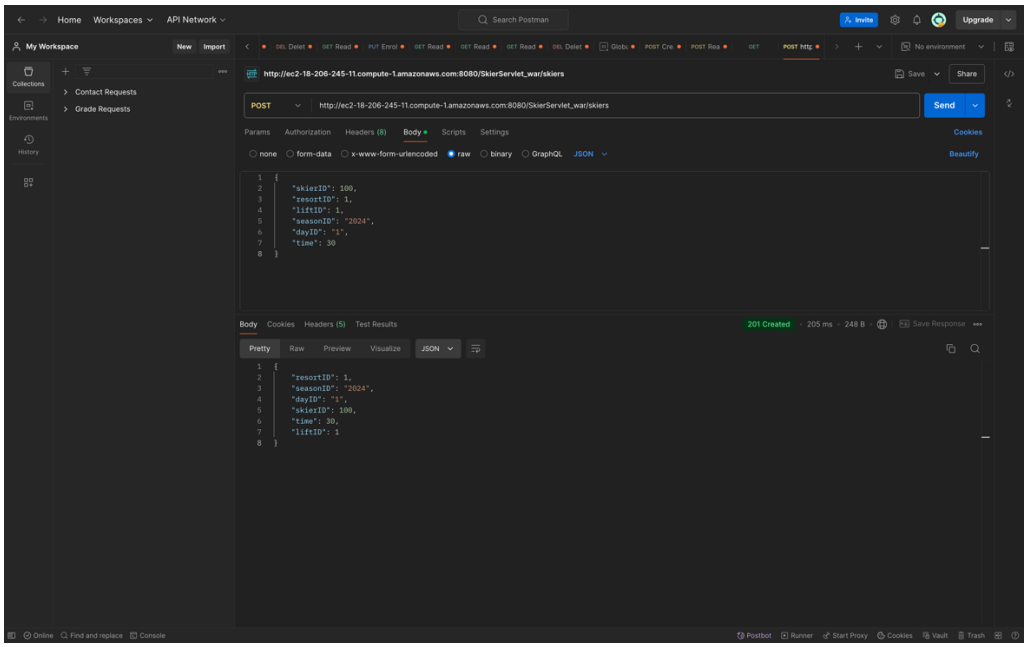
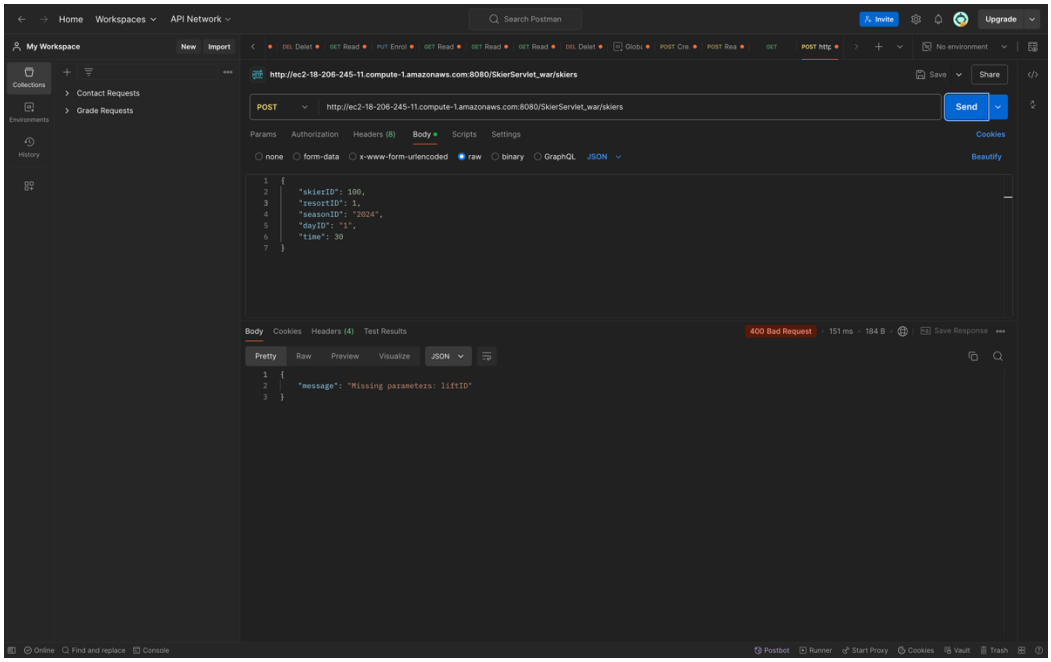
Assignment 1 report – Xujia Qin
Git repo: https://github.com/xq443/distributed-system/tree/main/Assignment/1

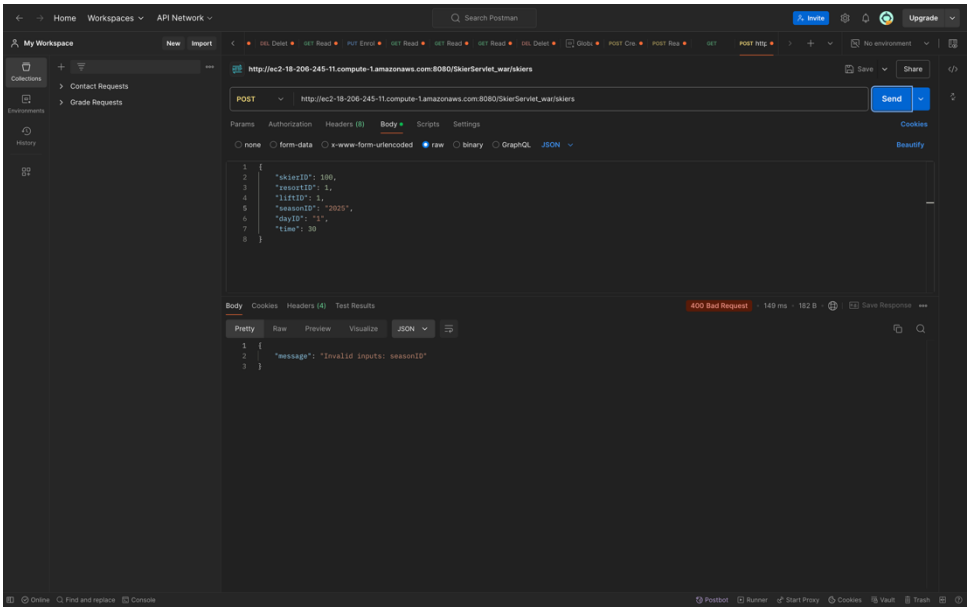
1. Server implementation working (10 points)
   test case1: 201 Created



POSTMAN test case2: missing parameters 400 Bad Request



POSTMAN test case3: invalid input 400 Bad Request



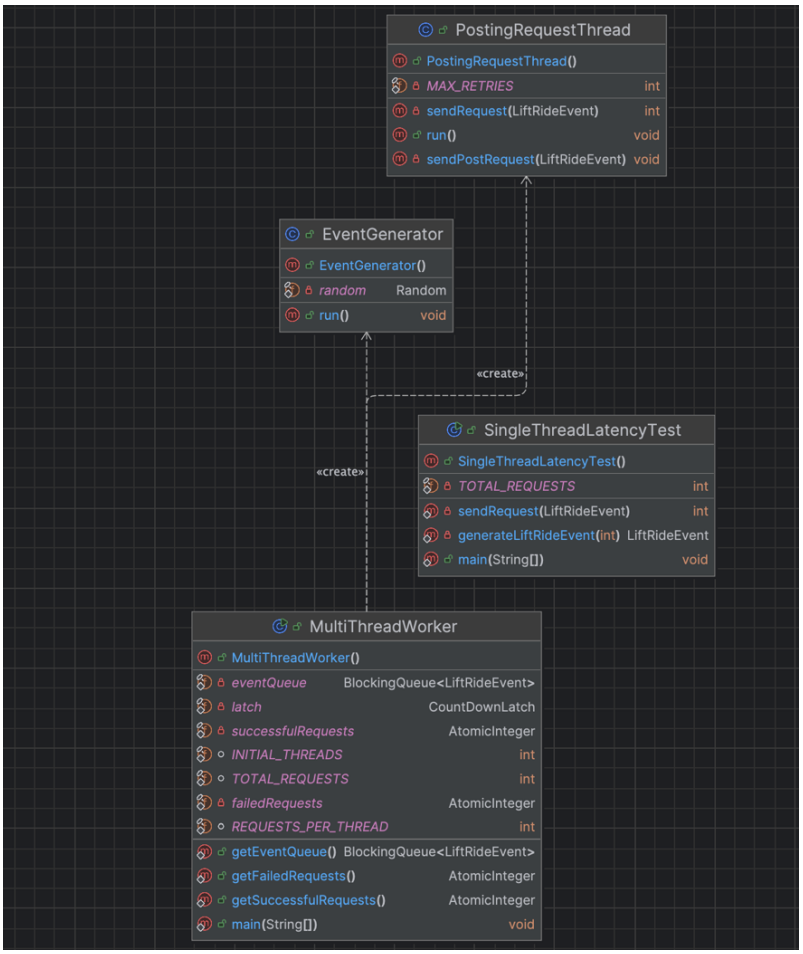
2. Client design description (5 points) - clarity of description, good design practices used

    a. EventGenerator.java: Generates a specified number of LiftRide events and adds them to a BlockingQueue.
        1. Implements the Runnable interface for concurrent task execution.
        2. Simulates diverse lift rides by populating LiftRide objects with random.

    b. PostingThread.java: Retrieves LiftRide events from the BlockingQueue and posts them to the server via the SkiersApi.
        1. Implements the Runnable interface for concurrent task execution.
        2. Manages request retries for failures and collects latency metrics for performance analysis.

    c. MultiThreadLiftRideClient.java: Initializes the event generation process and coordinates the multi-threaded POST operations.
        1. Configures the ThreadPoolExecutor to manage concurrent threads.
        2. Waits for all tasks to finish and outputs the results.

    d. UML Diagram:



e. **Multi-threaded Execution with ThreadPoolExecutor:** The client employs ThreadPoolExecutor to manage concurrent HTTP requests efficiently. Each thread within the pool runs an instance of the PostingThread class, functioning as an individual client (SingleClient) that handles specific HTTP tasks. This design allows for scalable, parallel execution, optimizing throughput while managing system resources.

f. **CountDownLatch for Synchronization:** A CountDownLatch is used to coordinate the execution of multiple threads, ensuring that the main program flow does not proceed until all threads have completed their tasks. This synchronization mechanism helps ensure precise performance measurements by preventing premature termination or incomplete aggregation of results.

g. **Thread-safe Request Tracking with AtomicInteger:** To track the number of successful and failed requests across threads, the client utilizes AtomicInteger, because this thread-safe class allows shared counters to be updated concurrently without introducing race conditions, ensuring that the metrics collected reflect the actual state of operations.

h. **Task Queue with BlockingQueue:** A BlockingQueue is used to store LiftRide objects, encapsulating detailed information about each HTTP request. This thread-safe queue enables threads to safely add tasks, facilitating thorough performance analysis after execution. The non-blocking nature of the queue ensures smooth data handling across multiple threads.

3. Client Part 1 - (10 points) - Output window showing best throughput. Points deducted if actual throughput not close to Little's Law predictions.

   a. Single thread with 10k POST requests

```
Total time for 10000 requests: 257595 ms
Average latency per request: 25.76 ms
Number of successful requests: 10000
Number of unsuccessful requests: 0
Total throughput: 38.82 requests/second
```

   Avg Response Time (W) = 0.02576 s / req

   b. 200k POST requests with 32 thread pool size **(queue size = total req)**

```
Successful requests: 200000
Failed requests: 0
Total time: 152616 ms
Throughput: 1310.4785867798919 requests/second
```

   Assume the previous response time W is our baseline,
   By using little's law,
   The Expected throughput (λ) = N / W = 32 / 0. 02576 s / req
   = 1242.24 req / s

Which is pretty like the throughput in this case.

c. 200k POST requests with 80 thread pool size (queue size = 3 * total req)

```
Successful requests: 200000
Failed requests: 0
Total time: 47391 ms
Throughput: 4220.2105885083665 requests/second
```

If we increase the thread pool size to 80 and blocking queue (event generator) capacity to 3 times, the throughput will also increase by around three times.

d. 200k POST requests with 100 thread pool size (queue size = 3 * total req)

```
Successful requests: 200000
Failed requests: 0
Total time: 42388 ms
Throughput: 4718.316504671134 requests/second
```

e. 200k POST requests with 190 thread pool size (queue size = 4 * total req)

```
Successful requests: 200000
Failed requests: 0
Total time: 31090 ms
Throughput: 6432.9366355741395 requests/second
```

Peak.

f. 200k POST requests with 200 thread pool size (queue size = 4 * total req)

```
Successful requests: 200000
Failed requests: 0
Total time: 31392 ms
Throughput: 6371.049949031601 requests/second
```

Increasing the capacity of the thread pool size here does not lead to continuous increase in throughput.

g. Summary:

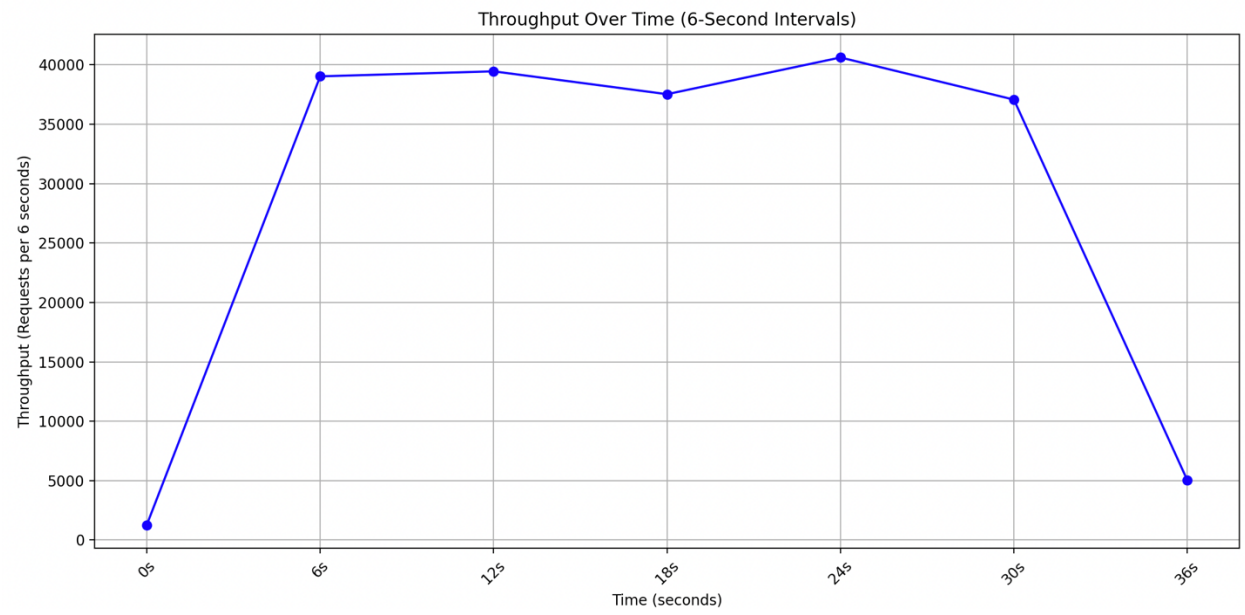
With deploying .war file to the EC2 server, the test result turns out that sending total 200k POST requests with 190 thread pool size (queue size = 4 * total req) is the best combination for the highest empirical throughput (**6432 requests/sec**)

4. Client Part 2 - (10 points) - 5 points for throughput within 5% of Client Part 1.
   5 points for calculations of mean/median/p99/max/throughput

```
Successful requests: 200000
Failed requests: 0
Total time: 31633 ms
Throughput: 6322.511301488951 requests/second
Mean response time: 30 ms
Median response time: 28 ms
p99 response time: 62 ms
Min response time: 10 ms
Max response time: 510 ms
```

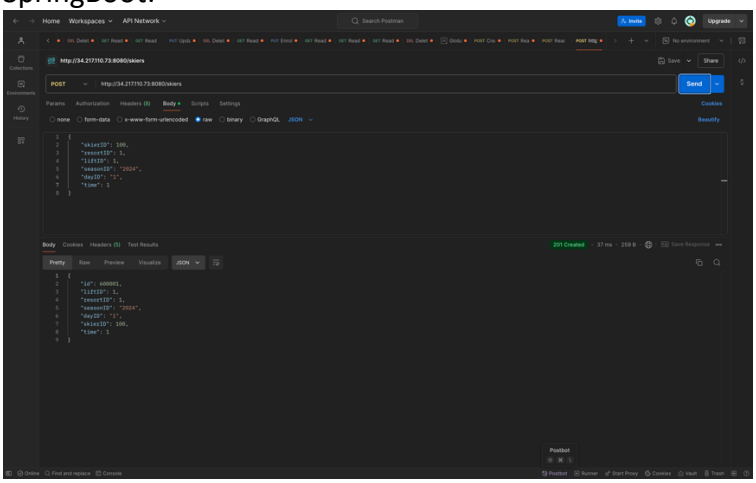
6432 – 6322 / 6432 = 1.7% throughput decrease

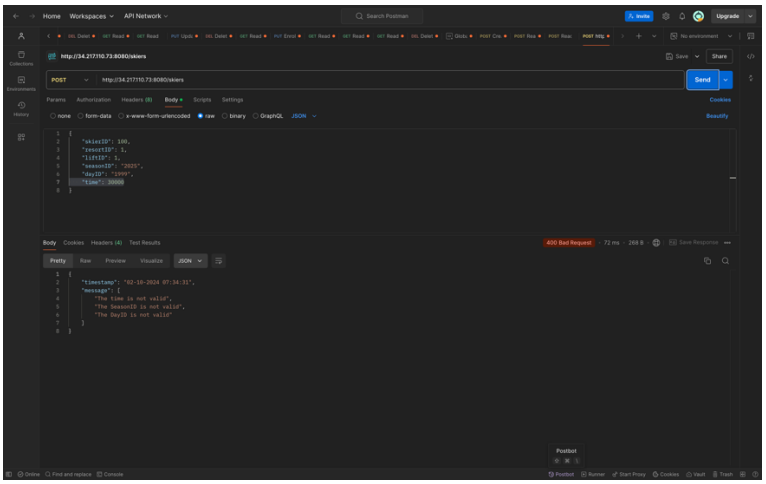
5. Plot of throughput over time (5 points)

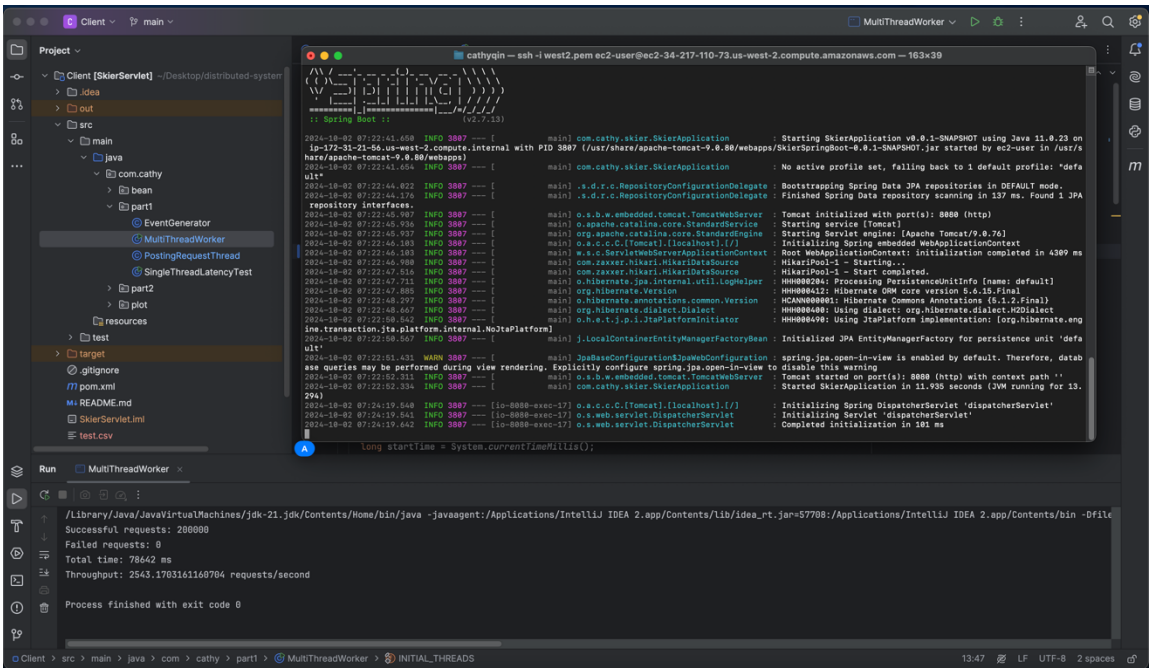


Since the total execution time is less than 36000 ms (36s), the chart utilizes 6s as time intervals.

6. SpringBoot:







Observation:
The throughput is around 2543 req/s, which is pretty much slower than servlet.
This is because Spring Boot adds layers of abstraction that simplify development but may introduce overhead, like dependency injection, AOP, and a set of libraries, which can lead to slower performance in some cases. Additionally, Spring Boot applications generally have a longer startup time due to the auto-configuration and bean initialization processes. This can be an issue in environments where quick response times are critical.