CS6650 Assignment 2 Report

Shengdi Wang - wang.shengd@northeastern.edu

1. GitHub Repo URL

https://github.com/leowang396/distributed-music-service

2. Description of data model

Image used

Used a 67-byte PNG image instead of the stock image. The image is stored in the GitHub repo's client directory for reference.

Database solution

AWS DynamoDB.

Data model

Since we are using the AWS DynamoDB (NoSQL) as the database, we apply the document data model to represent our data. The database models our data as a single album entity that contains the album information and album images posted by the clients. The album information is represented in the String fields "artist", "title", and "year", and the album image is stored in a binary field named "image".

By storing all fields in a single entity, we reduce the number of database write requests needed to fulfill the POST requests, which make up 50% of all client requests. To maintain a primary key for each record, we also generate a UUID for each record, which is stored in the "album_id" field in the database.

3. Output windows for the 3 client configuration tests run against a single server/DB

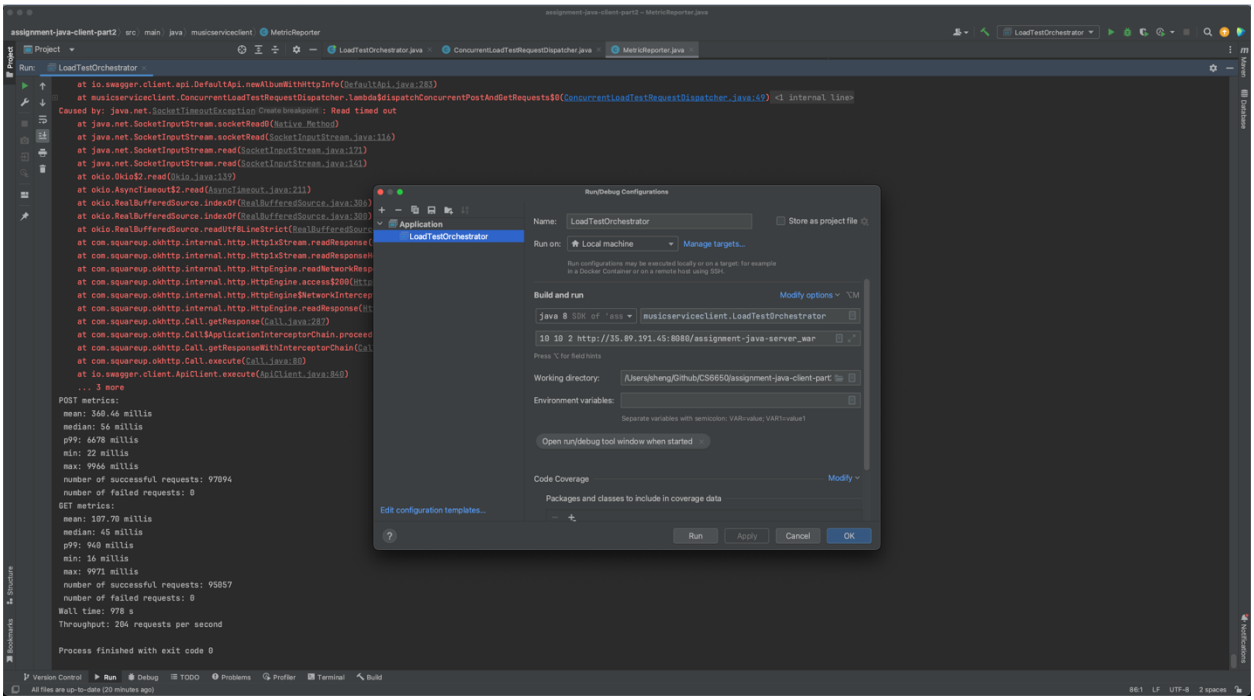

Figure 1: Single Server - 10 Thread Groups

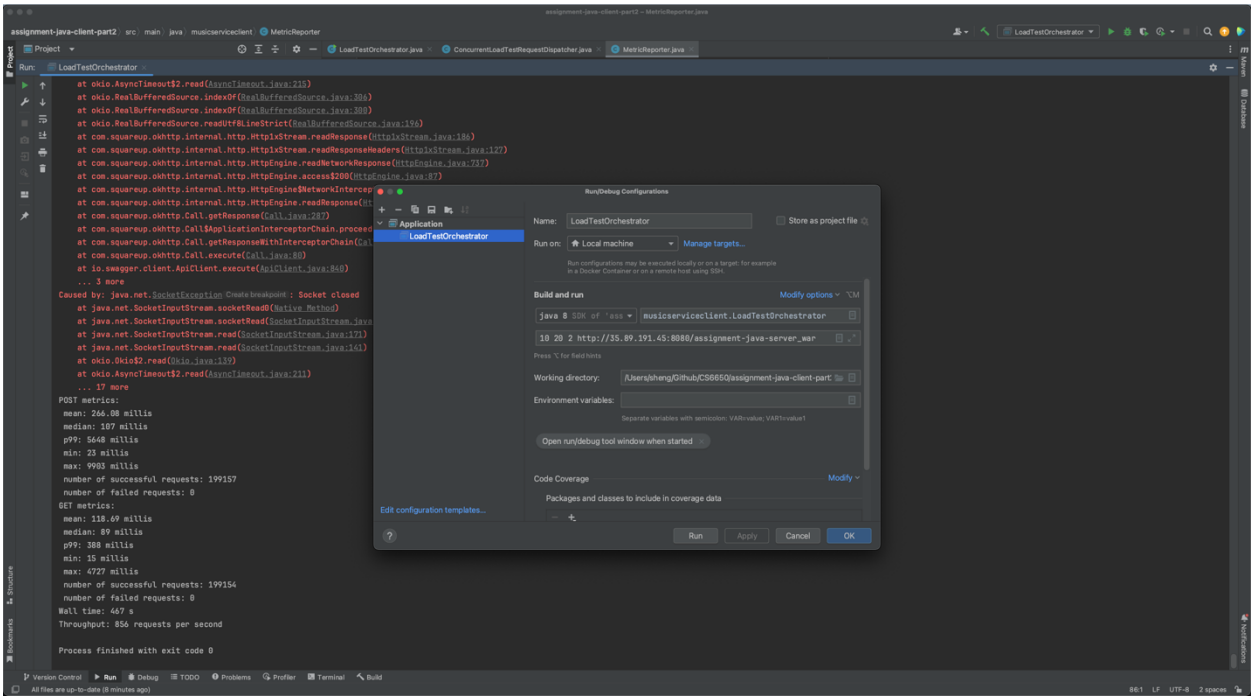

Figure 2: Single Server - 20 Thread Groups

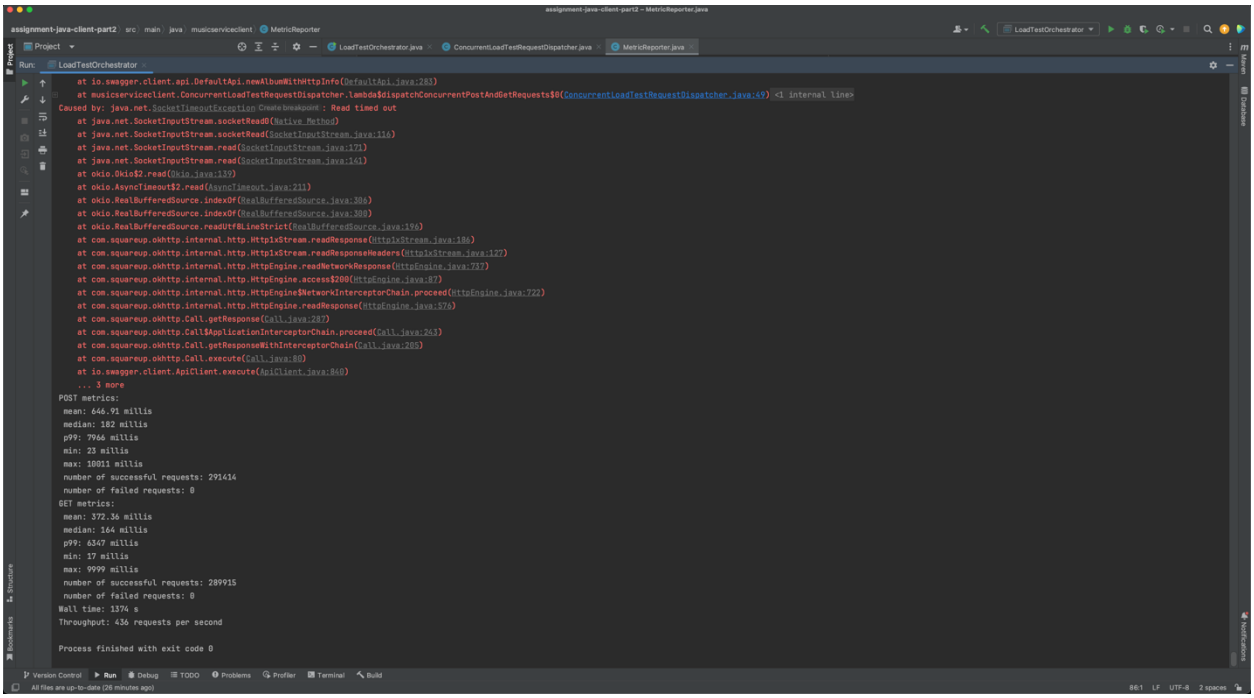

Figure 3: Single Server - 30 Thread Groups

Configuration		Wall Time (s)	Throughput
Single-server	**threadGroupSize = 10, numThreadGroups = 10, delay =**	978	204
	threadGroupSize = 10, numThreadGroups = 20, delay =	467	856
	threadGroupSize = 10, numThreadGroups = 30, delay =	1374	436

Figure 4: Table of results from Step 3

4. Output windows for the 3 client configuration tests run against two load-balanced servers/DB

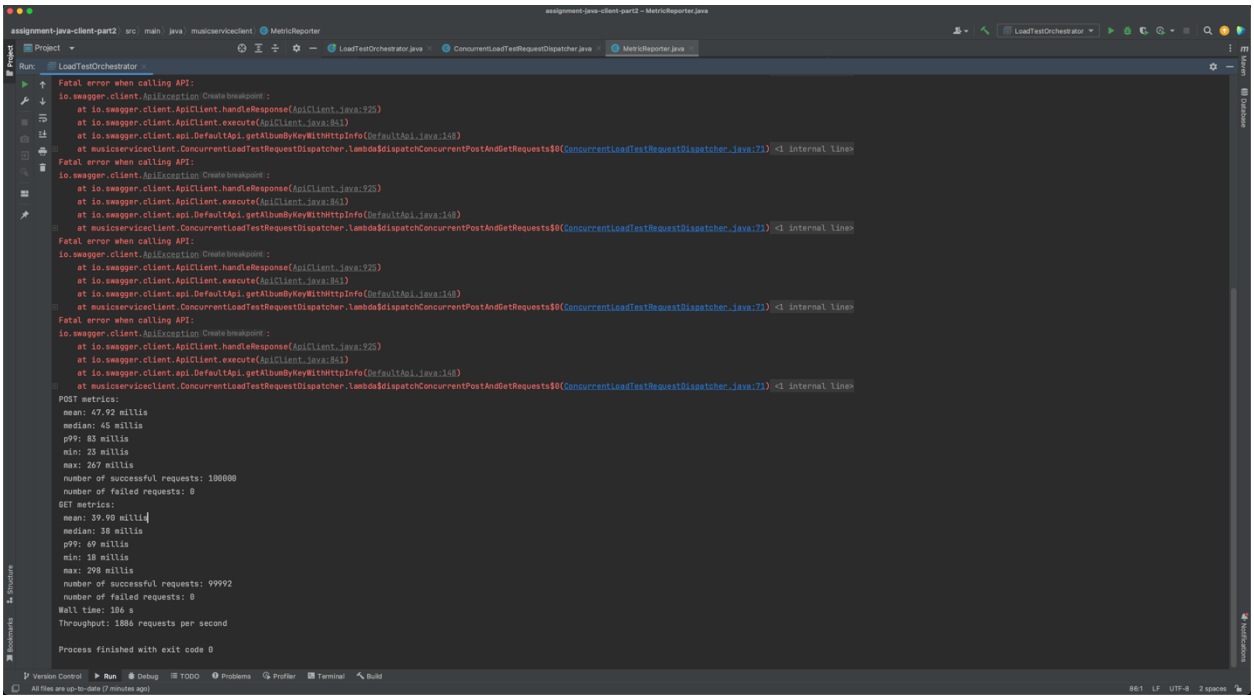

Figure 5: Two Load-balanced Servers - 10 Thread Groups

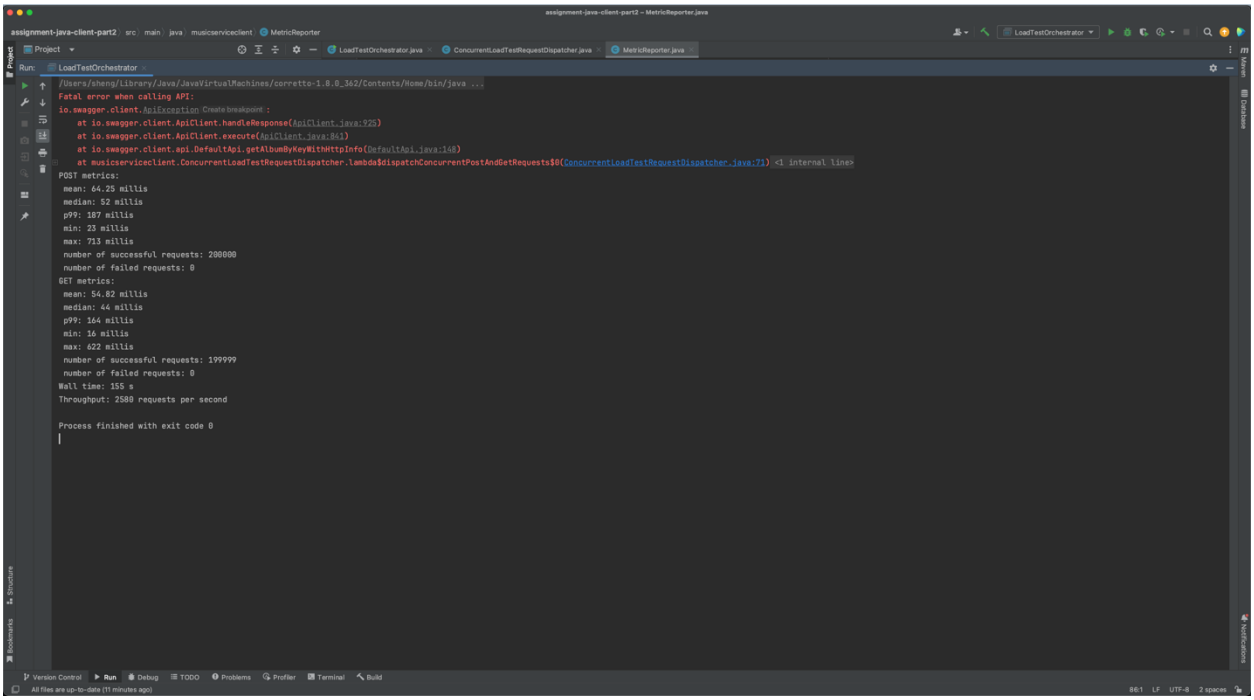

Figure 6: Two Load-balanced Servers - 20 Thread Groups

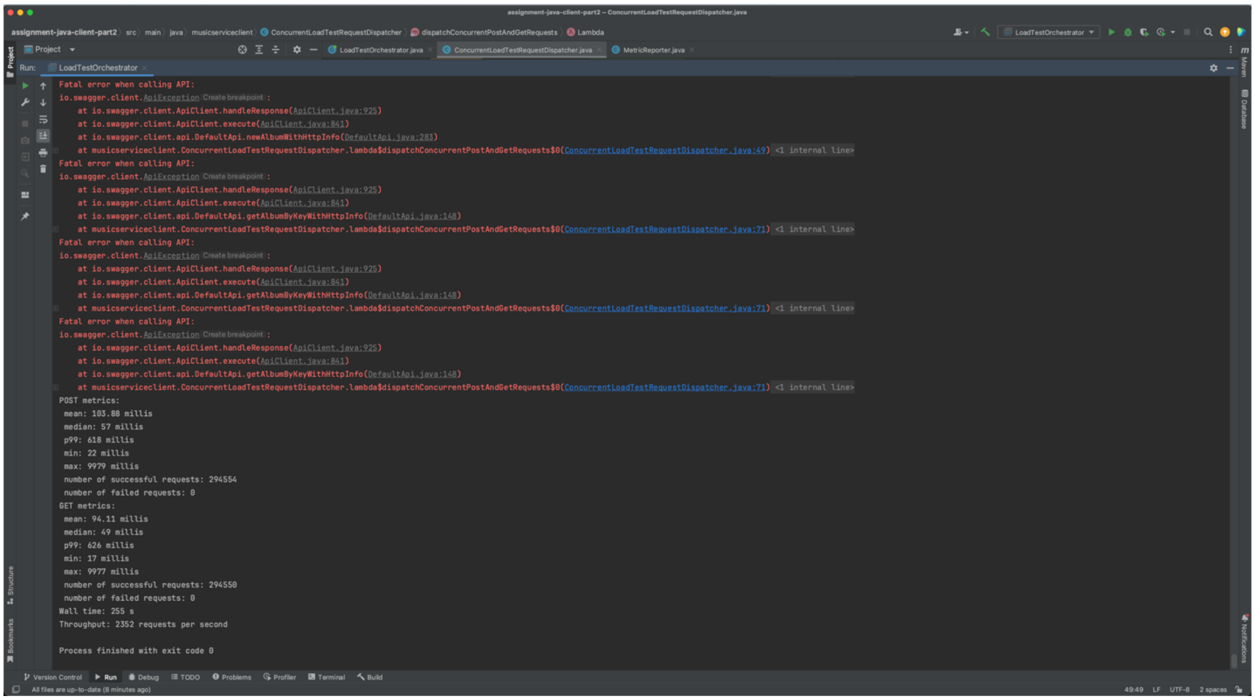

Figure 7: Two Load-balanced Servers - 30 Thread Grousp

Configuration		Wall Time (s)	Throughput
2-load-balanced-servers	threadGroupSize = 10, numThreadGroups = 10, delay =	106	1886
	threadGroupSize = 10, numThreadGroups = 20, delay =	155	2580
	threadGroupSize = 10, numThreadGroups = 30, delay =	255	2352

Figure 8: Tables of Results from Step 4

5. Output window for optimized server configuration for the client with 30 Thread Groups. Briefly describe what configuration changes you made and what % throughput improvement you achieved

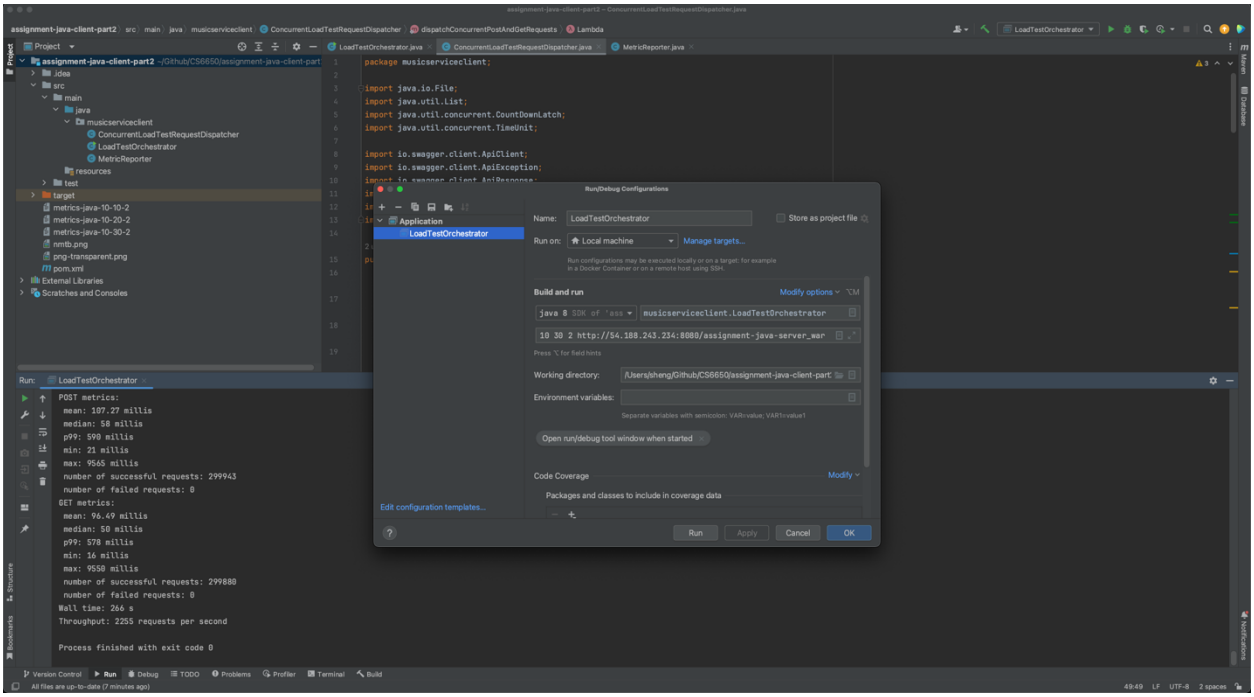

Figure 9: Optimization Configuration - 30 Thread Groups

Configuration		Wall Time (s)	Throughput
Optimized	**threadGroupSize = 10, numThreadGroups = 10, delay =**	94	2137
	threadGroupSize = 10, numThreadGroups = 20, delay =	155	2580
	threadGroupSize = 10, numThreadGroups = 30, delay =	266	2255

Figure 10: Table of Results from Step 5

Multiple iterations of changes were made. A few of these optimizations were done as a part of Steps 3 and 4, and the rest were done after ALB was added.

- I started off using **MySQL/AWS RDS and Java DBCP connection pool** as the database technology. However, despite achieving 2000+ throughput on a local MySQL DB, adding AWS RDS caused the server throughput to drop to ~**450** on EC2.
- After changing the **DB instance type** from db.t2.micro to db.t4g.micro, and swapping the stock image with the smaller **67-byte PNG image**, I was able to achieve a throughput of **600** with MySQL, a significant **33% improvement**.
- Since this is still short of the 1000 throughput target, I decided to try out DynamoDB. **DynamoDB** came with a default read/write capacity of 5 capacity units each, which proved to be way too small for even just 10 thread groups. To save time, I enabled auto-scaling and increased the minimum capacity for read/write from 1 to 100 capacity units. This led to a throughput that fluctuated between **200 to 900**. The wild fluctuation in throughput is due to incremental and often time-consuming responses with

DynamoDB's auto-scaling feature. If we take the highest throughput for comparison, this is a **50% improvement**.

- At this point, I observed the servlet server is getting the exception "java.io.IOException: Too many open file", which indicates that a single server is getting too busy and running out of file descriptors. I then proceeded to add a second server by using the **Application Load Balancer**. This led to a **100% improvement** in throughput from 900 to **1800** for the 10 Thread Groups experiment.
- To address the fluctuations in throughput and frequent reset of read/write capacity by DynamoDB's auto-scaling feature, I raised DynamoDB auto-scaling's **minimum read/write capacity to 2000** before I started the final experiments. This is the observed peak in the read/write capacity when fully managed by the auto-scaling feature. By doing so, I was able to cut away a few minutes of incremental ramp-up by DynamoDB's auto-scaling and achieve a consistent throughput of **over 2000**.

Configuration		Wall Time (s)	Throughput
Single-server	threadGroupSize = 10, numThreadGroups = 10, delay =	978	204
	threadGroupSize = 10, numThreadGroups = 20, delay =	467	856
	threadGroupSize = 10, numThreadGroups = 30, delay =	1374	436
2-load-balanced-servers	threadGroupSize = 10, numThreadGroups = 10, delay =	106	1886
	threadGroupSize = 10, numThreadGroups = 20, delay =	155	2580
	threadGroupSize = 10, numThreadGroups = 30, delay =	255	2352
Optimized	threadGroupSize = 10, numThreadGroups = 10, delay =	94	2137
	threadGroupSize = 10, numThreadGroups = 20, delay =	155	2580
	threadGroupSize = 10, numThreadGroups = 30, delay =	266	2255

Figure 11: Comparison of results across steps 3, 4, 5

6. Screenshots of the ALB setup

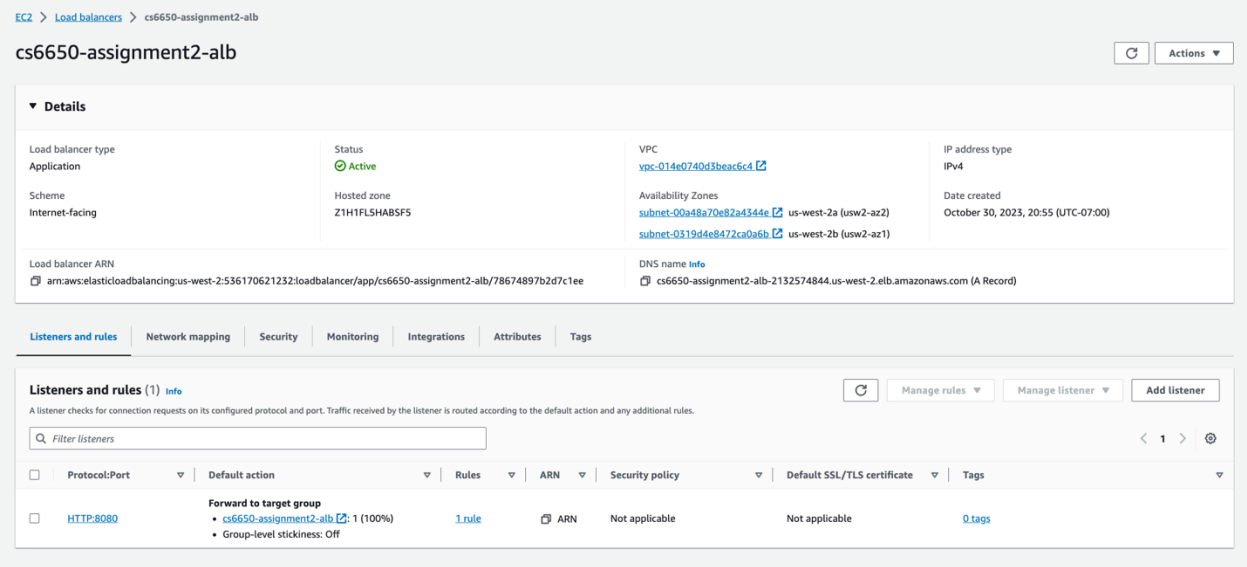

Figure 12: ALB Listener Setup

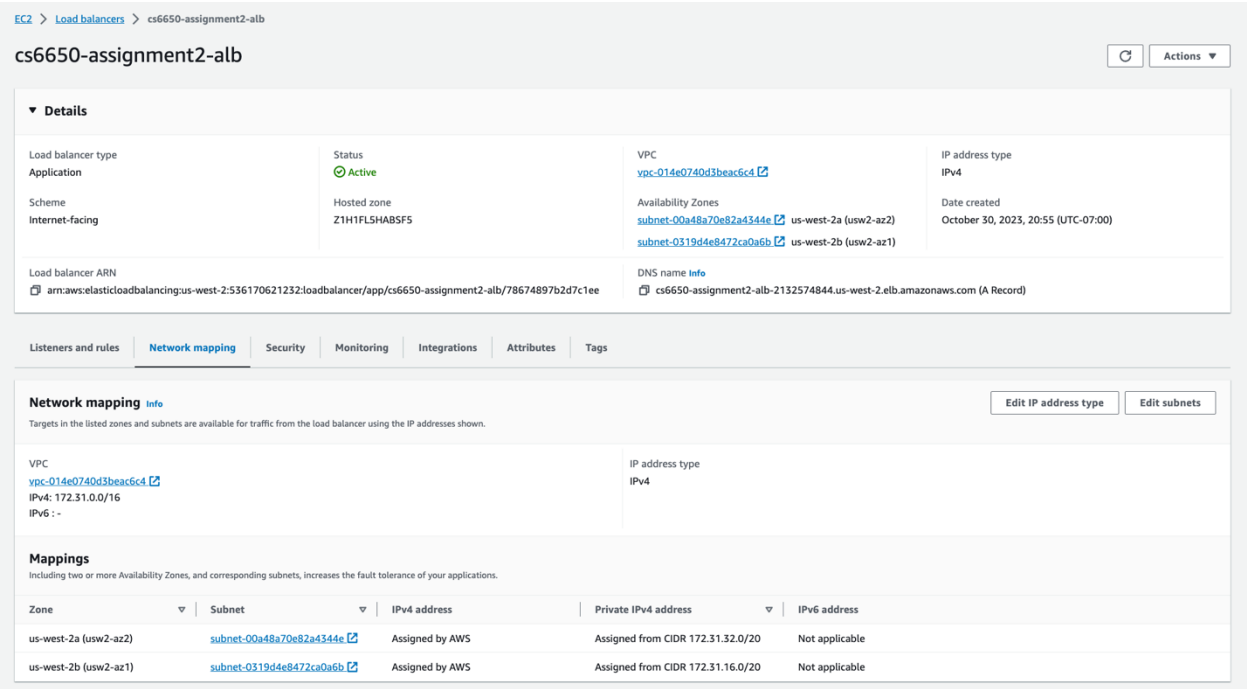

Figure 13: ALB Network Mapping Setup

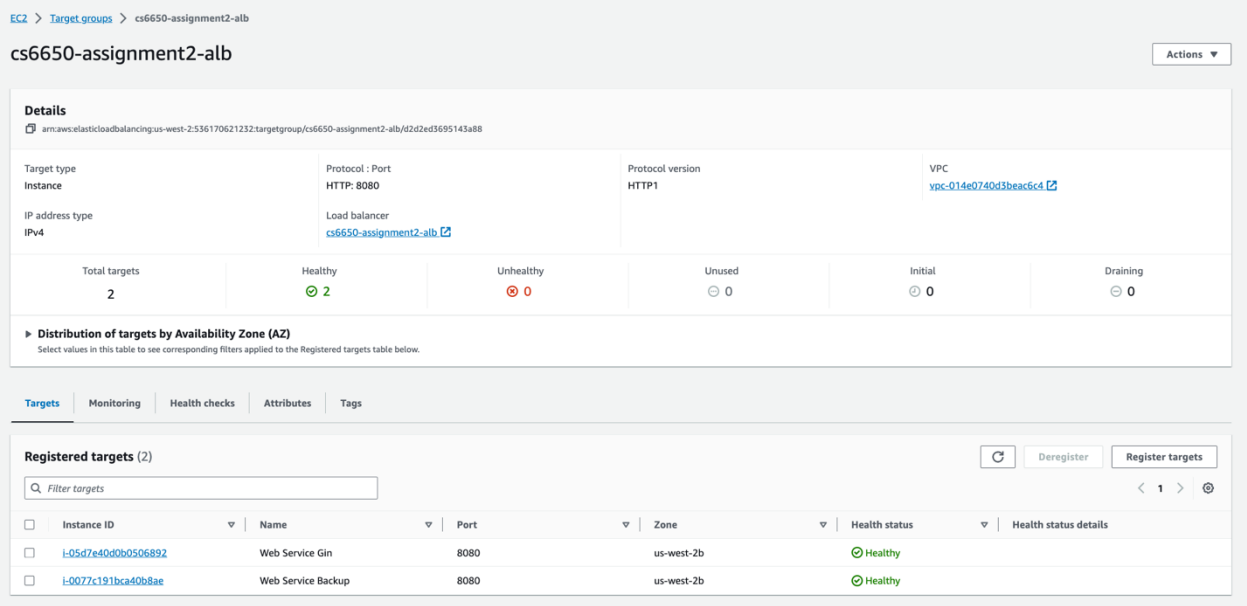

Figure 14: ALB Target Setup

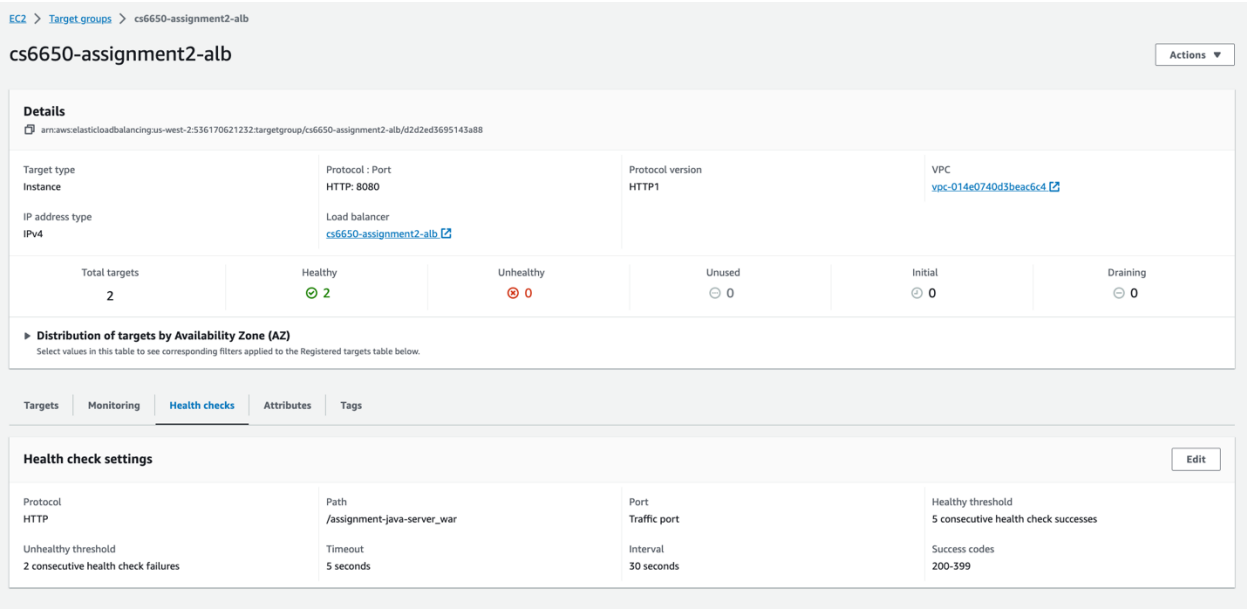

Figure 4: ALB Target Group Health Check Setup

7. Screenshots of the database after testing

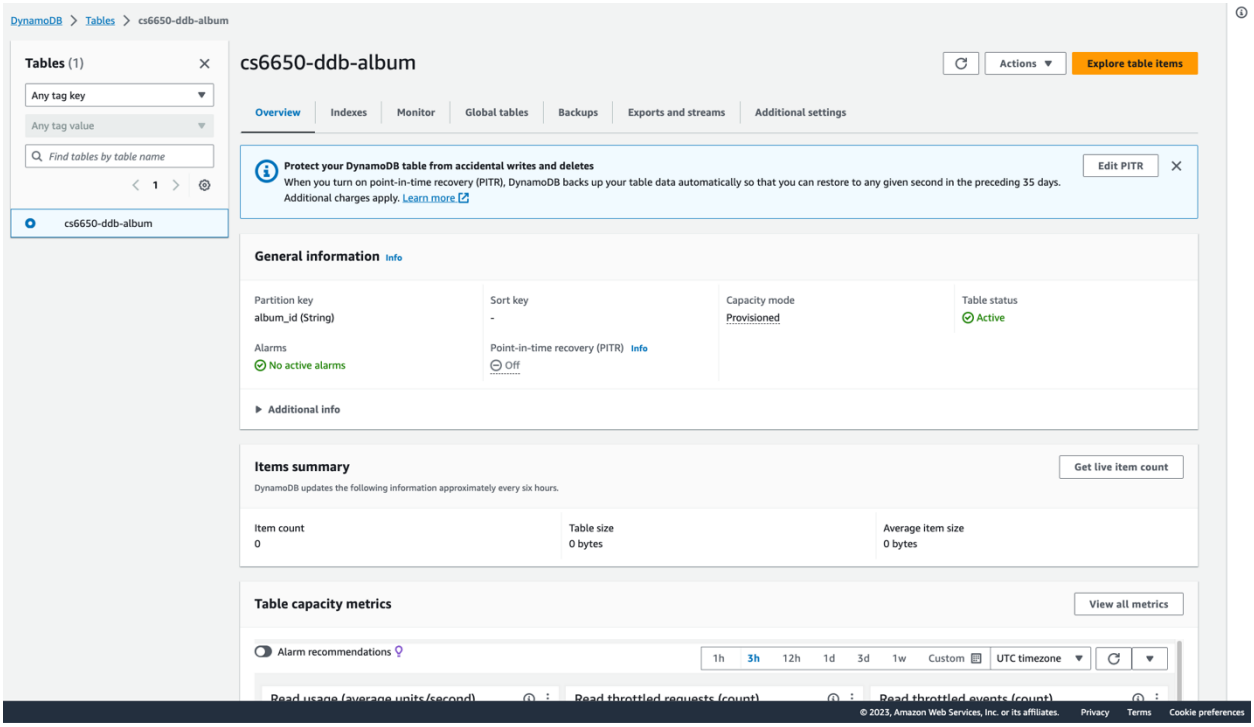

Figure 15: DynamoDB Overview of General Information

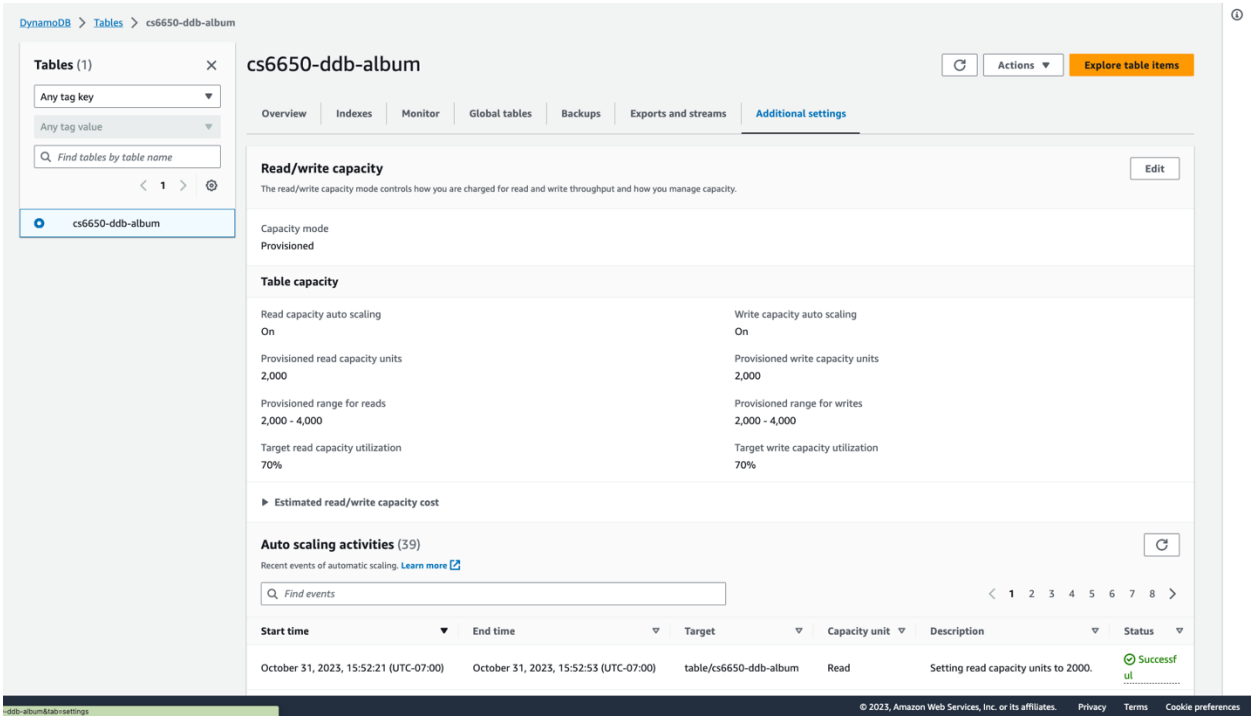

Figure 16: DynamoDB Additional Settings including Auto Scaling Settings

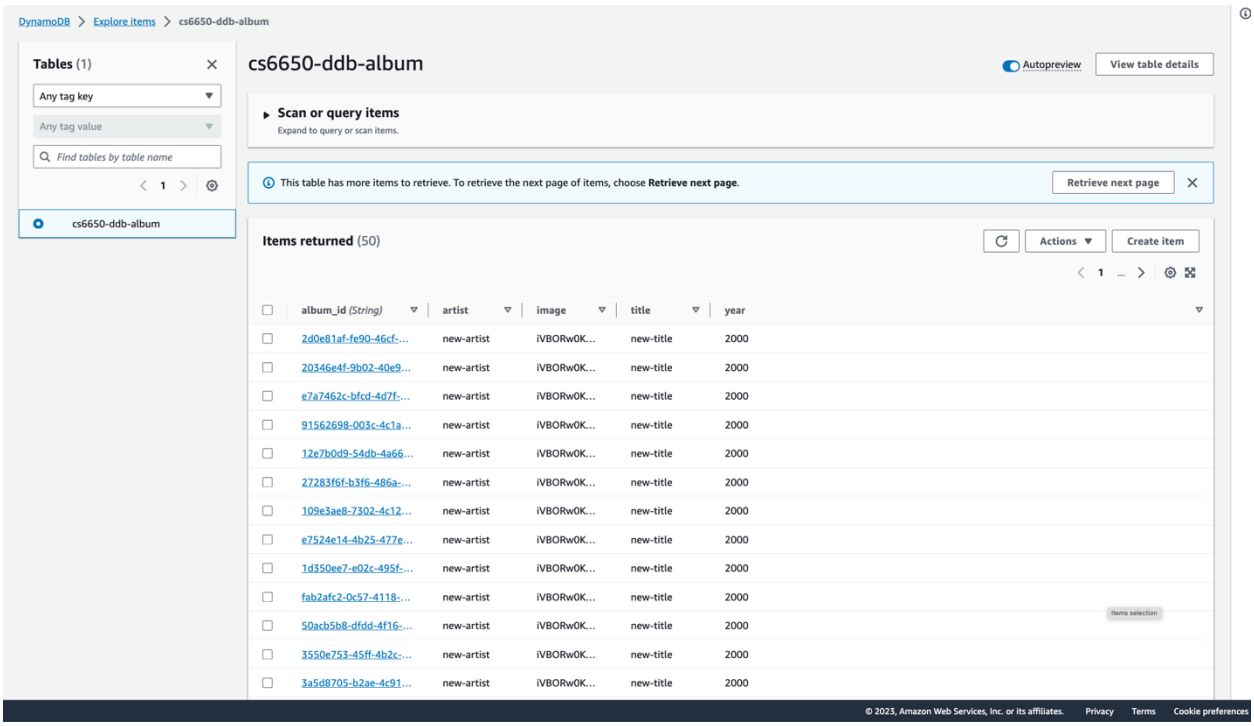

Figure 17: DynamoDB Samples of Items

8. Any other screenshot to demonstrate the effect of your configuration change for task 5.

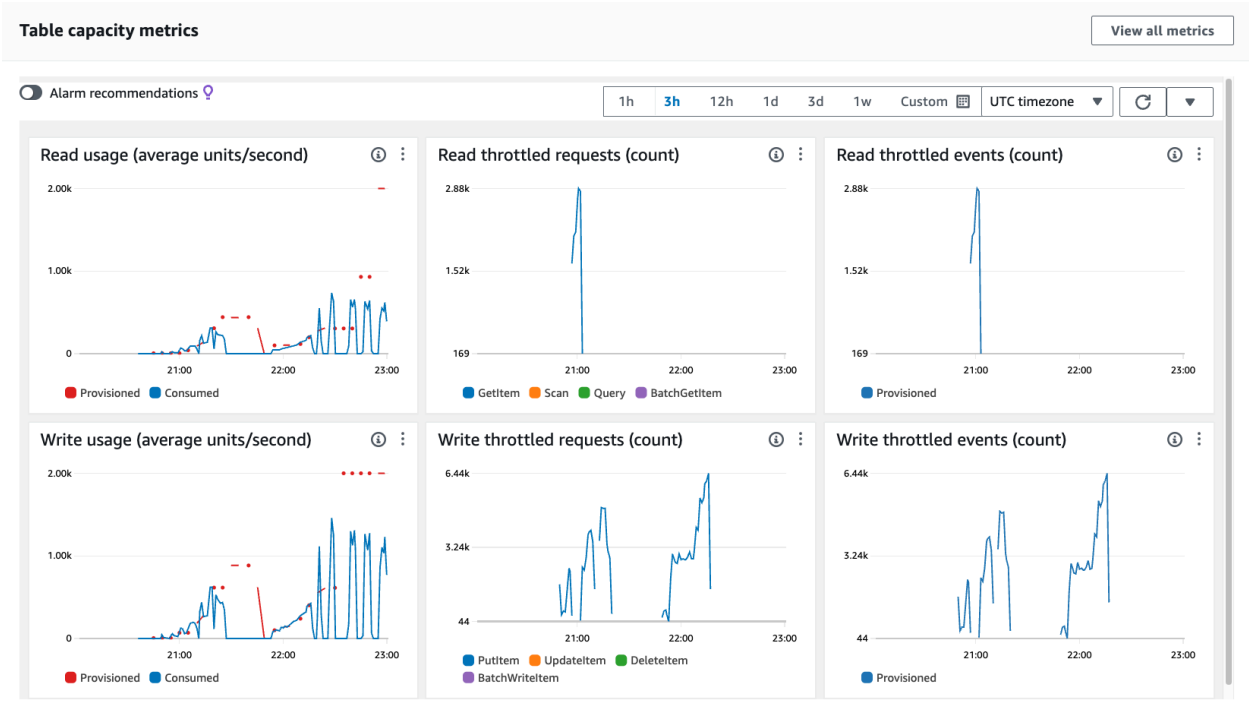

Figure 18: DynamoDB Basic Monitoring Graphs

As we can observe, before manually setting the minimum read/write capacity to a high enough value (i.e. 2000), auto-scaling took a number of minutes to effect one iteration of capacity increase. The capacity increases are also incremental in magnitude, which requires multiple iterations to reach a suitable capacity. As a result of this relatively slow increase in capacity, DynamoDB's consumed iread/write capacity is frequently limited by the provisioned capacity. We also observed a significant amount of throttled read and write requests.

In task 5 (at around time 23:00 in the graphs), I manually set the minimum read/write capacity to 2000, which is sufficiently large for our load tests. Read/write request throttling immediately dropped to 0. We therefore observed a consistent throughput above 2000 in our load tests.