**CS6650 Assignment 4**

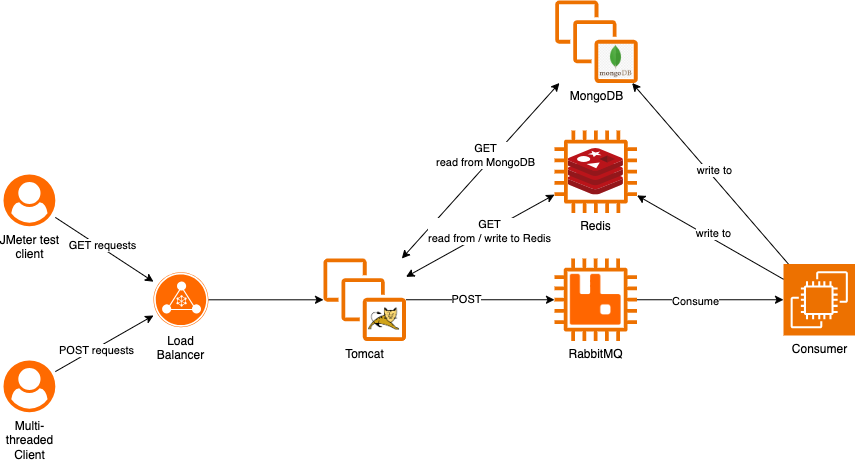
Team RabbitRush

Xi Yang, Yuan Shen, Zhixuan Liu, Nahai Gu

**1. Assignment design**

The system for this assignment is designed to prioritize scalability. It is also optimized for handling both read-heavy and write-heavy workloads, using a microservice-style backend and asynchronous messaging patterns to achieve high throughput and low latency.

**Figure 1. System Design**



**Components Overview**

* **Client**

The multithreaded client for this assignment is identical to the client in assignment 3.

* **Test Client**

The test client is set up using Apache JMeter. A dedicated Thread Group is used for testing each GET API respectively. A concurrent GET request load test that executes all Thread Groups simultaneously is also carried out to further test server’s response to heavy load.

* **Load Balancer**

Routes incoming traffic (both GET and POST) from clients to Tomcat servers for processing. It ensures scalability and fault tolerance.

* **Tomcat Servers**

The GET requests (primarily from the JMeter test client) query cached data in Redis for speed, falling back to MongoDB on cache misses. The POST requests (from a multi-threaded client) are published to RabbitMQ broker to decouple ingestion from processing.

* **RabbitMQ**

Asynchronous message broker that queues POST requests for background processing, enhancing responsiveness and throughput.

* **RabbitMQConsumer**

A background service on a dedicated EC2 instance that consumes messages from RabbitMQ. It is responsible for processing messages, caching data for Read operations to Redis, and persisting data to MongoDB.

* **Redis**

The Redis instance caches both data needed for GET request in Assignment 3 and Assignment 4. A single instance proofs to be effective since the total caches only accounts for 10% of the total available memory. This serves as a fast in-memory cache for frequently accessed GET request data. GETs are served from Redis when possible. On cache misses, data is fetched from MongoDB and cached in Redis asynchronously. This design helps ensure low response times for read-heavy workloads.

* **MongoDB**

It acts as the system of record and source of truth, providing strong consistency for persistent storage. Reads for cache misses and writes from the *RabbitMQConsumer* are directed here.

**Design Goals**

* **High Throughput & Low Latency for Reads**

This is achieved through Redis caching and parallel Tomcat instances behind a load balancer.

* **Write Decoupling**

POST requests are offloaded to RabbitMQ and processed asynchronously by the Consumer, preventing bottlenecks during high-load periods.

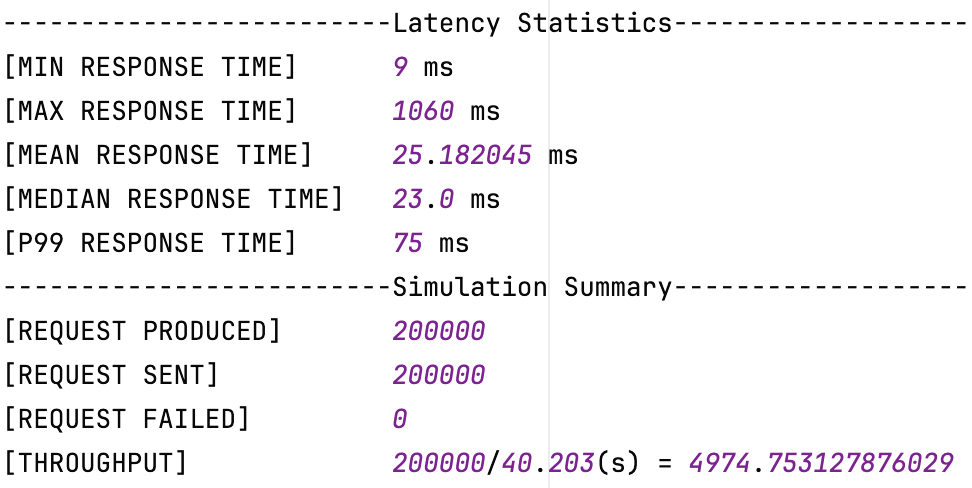
* **Consistency and Partition Tolerance (CP)**

Current design prioritizes consistent views of data across services.

**2. Updated POST result**

The integration of an Application Layer Load Balancer with four Tomcat server instances led to a 21% improvement in previous request throughput (4100.5745 request/s) and a 5% reduction in the best P99 response time (79 ms), further optimizing the system’s scalability and latency under load.

**Figure 2. POST statistics**



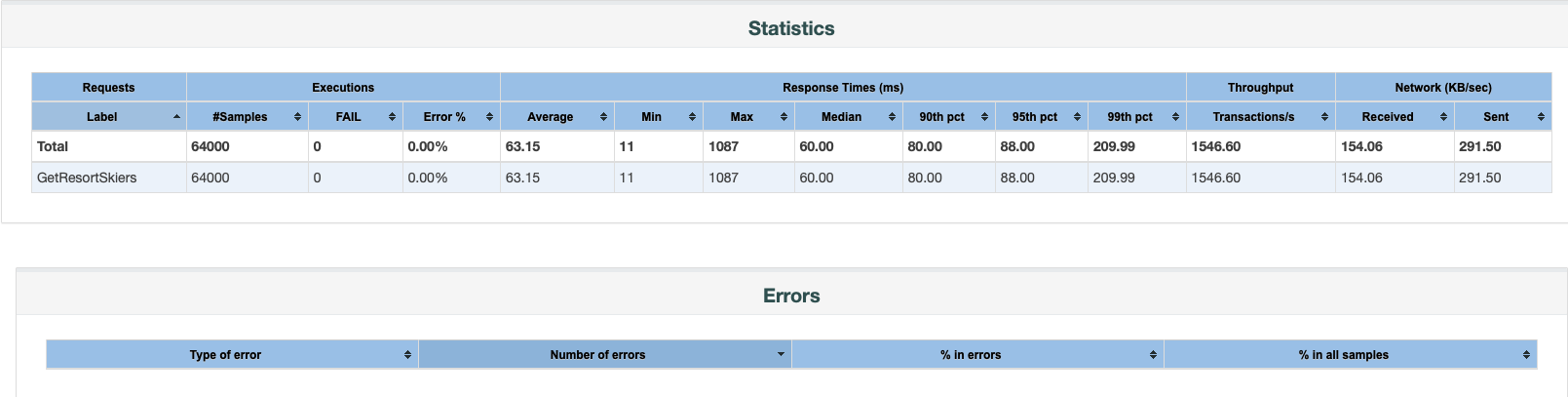
**3. Apache JMeter Test**

There are in total five test runs for the requested GET APIs.

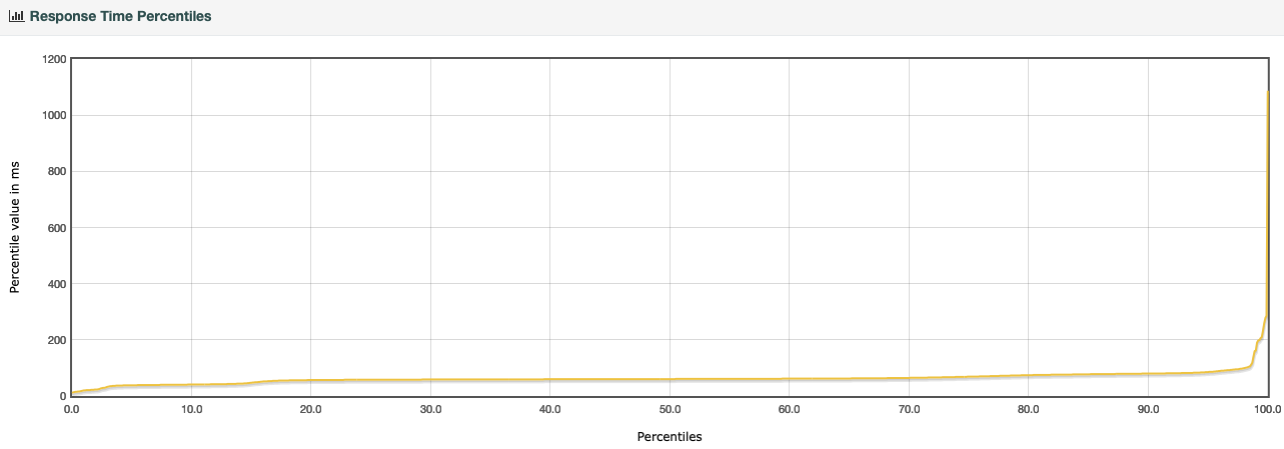
* **Test for *GET /resorts/{resortID}/seasons/{seasonID}/day/{dayID}/skiers***

The table and figures below show that the **average** and **median** response time are respectively **63** and **60** milliseconds. The **P99 percent response time** is around **210** milliseconds. This result indicates a generally well-performing system with occasional outliers.

**Table 1. GET statistics for GetResortSkiers**



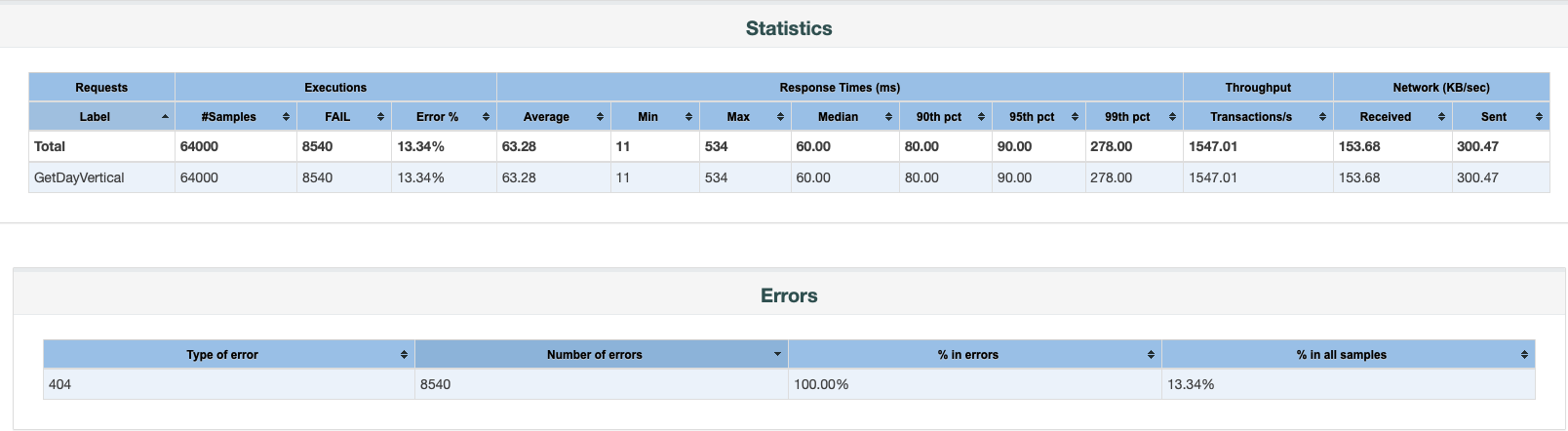
**Figure 3. Response Time Distribution for GetResortSkiers**



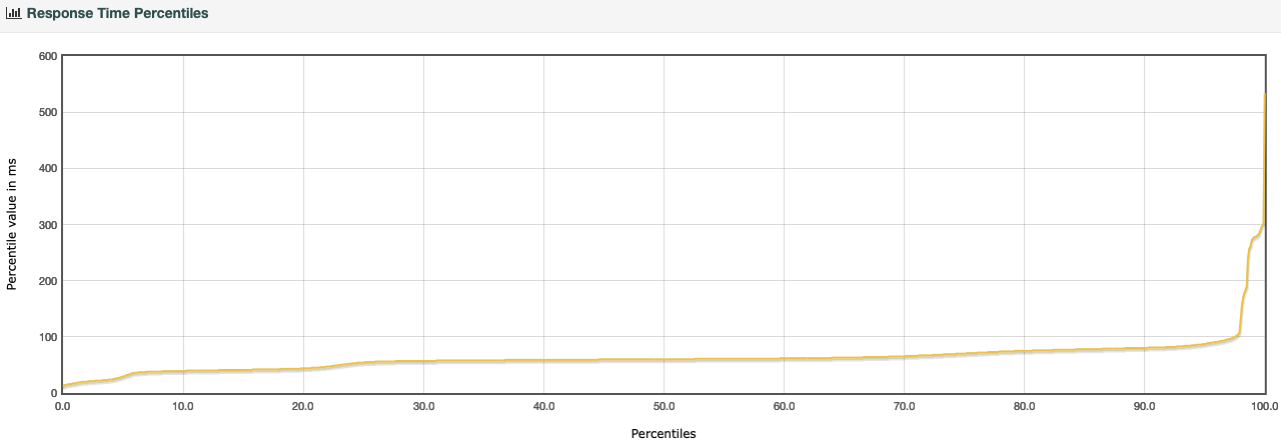
* **Test for *GET /skiers/{resortID}/seasons/{seasonID}/days/{dayID}/skiers/{skierID}***

The table and figures below show that the **average** and **median** response time are respectively **63** and **60** milliseconds. The **P99 percent response** **time** is around **278** milliseconds. Note that even though **13%** of requests did not return any result (404), the overall performance remains stable. This indicates that the system is gracefully handling request that miss both cache and database without significant performance degradation.

**Table 2. GET statistics for GetDayVertical**



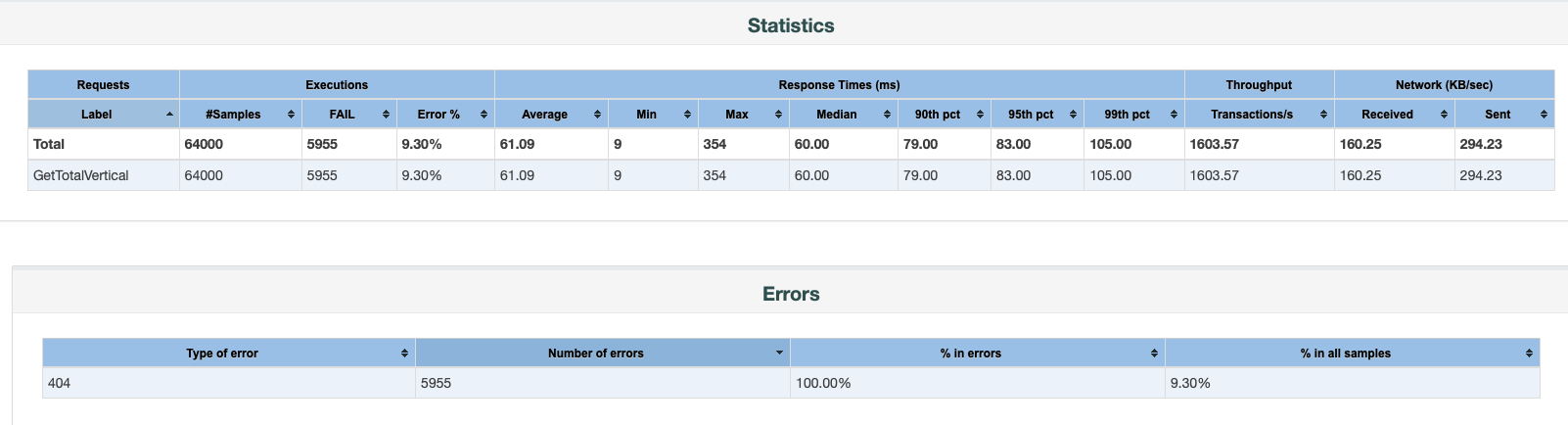
**Figure 4. Response Time Distribution for GetDayVertical**



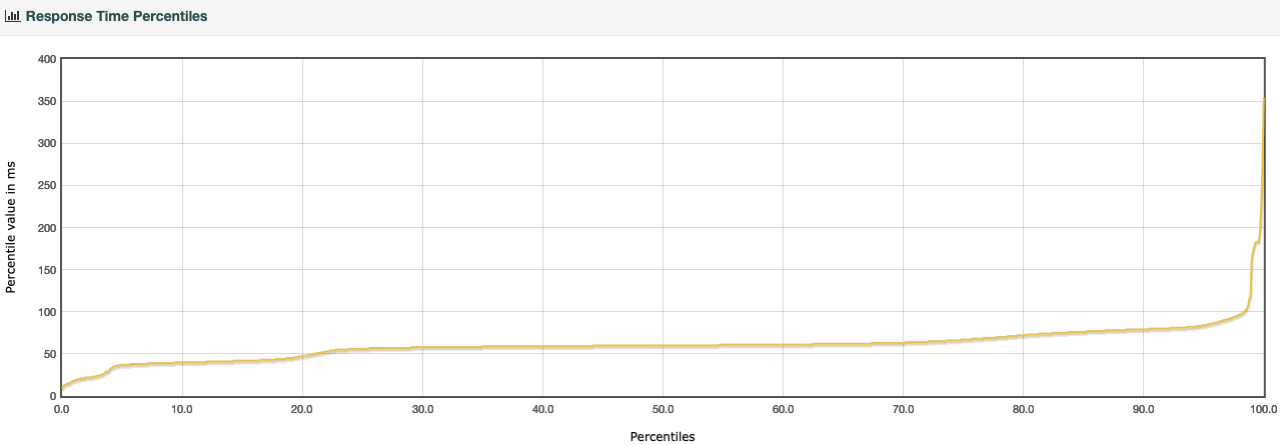
* **Test for *GET /skiers/{skierID}/vertical?resort={resortID}***

The table and figures below show that the **average** and **median** response time are respectively **61** and **60** milliseconds. The **P99 percent response time** is around **105** milliseconds. Even though **9%** of the requests did not return any result (404), the overall performance remains stable.

**Table 3. GET statistics for GetTotalVertical**



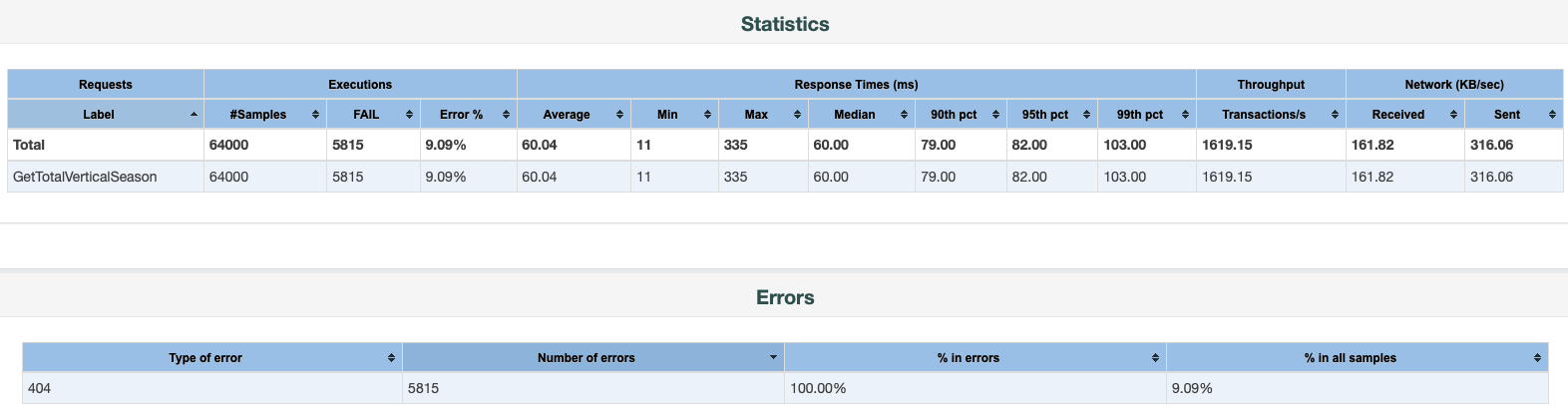
**Figure 5. Response Time Distribution for GetTotalVertical**



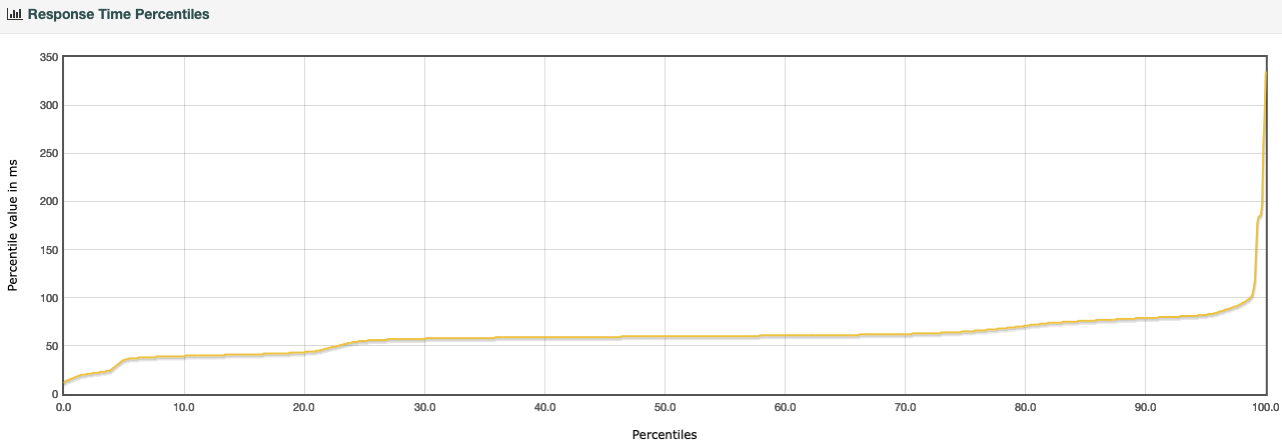
* **Test for *GET /skiers/{skierID}/vertical?resort={resortID}&season={seasonID}***

The table and figures below show that the **average** and **median** response time are **both** **around** **60** milliseconds. **The P99 percent response time** is around **103** milliseconds. Again, **9%** of the requests did not return any result (404) for this test, however, there is no signs of performance degradation.

**Table 4. GET statistics for GetTotalVerticalSeason**



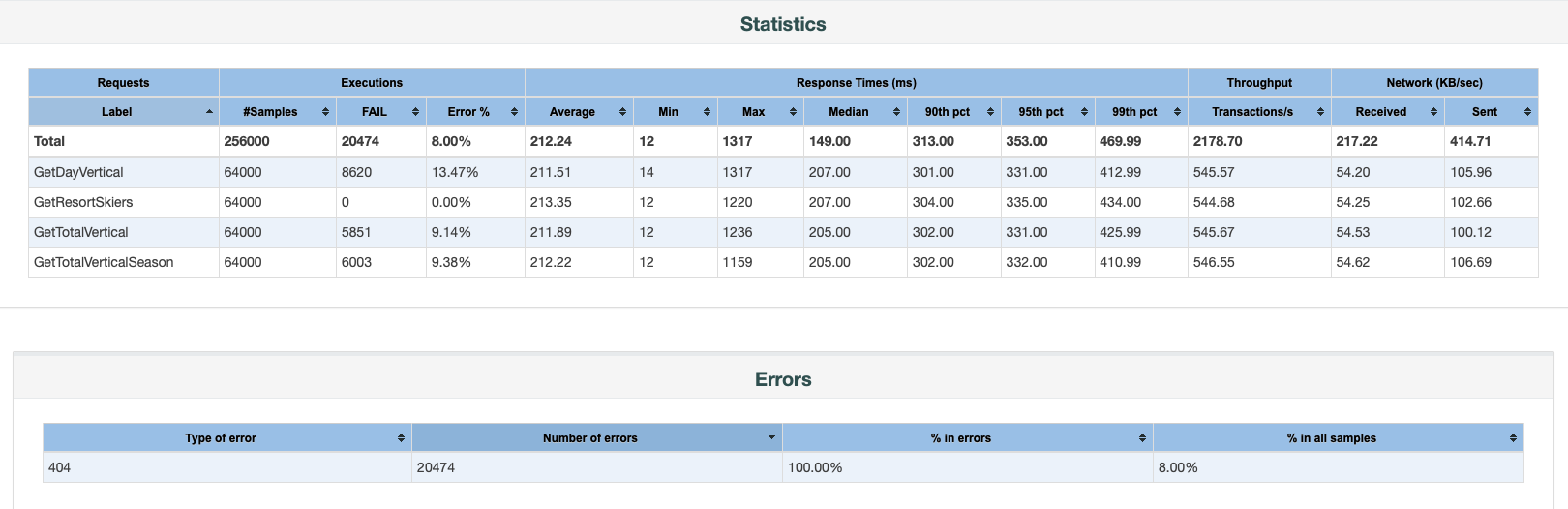
**Figure 6. Response Time Distribution for GetTotalVerticalSeason**



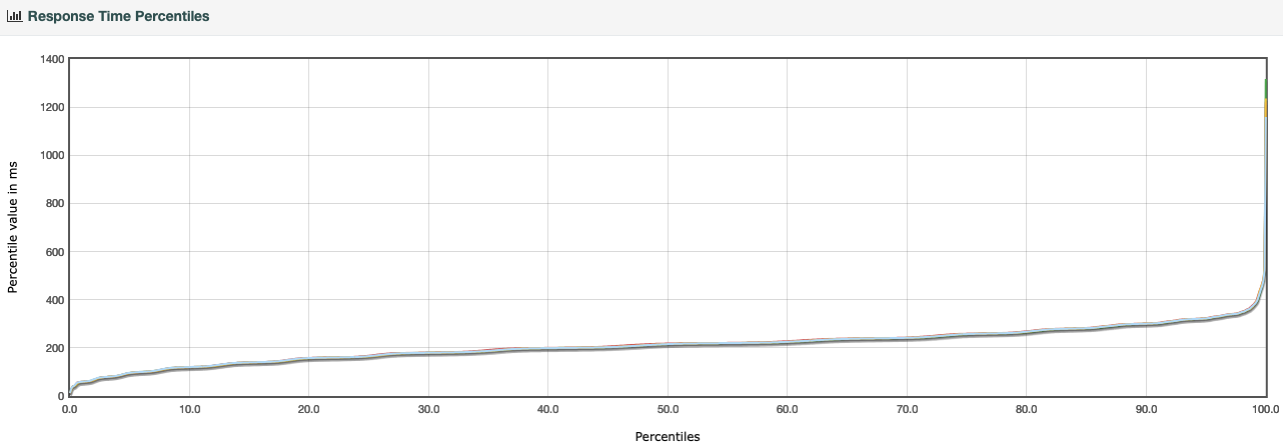
* **Concurrent test for all four APIs**

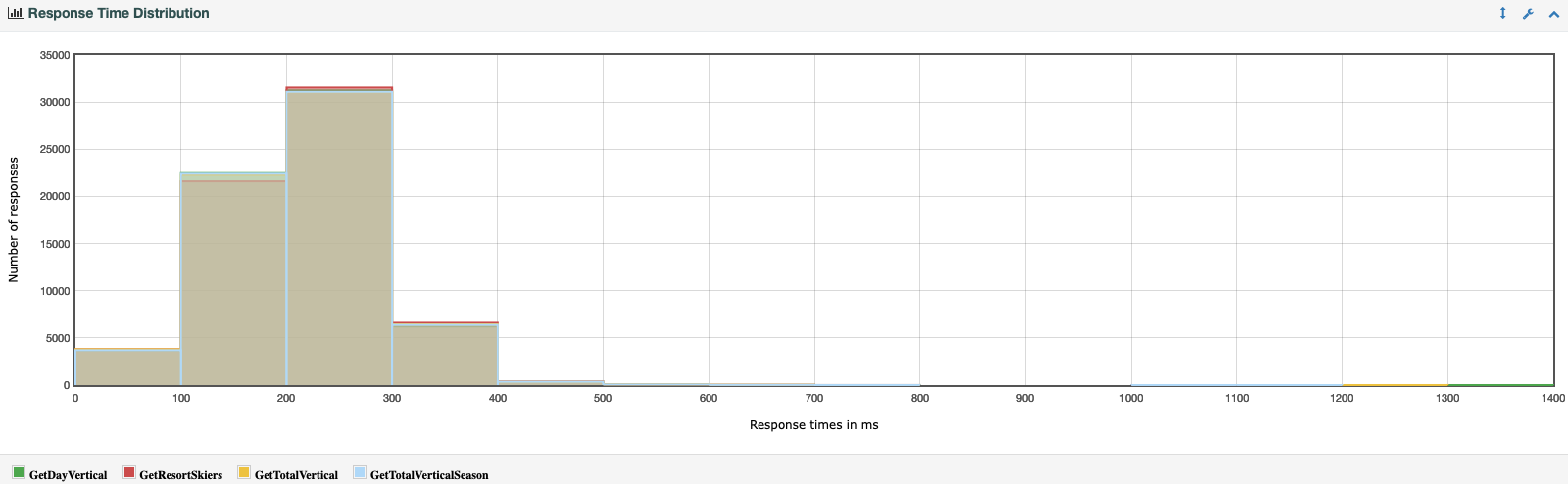
The table and figures below show that the **average** and **median** response time are around **212** and **149** milliseconds. **The P99 percent response time** is around **470** milliseconds. In addition, **8%** of the requests did not return any result (404) for the overall test. There is a significant increase in response time when the server is under more significant load. The fact that median is less than the average suggests a right skewed response time distribution. This skewness suggests bottlenecks in downstream of the system, potentially cache or database access. There exists a small amount of high response time requests that increases the average response time significantly.

**Table 5. GET statistics for Total**



**Figure 7. Response Time Distribution for Total**





**4. Tradeoffs**

Current design can be further optimized based on insights gained from these important test metrics. Here are some considerations for future improvements.

**Table 6. Consideration for Future Improvements**

|  |  |  |
| --- | --- | --- |
|  | Current implementation | Consideration for improvement |
| Real-time vs. Batch Writes | Batch write for better performance | Real-time write to improve consistency |
| Manual EC2 Deployment vs. Auto Scaling | Manual deployment for full control over provisioning | Auto Scaling for flexibility and convenience |
| Cache Accuracy vs. Infrastructure Cost | Keep Redis in sync on every POST request, “read what you write” | High volume of requests requires strong instance configuration, further optimization on resource usage needed |
| Speed vs. Safety | - Single AZ for low latency  - Publisher confirm off | - Deployment across different AZ to increase availability.  - Auto-Ack for higher throughput |

**5. Conclusion**

The current system design prioritizes high throughput, database consistency, and scalability while acknowledging the need for future refinements based on deeper business logic analysis. For further optimizations, we propose the following key considerations.

* **Balance fault tolerance and redundancy**

When message durability is not a strict requirement, Redis can serve as both a message broker and a cache, simplifying the overall architecture. This approach is viable if:

* Throughput and latency requirements are consistently met.
* The system can tolerate potential message loss during failures.
* **Design an Optimal Data Model**

A well-structured data model is essential to strike the right balance between performance, security, and analytical capabilities:

* Performance: Enable low-latency reads and writes through efficient indexing and schema design.
* Security: Implement robust access controls and ensure auditability.
* Analytics: Support real-time insights, aggregations, and the ability to analyze historical trends.
* **Optimize for Memory and Availability Constraints**

Caching and database strategies should be aligned with the system’s memory and availability needs:

* Select between Redis RDB and AOF based on recovery priorities.
* Define TTLs and eviction policies to ensure that data freshness aligns with application needs.
* Consider leaderless database architecture to maximize availability in distributed environments.
* **Scale Strategically with Sharding and Replication**

To achieve horizontal scalability and high availability:

* Use database and cache sharding to distribute workloads evenly and eliminate bottlenecks.
* Deploy read replicas to reduce query load on primary nodes and increase fault tolerance.