Assignment 3 Write Up

CS 6650, Spring 2022
Sean Stevens

Github Repository URL

https://github.com/seanmstevens/CS6650-Assignment3

Database Design

I went through a few different iterations of my database design and data model based on what I felt would be the best schema to leverage Redis' strengths.

My initial design was based on having composite keys with hashmap values. The key structure I intended to use was: skierID:seasonID:day:time. My assumption when implementing this schema was that a particular skier could only ride one lift at any given instant. The hashmap value would then contain the lift ID, the wait time, and the skier's vertical for that ride. The difficulty I found with this design came up when thinking about querying the data. To get aggregate statistics for a particular skier, I would need to get multiple values based on wildcard pattern matching on the keys. This isn't really supported well in the base Redis package. The alternative was to send multiple queries (either serialized or concurrent), which could result in prolonged blocking of the main Redis thread. Although the data model is efficient, the tradeoff is that querying is an expensive operation without the use of other Redis modules.

The next design I came up with was to use a combination of composite keys and varying data structures to make specific queries and updates very fast and efficient. Keys would use the following format: [nameOfQuery]:skierID:[otherData]. The name of the query is used here to avoid key collision, as the other data in the key may overlap even though the values are meant to represent different things. The other data is specific to the query, where this may be a day number if we're storing data related to the vertical totals for each ski day. While this approach is efficient at retrieving data for these specific queries, it isn't very scalable. Adding support for additional queries requires implementing complicated business logic to control how data is written to the Redis instance. Additionally, this requires the querying code to have specialized knowledge of the key syntax for each query.

For the above reasons, I ultimately decided on a much simpler approach that is closer to how the data was stored in a local hashmap in the previous assignment. Instead of using composite keys, I chose to use simple keys (for the skier microservice this is the skier ID, for the resorts service this is a composite key joining the season ID and day), with the values being a set of strings. Importantly, the string elements in the set are serialized JSON strings. A benefit of this design is that its querying code does not require knowledge of how keys are formed other than knowing the key is the skier ID. Queries can be *ad hoc*, and thus this design is more scalable than the other alternatives. Although queries will not be as efficient as the highly-indexed design

listed above, the number of unique elements for each skier will likely not be large, and as such serializing the JSON objects will not be a terribly expensive operation. The tradeoff between slightly longer querying times and reduced code complexity for queries is worth it in my view. Finally, there is room for optimization with this design. Using the RedisJSON module, the set elements could be converted to a native JSON array, making queries far more efficient and scalable.

AWS Deployment Topology

For deployment of the server, I decided to use a network load balancer to distribute requests over my EC2 fleet. The fleet consists of three t2.micro instances and one t3.medium instance. Both consumers (the skier microservice instance and the resorts microservice instance) and the instance hosting the RabbitMQ server are all t2.micro instances.

RabbitMQ Run Results

Given that I deployed my servers on multiple instances behind a network load balancer and each consumer runs on its own independent instance, I did not experience any service faults due to back pressure from an overloaded component in my system. Rather, the throughput was right around where I expected it to be for the various runs and the publication and consumption rates were very stable.

Despite this, I did find that queue sizes got up into the low triple digits when running both microservices at the same time. To alleviate this, I decided to change the instance types of the consumers up to t3.mediums. Compared to t2.micro instances, the medium instance type has double the vCPUs and four times the memory available for consuming messages from the queue. The results of this change are listed in the final section of the following run statistics, and there is a discussion included in the conclusion to this write up.

As the throughput was stable on the server side, I did not see a reason to implement a circuit breaker on the client or server instances.

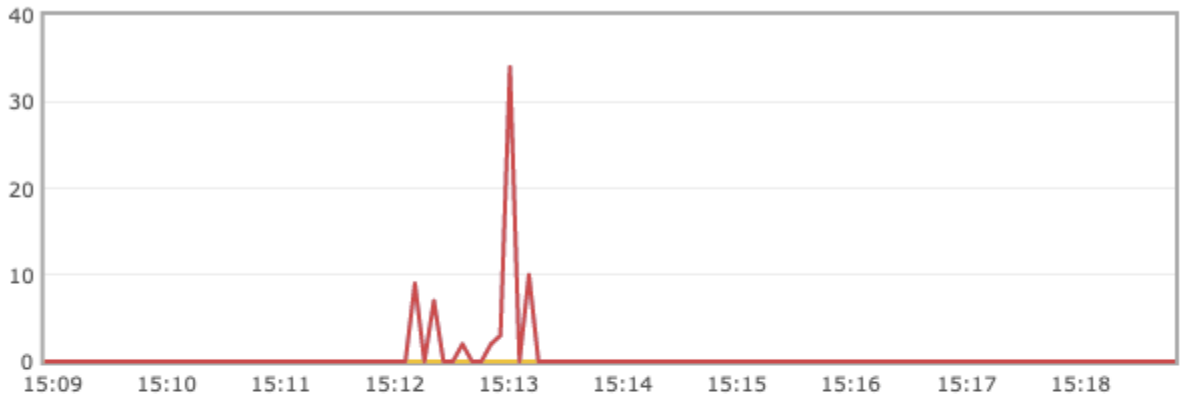
Another interesting configuration setting that I found made a significant difference to the performance of my consumers is the rate at which Redis writes a snapshot of the in-memory database to disk. Early in my test runs, I was finding that the results looked very stable up until the end (around the 1 minute mark), where suddenly the queue size would shoot up to around 60,000 messages, then would immediately come back down to normal levels before terminating. Changing this setting to force Redis to only write snapshots every 5 minutes if at least 100,000 keys were changed helped to fix this problem for the purposes of load testing. In production, further configuration would need to be done to ensure that queue sizes remained stable over longer periods of time.

Skier Microservice Results

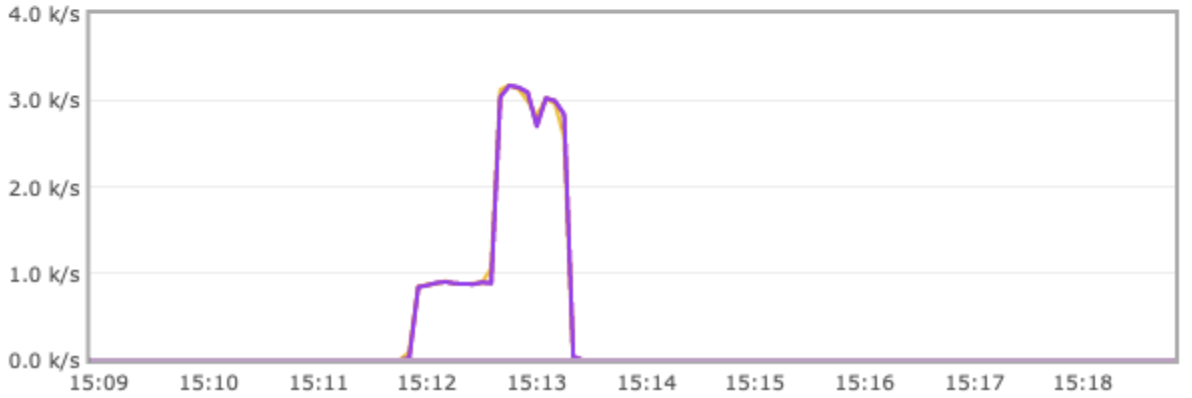
128 Threads (Load Balanced Fleet):

```
------ RUN STATISTICS ------

Total successes: 159820
Total failures: 0
Time elapsed (sec): 86.09
Total throughput (reqs/sec): 1856.4294

Max response time: 836
Min response time: 22
Mean response time: 40.77490927293205
Median response time: 39
99th percentile response time: 75
```

Queued messages last ten minutes ?

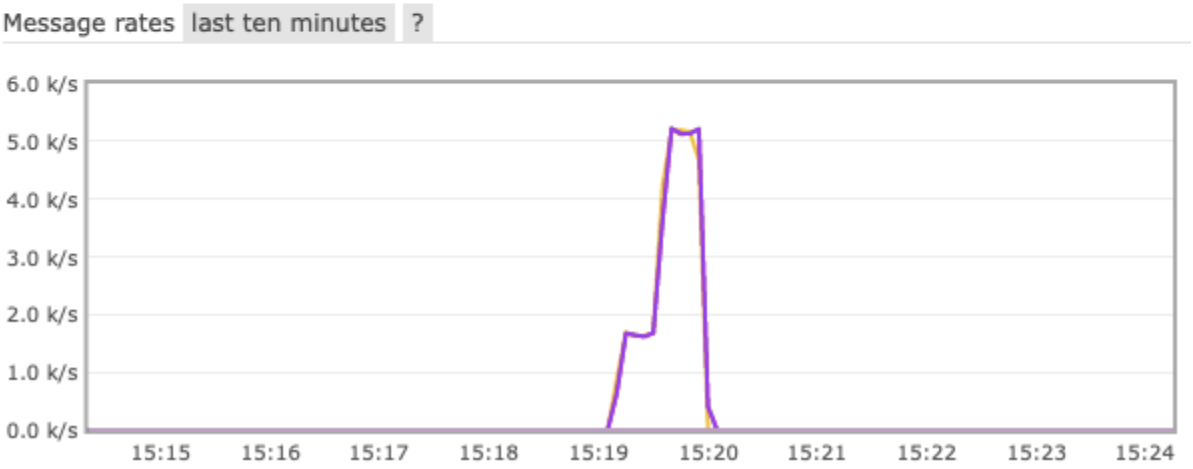


Message rates last ten minutes ?



256 Threads (Load Balanced Fleet):

```
------ RUN STATISTICS ------

Total successes: 159769
Total failures: 0
Time elapsed (sec): 48.034
Total throughput (reqs/sec): 3326.1648

Max response time: 359
Min response time: 23
Mean response time: 47.13489475430152
Median response time: 45
99th percentile response time: 87
```

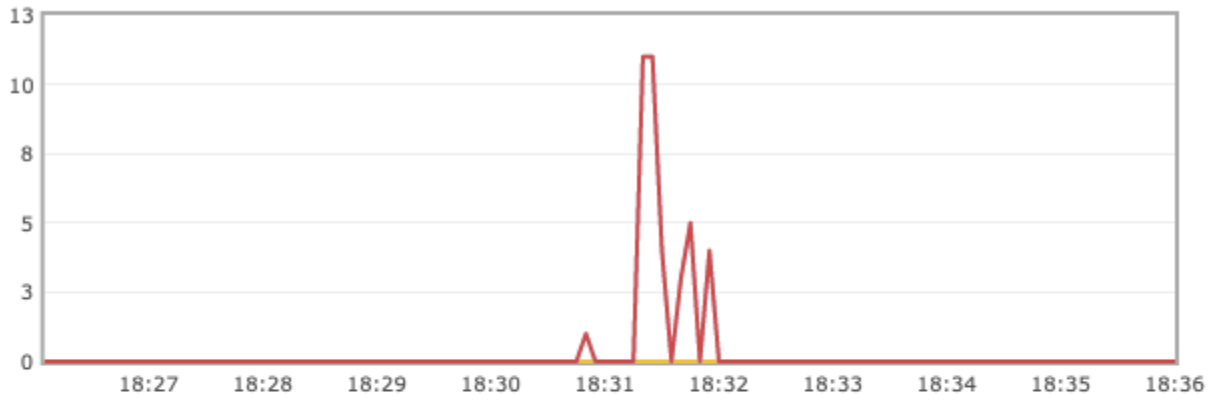


Resorts Microservice Results

128 Threads (Load Balanced Fleet):

```
------ RUN STATISTICS ------

Total successes: 159820
Total failures: 0
Time elapsed (sec): 89.972
Total throughput (reqs/sec): 1776.3304

Max response time: 1592
Min response time: 23
Mean response time: 43.69002002252534
Median response time: 41
99th percentile response time: 88
```

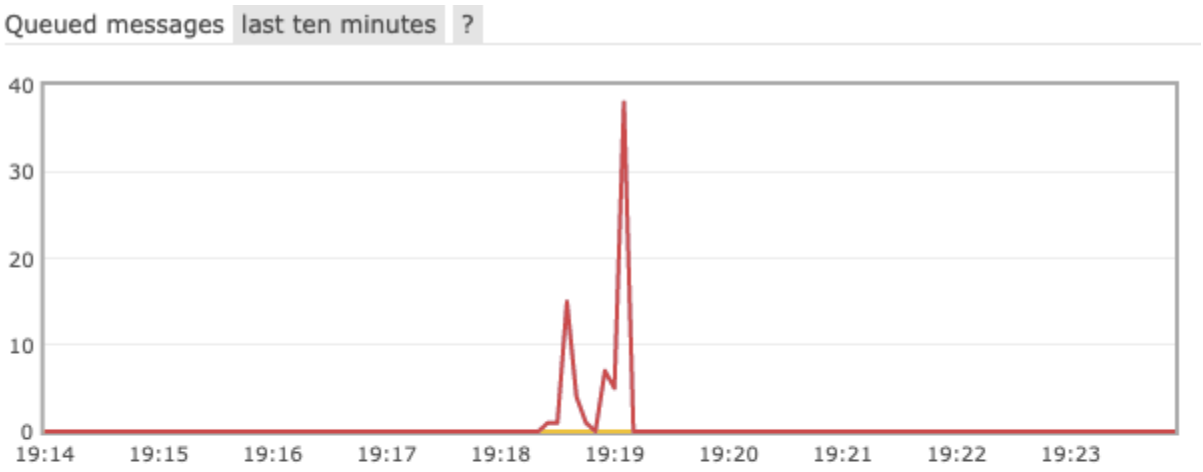
Queued messages last ten minutes ?

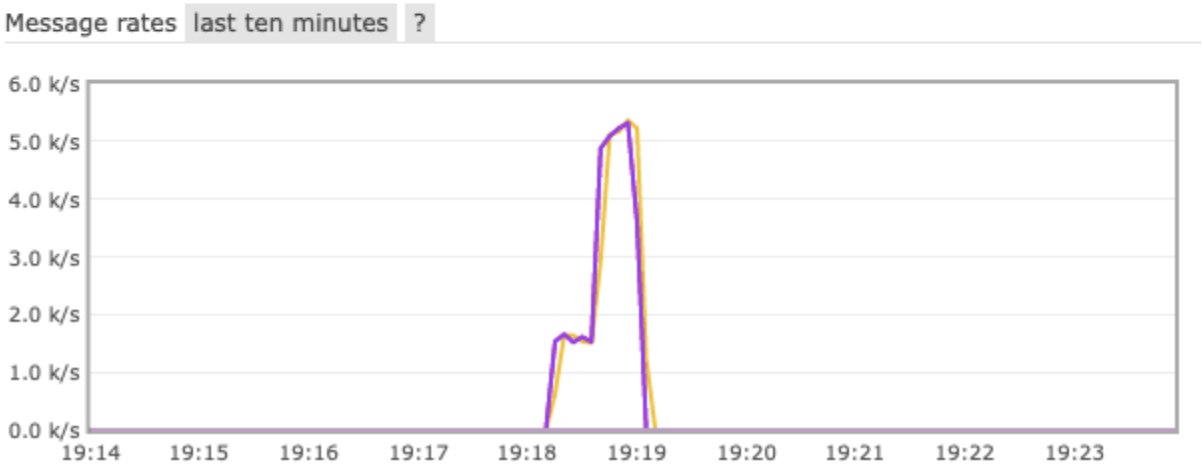


Message rates last ten minutes ?

256 Threads (Load Balanced Fleet):

```
------ RUN STATISTICS ------

Total successes: 159768
Total failures: 1
Time elapsed (sec): 51.186
Total throughput (reqs/sec): 3121.3223

Max response time: 2176
Min response time: 23
Mean response time: 51.688783461536055
Median response time: 47
99th percentile response time: 103
```



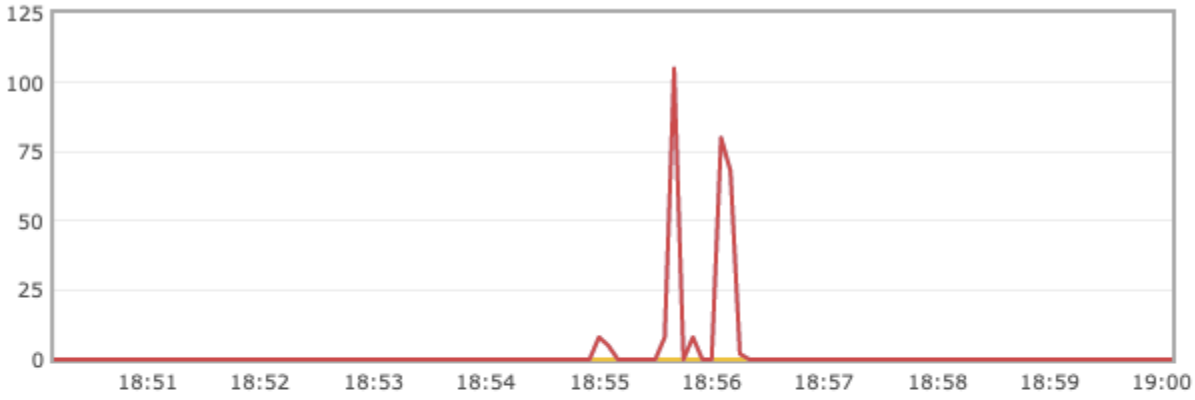


Both Microservices Results (No Mitigation)

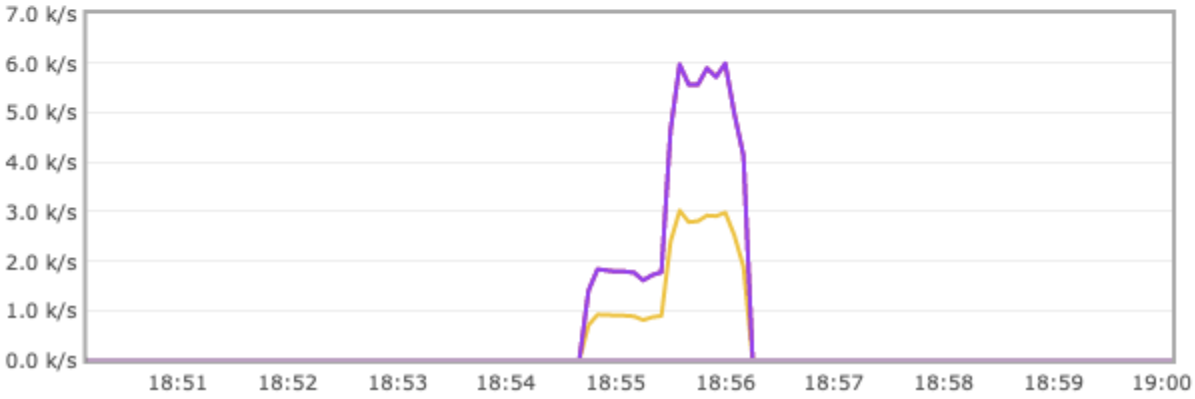
128 Threads (Load Balanced Fleet):

```
------ RUN STATISTICS ------

Total successes: 159820
Total failures: 0
Time elapsed (sec): 89.61
Total throughput (reqs/sec): 1783.5063

Max response time: 1448
Min response time: 20
Mean response time: 43.22335752721812
Median response time: 40
99th percentile response time: 158
```

Queued messages last ten minutes ?



Message rates last ten minutes ?

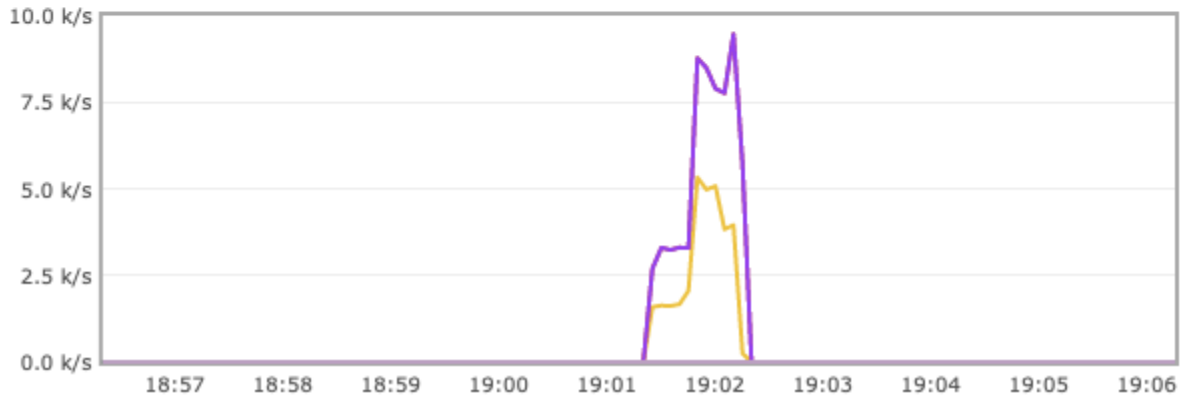


256 Threads (Load Balanced Fleet):

```
------ RUN STATISTICS ------

Total successes: 159767
Total failures: 2
Time elapsed (sec): 48.869
Total throughput (reqs/sec): 3269.2915

Max response time: 356
Min response time: 23
Mean response time: 47.835370547700855
Median response time: 46
99th percentile response time: 94
```

Queued messages last ten minutes ?

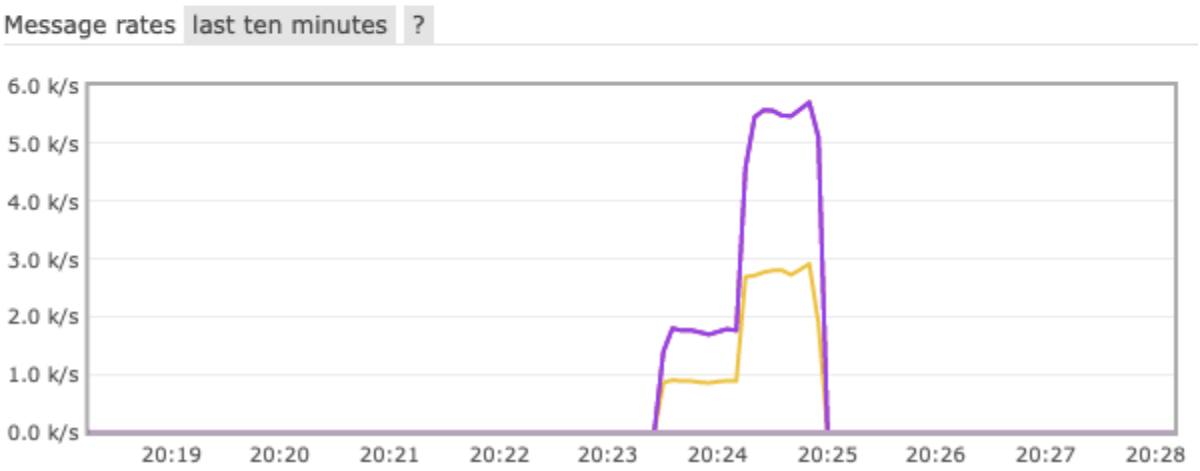
Message rates last ten minutes ?



Both Microservices Results (Larger Consumer Instances)

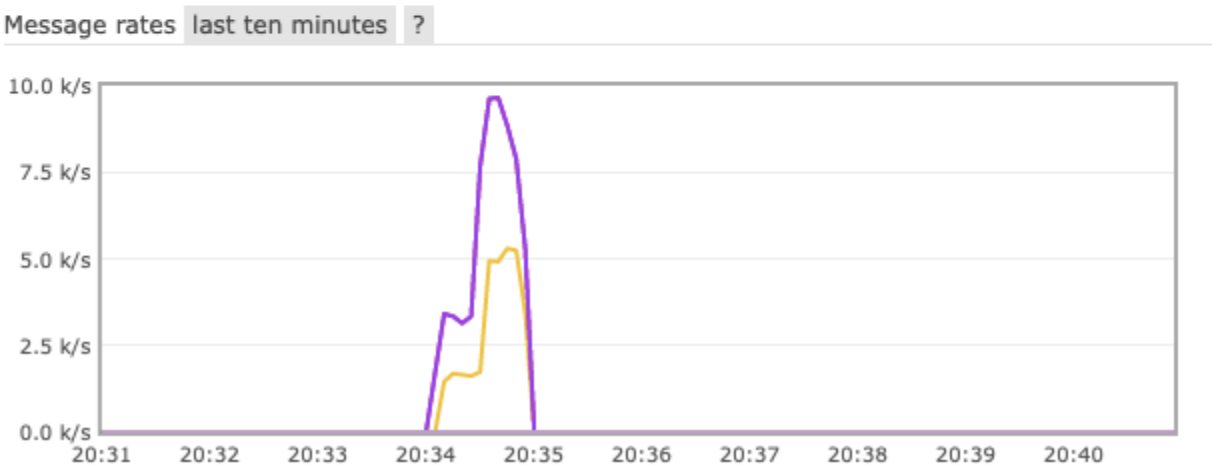
128 Threads (Load Balanced Fleet):

```
------ RUN STATISTICS ------

Total successes: 159819
Total failures: 1
Time elapsed (sec): 89.28
Total throughput (reqs/sec): 1790.0874

Max response time: 285
Min response time: 22
Mean response time: 43.523513364701174
Median response time: 42
99th percentile response time: 76
```



256 Threads (Load Balanced Fleet):

```
------ RUN STATISTICS ------

Total successes: 159769
Total failures: 0
Time elapsed (sec): 48.699
Total throughput (reqs/sec): 3280.7449

Max response time: 365
Min response time: 23
Mean response time: 47.59094692962965
Median response time: 45
99th percentile response time: 93
```



Conclusion

Through testing of my microservices, it is clear that the system can withstand relatively heavy request loads while maintaining stability. There are several factors that contribute to the high and nonvolatile throughput as seen in the test runs above.

The most important contributing factor to this stable system is likely the load balancer distributing requests among four dedicated server instances. In past assignments, I did not see the throughput I expected because the mean response time for larger request loads scaled in proportion with the number of threads generating requests. Here though, the load balancer is doing its job of taking pressure off of the individual instances very well, to the point where the server layer is no longer a bottleneck in achieving high throughput.

Another element of this system that deserves some attention is the relatively simple database design. Because each write to the database is adding an element to a set, which is a constant time operation under Redis' implementation, persisting the data did not cause the consumer's performance to suffer, and thus avoided backlogs in the queue.

The mitigation strategy I employed to further improve consumer performance also seems to have had a slight effect. When the consumers were on t2.micro instances, the peak queue backlogs were around double what they were on the t3.medium instance types, at least in the 256 client runs. Because throughput was high and stable from the client and server, a circuit breaker did not seem to be a good fit as a way to improve the performance of the system. Rate limiting would have reduced the total throughput of the system without providing much in the way of a stability benefit, so increasing consumer capacity seems to be the only mitigation to make in this case.