BACHELOR OF COMPUTER SCIENCE (HONS.) COMPUTER NETWORKS (CS255)

ITT440 – NETWORK PROGRAMING

INDIVIDUAL ASSIGNMENT

Title: Comprehensive Web Application Performance Testing & Analysis

GROUP: NBCS2555A

PREPARED BY:

NAME	STUDENT ID
NUR AMIRAH FIKRIYAH BINTI CHE JANTAN	2023650858

PREPARED FOR:
SIR SHAHADAN BIN SAAD

SUBMISSION DATE:
7 December 2025

Table of Contents

Introduction

To understand how a web application functions under various levels of user traffic, performance testing is essential. It guarantees that while real people engage with the system, it stays dependable, responsive, and stable. With the proliferation of applications over the internet, it is becoming imperative to understand how they would perform under different traffic patterns and whether they would conform to established standards for performance. This report describes in detail a performance testing exercise that was carried out on an openly accessible web application using Locust, an open-source performance testing tool. Three main types of tests were run which is Load Test, Stress Test, and Soak Test. Verifying several aspects of the application's performance, including system stability, breaking points, long-term endurance, and behaviour under typical usage, was the aim of each test.

These evaluations findings shed light on reaction time trends, mistake rates, throughput, and any performance constraints. This report describes the testing environment, methods utilized, and findings.

Tool Selection Justification

Locust was chosen for the current project because it is lightweight, very adaptable, and written in Python. It gives the user the ability to create unique user behaviour and realistic load scenarios that mimic actual usage patterns. Furthermore, Locust provides a web-based interface for monitoring performance indicators in real-time, which are essential for assessing an application's overall effectiveness. These metrics include response time, requests per second, and failure rates.

Locust built-in functionality for distributed load generation is another advantage. Because of this, it can mimic thousands of users at once, which is an essential feature for stress testing apps meant for heavy traffic. Locust outperforms alternative performance testing tools like JMeter or LoadRunner in terms of scripting ease, setup speed, and test type development scalability. Because of this, Locust is not only useful for professional developers but also perfect for educational settings where teachers wish to properly illustrate concepts related to performance testing. Both professionals and students can work with a cutting-edge tool that reinforces learning about performance indicators and how to conduct performance testing in real world scenarios by using Locust.

Test Environment and Web Target Application

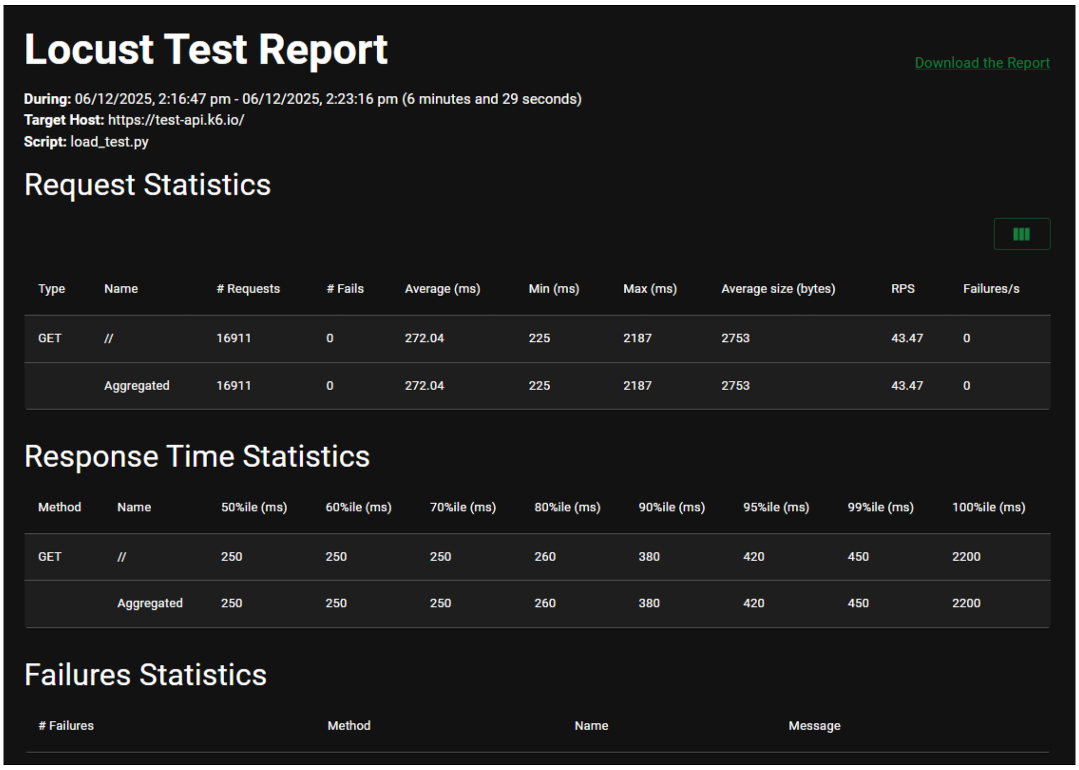
Locust served as the main load testing instrument for this project performance tests, which were carried out in a typical personal computer environment. The testing system had Python installed to run test scripts, a modern web browser to access the Locust web interface and a reliable internet connection, among other things. With Locust simulating user traffic and providing real time monitoring via it is integrated interface, all tests were conducted locally. Consistent and regulated test execution was guaranteed by this configuration. This made it possible to gather performance statistics without interference from background processing or other external system operations.

https://test-api.k6.io/ was the target web application selected for performance testing. It is a publicly available open API that is intended for load testing that is safe, moral, and compliant with the law. This API was developed with performance experimentation and benchmarking in mind, in contrast to the most operational websites that prohibit automated traffic. This API offers a several endpoints that replicate extremely realistic application interactions pertaining to user related queries, product browsing, and authentication procedures. This will provide the ideal framework for assessing real-world system performance metrics, including error rates, response times, throughputs, and behaviour under various load levels. This customized testing API will enable correct analysis and interpretation of the system performance characteristics, enabling these performance tests to yield meaningful and dependable results.

Test Scenarios

To fulfil the project requirements, three different types of performance which is stress test, soak test, and load test were designed and executed using separate Locust scripts.

Load Test



Locust Test Report

During: 06/12/2025, 2:16:47 pm - 06/12/2025, 2:23:16 pm (6 minutes and 29 seconds)
Target Host: https://test-api.k6.io/
Script: load_test.py

Request Statistics

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	//	16911	0	272.04	225	2187	2753	43.47	0
	Aggregated	16911	0	272.04	225	2187	2753	43.47	0

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
GET	//	250	250	250	260	380	420	450	2200
	Aggregated	250	250	250	260	380	420	450	2200

Failures Statistics

# Failures	Method	Name	Message

Figure 1 Load Test Statistics

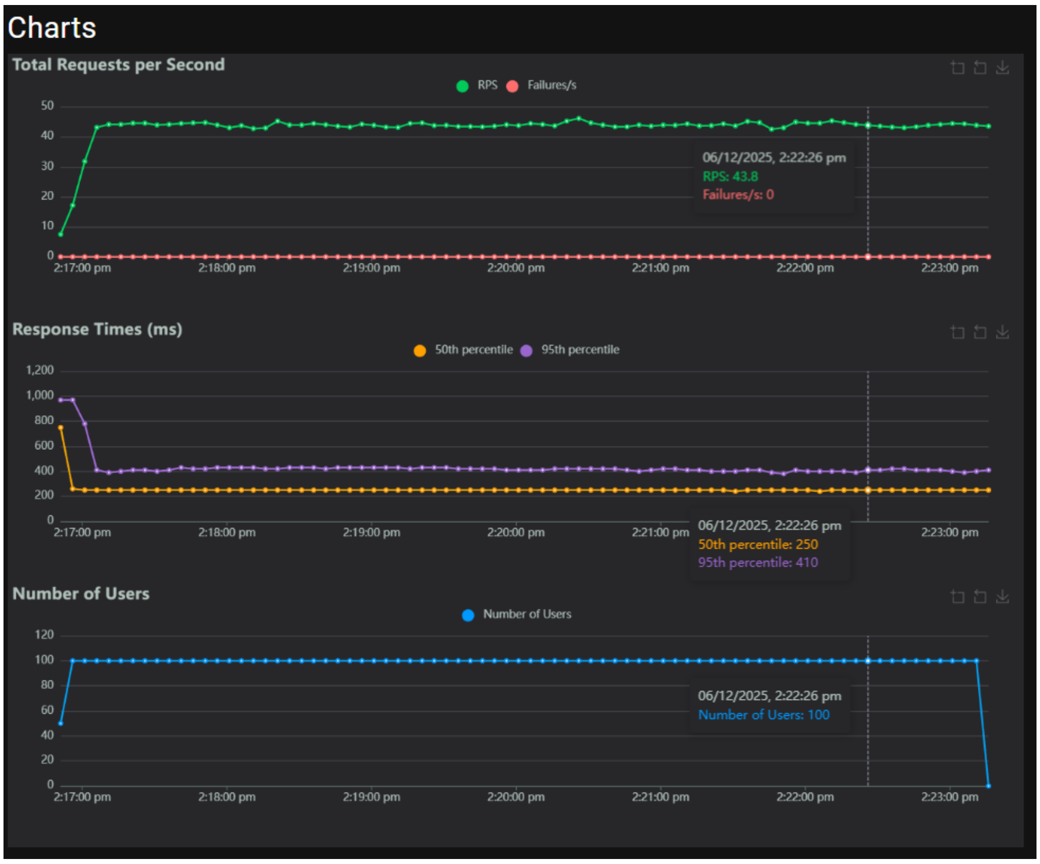
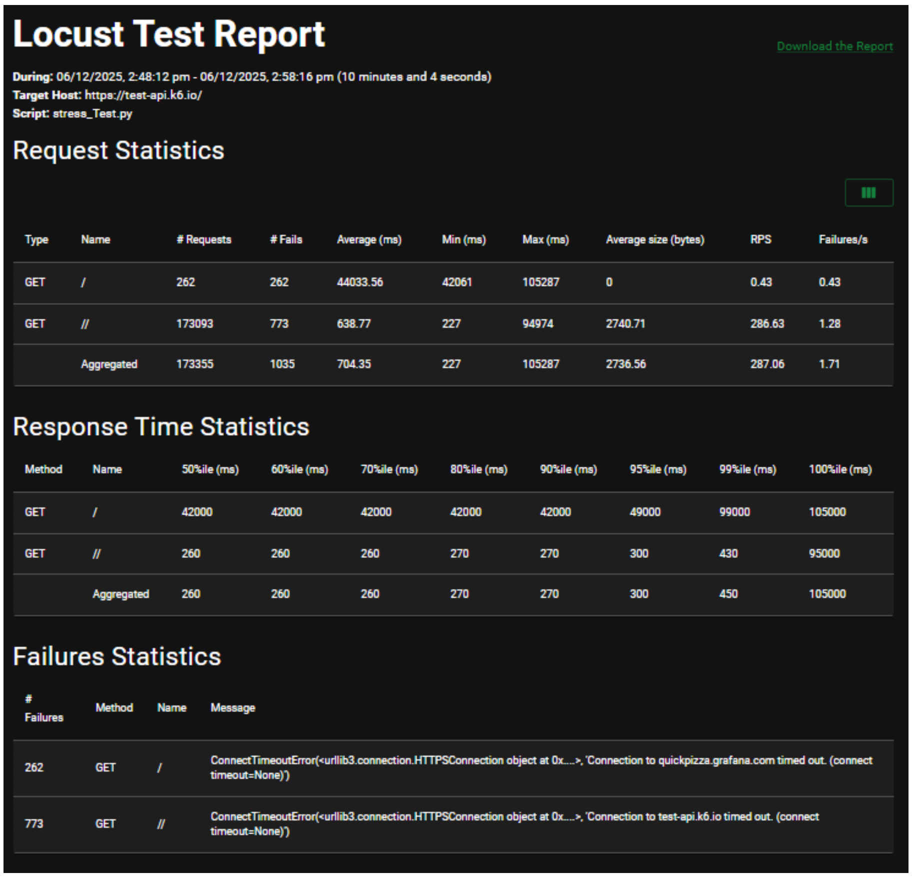


Figure 2 Load Test Chart

A load test with 100 virtual users was carried out for 6 minutes and 29 seconds to see how the test API at https://test-api.k6.io/ will perform under demand. During this test, Locust successfully completed 16,911 GET requests, demonstrating that the API could handle the amount of traffic thrown at it without any issues. The system's capacity to consistently manage high request volumes was demonstrated by the throughput, which was steady at 43–44 requests per second. The greatest response time was 2,187 ms, the median was 250 ms, and the 95th and 99th percentiles were 420 and 450 ms, respectively. As the load increased, latency briefly increased before levelling off and staying constant for the duration of the test. The user count graph, which demonstrates that all 100 virtual users were active during the test period, supports this. Overall, the findings demonstrate that the API managed the load during the users' prolonged activity with good stability, no errors, and predictable latency performance.

Stress Test



Locust Test Report

Download the Report

During: 06/12/2025, 2:48:12 pm - 06/12/2025, 2:58:16 pm (10 minutes and 4 seconds)
Target Host: https://test-api.k6.io/
Script: stress_Test.py

Request Statistics

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	/	262	262	44033.56	42061	105287	0	0.43	0.43
GET	//	173093	773	638.77	227	94974	2740.71	286.63	1.28
	Aggregated	173355	1035	704.35	227	105287	2736.56	287.06	1.71

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
GET	/	42000	42000	42000	42000	42000	49000	99000	105000
GET	//	260	260	260	270	270	300	430	95000
	Aggregated	260	260	260	270	270	300	450	105000

Failures Statistics

# Failures	Method	Name	Message
262	GET	/	ConnectTimeoutError(<urllib3.connection.HTTPSConnection object at 0x....>, 'Connection to quickpizza.grafana.com timed out. (connect timeout=None)')
773	GET	//	ConnectTimeoutError(<urllib3.connection.HTTPSConnection object at 0x....>, 'Connection to test-api.k6.io timed out. (connect timeout=None)')

Figure 3 Stress Test Statistics

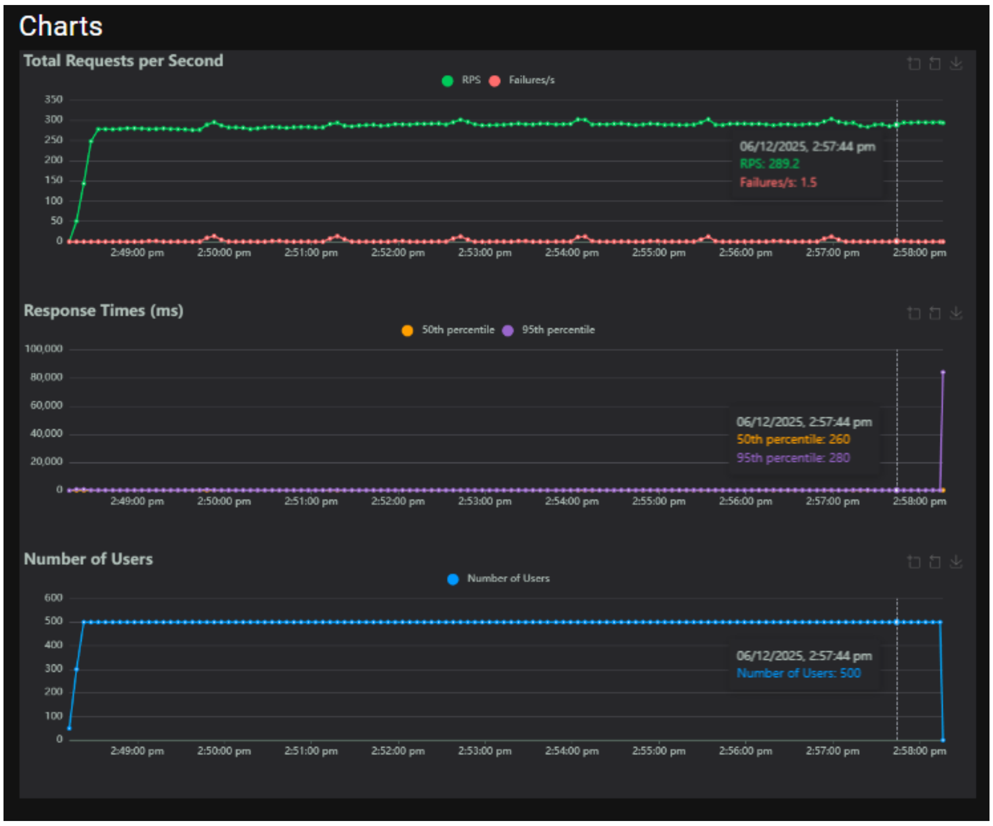
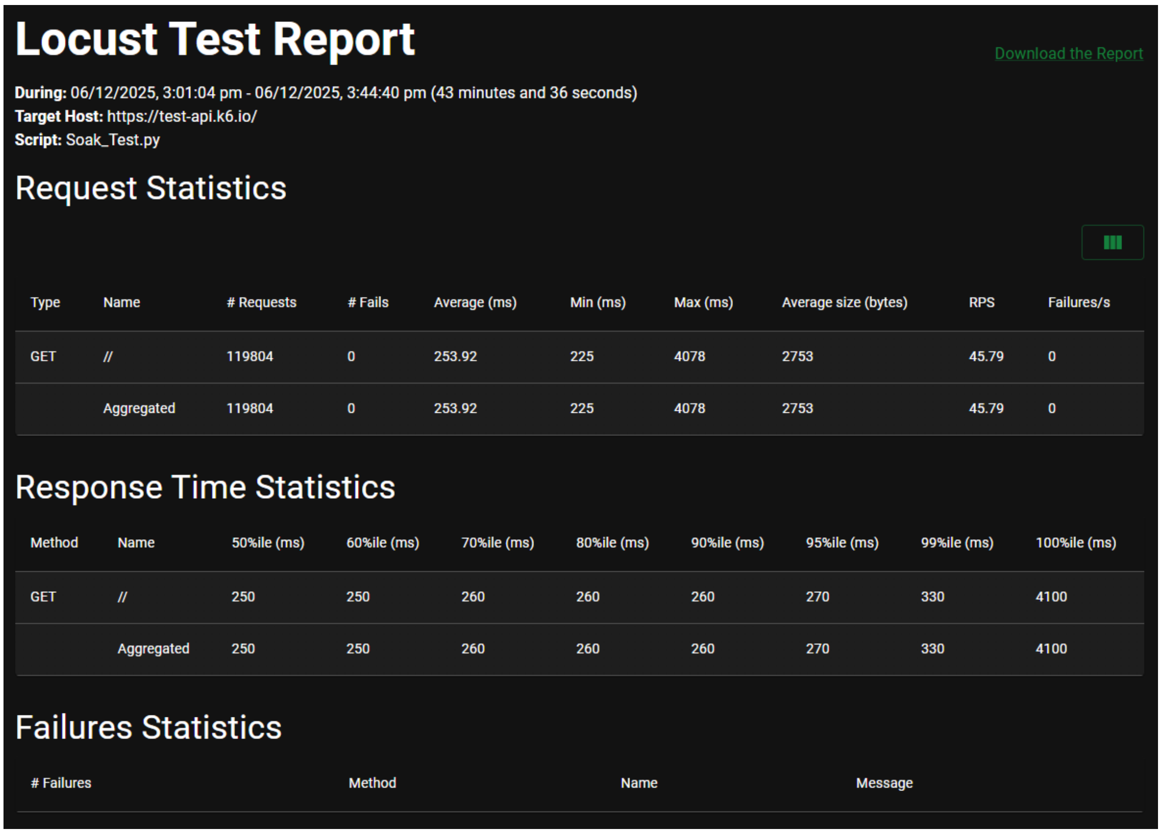


Figure 4 Stress Test Charts

Stress test was successfully pushed above typical utilization limits throughout the ten-minute stress test, which maintained 500 active virtual users and produced a total of 173,355 calls to the test API. The API survived the majority of the load, with 1,035 requests failing owing to ConnectTimeoutError, which indicates that the server or network started timing out under extreme strain. Although response times averaged 704 ms, the majority of requests continuously had latencies between 260 and 300 ms, according to the 50th and 95th percentile plots. The system proved unable to keep up after its capacity was exceeded, as seen by the highest reaction time during the test's peak conditions, which exceeded 105 seconds. Additionally, despite sporadic failure spikes that corresponded with connection timeouts, the Requests Per Second chart stayed steadily high at roughly 280–290 RPS. All things considered, the statistics would show that although the API can sustain large throughput and, for the most part, mild stress, it obviously reaches a breaking point under maximal load, with increased latency and sporadic connection failures.

Soak Test



Locust Test Report

Download the Report

During: 06/12/2025, 3:01:04 pm - 06/12/2025, 3:44:40 pm (43 minutes and 36 seconds)
Target Host: https://test-api.k6.io/
Script: Soak_Test.py

Request Statistics

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	//	119804	0	253.92	225	4078	2753	45.79	0
	Aggregated	119804	0	253.92	225	4078	2753	45.79	0

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
GET	//	250	250	260	260	260	270	330	4100
	Aggregated	250	250	260	260	260	270	330	4100

Failures Statistics

# Failures	Method	Name	Message

Figure 5 Soak Test Statistics

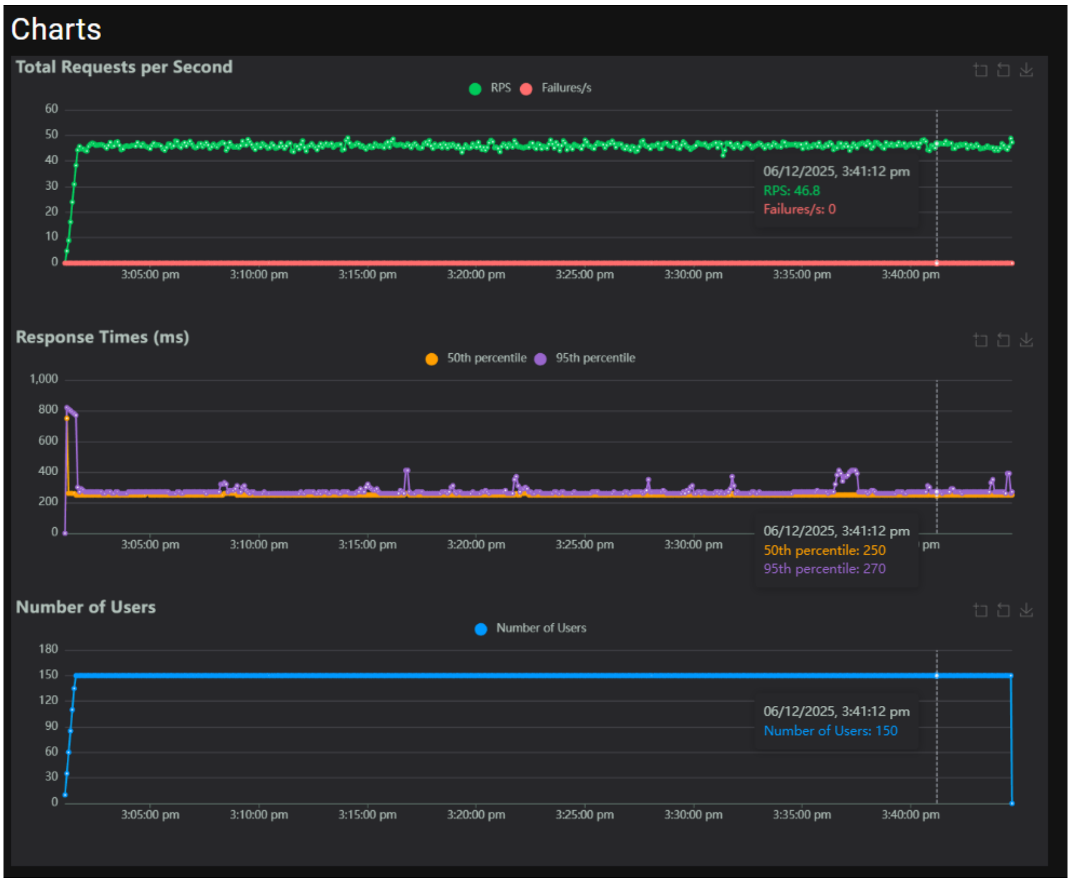


Figure 6 Soak Test Charts

To assess the API's long-term durability under constant demand, 150 virtual users took part in the soak test for 43 minutes and 36 seconds. The system demonstrated remarkable dependability and endurance by processing 119,804 GET requests over the test period without experiencing a single failure. Despite delays ranging from 225 ms to a maximum of 4,078 ms, the average response time for this was 253.92 ms. This implies that although there were spikes, overall performance remained unaffected. Additionally, percentile results exhibit stable and predictable reaction times for the sustained load, with the 50th percentile at 250 ms and the 95th percentile at 270 ms. The graphs confirm the same, the Requests Per Second graph shows a consistent throughput of 45–47 RPS, the Response Time graph mostly flat with minor spikes that flattened out, and the User Count graph stays consistently at 150 users throughout the test run. With no indications of deterioration, memory leaks, or rising error rates, these results show that the API maintains good performance and stability over an extended period, suggesting that the system is well-optimized for operation over an extended period under moderate load.

Analysis of Performance Data

Metric	**Load Test**	**Stress Test**	**Soak Test**
Total Requests	16,911	173,355	119,804
Failures / Error Rate	0 failures (0%)	1,035 failures (≈0.60%)	0 failures (0%)
Average Response Time	~253–272 ms	~704 ms	~254 ms
Minimum Response Time	225 ms	227 ms	225 ms
Maximum Response Time	2,187 ms	105,287 ms	4,078 ms
50th Percentile (Median)	250 ms	260 ms	250 ms
95th Percentile	420 ms	280 ms	270 ms
99th Percentile	450 ms	Extremely high (overload point)	270–300 ms
Throughput (RPS)	43–44 RPS	280–290 RPS	~5–47 RPS
System Stability	Stable under moderate load	Degraded under extreme load (timeouts)	Stable over long duration
Observed Bottlenecks	Minor initial latency spike	Severe latency spikes, timeouts	Occasional spikes but overall stable
Resource Utilization Insight	No signs of overload	Likely CPU/network saturation	No signs of memory leak or CPU stress

Hypothesis

It was anticipated that the API at https://test-api.k6.io/ would operate continuously and reliably under modest load before the performance tests. It was anticipated that response times would stay consistent and that there would be few failures during load and soak tests. The system was built to withstand traffic for extended periods of time without exhibiting any indications of degradation, such as increased latency or memory leaks. However, it was anticipated that early signs of overload, like a longer response time, poorer throughput, or more connection timeouts, would show up on the API during a stress test, when the number of virtual users and request rates were purposefully pushed above typical operational limitations. Overall, the hypothesis predicted that the system would function well under typical and prolonged use, with performance loss only happening when the system was overextended.

Identified Bottlenecks

Several bottlenecks have been found in all three performance tests Load, Stress, and Soak, which highlight the test API operational limitations at varying traffic volumes. Although sporadic latency spikes suggested the system still encounters brief periods of increased processing delay during rapid user ramp-up, the load test did not reveal any serious bottlenecks, with the API handling 100 concurrent users without its response time changing dramatically. When the API started to surpass its ideal capacity during the Stress Test, clear bottlenecks appeared.

When 500 virtual users overwhelmed the server, response times increased significantly, peaking at over 105 seconds, and connection timeout issues started to occur. Reduced responsiveness, server saturation, and an inability to manage high concurrency are some of the symptoms of these problems. During the lengthy test execution, the Soak Test discovered no functional bottlenecks. However, slight variations in response time and occasional latency spikes indicate that periodic load variability may have a brief effect on performance. In summary, studies have demonstrated that although the API functions effectively under heavy and continuous load, it becomes unstable with severe delays and connection failures when high amounts of stress are imposed. This indicates that system capacity and connection handling constraints are the primary obstacles.

Recommendations and improvements

The following enhancements may be recommended to improve the API capacity to tolerate higher loads while keeping constant response times, based on the performance test results. First, to help the system handle abrupt increases in traffic that are seen during the stress test, scaling solutions like autoscaling policies or horizontal scaling with extra server instances may be put in place. Reducing database query times, improving API logic, or adding caching layers are examples of backend infrastructure optimization that would significantly lower response times and prevent severe latency spikes that were observed under high demand. By enhancing connection handling and timeout configurations once the system reaches its maximum capacity, this could further lower connection timeout errors. Load balancing across several servers should be used to lessen bottlenecks and more equally distribute traffic loads. To identify early indicators of a decline in performance during extended sessions, track CPU use, memory usage, and network throughput using continuous monitoring tools such as Grafana, Prometheus, or APM suites. Last but not least, adding more intricate API flows to test scenarios such as data creation, authenticated requests, and interaction with numerous endpoints will assist uncover more serious performance problems and enable the system to operate in a wider variety of real-world scenarios.

Conclusion

In conclusion the comprehensive API performance testing may be found at https://test-api.k6.io/, It has shown the system general effectiveness, dependability, and capacity to manage various types of loads. The load test demonstrated that the API functions effectively with a reasonable number of users. It is steady in terms of response time and throughput and shows no failures, indicating that it is well optimized for common use scenarios. The soak test, which demonstrated that the API could withstand extended durations of continuous activity without experiencing memory problems, performance degradation, or rising error rates a hallmark of stability under real-world, long duration operating conditions further confirmed this.

The stress test, however, exposed the API real flaw excessive concurrency caused noticeable latency spikes, increasing response times, and connection timeouts indications that it is approaching and beyond its full capacity. A behavioural pattern like this clearly identifies the point at which the API becomes unreliable. All of the tests taken together have achieved the goal of assessing stability, responsiveness, and scalability under various loads as well as identifying the critical bottlenecks that require attention to improve the system resilience. Overall, this research emphasizes a number of useful insights that could further guide optimization efforts and architectural improvements. It also emphasizes the importance of performance testing as a preventative measure to ensure system stability.