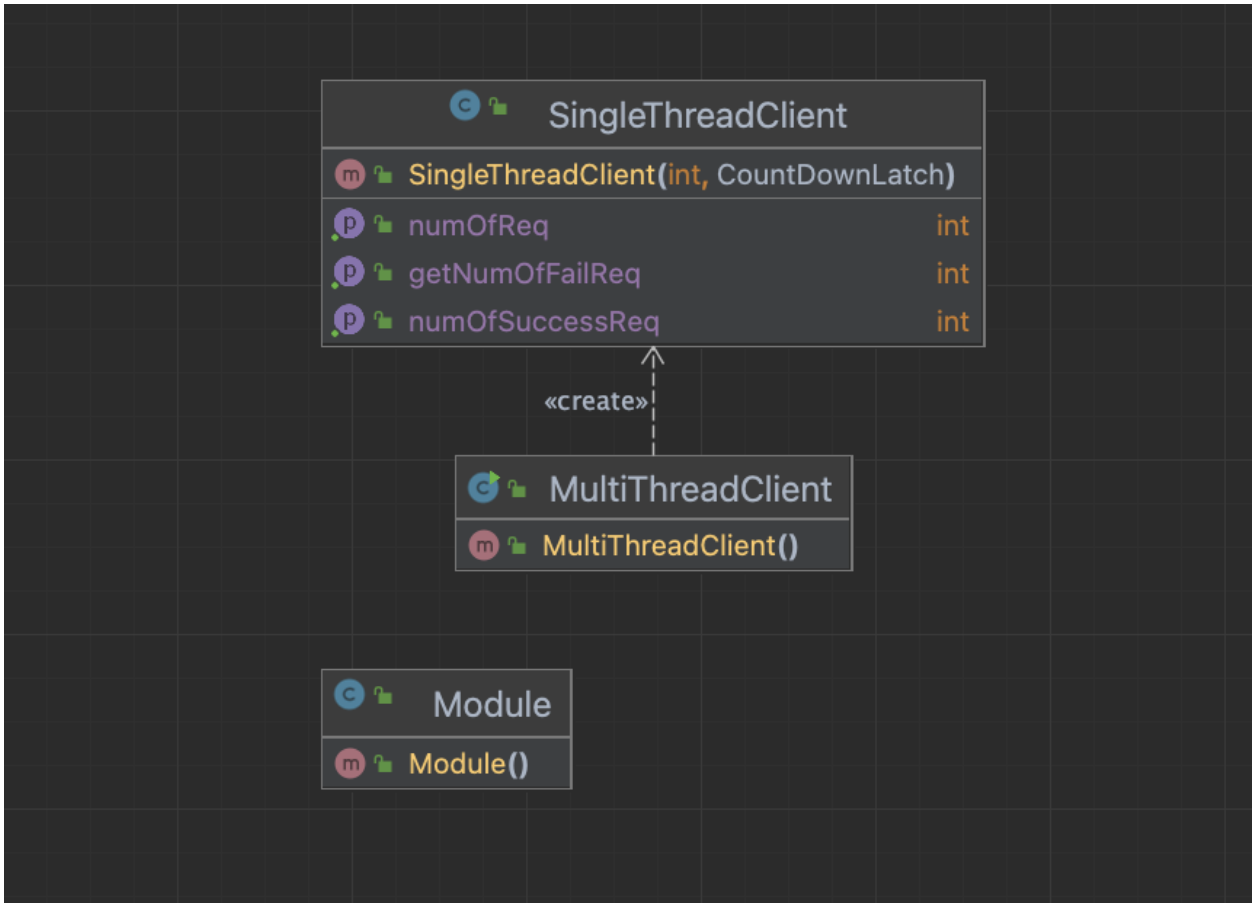
Submission details:

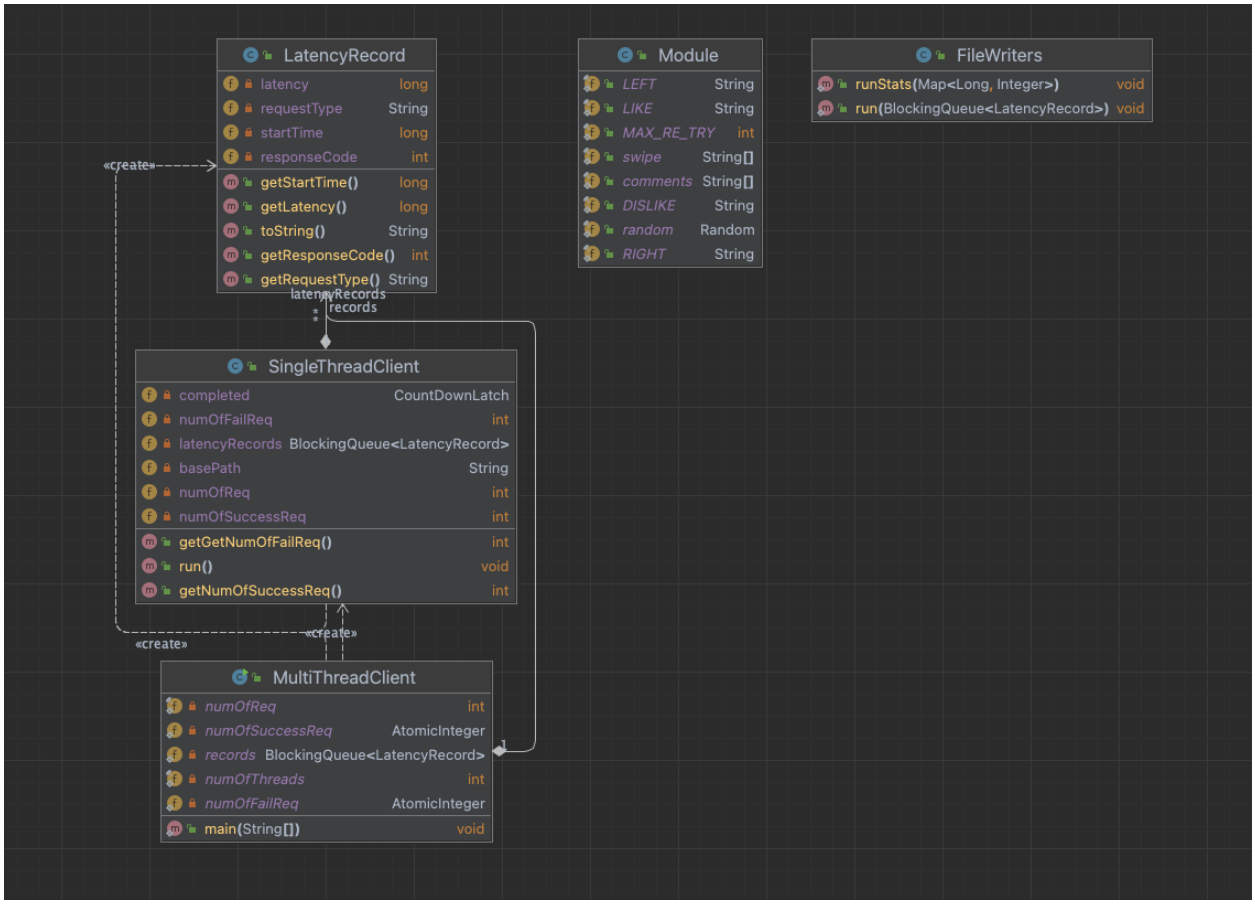
1. Assignment-1 repo URL:
   https://github.com/Oliver1024/DistributedSystemDatingApp/tree/main/assignment1

   Springboot URL:
   https://github.com/Oliver1024/DistributedSystemDatingApp/tree/main/assignment1SpringBoot2

2. Client design description:
   Part -1 uml:



   Part -2 uml:



The client part which I have is implemented multi-threading client-server architecture to simulate the client sending requests to the server. This client part2 contains 5 Java classes: FileWriters, LatencyRecord, Module, and MultiThreadClient and SingleThreadClient. I also have the same logic as in client part1.

SingleThreadClient class acts as a single client, which is executed by multiple threads. Each thread executes the run method of SingleThreadClient class and sends the number of requests specified in the numOfReq variable and the run method of the SingleThreadClient class sends the request to the server, records the latency of each request, and calculates the success and failure counts of the requests. This class creates an instance of a SwipeApi client using the Swagger API client library and sends a specified number of "swipe" requests to the API at the specified base URL. Each swipe request is represented by a "SwipeDetails" object, which includes the swiper ID, swipee ID, and a comment, all of which are randomly generated. If a swipe request fails, the code retries it a specified number of times before moving on to the next request.

FileWriters is a class that writes data to two separate CSV files. The first file records latency information, including "StartTime", "RequestType", "Latency", and "ResponseCode". The second file records performance statistics, including "Seconds" and "Throughput/second".

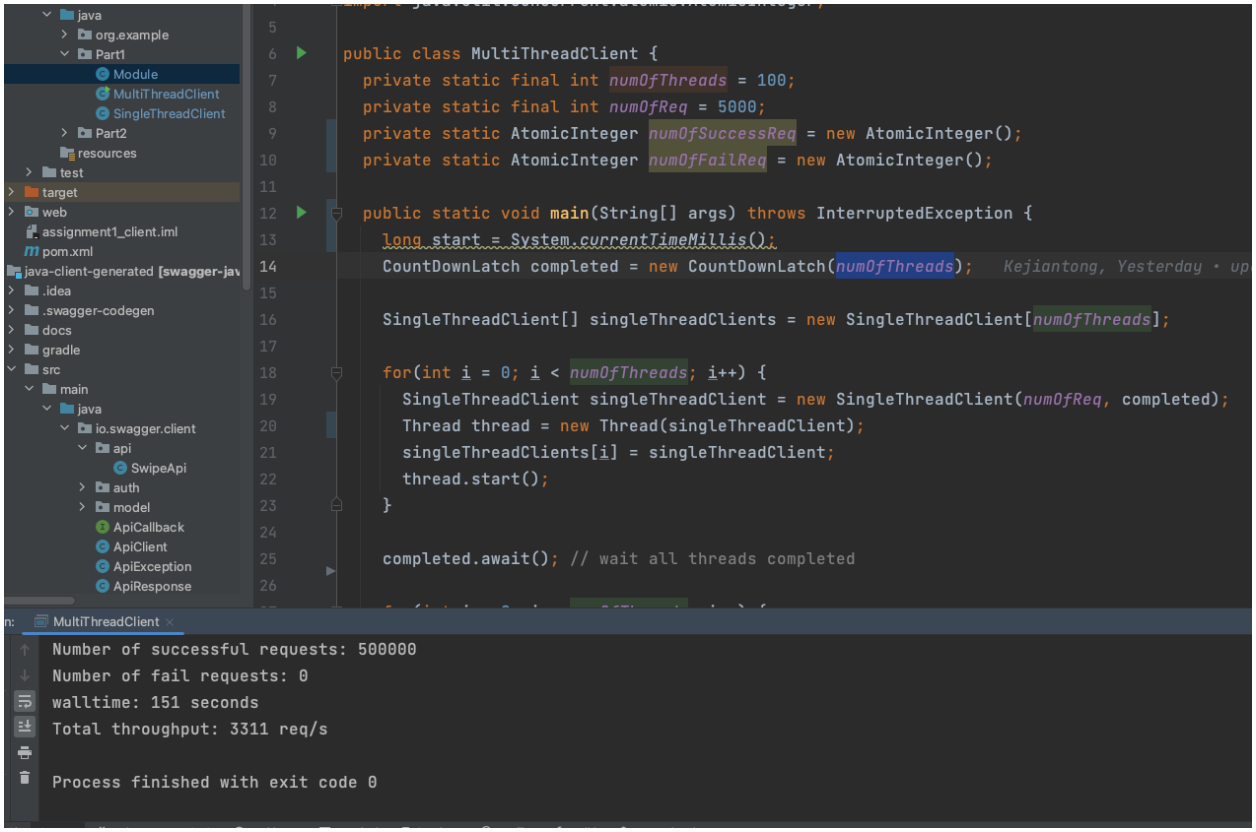
LatencyRecord is a simple data class that holds information about a latency record, including the start time, request type, latency, and response code.

Module is a class with several constants and a random number generator. The constants define swipe directions (LEFT, RIGHT), comments (LIKE, DISLIKE), and the maximum number of retries (MAX_RE_TRY).

MultiThreadClient is the main class that creates and runs multiple SingleThreadClient threads. Each SingleThreadClient makes multiple requests and records latency information and performance statistics. The MultiThreadClient class waits for all SingleThreadClient threads to complete and calculates the overall success rate of the requests and the wall-clock time. The final results are stored in two CSV files by the FileWriters class.

3. Client - Part 1:

Configure for client:
Number of threads: 100
Number of request per thread: 5000
Total requests: 500k
Thorughput: 3311 req/s



First, I send 10000 requests using 1 thread, and get the response time(w) = 18.6ms
Then, apply to little's law,
Expected Little's Law throughput = 100/18.6 = 5376 req/s

If I used a consumer-producer pattern, it could help me get better results and be closed to the little's laws' expected results. The producer generates threads firs and then adds them to a blocking queue, and the consumer retrieves threads from the blocking queue and processes them.

4. Client - Part2: 100 threads

```
Total throughput: 3333 req/s
Mean response time: 29 ms
Median response time: 28 ms
99th percentile response time: 76 ms
Max response time: 1882 ms
Min response time: 12 ms

Process finished with exit code 0
```

I have conducted numerous experiments to send 500,000 requests, utilizing various configurations such as 10, 20, 30, 50, 80, 100, 200, 250, and 300 threads. Please refer to the following detailed comparison for more information.

numOfThreads: 10
Total throughput: 605 req/s

```
Total throughput: 605 req/s
Mean response time: 16 ms
Median response time: 15 ms
99th percentile response time: 29 ms
Max response time: 1226 ms
Min response time: 11 ms

Process finished with exit code 0
```

numOfThreads: 20
Total throughput: 1196 req/s

```
Total throughput: 1196 req/s
Mean response time: 16 ms
Median response time: 16 ms
99th percentile response time: 31 ms
Max response time: 672 ms
Min response time: 11 ms

Process finished with exit code 0
```

numOfThreads: 30
Total throughput: 1724 req/s

```
Total throughput: 1724 req/s
Mean response time: 17 ms
Median response time: 16 ms
99th percentile response time: 35 ms
Max response time: 839 ms
Min response time: 11 ms

Process finished with exit code 0
```

numOfThreads: 50
Total throughput: 2538 req/s

```
Total throughput: 2538 req/s
Mean response time: 19 ms
Median response time: 18 ms
99th percentile response time: 46 ms
Max response time: 1309 ms
Min response time: 12 ms

Process finished with exit code 0
```

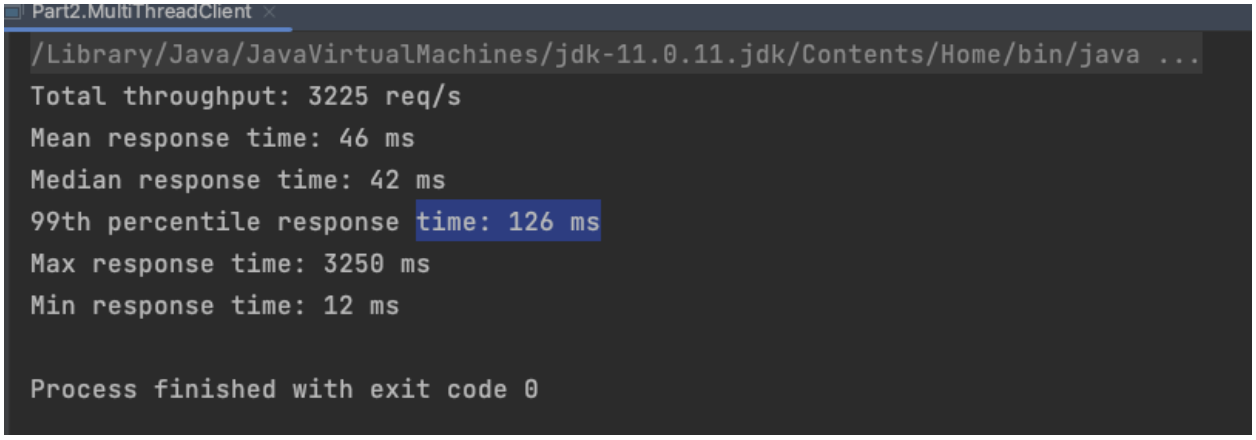
numOfThreads: 80
Total throughput: 3184 req/s

```
Total throughput: 3184 req/s
Mean response time: 24 ms
Median response time: 23 ms
99th percentile response time: 70 ms
Max response time: 1401 ms
Min response time: 12 ms

Process finished with exit code 0
```

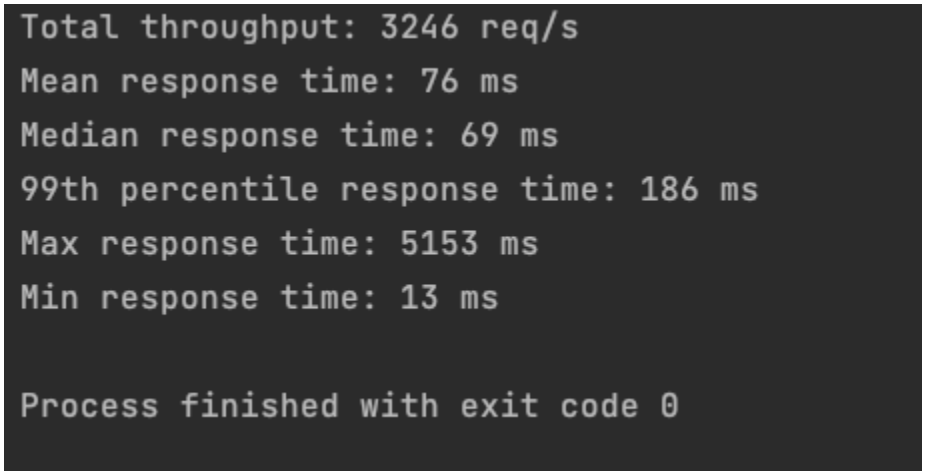
numOfThreads: 100
Total throughput: 3333 req/s

```
Total throughput: 3333 req/s
Mean response time: 29 ms
Median response time: 28 ms
99th percentile response time: 76 ms
Max response time: 1882 ms
Min response time: 12 ms

Process finished with exit code 0
```

numOfThreads: 200
Total throughput: 3225 req/s



numOfThreads: 250
Total throughput: 3246 req/s

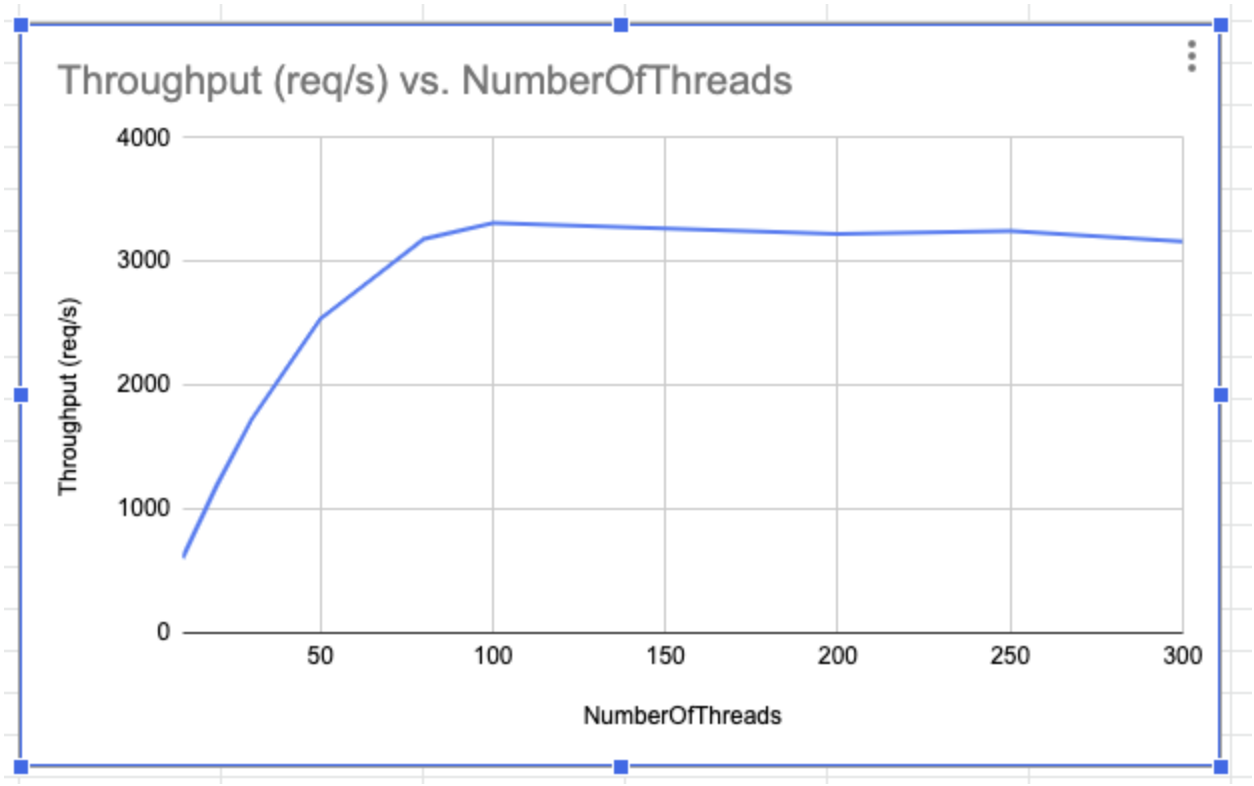


numOfThreads: 300
Total throughput: 3164 req/s

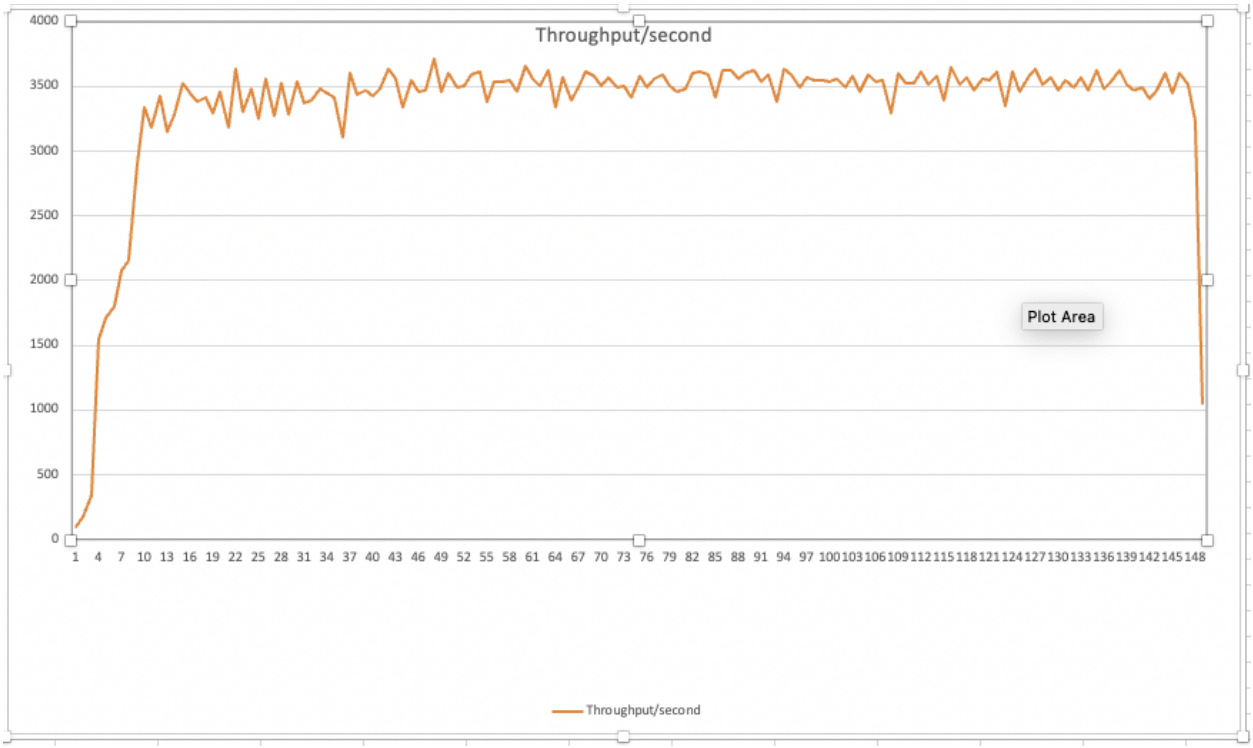
```
Total throughput: 3164 req/s
Mean response time: 92 ms
Median response time: 83 ms
99th percentile response time: 235 ms
Max response time: 5288 ms
Min response time: 12 ms

Process finished with exit code 0
```

Finally, I obtained intriguing statistics regarding the relationship between the number of threads and throughput. Surprisingly, the throughput significantly increased when I increased the number of threads from 10 to 100. However, despite the default maximum number of threads for Tomcat being set at 200, the best throughput was achieved with 100 threads for my configuration and client. When I tried to increase the number of threads from 100 to 300, the throughput did not increase and instead, it even decreased slightly.

Additionally, other factors such as AWS networking and time of day may also impact the throughput. For example, the results may vary if the test is conducted in the early morning or late night.


Throughput (req/s) vs. NumberOfThreads
4000
3000
2000
1000
0
Throughput (req/s)
50
100
150
200
250
300
NumberOfThreads

Task 4:



Spring Boot part:

numOfThreads: 10
Total throughput: 615 req/s

```
Total throughput: 615 req/s
Mean response time: 16 ms
Median response time: 15 ms
99th percentile response time: 29 ms
Max response time: 1389 ms
Min response time: 11 ms

Process finished with exit code 0
```

numOfThreads: 20
Total throughput: 1207req/s

```
Total throughput: 1207 req/s
Mean response time: 16 ms
Median response time: 16 ms
99th percentile response time: 31 ms
Max response time: 663 ms
Min response time: 11 ms

Process finished with exit code 0
```

numOfThreads: 30
Total throughput: 1730 req/s

```
Total throughput: 1730 req/s
Mean response time: 17 ms
Median response time: 16 ms
99th percentile response time: 35 ms
Max response time: 1115 ms
Min response time: 11 ms

Process finished with exit code 0
```

numOfThreads: 50
Total throughput: 2512 req/s

```
Total throughput: 2512 req/s
Mean response time: 19 ms
Median response time: 18 ms
99th percentile response time: 51 ms
Max response time: 1229 ms
Min response time: 11 ms

Process finished with exit code 0
```

numOfThreads: 80
Total throughput: 3267 req/s

```
Total throughput: 3267 req/s
Mean response time: 24 ms
Median response time: 23 ms
99th percentile response time: 65 ms
Max response time: 1755 ms
Min response time: 12 ms

Process finished with exit code 0
```

numOfThreads: 100
Total throughput: 3311 req/s

```
Total throughput: 3311 req/s
Mean response time: 30 ms
Median response time: 28 ms
99th percentile response time: 75 ms
Max response time: 2208 ms
Min response time: 12 ms

Process finished with exit code 0
```

numOfThreads: 150
Total throughput: 3355 req/s

```
Total throughput: 3355 req/s
Mean response time: 44 ms
Median response time: 41 ms
99th percentile response time: 108 ms
Max response time: 2999 ms
Min response time: 12 ms

Process finished with exit code 0
```

numOfThreads: 200
Total throughput: 3267 req/s

```
Total throughput: 3267 req/s
Mean response time: 60 ms
Median response time: 55 ms
99th percentile response time: 163 ms
Max response time: 3529 ms
Min response time: 12 ms

Process finished with exit code 0
```

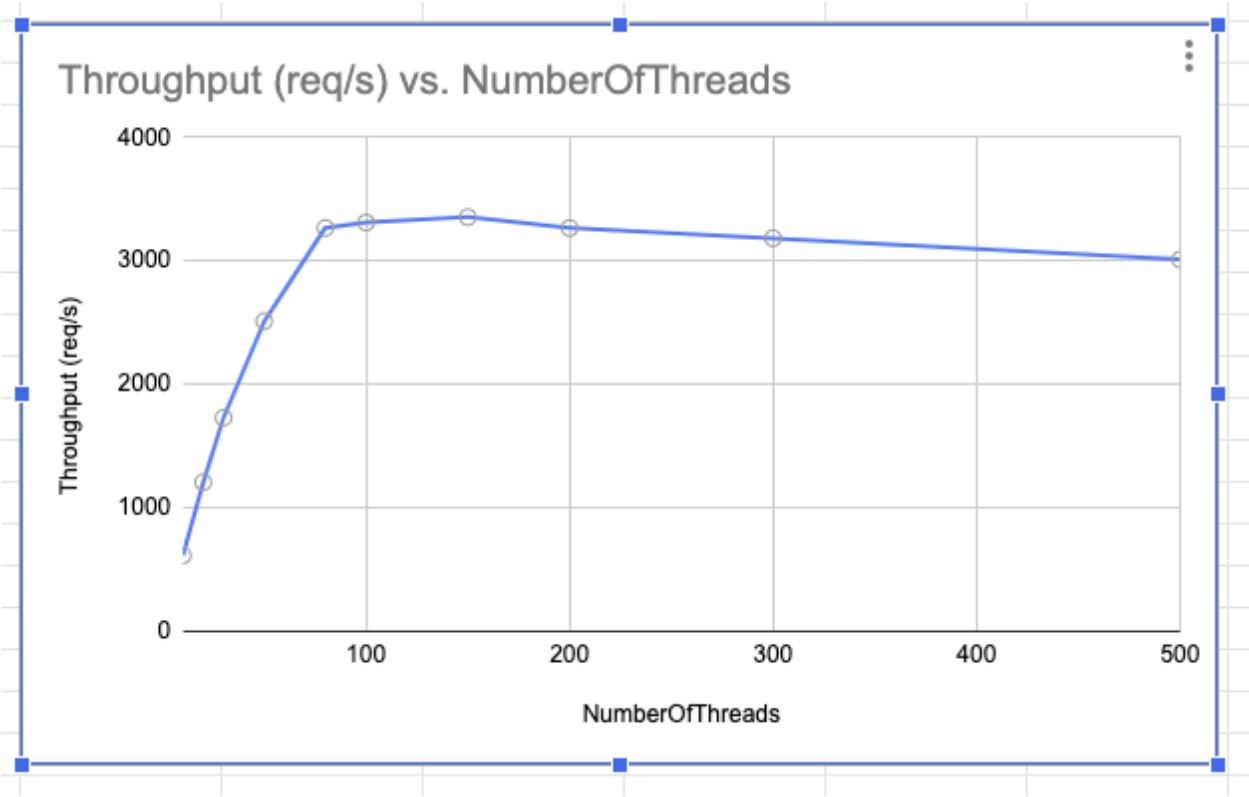
numOfThreads: 300
Total throughput: 3184 req/s

```
Total throughput: 3184 req/s
Mean response time: 91 ms
Median response time: 82 ms
99th percentile response time: 225 ms
Max response time: 4291 ms
Min response time: 12 ms

Process finished with exit code 0
```

numOfThreads: 500
Total throughput: 3012 req/s

```
Total throughput: 3012 req/s
Mean response time: 161 ms
Median response time: 107 ms
99th percentile response time: 817 ms
Max response time: 9311 ms
Min response time: 12 ms

Process finished with exit code 0
```

Compared to my previous server, Spring Boot also demonstrates a similar trend where an increase in the number of threads results in a corresponding increase in throughput. However, once the number of threads surpasses 100, the trend begins to resemble that of my previous server, with the highest throughput being similar to the best throughput previously recorded.


Throughput (req/s) vs. NumberOfThreads
4000
3000
2000
1000
0
Throughput (req/s)
100
200
300
400
500
NumberOfThreads