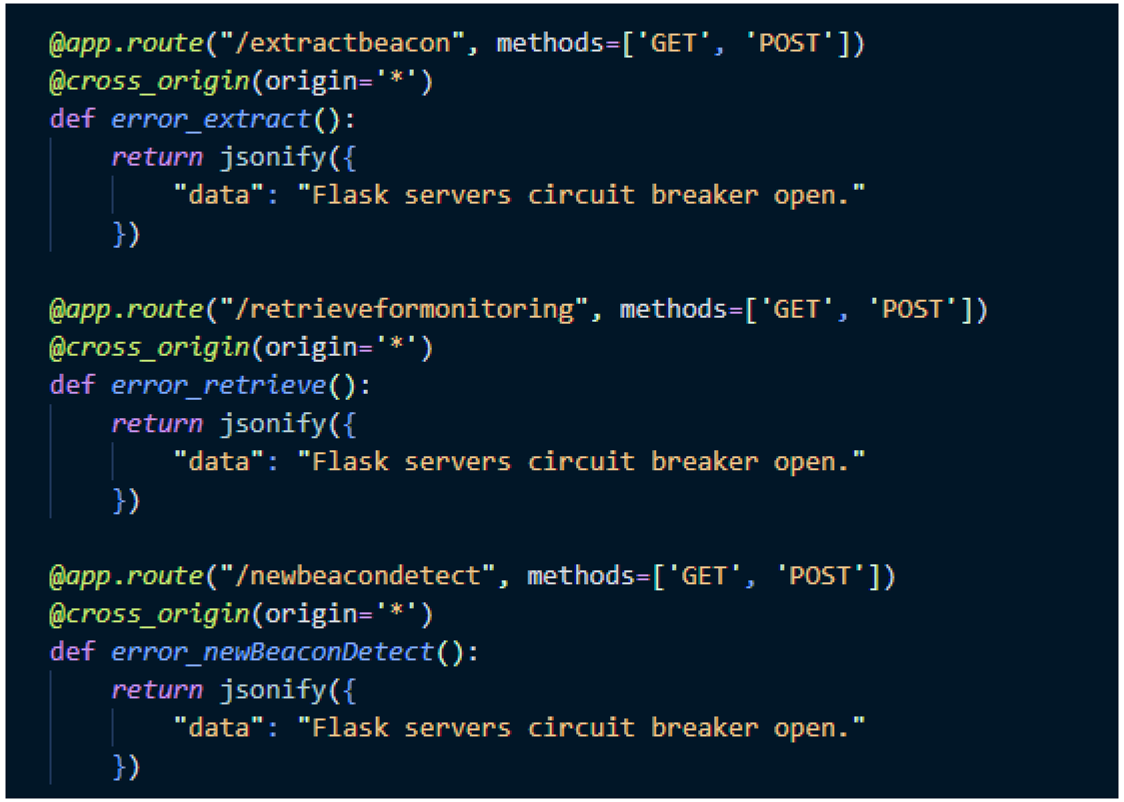


Not secure | 13.229.113.51:4000/monitoring

Monitoring Page

Staff id: 1	Staff id: 0
Recent Locations:	Recent Locations:
level: unknown	level: unknown
Location: Team 6 Beacon 1	Location: Team 6 Beacon 1
Timestamp: 1637642455	Timestamp: 1637642350
Rssi: -20	Rssi: -20
mac: C2A628384B08	mac: C2A628384B08

Monitoring page throws instant error response if circuit is open



```python
@app.route("/extractbeacon", methods=['GET', 'POST'])
@cross_origin(origin='*')
def error_extract():
    return jsonify({
        "data": "Flask servers circuit breaker open."
    })

@app.route("/retrieveformonitoring", methods=['GET', 'POST'])
@cross_origin(origin='*')
def error_retrieve():
    return jsonify({
        "data": "Flask servers circuit breaker open."
    })

@app.route("/newbeacondetect", methods=['GET', 'POST'])
@cross_origin(origin='*')
def error_newBeaconDetect():
    return jsonify({
        "data": "Flask servers circuit breaker open."
    })
```

Circuit Breaker (Cont)

The team used the monitoring page as a health check page for the system. The moment app1 and app2 instances are down, the circuit breaker will enter the open state and redirect requests to the monitoring page. The monitoring page does not perform heavy computation and will just return a lightweight json response with the error code for each request as shown above. This will inform the user that the server is down and will not cause a long delay waiting for the flasks server to respond.

Performance Optimization Testing: API Gateway

Before Optimisation (Inserting same staff id for all records)

PERF-TP-1:
Throughput for inserting detected beacon information from mobile to flask server

Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads = 210 Ramp up = 60	3.52	13	74	19.41
Threads = 2100 Ramp up = 60	34.49	12	95	17.25

PERF-TP-2:
Throughput for extracting all user locations between flask server to HACW server.

Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads = 86 Ramp up = 60 DB Data = 2310	1.45	98	304	152.19
Threads = 860 Ramp up = 60 DB Data = 2310	10.08	145	25366	**12810.52**

After Optimisation (Inserting same staff id for all records)

PERF-TP-1:
Throughput for inserting detected beacon information from mobile to flask server

Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads = 210 Ramp up = 60	3.52	18	58	23.17
Threads = 2100 Ramp up = 60	34.50	15	109	21.34

PERF-TP-2:
Throughput for extracting all user locations between flask server to HACW server.

Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads = 86 Ramp up = 60 DB Data = 2310	1.45	107	364	161.09
Threads = 860 Ramp up = 60 DB Data = 2310	12.39	138	9645	**4592.05**

Result Analysis

We performed a basic throughput test for both read and write capabilities to confirm if our planned architecture can withstand the minimum throughput required to handle our workload model. An additional test of x10 the original throughput is also performed to see how much more load we can actually take in each scenario.

For TP-1, write operations for both versions are able to handle the required 3.5 transaction per second of beacon insertions from the mobile phone to the flask server. A slight noticeable difference can be observed in the optimised version which could be due to the overhead of having a new mongos router in between the flask server and the database. We will be exploring the benefits of the mongos and sharding in the following tests. We can conclude that even under 10x the throughput for write, the flask server will still be operational.

For TP-2, read operations for the optimised version performed slightly better compared to the original version. This change could be due to the database sharding and additional mongos splitting the database request load, allowing for better response times. We can conclude that both setups will face a degradation of throughput when performing 10x the read operation.

Performance Optimization Testing: API Gateway

Before Optimisation (Inserting same staff id for all records)

PERF-RT-1:
Response time for inserting detected beacon information from mobile to flask server

Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads=2100 Ramp up = 60	34.58	12	322	18.93
Threads=2520 Ramp up = 60	41.67	12	108	17.66
Threads=2940 Ramp up = 60	49.94	12	117	18.63
Threads=3360 Ramp up = 60	55.55	11	103	19.35
Threads=3780 Ramp up = 60	62.48	11	299	24.76

After Optimisation (Inserting same staff id for all records)

PERF-RT-1:
Response time for inserting detected beacon information from mobile to flask server

Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads=2100 Ramp up = 60	34.58	16	173	**20.50**
Threads=2520 Ramp up = 60	41.67	15	110	**20.90**
Threads=2940 Ramp up = 60	50.00	15	128	**20.11**
Threads=3360 Ramp up = 60	55.55	15	200	**21.58**
Threads=3780 Ramp up = 60	62.50	15	328	**26.30**

Result Analysis

Both versions have implemented the API gateway pattern and it is hard to showcase the benefits of the API gateway as such. In an earlier D1 study, having multiple services with and API gateway helped increase the maximum stress test limit of 2100 running threads at a throughput of 34 transactions per second. For the write operation In D2, we plan to test the limits further on both implementations with the API gateway. Thus the RT-1 has been performed. The test starts from our original limit of 2100 and increases the number of threads by 20% of the original per round.

At first glance, we can see that the optimised version is performing yet again slightly slower compared to the original version. As mentioned earlier, this could be the overhead of the mongos. In this test, we are simulating the mobile phone sending write requests to the flask server. Both setups show the efficiency of the API gateway being able to handle way more workload than we planned for. This assures us that our workload model for insert requests can be satisfied even when the load is more than 10x the expected amount.

Performance Optimization Testing: Circuit Breaker Pattern

Before Optimisation (Inserting same staff id for all records)

PERF-RT-2:
Response time for extracting all user locations between flask server to HACW server.

Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads = 860 Ramp up = 60 DB Data = 600	14.29	41	145	51.17
Threads = 860 Ramp up = 60 DB Data = 1200	14.28	66	228	107.33
Threads = 860 Ramp up = 60 DB Data = 1800	12.47	116	9094	4911.71
Threads = 860 Ramp up = 60 DB Data = 2400	9.62	187	**29249**	**15048.17**
Threads = 860 Ramp up = 60 DB Data = 3000	7.79	229	**51651**	**25692.12**

After Optimisation (Inserting same staff id for all records)

PERF-RT-2:
Response time for extracting all user locations between flask server to HACW server.

Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads = 860 Ramp up = 60 DB Data = 600	14.29	43	290	52.24
Threads = 860 Ramp up = 60 DB Data = 1200	14.29	59	164	75/39
Threads = 860 Ramp up = 60 DB Data = 1800	14.27	76	370	128.53
Threads = 860 Ramp up = 60 DB Data = 2400	14.30	12	**20028**	**4936.30**
Threads = 860 Ramp up = 60 DB Data = 3000	14.30	12	**20022**	**3810.64**

Result Analysis

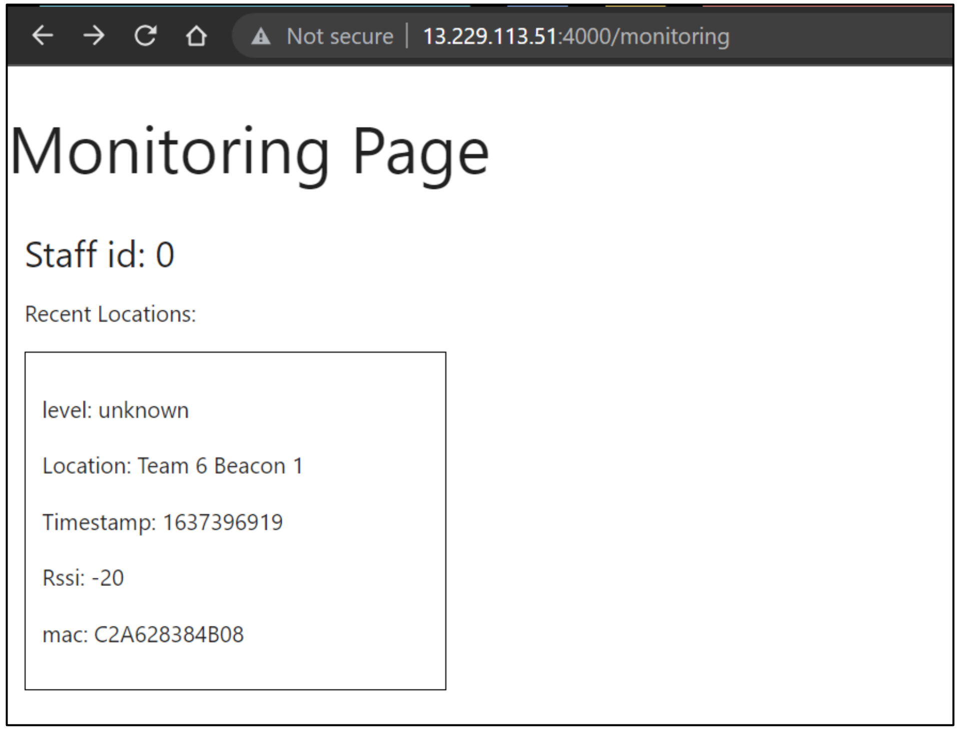
In the D1 study, it has shown that the more data is stored in the database, the longer it will take to retrieve all the data, eventually causing failures to occur. Having the possibility of our service failing will cause long response times if no measures are set. Thus we implemented the circuit breaker to curb the issue. In D1, before implementing the API gateway, a single flask server service can only perform 840 thread request retrieving 300 database records before it starts to fail. With the API gateway, the limits have been increased and we will be testing the new limit against RT-2, as well as to check if our circuit breaker implementation will improve overall response times.

In both scenarios, we are now running 860 threads to adapt to our updated workload and will be inserting 600 records of the same staff_id. The reason why we use the same staff_id is so that sharding will not be affecting the result, since the data are all allocated to the same shard database based on the staff_id shard key. At 2400 and 3000 database records, both setups faced performance degradation. However in the optimised version, the implemented circuit breaker was opened, directing the request to an error page immediately and reducing overall response time. The improvement in average response time for 2400 records is reduced **about a third** and for 3000 records the reduction is **twice that amount (6 times)**.

This shows that the circuit breaker implementation is operationally working and is effective in reducing unwanted error response times.

Flask Server Optimization 3: Separation of Services

Monitoring Service
Instance Port
4000:4000



Monitoring Page

Staff id: 0

Recent Locations:

level: unknown

Location: Team 6 Beacon 1

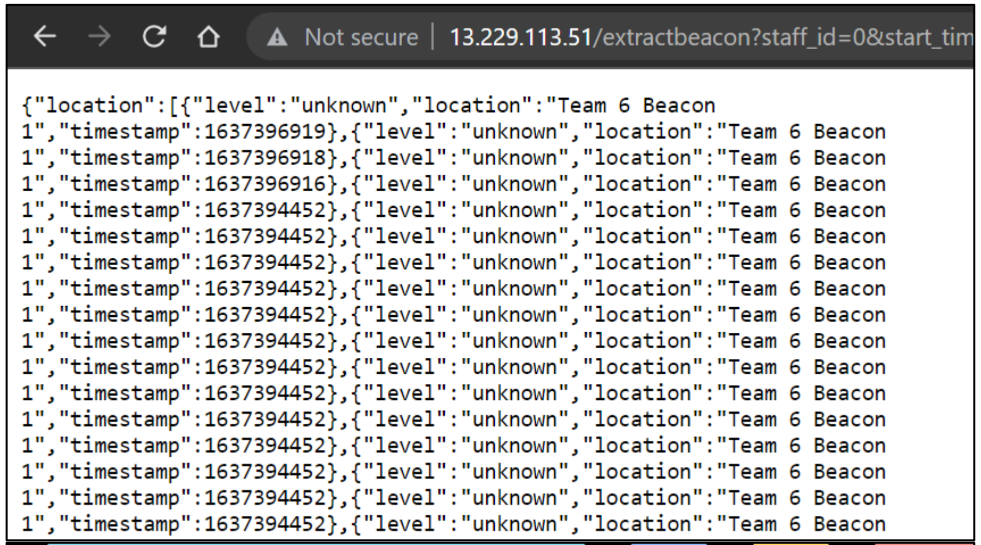
Timestamp: 1637396919

Rssi: -20

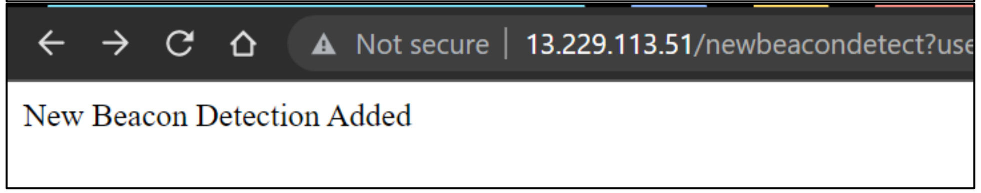
mac: C2A628384B08

Flask Service 1
Instance Port
5001:5000

Flask Service 2
Instance Port
5002:5000


Not secure | 13.229.113.51/extractbeacon?staff_id=0&start_tim

```
{"location":[{"level":"unknown","location":"Team 6 Beacon
1","timestamp":1637396919},{"level":"unknown","location":"Team 6 Beacon
1","timestamp":1637396918},{"level":"unknown","location":"Team 6 Beacon
1","timestamp":1637396916},{"level":"unknown","location":"Team 6 Beacon
1","timestamp":1637394452},{"level":"unknown","location":"Team 6 Beacon
1","timestamp":1637394452},{"level":"unknown","location":"Team 6 Beacon
1","timestamp":1637394452},{"level":"unknown","location":"Team 6 Beacon
1","timestamp":1637394452},{"level":"unknown","location":"Team 6 Beacon
1","timestamp":1637394452},{"level":"unknown","location":"Team 6 Beacon
1","timestamp":1637394452},{"level":"unknown","location":"Team 6 Beacon
1","timestamp":1637394452},{"level":"unknown","location":"Team 6 Beacon
1","timestamp":1637394452},{"level":"unknown","location":"Team 6 Beacon
1","timestamp":1637394452},{"level":"unknown","location":"Team 6 Beacon
1","timestamp":1637394452},{"level":"unknown","location":"Team 6 Beacon
1","timestamp":1637394452},{"level":"unknown","location":"Team 6 Beacon
1","timestamp":1637394452},{"level":"unknown","location":"Team 6 Beacon
1","timestamp":1637394452},{"level":"unknown","location":"Team 6 Beacon
```



New Beacon Detection Added

Originally, both the monitoring page and the business logic were combined into a single service. The team wants to explore if separating the services would lead better read/write operation response times if the load of the monitoring page is removed from the service.

As the services are now separated, the monitoring page will have to make an ajax call towards the API gateway to retrieve data. The API gateway will allocate a flask server with the least connections to the monitoring page for it to perform it's API calls for latest beacon data. The flask server will return the latest records within a 15 minute timeframe to the monitoring service client side ajax call. A strategy applied here is that we shifted the computational load of displaying each staff's single latest location to the monitoring page's user where the client side JavaScript will handle the logic to only show each staff's latest location.

Due to the limited memory restrictions, the team has not implemented a separate container service for read or write functions individually. Instead, we decided to go with 2 identical servers that handle both read and write capabilities which will be load balanced based on least connections. This is for availability of both functions at all times.

Performance Optimization Testing: Separation of services (Write First Pass)

Before Optimisation (Inserting different staff id for all records)

PERF-RT-1-1:
Response time for inserting detected beacon information from mobile to flask server

Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads=2100 Ramp up = 60	34.58	12	235	16.49
Threads=2520 Ramp up = 60	41.68	12	98	**15.90**
Threads=2940 Ramp up = 60	50.00	12	177	**19.55**
Threads=3360 Ramp up = 60	55.56	11	170	**17.46**
Threads=3780 Ramp up = 60	62.50	11	423	**20.94**

After Optimisation (Inserting different staff id for all records)

PERF-RT-1-1:
Response time for inserting detected beacon information from mobile to flask server

Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads=2100 Ramp up = 60	34.58	13	1021	22.32
Threads=2520 Ramp up = 60	41.68	12	98	**15.63**
Threads=2940 Ramp up = 60	50.01	11	107	**16.31**
Threads=3360 Ramp up = 60	55.56	12	164	**16.11**
Threads=3780 Ramp up = 60	62.50	11	207	**16.11**

Result Analysis

To correctly perform this test, we have reverted the new architecture's database and NGINX configurations to the same as the old architecture. This will ensure that we are testing the difference of the flask servers only. Also notice that we are running a variation of RT-1 which inserts different staff_id for every group of threads.

By separating the frontend from the backend service, we were hoping for stronger improvement in response times. For the write operations, average response times have improved in the optimised version but only by a few microseconds. Currently, it is unclear how many features the monitoring pages will provide in the future but the page should ideally be lightweight and should not hold any business logic or heavy processing. With more features being added in the future, video monitoring etc, having it in the flask server would ultimately affect the performance of the flask server. Thus, this little improvement shows signs that the separation of service would benefit the service as a whole in the long run.

We will perform a second pass with more monitoring pages open to simulate more load on the monitoring page to properly test the difference of the separation of services.

Performance Optimization Testing: Separation of services (Write Second Pass)

Before Optimisation (Inserting different staff id for all records)

PERF-RT-1-1: Response time for inserting detected beacon information from mobile to flask server				
Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads=2100 Ramp up = 60	34.58	13	1024	21.16
Threads=2520 Ramp up = 60	41.67	12	291	**23.73**
Threads=2940 Ramp up = 60	50.00	13	1171	**34.82**
Threads=3360 Ramp up = 60	55.55	12	272	**30.82**
Threads=3780 Ramp up = 60	62.50	13	291	**34.22**

After Optimisation (Inserting different staff id for all records)

PERF-RT-1-1: Response time for inserting detected beacon information from mobile to flask server				
Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads=2100 Ramp up = 60	34.62	13	1084	21.87
Threads=2520 Ramp up = 60	41.68	13	145	**18.56**
Threads=2940 Ramp up = 60	49.99	12	1029	**20.90**
Threads=3360 Ramp up = 60	55.55	12	305	**25.28**
Threads=3780 Ramp up = 60	62.50	12	238	**28.05**

Result Analysis

To correctly perform this test, we have reverted the new architecture's database and NGINX configurations to the same as the old architecture. This will ensure that we are testing the difference of the flask servers only. Also notice that we are running a variation of RT-1 which inserts different staff_id for every group of threads. We have also started **10 monitoring pages** instead of 1 to test if there if performance improvements when load lifted from the flask servers and to simulate more load on the monitoring page.

Each monitoring page will refresh it’s data every 30 seconds and display all the latest staff information within a 15 min timeframe, which can be a costly retrieve operation when multiple instances are spawned. Do note that the HACS server is also running in this test to ensure operational efficacy is true. In both cases, we can see an increase in response times when more monitoring pages are running concurrently. This does show that the monitoring service does potentially affect the overall performance of the application. Once more features are included to the monitoring page, the load will be more significant than the current test performing only one retrieves every 30 seconds. It is unfortunate that the response times for the optimised setup increased too here as we were expecting it to remain closely similar to the previous test due to the separation of the service. We will test again for the read operation if this is still the case.

We can conclude that the effects of this change has a minor improvement but could potentially save the flask services some load once the need for more monitoring page features are implemented.

Performance Optimization Testing: Separation of services (Read First Pass)

Before Optimisation (Inserting different staff id for all records)

PERF-RT-2-1:
Response time for extracting all user locations between flask server to HACW server.

Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads = 860 Ramp up = 60 DB Data = 600	14.29	41	130	**50.18**
Threads = 860 Ramp up = 60 DB Data = 1200	14.29	42	138	**50.91**
Threads = 860 Ramp up = 60 DB Data = 1800	14.29	43	163	**54.24**
Threads = 860 Ramp up = 60 DB Data = 2400	14.29	42	269	**54.49**
Threads = 860 Ramp up = 60 DB Data = 3000	14.29	43	275	**55.00**

After Optimisation (Inserting different staff id for all records)

PERF-RT-2-1:
Response time for extracting all user locations between flask server to HACW server.

Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads = 860 Ramp up = 60 DB Data = 600	14.29	39	88	**48.38**
Threads = 860 Ramp up = 60 DB Data = 1200	14.29	39	113	**46.70**
Threads = 860 Ramp up = 60 DB Data = 1800	14.29	37	150	**47.54**
Threads = 860 Ramp up = 60 DB Data = 2400	14.29	38	156	**48.16**
Threads = 860 Ramp up = 60 DB Data = 3000	14.29	39	171	**48.17**

Result Analysis

Similar to the previous test, we have reverted the new architecture's database and NGINX configurations to the same as the old architecture. This will ensure that we are testing the difference of the flask servers only.

In order not to fail the system, we will be using the variation of RT-2 where we insert different staff id per round. The test will only retrieve staff_id 0 and will focus on testing the improvements of the read operation to see if any improvements have been made from separating the monitoring page from the flask server service. Again, the results have improved only by a few microseconds.

We will perform a second pass with more monitoring pages open to simulate more load on the monitoring page to properly test the difference of the separation of services.

Performance Optimization Testing: Separation of services (Read Second Pass)

Before Optimisation (Inserting different staff id for all records)

PERF-RT-2-1: Response time for extracting all user locations between flask server to HACW server.				
Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads = 860 Ramp up = 60 DB Data = 600	14.30	43	128	**51.65**
Threads = 860 Ramp up = 60 DB Data = 1200	14.29	44	1294	**68.68**
Threads = 860 Ramp up = 60 DB Data = 1800	14.29	44	295	**58.78**
Threads = 860 Ramp up = 60 DB Data = 2400	14.29	44	781	**68.48**
Threads = 860 Ramp up = 60 DB Data = 3000	14.29	45	316	**65.24**

After Optimisation (Inserting different staff id for all records)

PERF-RT-2-1: Response time for extracting all user locations between flask server to HACW server.				
Data Sample	Throughput Trans/sec	Min Response (ms)	Max Response (ms)	Average Response (ms)
Threads = 860 Ramp up = 60 DB Data = 600	14.29	38	117	**49.06**
Threads = 860 Ramp up = 60 DB Data = 1200	14.29	41	635	**54.02**
Threads = 860 Ramp up = 60 DB Data = 1800	14.29	41	386	**51.43**
Threads = 860 Ramp up = 60 DB Data = 2400	14.29	42	391	**52.75**
Threads = 860 Ramp up = 60 DB Data = 3000	14.29	40	225	**51.37**

Result Analysis

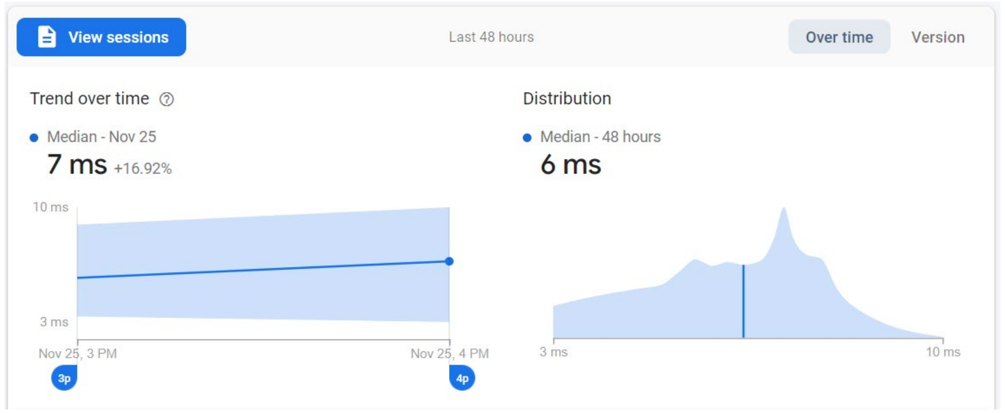
Similar to the previous test, we have reverted the new architecture's database and NGINX configurations to the same as the old architecture. This will ensure that we are testing the difference of the flask servers only. Also notice that we are running a variation of RT-2 similar to the previous test. We have also started **10 monitoring pages** instead of 1 to test if there if performance improvements when load lifted from the flask servers and to simulate more load on the monitoring page.

Each monitoring page will refresh it's data every 30 seconds and display all the latest staff information within a 15 minute timeframe, which can be a costly retrieve operation when multiple instances are spawned. Do note that the HACS server is also running in this test to ensure operational efficacy is true. Similar to the write operation test, both setup's have increased in response times. However the increase in the original setup seems to be a lot higher compared to the optimised setup, with differences up to 10 ms as compared to 3ms.

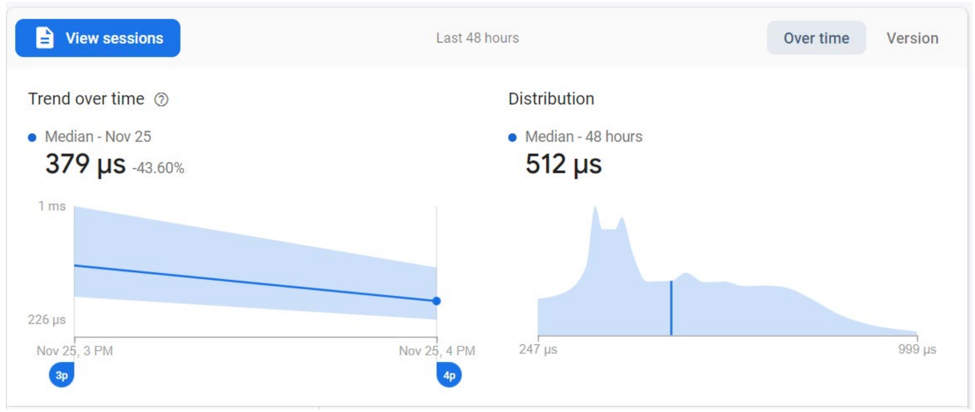
Once again, we can conclude that the effects of this change has a minor improvement but could potentially save the flask services some load once the need for more monitoring page features are implemented.

D2 Appendices

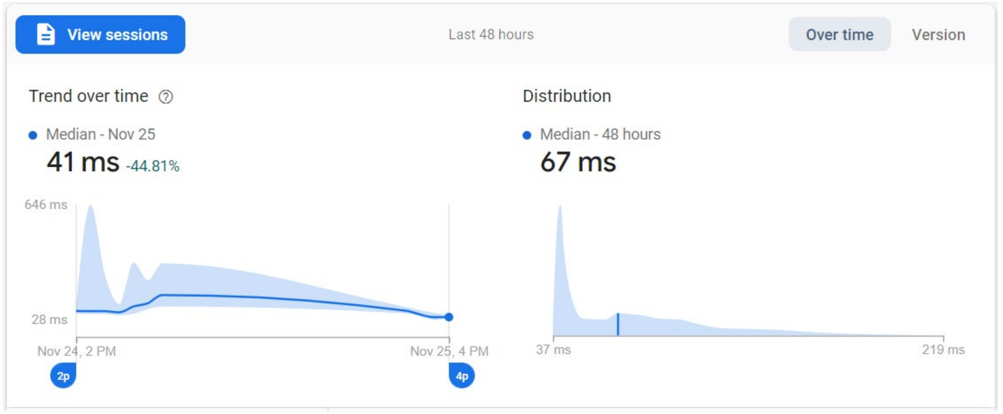
Appendices A: Firebase Performance Monitoring Console Data



Before Optimization: API Request Response Time

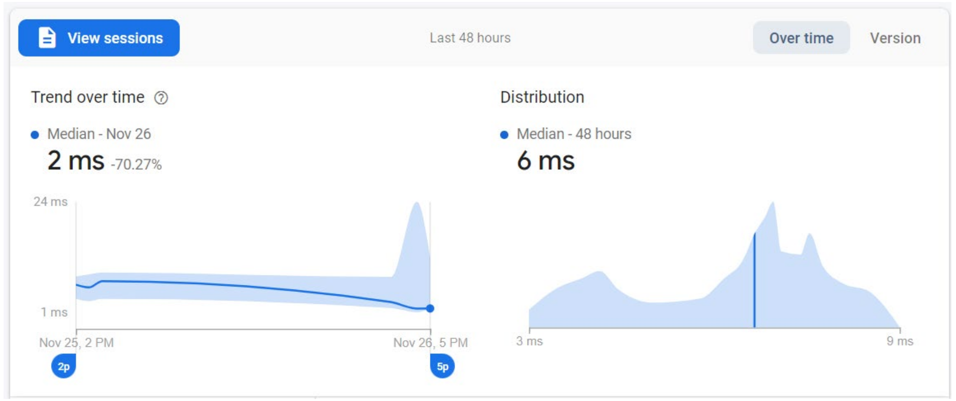


Before Optimization: Data Processing Response Time

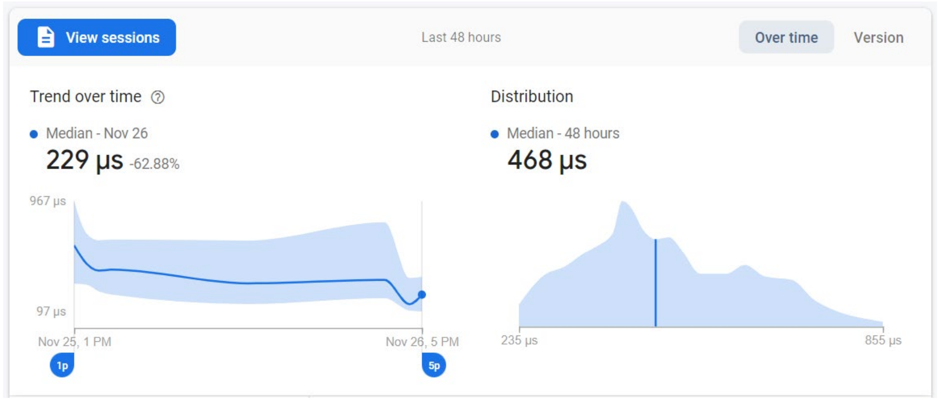


Before Optimization: Network Trace Request and Response Time

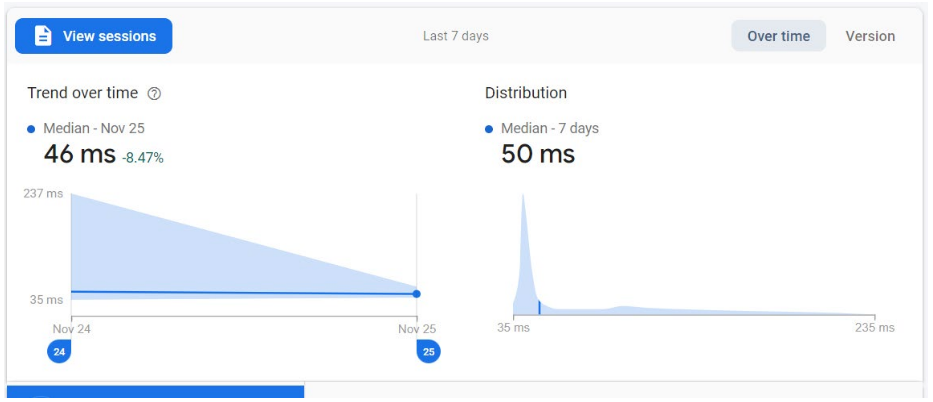
Appendices A: Firebase Performance Monitoring Console Data



After Optimization: API Request Response Time



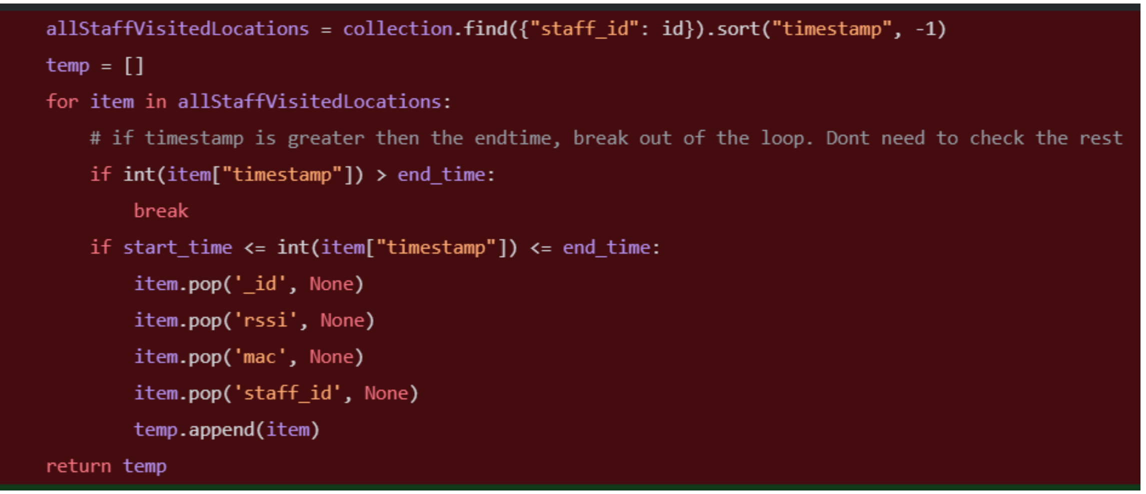
After Optimization: Data Processing Response Time



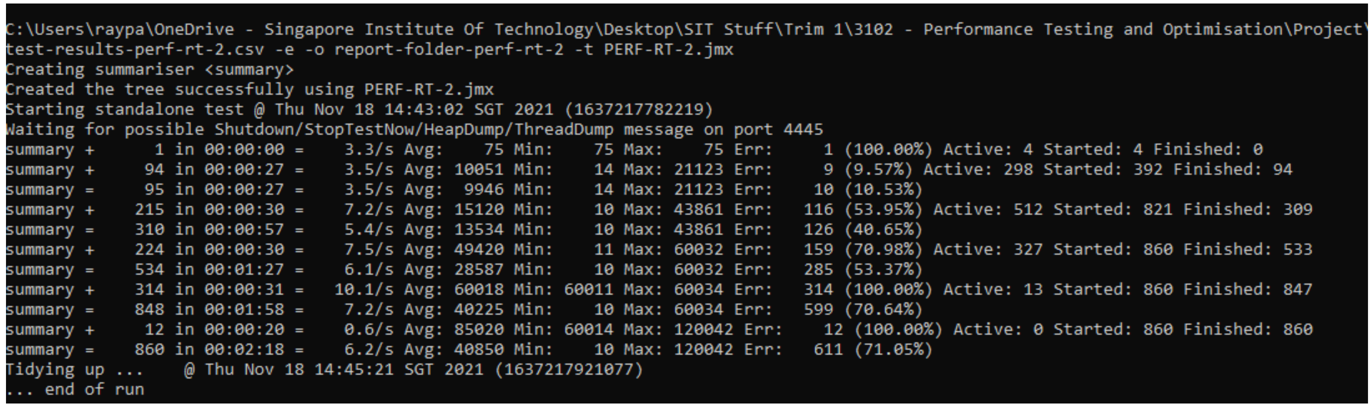
After Optimization: Network Trace Request and Response Time

Appendices B: Bad MongoDB Query Optimization

Before Optimization, query was not optimal

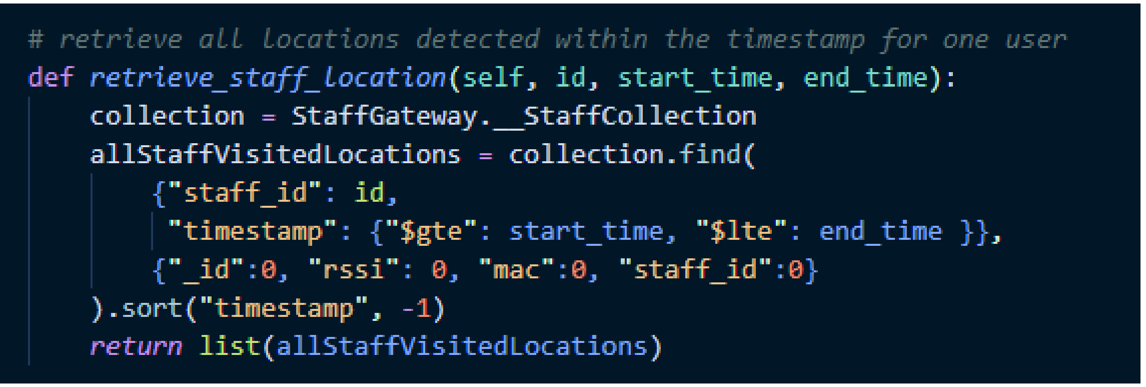


```
allStaffVisitedLocations = collection.find({"staff_id": id}).sort("timestamp", -1)
temp = []
for item in allStaffVisitedLocations:
    # if timestamp is greater then the endtime, break out of the loop. Dont need to check the rest
    if int(item["timestamp"]) > end_time:
        break
    if start_time <= int(item["timestamp"]) <= end_time:
        item.pop('_id', None)
        item.pop('rssi', None)
        item.pop('mac', None)
        item.pop('staff_id', None)
        temp.append(item)
return temp
```

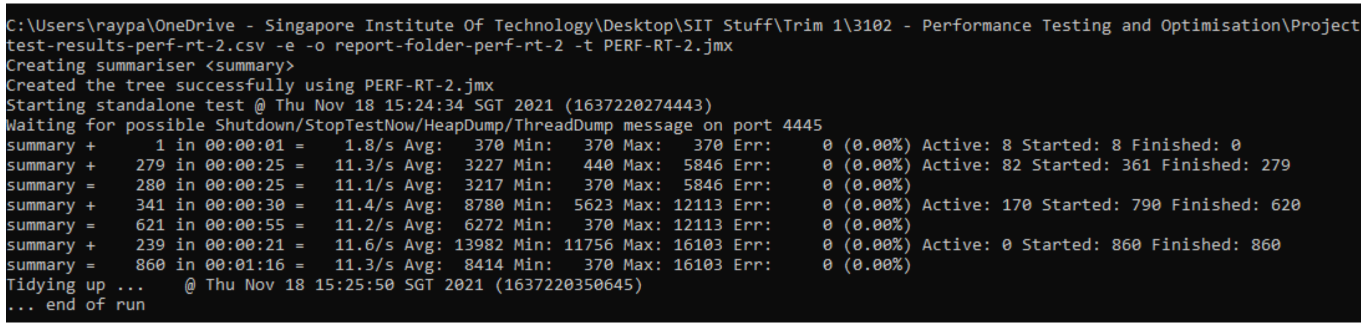


```
C:\Users\raypa\OneDrive - Singapore Institute Of Technology\Desktop\SIT Stuff\Trim 1\3102 - Performance Testing and Optimisation\Project
test-results-perf-rt-2.csv -e -o report-folder-perf-rt-2 -t PERF-RT-2.jmx
Creating summariser <summary>
Created the tree successfully using PERF-RT-2.jmx
Starting standalone test @ Thu Nov 18 14:43:02 SGT 2021 (1637217782219)
Waiting for possible Shutdown/StopTestNow/HeapDump/ThreadDump message on port 4445
summary +      1 in 00:00:00 =    3.3/s Avg:    75 Min:    75 Max:    75 Err:     1 (100.00%) Active: 4 Started: 4 Finished: 0
summary +     94 in 00:00:27 =    3.5/s Avg: 10051 Min:    14 Max: 21123 Err:     9 (9.57%) Active: 298 Started: 392 Finished: 94
summary =     95 in 00:00:27 =    3.5/s Avg:  9946 Min:    14 Max: 21123 Err:    10 (10.53%)
summary +    215 in 00:00:30 =    7.2/s Avg: 15120 Min:    10 Max: 43861 Err:   116 (53.95%) Active: 512 Started: 821 Finished: 309
summary =    310 in 00:00:57 =    5.4/s Avg: 13534 Min:    10 Max: 43861 Err:   126 (40.65%)
summary +    224 in 00:00:30 =    7.5/s Avg: 49420 Min:    11 Max: 60032 Err:   159 (70.98%) Active: 327 Started: 860 Finished: 533
summary =    534 in 00:01:27 =    6.1/s Avg: 28587 Min:    10 Max: 60032 Err:   285 (53.37%)
summary +    314 in 00:00:31 =   10.1/s Avg: 60018 Min: 60011 Max: 60034 Err:   314 (100.00%) Active: 13 Started: 860 Finished: 847
summary =    848 in 00:01:58 =    7.2/s Avg: 40225 Min:    10 Max: 60034 Err:   599 (70.64%)
summary +     12 in 00:00:20 =    0.6/s Avg: 85020 Min: 60014 Max: 120042 Err:    12 (100.00%) Active: 0 Started: 860 Finished: 860
summary =    860 in 00:02:18 =    6.2/s Avg: 40850 Min:    10 Max: 120042 Err:   611 (71.05%)
Tidying up ...    @ Thu Nov 18 14:45:21 SGT 2021 (1637217921077)
... end of run
```

After Optimization, less crashes were observed. Did not add this into main report due to the fact that it was poorly coded from the beginning, and we don't want to force out fake optimization techniques.



```
# retrieve all locations detected within the timestamp for one user
def retrieve_staff_location(self, id, start_time, end_time):
    collection = StaffGateway.__StaffCollection
    allStaffVisitedLocations = collection.find(
        {"staff_id": id,
         "timestamp": {"$gte": start_time, "$lte": end_time }},
        {"_id":0, "rssi": 0, "mac":0, "staff_id":0}
    ).sort("timestamp", -1)
    return list(allStaffVisitedLocations)
```



```
C:\Users\raypa\OneDrive - Singapore Institute Of Technology\Desktop\SIT Stuff\Trim 1\3102 - Performance Testing and Optimisation\Project
test-results-perf-rt-2.csv -e -o report-folder-perf-rt-2 -t PERF-RT-2.jmx
Creating summariser <summary>
Created the tree successfully using PERF-RT-2.jmx
Starting standalone test @ Thu Nov 18 15:24:34 SGT 2021 (1637220274443)
Waiting for possible Shutdown/StopTestNow/HeapDump/ThreadDump message on port 4445
summary +      1 in 00:00:01 =    1.8/s Avg:   370 Min:   370 Max:   370 Err:     0 (0.00%) Active: 8 Started: 8 Finished: 0
summary +    279 in 00:00:25 =   11.3/s Avg:  3227 Min:   440 Max:  5846 Err:     0 (0.00%) Active: 82 Started: 361 Finished: 279
summary =    280 in 00:00:25 =   11.1/s Avg:  3217 Min:   370 Max:  5846 Err:     0 (0.00%)
summary +    341 in 00:00:30 =   11.4/s Avg:  8780 Min:  5623 Max: 12113 Err:     0 (0.00%) Active: 170 Started: 790 Finished: 620
summary =    621 in 00:00:55 =   11.2/s Avg:  6272 Min:   370 Max: 12113 Err:     0 (0.00%)
summary +    239 in 00:00:21 =   11.6/s Avg: 13982 Min: 11756 Max: 16103 Err:     0 (0.00%) Active: 0 Started: 860 Finished: 860
summary =    860 in 00:01:16 =   11.3/s Avg:  8414 Min:   370 Max: 16103 Err:     0 (0.00%)
Tidying up ...    @ Thu Nov 18 15:25:50 SGT 2021 (1637220350645)
... end of run
```

D1 Appendices

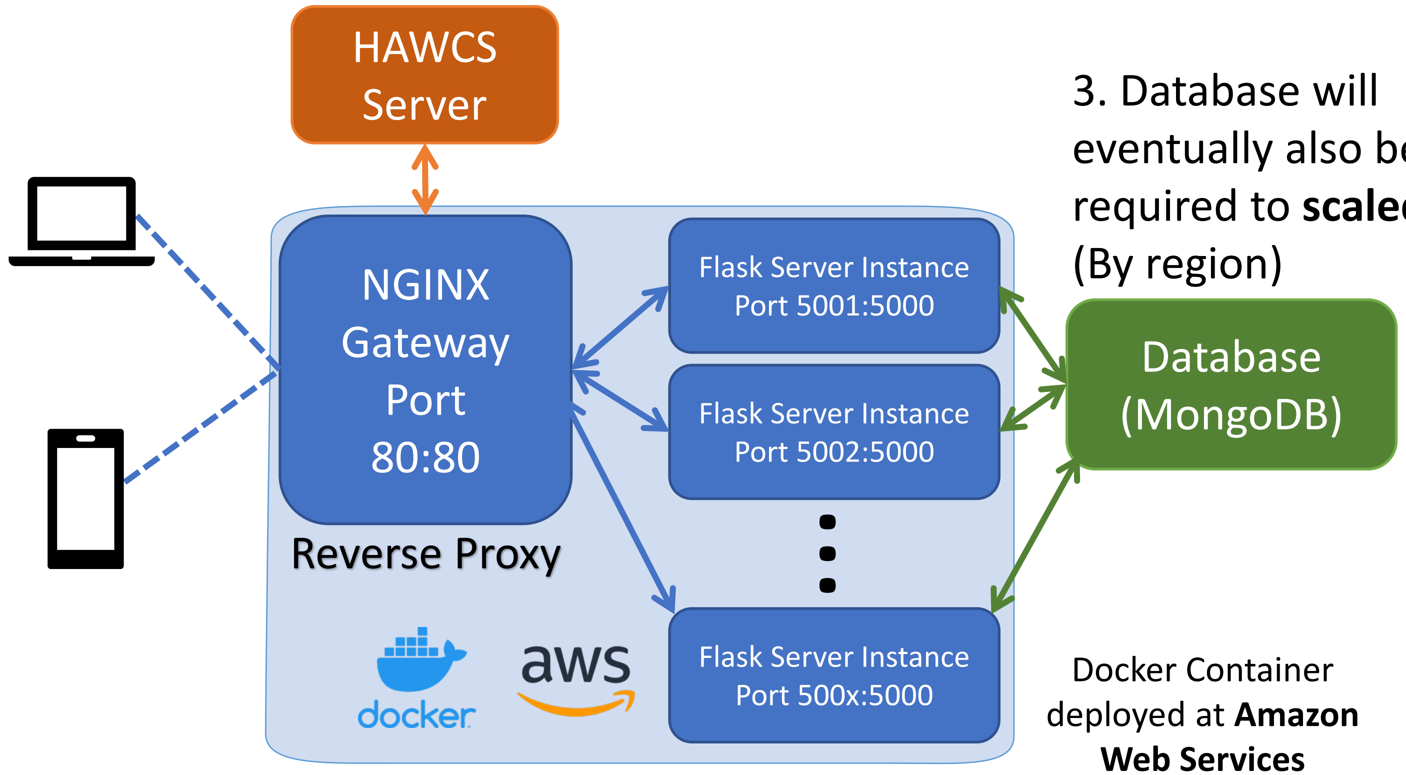
Do note that D1 contents are only added for reference for the commenting team and have not been updated based on feedback provided.

Appendices C: D1 Architecture

Production Environment Scaling

1. Application will utilise a reverse proxy NGINX to **load balance** incoming requests to multiple docker container server instances
2. Application will **scale out** by adding more instances (2 or more docker application containers)

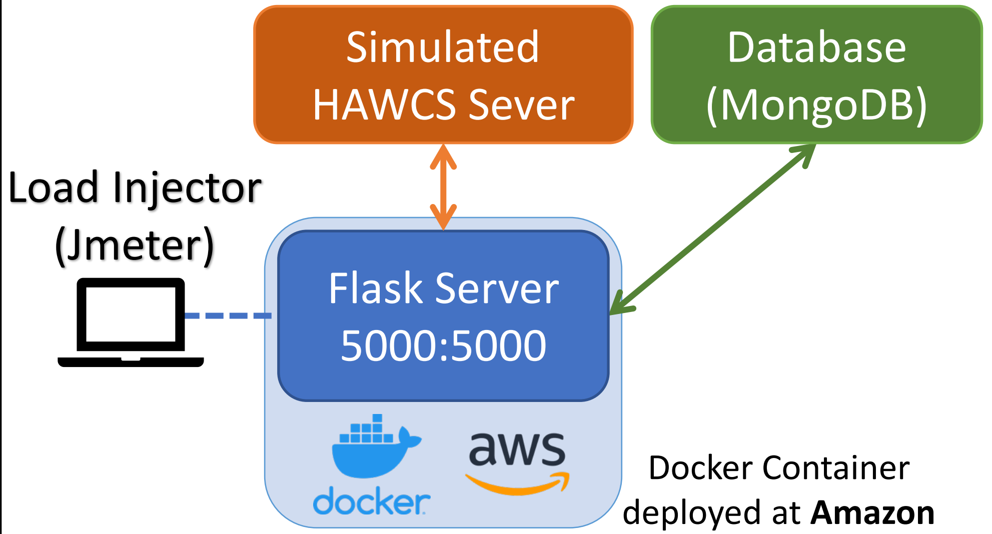
3. Database will eventually also be required to **scaled up**. (By region)


HAWCS Server
NGINX Gateway Port 80:80
Reverse Proxy
Flask Server Instance Port 5001:5000
Flask Server Instance Port 5002:5000
Flask Server Instance Port 500x:5000
Database (MongoDB)
docker
aws

Docker Container deployed at **Amazon Web Services** EC2 T2.Micro

Test Environment

In relation to the production environment with **2 Flask Servers**, the expected load should **scaled down** by $\frac{1}{2}$ due to the production version scaling of containers. The load will be projected by the load injector towards the Flask server as well as the simulated HAWCS Server.



Docker Container deployed at **Amazon Web Services** EC2 T2.Micro

Appendices D: D1 Requirements Under Test

Note: See Estimated Workload for Estimate Workload for SDA (Slide 4) for justifications and sources for SDA numbers.

ID	Feature Description	Flask Server API Endpoint	Frequency Estimation	Related Test Requirement and test
R1	Retrieving of recent location data for a single user given the inputs for start time and end time (timeframe)	/extractbeacon	The automatic update request from HAWCS Server to Flask server is every **10 second**. Working For 1 SDA staff = 6 Request/minute or **0.1 Calls/Second** For 14 SDA Staff = 84 Request/minute or **1.4 Calls/Second**	**PERF-TP-2** Load Testing **PERF-RT-2** Stress Testing
R2	Adding a new beacon detection to the database given the inputs from the mobile application.	/newbeacondetect	The operation is to be invoked automatically whenever a beacon is detected every **4 second for each SDA**. Working **Peak**: 15 Calls/Minutes X 2 SDA X 7 Levels = 210 Calls/Minute OR **3.5 Calls/Seconds** **Non-Peak**: 15 Calls/Minute X 2 SDA = 30 Calls/Minute OR **0.5 Calls/Seconds**	**PERF-TP-1** Load Testing **PERF-RT-1** Stress Testing

Appendices D: D1 Requirements Under Test

Note: See Estimated Workload for Estimate Workload for Admin Monitoring (Slide 5) for justifications and sources.

ID	Feature Description	Flask Server API Endpoint	Frequency Estimation	Related Test Requirement and test
R3	Display all beacon user and recent location details on the monitoring page	/	The automatic page refresh for the admin monitoring page is every **300 seconds** (5min) <u>Working</u> Automated page refresh 1/5 call every min or **0.0033 Calls/Second** Manual page refresh (peak and non-peak) ~4 Calls every Hour or **0.0011 Calls/Second** Total rate = **0.0033 + 0.0011 Calls/Second** **= 0.0044 Calls/Seconds** Or 0.264 Calls/Minute Or 15.84 Calls/Hour	**PERF-RT-3** Stress Testing

Appendices E: D1 Performance Requirements

Requirement No.	PERF-TP-1
Title	Throughput for uploading detected beacon information
Statement of Requirement	In 99% of all cases, the operation shall have a transaction rate of **3.5 Calls/Second** (Peak) and **0.5 Calls/Second** (Non-peak) for uploading information for detected beacon
Supporting Commentary	The test will monitor the throughput of inserting new information of detected beacons into the database
Sources	Derived from justification from Slide **4**
Measurement	The throughput will be measured using Jmeter
Derivation of Quantities	Calculation of throughput is derived from Slide **4**, Estimated Workload (SDA) Peak: 15 Calls/Minutes X 2 SDA X 7 Levels = 210 Calls/Minute OR **3.5 Calls/Seconds** Non-Peak: 15 Calls/Minute X 2 SDA = 30 Calls/Minute OR **0.5 Calls/Second**
Assumptions	Mokobeacon mac address is within the application's address whitelist
Subject Matter Expert	Pak Shao Kai

Appendices E: D1 Performance Requirements

Requirement No.	PERF-TP-2
Title	Throughput for viewing location history of a user
Statement of Requirement	In 99% of all cases, the operation shall have an average transaction rate of **1.4 Calls/Second** to view the location history of a staff
Supporting Commentary	The test will monitor the throughput of retrieving all recent user location information from the database
Sources	Derived from justification from Slide 5
Measurement	The throughput will be measured using Jmeter
Derivation of Quantities	Number of HAWCS Server staffs are derived from Slide **5** , Estimated Workload (Admin Monitoring), Working For 1 SDA staff = 6 Request/minute or **0.1 Calls/Second** For 14 SDA Staff = 84 Request/minute or **1.4 Calls/Second**
Assumptions	Timeframe of retrieved data is a constant 10 seconds
Subject Matter Expert	Pak Shao Kai

Appendices E: D1 Performance Requirements

Requirement No.	PERF-RT-1
Title	Response time for uploading detected beacon information
Statement of Requirement	In 99% of all cases, the operation shall have a response time of no more than **6 Seconds** to upload the information of the detected beacon
Supporting Commentary	Computation time may take longer due to network delays in request from the mobile app
Sources	MongoDB Mean Insert Query Performance: Performance evaluation for CRUD operations inasynchronously replicated document orienteddatabase; Truică, Ciprian-Octavian & Rădulescu, Florin & Boicea, Alexandru & Bucur, Ion. (2015)
Measurement	The response time will be measured using Jmeter
Derivation of Quantities	The operation inserts beacon location data to the database for a selected user. It is found that a single instance of Mongodb can perform **100,000 insert operations in 5282.5ms** (5.28 seconds). Thus, the operation should take a **maximum of 6000 ms** to perform
Assumptions	Bandwidth of the server is constant
Subject Matter Expert	Raynold Tan Yong Ren

Appendices E: D1 Performance Requirements

Requirement No.	PERF-RT-2
Title	Response time for viewing location history of a user
Statement of Requirement	In 99% of all cases, the operation shall have a response time of no more than **1 Second** to view the location history of a user
Supporting Commentary	Computation time may take longer due to network delays between requesting HAWCS Server and the Flask Server
Sources	MongoDB Mean Select/Retrieve Query Performance: Performance evaluation for CRUD operations inasynchronously replicated document orienteddatabase; Truică, Ciprian-Octavian & Rădulescu, Florin & Boicea, Alexandru & Bucur, Ion. (2015)
Measurement	The response time will be measured using Jmeter
Derivation of Quantities	The operation retrieves information from the database and checks for a selected user within a stated timeframe. It is found that a single instance of MongoDB can perform **100,000 select operations in 43.5ms**. Thus, the operation should take a **maximum of 1 seconds** to perform
Assumptions	Bandwidth of the server is constant
Subject Matter Expert	Raynold Tan Yong Ren

Appendices F: Other Resources

Bluetooth Inquiry Time Characterization and Selection. (2021). Retrieved 30 September 2021,

from https://ieeexplore.ieee.org/document/1661527

Performance Evaluation for CRUD Operations in Asynchronously Replicated Document Oriented Database. (2021). Retrieved 30 September 2021,

from https://www.researchgate.net/publication/280832003_Performance_Evaluation_for_CRUD_Operations_in_Asynchronously_Replicated_Document_Oriented_Database

Response Time Limits: Article by Jakob Nielsen. (2021). Retrieved 30 September 2021,

from https://www.nngroup.com/articles/response-times-3-important-limits/

The Bluetooth Wireless Technology Overview. (2021). Retrieved 30 September 2021,

from http://oscar.iitb.ac.in/onsiteDocumentsDirectory/Bluetooth/Bluetooth/Help/Technology%20Overview.htm