CS6650 Final Project Report

Yihan Tian
Yifei Wang

Introduction to our works:

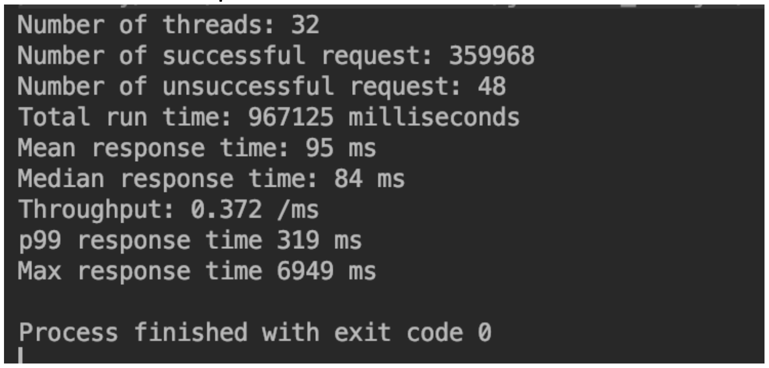
We rewrote our server code to Node.js and uploaded it to AWS Lambda. We used API Gateway to handle Get and Post requests from the client. And we change our database from SQL to AWS DynamoDB.

URL to A4 Server: https://github.ccs.neu.edu/yihtian/CS6650/tree/master/Assignment4

Our design:

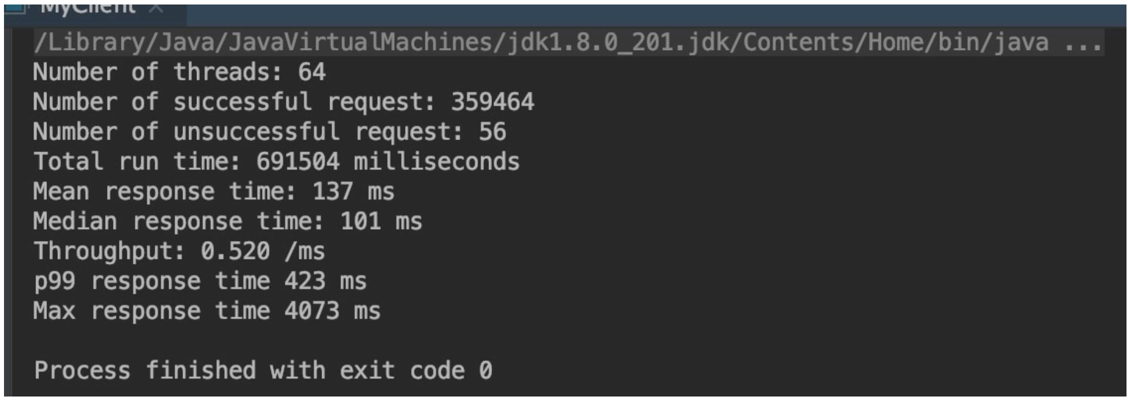
Our client part remains the same. The http requests are directed to AWS API Gateway, in which we allocated resources and created corresponding methods to handle required URLs. We used a {proxy+} function of API Gateway to process all endpoints with URL path start with skiers/ using our Lambda functions.
To use AWS Lambda, we created a TypeScript project, which could be compiled to Node.js and directly used as Lambda function. Apart from the function itself, we add API Gateway and DynamoDB as triggers so that out Lambda function can be called and write data to the database accordingly. In our server-side code, what we have done follows similar ideas as before. First, we parse the URL path and JSON body to get the correct data. And then we use those derived data to connect to the database and make read and write operations. If the operation is successfully executed, the server will return a 200-status code, otherwise return a 400-status code.
The most challenging part of our design is the database part. Since DynamoDB is No-SQL database, it stores data as key-value format, it is only possible to make query with key data, otherwise it will be extremely time-consuming. Therefore, we need to redesign our database structure. Because DynamoDB allows at most two indices to comprise the primary key, and we need to make queries using SkierId and DayId, we decide to use those two attributes as the compound primary key. According to the documentation of DynamoDB, among the two indices, one is partition key which can have duplications, and the other is sort key which must be unique. We choose SkierId as the partition key, and we add a random 6-digit decimal number to DayId to make it unique. When making queries, we use SkierId and the integer part of DayId. Thus, we guarantee the efficiency of making queries as well as conforming to DynamoDB schemas.

Results and comparisons:



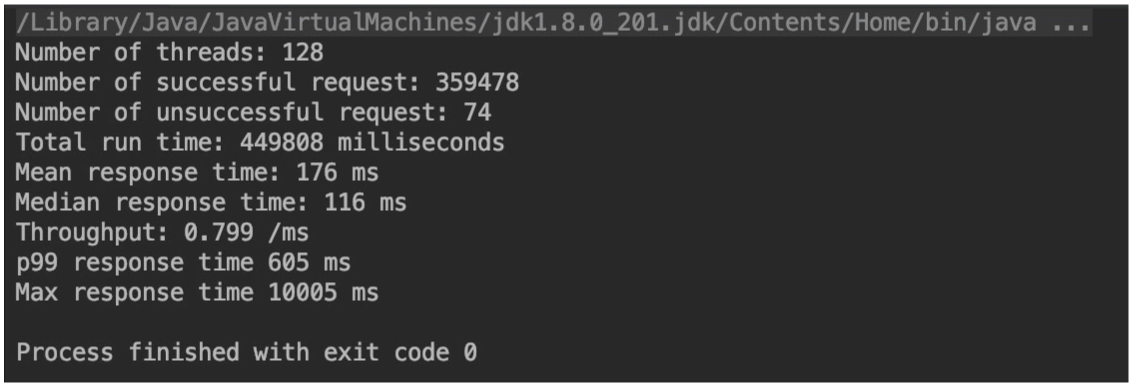
```
Number of threads: 32
Number of successful request: 359968
Number of unsuccessful request: 48
Total run time: 967125 milliseconds
Mean response time: 95 ms
Median response time: 84 ms
Throughput: 0.372 /ms
p99 response time 319 ms
Max response time 6949 ms

Process finished with exit code 0
```



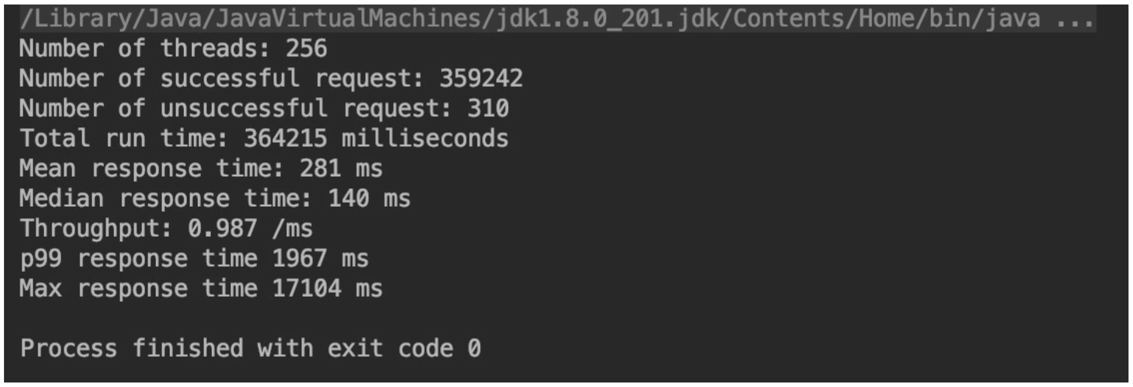
```
/Library/Java/JavaVirtualMachines/jdk1.8.0_201.jdk/Contents/Home/bin/java ...
Number of threads: 64
Number of successful request: 359464
Number of unsuccessful request: 56
Total run time: 691504 milliseconds
Mean response time: 137 ms
Median response time: 101 ms
Throughput: 0.520 /ms
p99 response time 423 ms
Max response time 4073 ms

Process finished with exit code 0
```



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_201.jdk/Contents/Home/bin/java ...
Number of threads: 128
Number of successful request: 359478
Number of unsuccessful request: 74
Total run time: 449808 milliseconds
Mean response time: 176 ms
Median response time: 116 ms
Throughput: 0.799 /ms
p99 response time 605 ms
Max response time 10005 ms

Process finished with exit code 0
```

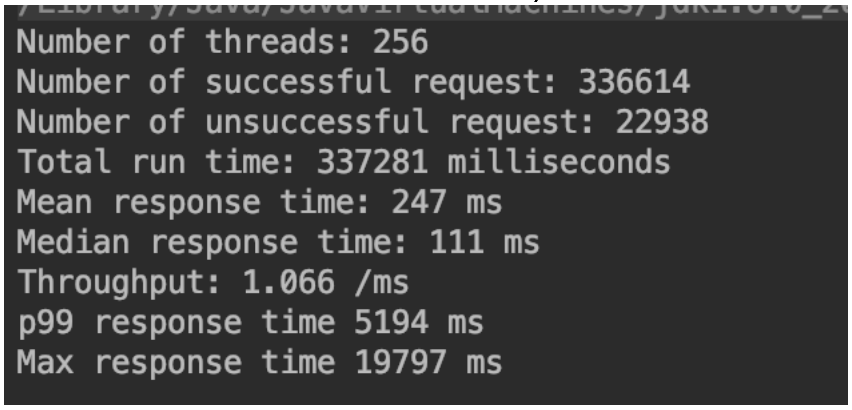


```
/Library/Java/JavaVirtualMachines/jdk1.8.0_201.jdk/Contents/Home/bin/java ...
Number of threads: 256
Number of successful request: 359242
Number of unsuccessful request: 310
Total run time: 364215 milliseconds
Mean response time: 281 ms
Median response time: 140 ms
Throughput: 0.987 /ms
p99 response time 1967 ms
Max response time 17104 ms

Process finished with exit code 0
```

Above are the results using AWS Lambda and DynamoDB. We can see that the performance is relatively good for 256, 128 and 64 threads, but 32 thread is not performing well. The mean and median response times are all in a reasonable range.

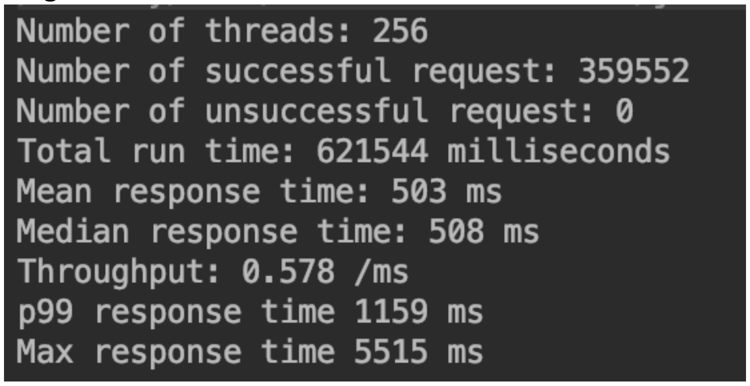
We focus on 256 threads results for analysis. Below is the result of using GAE and Cloud SQL.



```
Number of threads: 256
Number of successful request: 336614
Number of unsuccessful request: 22938
Total run time: 337281 milliseconds
Mean response time: 247 ms
Median response time: 111 ms
Throughput: 1.066 /ms
p99 response time 5194 ms
Max response time 19797 ms
```

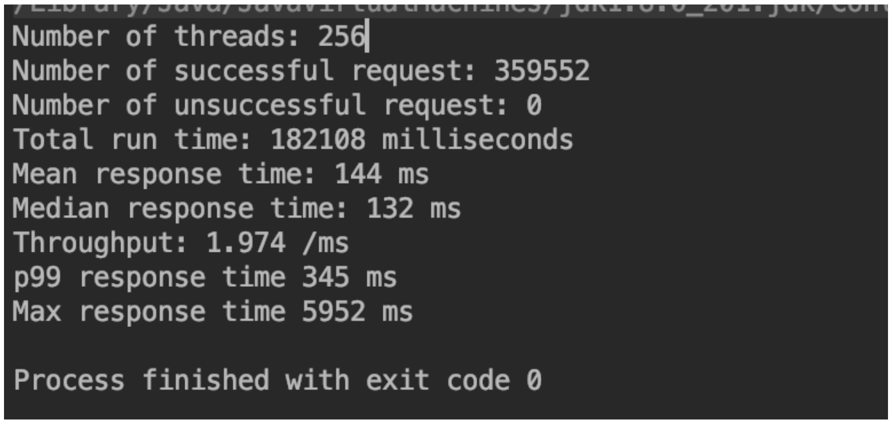
We can see that the total wall time and mean time is slightly faster. But it is less reliable.

Below is the result of using AWS EC2 and RDS.
Single server:



```
Number of threads: 256
Number of successful request: 359552
Number of unsuccessful request: 0
Total run time: 621544 milliseconds
Mean response time: 503 ms
Median response time: 508 ms
Throughput: 0.578 /ms
p99 response time 1159 ms
Max response time 5515 ms
```

Load balanced server:



```
Number of threads: 256
Number of successful request: 359552
Number of unsuccessful request: 0
Total run time: 182108 milliseconds
Mean response time: 144 ms
Median response time: 132 ms
Throughput: 1.974 /ms
p99 response time 345 ms
Max response time 5952 ms

Process finished with exit code 0
```

We can see that using Lambda can beat EC2 single server, but the performance is not so well as using load balanced servers in both time and reliability.

Based on our analysis, the using Lambda exceeds the performance of EC2 because of its serverless characteristics. All requests could be handled on demand in an auto-scaling way so that the configuration of server is not a restriction anymore. Comparing choosing specific EC2 machine configuration, it saves cost and resources but also achieves better performance. It is convenient to use API Gateway to handle all requests instead of creating servlets for each URL endpoint. Using NoSQL databases like DynamoDB could also getting rid of DAOs and models. The server code is now in a clearer structure and is easy to debug.

There are some drawbacks using Lambda. Although there is auto-scaling, it is not easy to make further optimizations in Lambda such as load balancing. There is a max limit of concurrent executions in Lambda, and it's hard to exceed that limit by making changes to server. Also, there exists cold start in Lambda, which generates extra latency when start up the service. Besides, since it is stateless, all dependent packages need to be uploaded in a zip file and installed each time, which is less effective than installing packages in EC2 and GAE machine.

To sum up, the performance using Lambda is better than using single server EC2 and is similar as using GAE based on wall time. It performs better than GAE based on reliability. Building and deploying server in Lambda is different than previous experience and takes some time to learn, but it is not very complex.

What we learned:

In this assignment, we learned how to use AWS Lambda and make it connect to API Gateway and DynamoDB. We have gained some valuable experience in getting proficient with new things in a short time. For example, both of us are not familiar with TypeScript and Node.js, but based on the docs and online resources, we managed to finish our project using Node.js. We also learned the implementation of serverless structures. It is exciting to see that a bunch of server code can be transformed into a few numbers of lines using serverless services and achieving satisfying performance.