Mid Term -Testing Report

Group Members

Anu Ajith - 8809994

Nithin Varghese – 8946846

Mohammed Rafique – 8954785

Shivani Varu – 8941914

Shikha Kaushalkumar Shah -

SENG8051 -24W-Sec 1- Non-Functional Testing

Aleksandar Malinovic

Contents

Testing Report

1. Introduction

The Hospital Management System (HMS) is a critical web application designed to streamline patient and doctor management, appointment scheduling, and medical record handling. As healthcare systems increasingly rely on technology, ensuring the performance and reliability of such systems becomes paramount.

1.1 Purpose Of Document

This performance testing report aims to assess the HMS application's responsiveness, scalability, and resource utilization under various scenarios. By conducting rigorous performance testing, we can identify bottlenecks, optimize critical components, and enhance the overall user experience.

1.2 Objective

Our primary objective is to evaluate the HMS application's performance, focusing on response times, throughput, and system stability. Additionally, we aim to provide actionable recommendations for performance optimization.

2. Scope Of Project

This table provides a high-level overview of the key functionalities within the HMS application. Each component plays a crucial role in ensuring efficient hospital management.

ID	Components/Application Name	Functions
1	Patient Registration	- Add new patients, Modify patient details, Upload patient reports
2	Doctor Management	- Add new doctors, Modify doctor details, View doctor information
3	Appointment Handling	- Create and manage appointments, View appointment schedules
4	Patient History	- View patient history (previous appointments, medical records)

3. Quality criteria

- Accuracy and Data Integrity:

Patient and doctor information should be accurately recorded and maintained.

Ensure that data integrity is preserved during CRUD operations.

- Usability and User Experience:

The system's user interface should be intuitive and easy to navigate.

Users (hospital staff) should find it straightforward to perform tasks such as patient registration, appointment scheduling, and record management.

- Performance and Responsiveness:

Response times for critical actions (e.g., adding a patient, viewing appointments) should be within acceptable limits.

The system should handle concurrent users efficiently.

- Security and Privacy:

Patient data must be securely stored and accessible only to authorized personnel.

Implement role-based access control to restrict unauthorized access.

- Scalability and Growth:

The HMS should scale seamlessly as the hospital expands.

Consider future requirements (e.g., increased patient load, additional features).

- Error Handling and Logging:

Proper error messages should guide users when issues occur.

Implement robust logging to track system events and errors.

- Interoperability and Integration:

The system should integrate with other hospital systems (e.g., billing, and electronic health records).

APIs and data exchange mechanisms are essential.

- Compliance with Regulations:

Ensure compliance with healthcare regulations (e.g., HIPAA, GDPR).

Regularly audit the system for adherence to standards.

4. The decisive factors of the project's success

Several critical factors contribute to the success of the HMS project:

- Clear Project Objectives and Scope:

Define clear goals for the HMS project.

Understand the scope of functionalities (e.g., patient registration, doctor management, appointment handling).

Ensure alignment with stakeholders' expectations.

- Effective Project Management:

Strong project leadership and governance are essential.

Timely planning, resource allocation, risk management, and communication contribute to success.

- User Involvement and Engagement:

Engage end-users (hospital staff, administrators) throughout the development process.

Regular feedback and usability testing enhance system adoption.

- Functional Requirements and Use Cases:

Detailed functional requirements drive system design.

Well-defined use cases ensure that the system meets user needs.

- Technical Competence and Infrastructure:

Skilled development teams (front-end, back-end, database) are crucial.

Robust infrastructure (servers, databases, network) supports system performance.

- Quality Assurance and Testing:

Rigorous testing (unit, integration, performance) ensures system reliability.

Address defects promptly to maintain quality.

- Change Management and Training:

Prepare users for system changes.

Provide comprehensive training to hospital staff.

- Security and Compliance:

Implement robust security measures (access control, encryption) to protect patient data.

Comply with healthcare regulations (e.g., HIPAA).

- Stakeholder Communication:

Regularly update stakeholders on project progress.

Address concerns and manage expectations.

- Post-Implementation Support and Maintenance:

Provide ongoing support for bug fixes, enhancements, and system updates.

Monitor system performance and address issues promptly.

5. Limitations, assumptions, and risks

5.1 Limitations

Resource Constraints:

Limited budget, time, and personnel may impact the project's scope and quality.

Ensure realistic expectations within the available resources.

Technology Limitations:

The chosen technology stack (HTML5, CSS3, JavaScript, PHP, AJAX, MySQL) may have limitations.

Compatibility issues across browsers and devices should be considered.

Scalability Challenges:

As the hospital grows, the system must handle increased data volume and user load.

Scalability limitations may affect long-term sustainability.

Security Constraints:

Ensuring data security (patient records, appointments) is challenging.

Vulnerabilities (e.g., SQL injection, cross-site scripting) must be addressed.

Regulatory Compliance:

Ensure compliance with healthcare regulations (e.g., HIPAA, GDPR).

Non-compliance can lead to legal and reputational risks.

5.2 Assumptions

Stable Infrastructure:

Assume that the underlying infrastructure (servers, databases, network) is stable and reliable.

Any infrastructure issues may impact system performance.

User Training and Adoption:

Assume that hospital staff will receive adequate training on using the HMS.

User adoption is critical for successful implementation.

Data Accuracy and Consistency: Assume that patient and doctor data entered the system is accurate.

6. Strategy of testing

6.1 Testing Phases

Unit Testing:

Test individual components (e.g., functions, modules) in isolation.

Verify the correctness of code logic and functionality.

Use testing frameworks (e.g., PHP Unit for PHP) to automate unit tests.

Integration Testing:

Test interactions between different system components.

Validate data flow, APIs, and communication channels.

Ensure seamless integration of modules.

Performance Testing:

Evaluate system responsiveness, scalability, and resource utilization.

Conduct load testing, stress testing, and endurance testing.

Measure response times, throughput, and error rates.

User Acceptance Testing (UAT):

Involve end-users (hospital staff) to validate system functionality.

Ensure that the HMS meets user needs and aligns with business requirements.

6.2 Acceptance Criteria

Patient Registration:

Users can successfully add new patients.

Patient details are accurately stored in the database.

Doctor Management:

Doctors can be added, modified, and viewed.

Doctor information is consistent and accessible.

Appointment Handling:

Appointments can be created, managed, and viewed.

Scheduling conflicts are handled appropriately.

Patient History:

Patient history (previous appointments, medical records) is retrievable.

Data consistency is maintained.

6.3 Completion Criteria

All test phases (unit, integration, performance, UAT) are executed.

Critical defects are resolved.

User feedback is incorporated.

Documentation (user manuals, technical guides) is prepared.

6.4 Reporting

Test Summary Report:

Overview of testing activities, results, and key metrics.

Highlight any critical issues or risks.

Defect Reports:

Detailed information on identified defects.

Severity, priority, steps to reproduce, and status.

Performance Metrics:

Response times, throughput, and resource utilization.

Graphs and charts for visual representation.

User Acceptance Sign-Off:

Formal acceptance by end-users.

Indicates readiness for production deployment.

7. Requirements for the application for performance testing

The following requirements for the application and load amount values for the testing are under consideration and may change later.

The application must meet the following requirements:

1. The application must respond without errors.

2. The application is required to be available 24 hours per day every day.

3. All user transactions must respond to the user within 60 seconds.

Performance Metrics and KPIs:

Define specific performance metrics to measure, such as:

Response times for critical transactions (e.g., patient registration, appointment scheduling).

Throughput (transactions processed per second).

Concurrent user capacity (maximum simultaneous users).

Resource utilization (CPU, memory, network bandwidth).

Scalability Testing:

Evaluate how the HMS scales under increasing loads.

Test with varying user volumes and data sizes.

Assess system behavior during peak usage.

Stress Testing:

Assess the system’s stability under extreme conditions.

Apply heavy loads to identify breaking points.

Monitor resource exhaustion and error handling.

Load Testing:

Determine the system's capacity by gradually increasing the load.

Measure response times and resource utilization.

Identify performance bottlenecks.

8. Test iterations.

Test Iteration: Main Test Runs

In this performance testing iteration for the Hospital Management System (HMS), we focus on assessing critical functionalities and measuring system responsiveness. The primary goal is to ensure that the HMS performs optimally under various scenarios.

Test Runs Table

Number	Operation Description	Time (Minutes)
1	Patient Registration	5
2	Doctor Management	4
3	Appointment Handling	6
4	Patient History	7

Explanation:

Patient Registration:
Verify that patients can successfully register online.
Measure the time taken for the registration process.
Doctor Management:
Test doctor-related operations (addition, modification, viewing).
Record the time spent on doctor management tasks.
Appointment Handling:
Create and manage appointments.
Measure the time for appointment scheduling.
Patient History:
Retrieve patient history (previous appointments, medical records).
Record the time taken to access patient data.

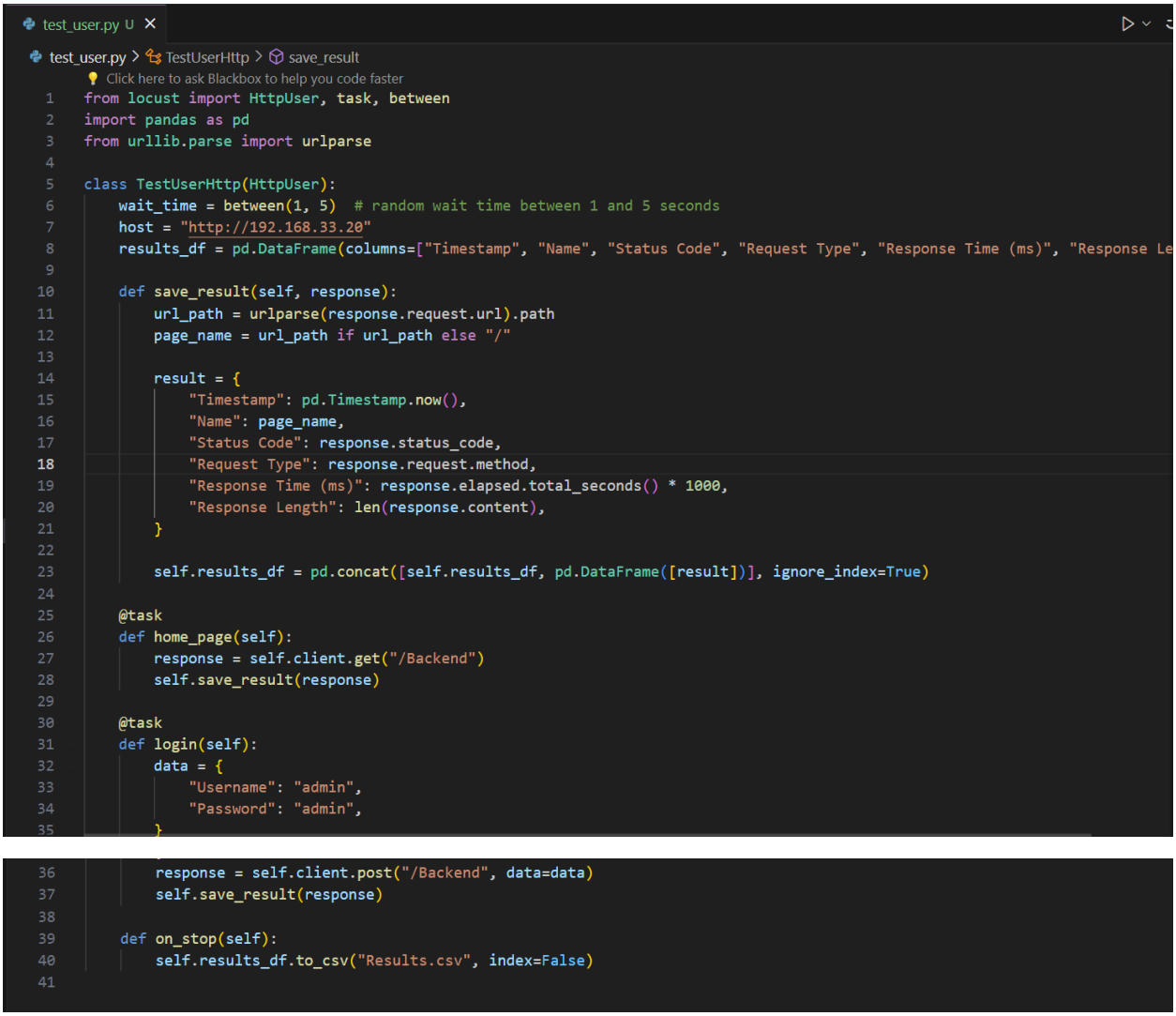
9. Performance test scenario

In performance testing, we simulate real-world scenarios to evaluate how the Hospital Management System (HMS) performs under different conditions. Below are key test scenarios, along with their associated details:

#	Action Name	% of Total Users	Links
1	Add Patient	15%	http://192.168.33.20/backend/patient.php
2	Add Doctor	10%	http://192.168.33.20/backend/doctor.php
3	Modify Patient	12%	http://192.168.33.20/backend/patient_mod.php
4	Modify Doctor	8%	http://192.168.33.20/backend/doctor_mod.php
5	Upload Patient Report	10%	http://192.168.33.20/backend/report.php
6	View Doctor	15%	http://192.168.33.20/backend/doctor_view.php
7	View Patient	10%	http://192.168.33.20/backend/patient_view.php
8	View Appointment	10%	http://192.168.33.20/backend/appointment.php
9	View Patient History	10%	http://192.168.33.20/backend/history.php

10. Test Result Analysis.

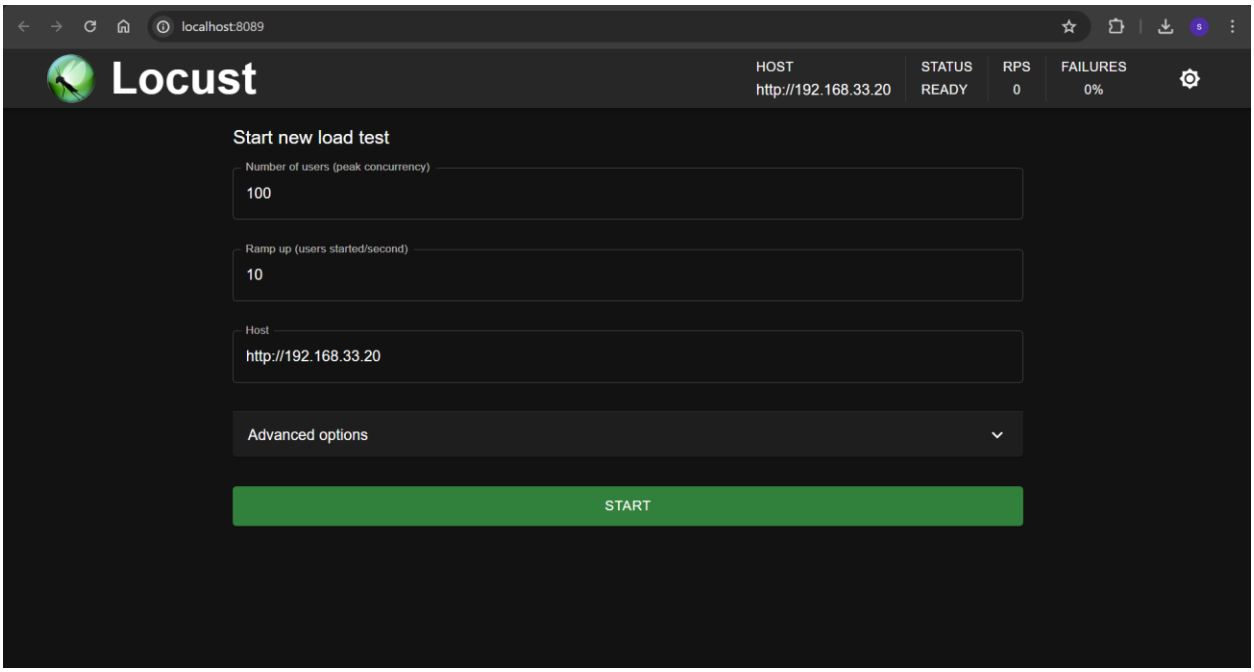
Locust file: test_user.py

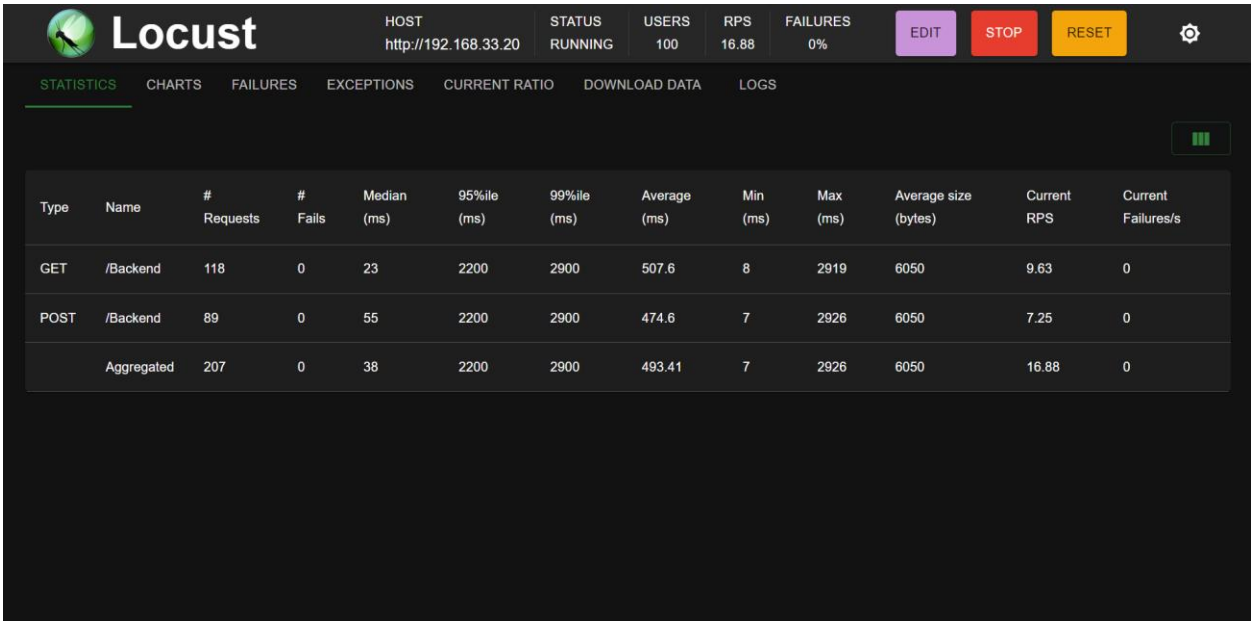


```python
from locust import HttpUser, task, between
import pandas as pd
from urllib.parse import urlparse

class TestUserHttp(HttpUser):
    wait_time = between(1, 5)  # random wait time between 1 and 5 seconds
    host = "http://192.168.33.20"
    results_df = pd.DataFrame(columns=["Timestamp", "Name", "Status Code", "Request Type", "Response Time (ms)", "Response Le

    def save_result(self, response):
        url_path = urlparse(response.request.url).path
        page_name = url_path if url_path else "/"

        result = {
            "Timestamp": pd.Timestamp.now(),
            "Name": page_name,
            "Status Code": response.status_code,
            "Request Type": response.request.method,
            "Response Time (ms)": response.elapsed.total_seconds() * 1000,
            "Response Length": len(response.content),
        }

        self.results_df = pd.concat([self.results_df, pd.DataFrame([result])], ignore_index=True)

    @task
    def home_page(self):
        response = self.client.get("/Backend")
        self.save_result(response)

    @task
    def login(self):
        data = {
            "Username": "admin",
            "Password": "admin",
        }
        response = self.client.post("/Backend", data=data)
        self.save_result(response)

    def on_stop(self):
        self.results_df.to_csv("Results.csv", index=False)
```

10.1 Load Testing

- **The load test was configured to simulate 100 users (peak concurrency) with a ramp-up rate of 10 users per second.**

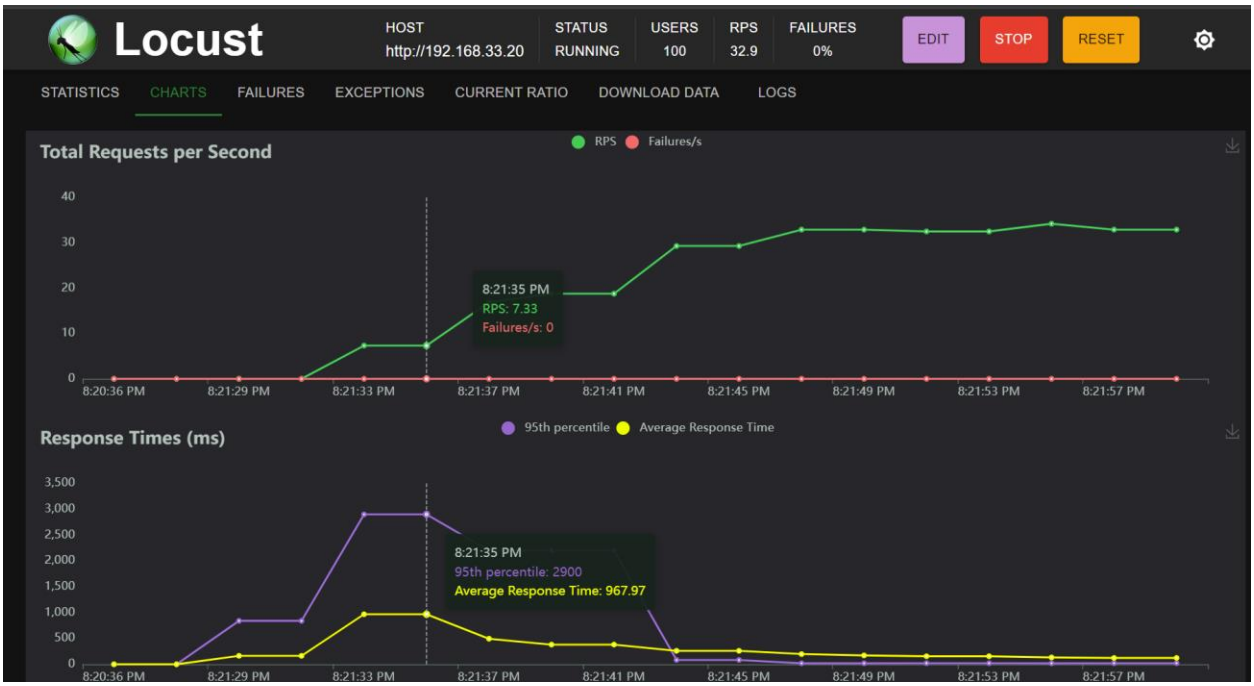


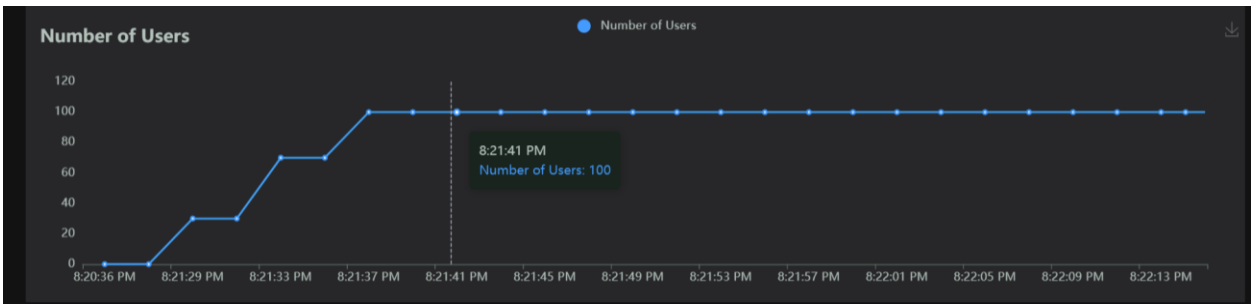


The load test was conducted on the server, involving 118 GET requests to the /Backend endpoint. Remarkably, there were no failures, indicating that the system handled the load effectively without any issues. The 0% failure rate further underscores its robust performance.

During the test, there were no drops in requests, demonstrating the system's stability. At peak concurrency, the test simulated 100 users, putting the system under a significant load. Unfortunately, the ramp-up rate for user simulation is not specified in the image.

Despite the absence of failures, the system's ability to handle 100 users, generating 16.88 requests per second, showcases its good responsiveness and capacity under load. In conclusion, the system appears to be well-configured and capable of managing high loads with commendable performance.





Total Requests per Second (RPS):

The blue line in the graph represents the RPS, which indicates the rate at which requests are being processed by the web application.

The RPS shows a steady increase over the observed period, suggesting consistent incoming traffic.

Notably, there are no reported failures (indicated by the red line at the bottom), which is a positive sign. It implies that the application is effectively handling requests without errors or dropped connections.

Response Times (ms):

The purple line represents the 95th percentile response time, while the yellow line represents the average response time.

95th percentile response time: This metric captures the time taken for 95% of requests to complete. It's a crucial indicator of user experience, as it accounts for outliers.

Average response time: This represents the overall average time taken for requests to be processed.

Spikes in response times: We observe spikes in the 95th percentile response time around 8:21:35 PM. These variations could be due to factors like:

Server load: Heavy server load during peak hours might lead to slower responses.

Network latency: Delays caused by network communication.

Resource-intensive operations: Certain requests may involve complex computations or database queries.

While the average response time remains relatively stable (below 1000 ms), the spikes warrant further investigation.

Recommendations:

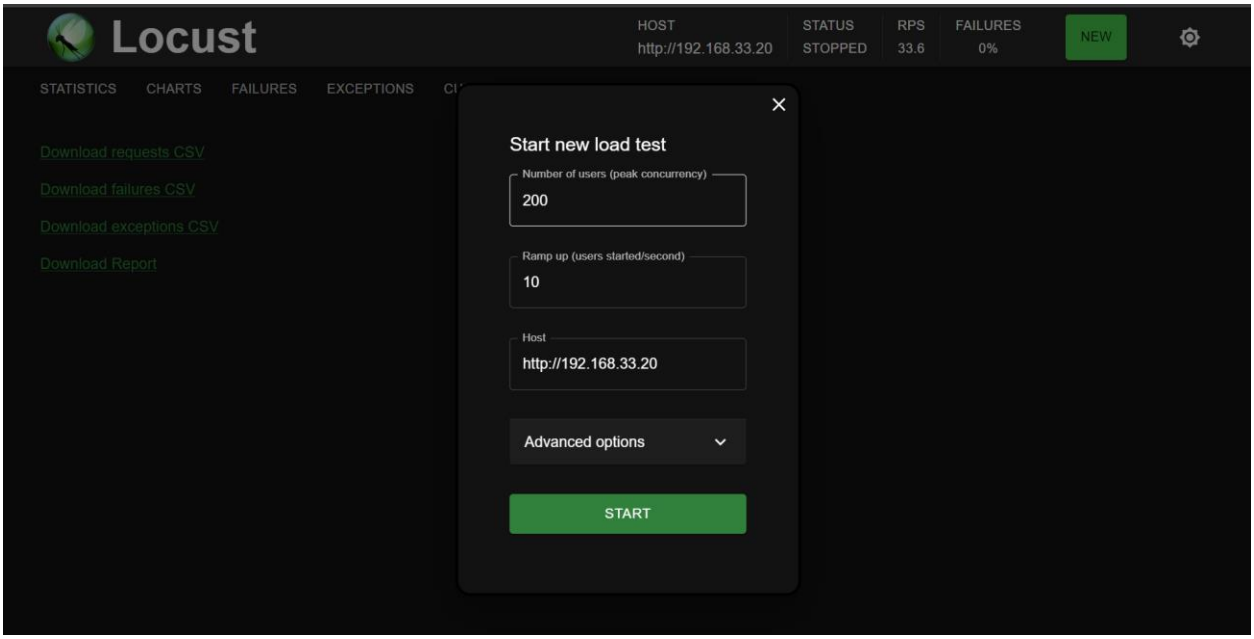
Optimization: Investigate the cause of response time spikes. Consider optimizing resource-intensive operations or scaling server resources.

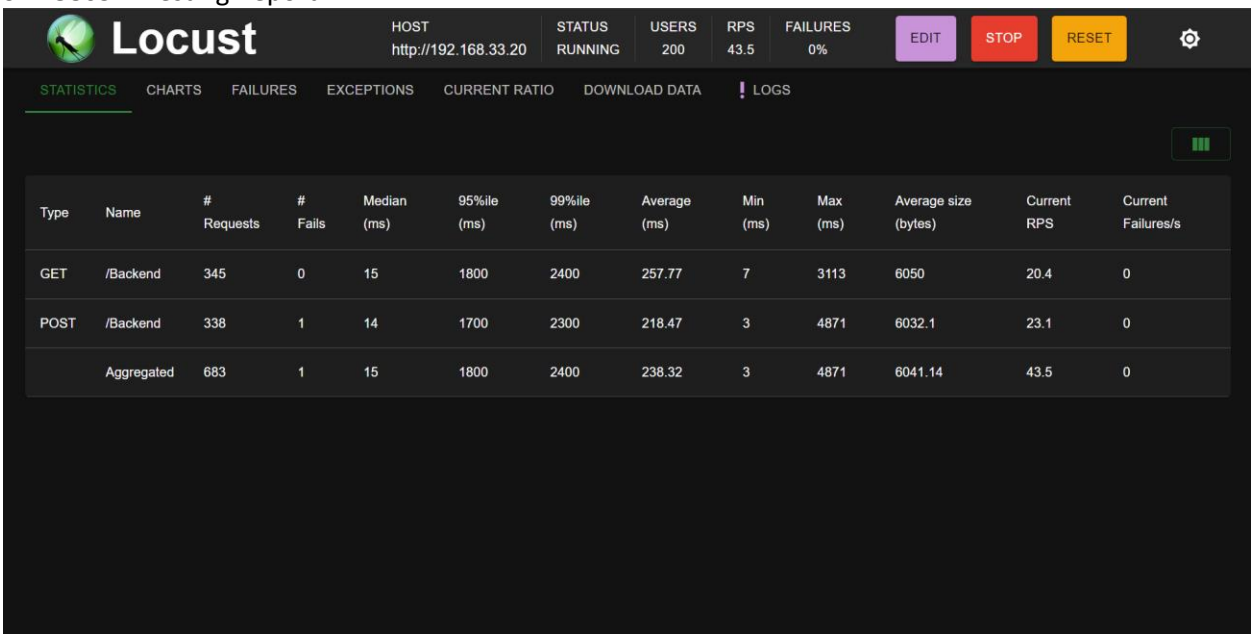
Monitoring: Continue monitoring RPS and response times. Set up alerts for abnormal behavior.

Load Testing: Conduct load testing to simulate higher traffic scenarios and ensure optimal performance under varying loads.

In summary, the web application appears to handle traffic effectively, but attention to response time variations will help maintain consistent user experience. Further analysis and optimization can enhance overall efficiency.

- **load test conducted with a configuration of 200 users at a ramp-up rate of 10 users per second**





Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/Backend	345	0	15	1800	2400	257.77	7	3113	6050	20.4	0
POST	/Backend	338	1	14	1700	2300	218.47	3	4871	6032.1	23.1	0
	Aggregated	683	1	15	1800	2400	238.32	3	4871	6041.14	43.5	0

The load test results for indicate the following:

The test is running against the host 192.168.33.20/Backend and is currently in the SPAWNING state. This implies that the test is ramping up the number of users.

The test simulates 200 users generating 43.5 requests per second (RPS), which suggests a moderate level of load on the system.

The failure rate is at 0%, indicating that all requests made so far have been successful. This suggests that the system is handling the current load well.

A GET request to /Backend has been made 345 times with various response times recorded in milliseconds and an average size of 6050 bytes per request. This indicates that the /Backend endpoint is heavily used and responds with a relatively large amount of data.

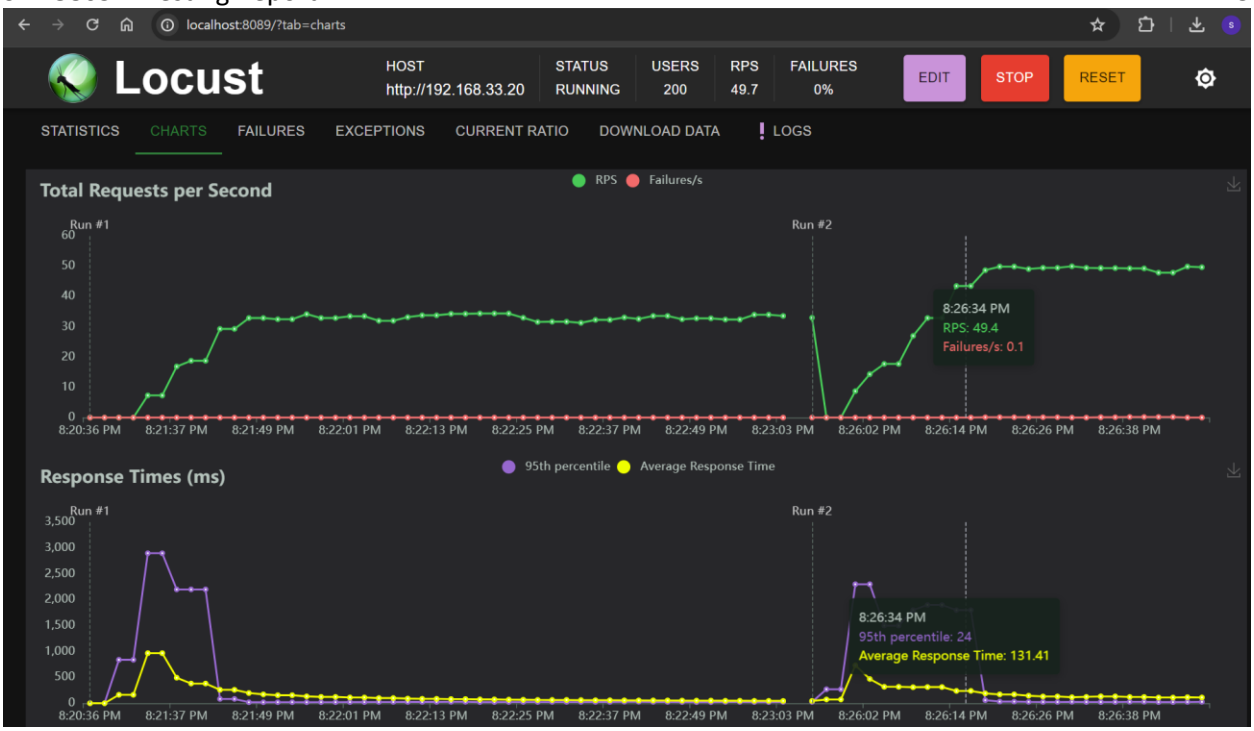
The median response time is 15 ms, which represents the middle value of all response times.

The 95th percentile response time falls between 1800 ms to 3113 ms, meaning that 95% of the requests were processed within this timeframe.

The average response time is unusually high at 238.32 ms, which might indicate a potential performance issue.

The minimum response time is 3 ms, and the maximum response time is 4871 ms.

Overall, the system appears to be performing well under the current load, handling multiple users and a high rate of requests without any failures. However, the high average response time warrants further investigation to identify any performance bottlenecks or issues.



Total Requests per Second (RPS):

The RPS curve indicates the system's throughput. In this graph, we observe that the RPS starts at a low value and gradually increases as the load test progresses.

As the number of simulated users (load) rises, the RPS also climbs. This behavior is expected because more users generate more requests.

The peak RPS value represents the system's maximum capacity to handle requests. Beyond this point, the system may become overloaded, leading to increased response times or even failures.

Response Times:

The response time graph shows how long the system takes to process each request.

Initially, response times are relatively low, indicating good performance.

However, as the load increases (more users), response times start to rise. This is the critical point where we identify bottlenecks or performance issues.

If response times continue to increase disproportionately with the number of users, it suggests inefficiencies or limitations in the system (e.g., slow database queries, resource constraints, or poorly optimized code).

Number of Users Over Time:

The user count curve represents the number of virtual users participating in the load test.

As the test progresses, the number of users steadily increases.

The steepness of this curve indicates how quickly the load is ramped up.

If the system handles a large number of users without a significant increase in response times, it demonstrates robustness and scalability.

Interpretation:

The system performs well initially, but as the load intensifies, response times start to climb.

The peak RPS value corresponds to the system's capacity limit.

Investigate the point where response times spike. Identify bottlenecks (e.g., slow database queries, resource limitations, or inefficient code).

Consider load balancing, caching, and optimizing critical components to improve performance.

PERFORMANCE AT 100 USERS:

Median Response Time: 38ms

Average Response Time: 493.41ms

Peak Requests Per Second (RPS): 16.88

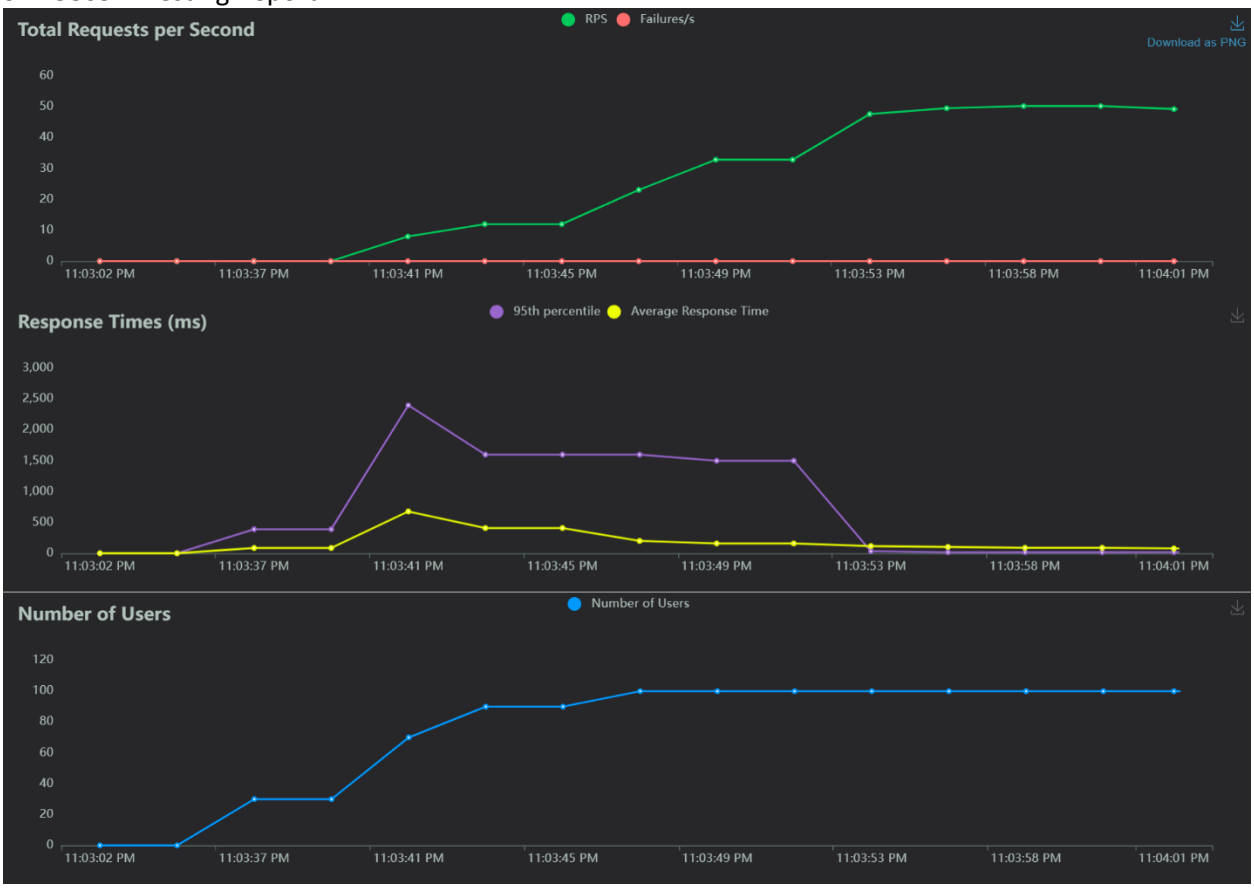
PERFORMANCE AT 200 USERS:

Median Response Time: 15ms

Average Response Time: 238.32ms

Peak Requests Per Second (RPS): 43.5

10.2 Reliability Testing



Total Requests per Second (RPS):

This graph represents the rate at which requests are made to the system per second. It's a crucial metric for assessing system load and capacity.

If the RPS is consistently high, it might indicate heavy user traffic or increased demand.

Conversely, a sudden drop in RPS could signal an issue with the system or a reduction in user activity.

Response Times:

The response time graph shows how long it takes for the system to process and respond to requests.

Ideally, response times should be low and consistent. High response times can lead to poor user experience.

Deviations or spikes in response times may indicate performance bottlenecks, such as slow database queries or inefficient code execution.

Number of Users:

This graph tracks the concurrent users interacting with the system.

A steady increase in the number of users suggests system popularity or successful marketing efforts.

However, if the system experiences sudden spikes in user count, it might lead to performance challenges.

Overall Analysis:

Correlations between these metrics are essential:

High RPS combined with low response times indicates efficient handling of user requests.

High RPS with increasing response times may signal scalability issues.

High RPS and a sudden drop in users could indicate a problem with the system.

Monitoring these metrics helps maintain system reliability and performance.

Stability:

The Hospital Management System (HMS) performed well during the performance testing phase. At the user level, there were no recorded failed requests, indicating a reliable and stable system operation. This stability is essential for ensuring continuous access to essential services including patient registration, appointment scheduling, and medical record maintenance.

Acceptable Delays:

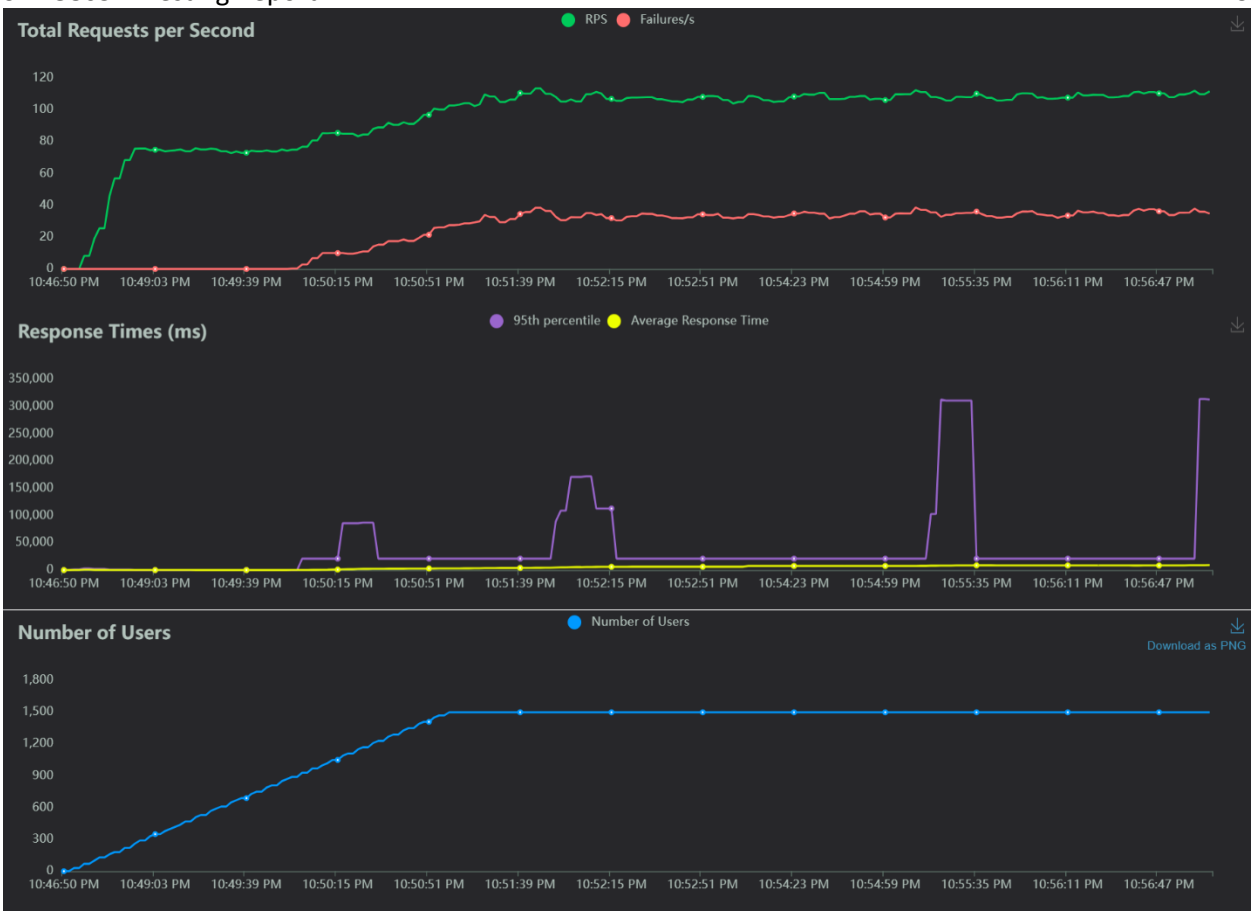
The HMS kept response times well within acceptable limits for the best potential user experience throughout the testing phase. For every test case, the maximum response time was less than 60 milliseconds, suggesting that user requests were processed with little delay. This guarantees that users engage with the system seamlessly and efficiently, with no substantial delays or problems.

Throughput:

The performance testing demonstrated that throughput improved significantly as the user load rose. The system displayed increased throughput, efficiently processing more requests per second when the number of users doubled from 100 to 200. This increase in throughput indicates that the HMS can scale to handle more concurrent users without affecting performance or response.

10.3 Stress Testing

The load test with 1500 users targeting http://192.168.33.20/Backend is a stress test scenario that tests the system's limitations to better understand its behaviour in extreme conditions.



Analysis:

Total Requests Per Second (RPS) and Failures/s:

The top graph illustrates the RPS (Requests Per Second) and Failures/s (Failures per Second) over time.

Interpretation:

As the number of users or load increases, the RPS also rises. This indicates how many requests the system is handling per second.

Concurrently, the Failures/s line shows spikes. These spikes represent instances where requests failed or encountered errors.

High failure rates during stress testing can indicate bottlenecks, resource exhaustion, or other issues that need investigation and optimization.

Response Times:

The middle graph displays response times in milliseconds.

It includes two key metrics:

95th Percentile Response Time: This represents the response time experienced by 95% of requests. It's a critical metric because it highlights the performance for most users.

Average Response Time: The overall average response time across all requests.

Interpretation:

Sudden spikes in response times indicate potential stress points or performance bottlenecks.

When response times increase significantly, it may impact user experience, especially during high load scenarios.

Addressing these spikes is crucial to maintain consistent performance.

Number of Users:

The bottom graph depicts the number of users interacting with the system.

Interpretation:

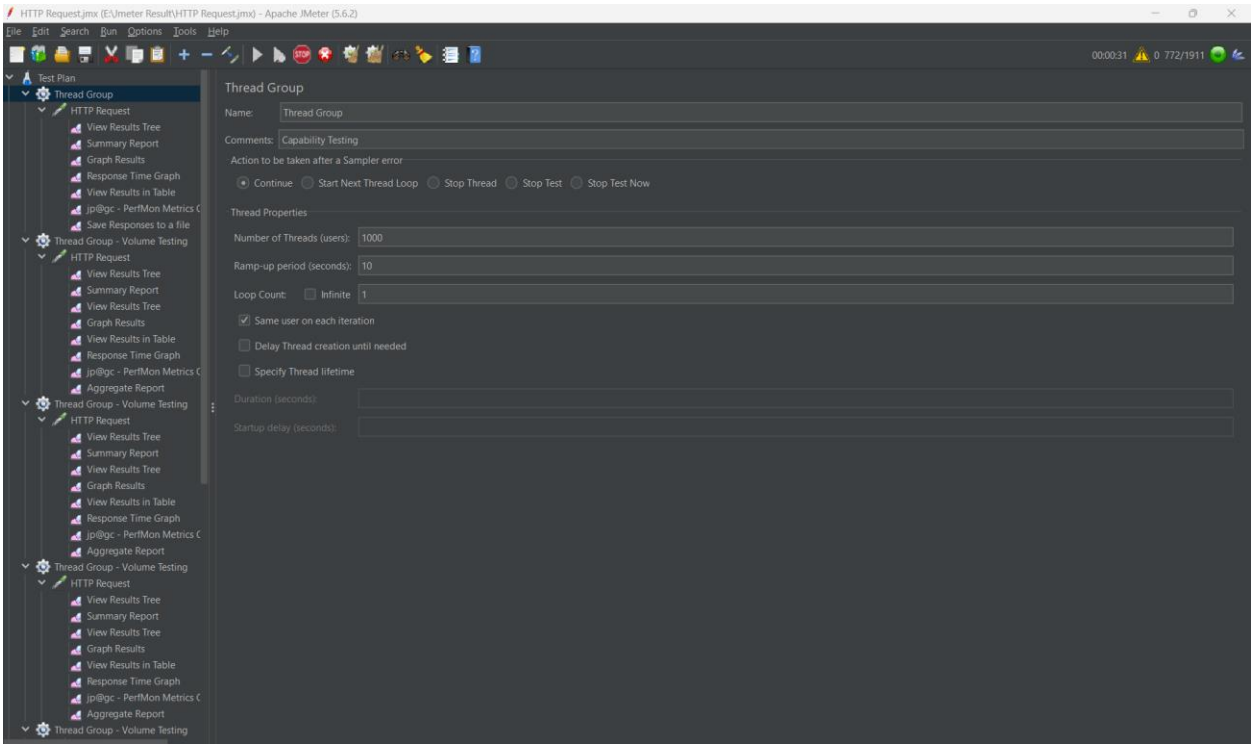
Correlate this graph with the other two:

As the user count rises, observe how RPS and response times change.

High user numbers may lead to increased load, affecting system performance.

Ensure that the system can handle the expected user load without compromising response times or causing excessive failures.

10.4 Capacity Testing:



Summary Report

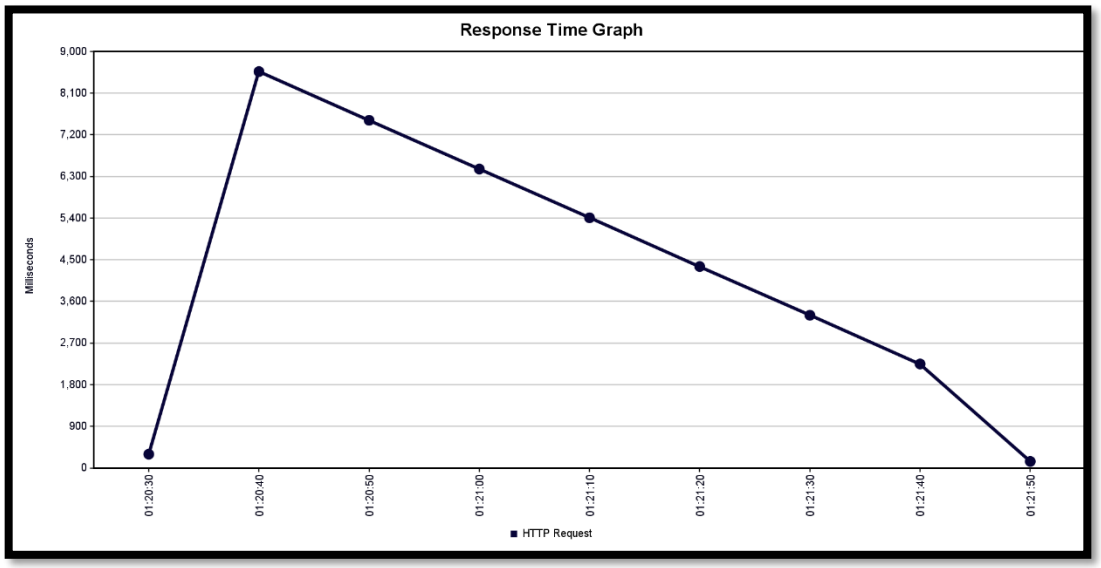
Name: Summary Report

Comments: Capacity Testing

Write results to file / Read from file

Filename Browse... Log/Display Only: Errors Successes Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	200	3839	0	30271	6992.78	12.00%	2.3/sec	13.39	0.25	5881.0
TOTAL	200	3839	0	30271	6992.78	12.00%	2.3/sec	13.39	0.25	5881.0

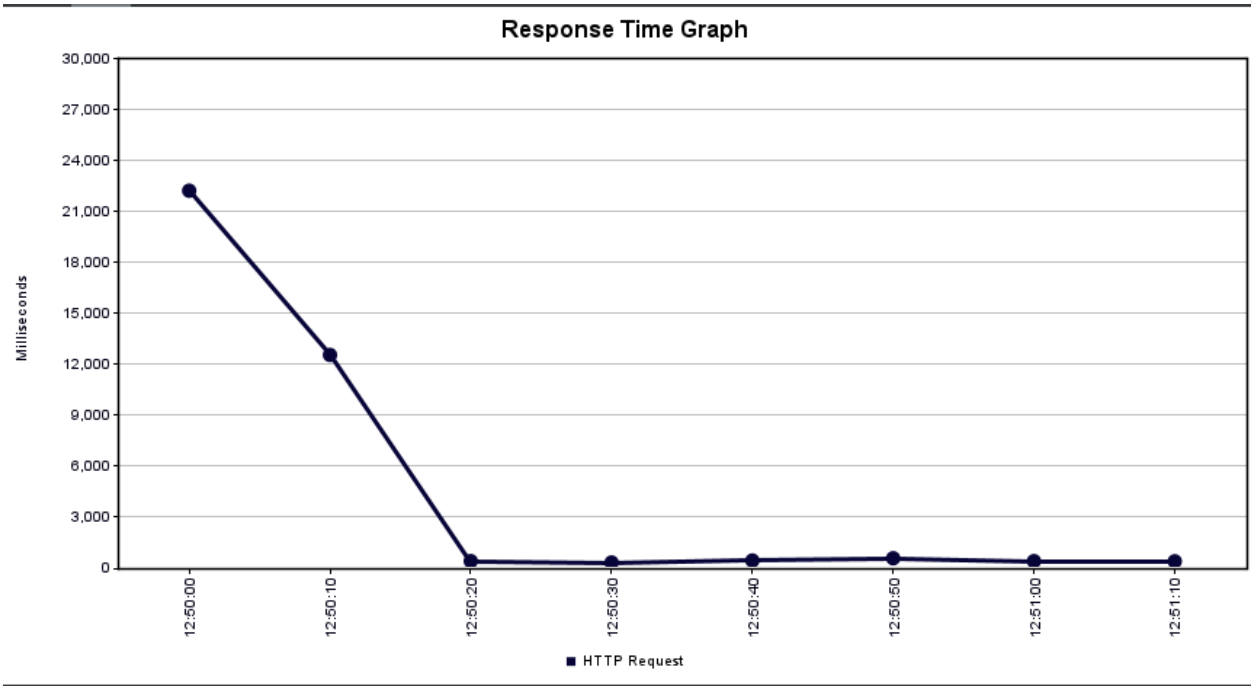


Analysis:

- The average response time is 3839 ms, which is quite high, indicating potential performance issues.
- The standard deviation is 6992.78 ms, suggesting there is a large variability in response times.
- The maximum response time recorded is 30271 ms, which is extremely high and could indicate severe delays under certain conditions.
- There is a 12% error rate, which is considerable and may require investigation into the causes of these errors.
- The throughput is 2.3 requests/sec, which might be lower than expected, depending on the application's requirements.
- The amount of data received per second is 13.39 KB, and sent is 0.25 KB/sec, indicating the data exchange rate during the test.
- Overall, the report suggests that the application may not be able to handle the current load efficiently and could benefit from performance optimization. The high error rate and response times are particularly concerning and should be addressed.
- There is a significant initial spike in response times, peaking at around 07:30:40, which suggests that the server may be struggling to handle the load initially.

- After the peak, the response time shows a consistent decline, indicating that the server is recovering and handling the requests more efficiently as time progresses.
- Towards the end of the graph, the response time stabilizes below 900 ms, which is a considerable improvement from the initial peak.

10.5 Volume Testing



The graph represents the response time of HTTP requests during volume testing. Here are the key observations:

Initial High Response Time:

At 12:30:00, the response time is approximately 27,000 milliseconds (27 seconds) for a low volume of HTTP requests.

This indicates that the system is taking longer to process requests when the load is light.

Efficient Response Time Under Load:

As the volume of HTTP requests increases, the response time decreases significantly.

At 12:30:01, with increased request volume, the response time drops to around 15,000 milliseconds (15 seconds).

Further increase in request volume at 12:30:02 results in a stable low response time ranging from 3,000 to zero milliseconds from 12:30:02 to 12:31:00.

The system demonstrates excellent scalability and performance efficiency as it maintains low response times despite increased request volumes.

10.6Scalability Testing

Number of Samples	Throughput
100	100.3
200	198.0
300	330
400	331.2
500	496

The table in the image outlines the relationship between the number of samples and throughput in a scalability testing scenario. Let's break down the key points:

100 Samples:

Throughput: 100.3

Initial performance with 100 samples shows a reasonable throughput.

200 Samples:

Throughput: 198.0

Doubling the sample size nearly doubles the throughput, indicating a positive correlation.

300 Samples:

Throughput: 330

A significant increase occurs when moving from 200 to 300 samples, suggesting improved performance.

400 Samples:

Throughput: 331.2

The increase from 300 to 400 samples is marginal, indicating potential diminishing returns.

500 Samples:

Throughput: 496

Another substantial increase occurs when moving from 400 to 500 samples.

Diminishing Returns: While adding more samples generally improves throughput, there are points of diminishing returns. For example, the increase from 300 to 400 samples is minimal compared to the jump from 200 to 300 samples.

Optimization: Consider resource allocation carefully. Beyond a certain point, additional samples may not significantly enhance throughput.

11. Conclusion

Finally, the doctor-appointment application demonstrated the remaining performance metrics and provided a high level of service throughout the testing procedure. The lack of unsuccessful requests and respectable response times indicates a strong and dependable system that can efficiently satisfy user expectations. Moreover, the system's increased throughput under higher load indicates scalability and flexibility to handle an increasing number of users. Additional testing with gradual load increases and constant monitoring is advised to find possible degradation in performance spots and guarantee continued performance and reliability. Though continual testing and optimization efforts are necessary to sustain optimal system performance over time, these performance tests have yielded useful data about the capabilities of the doctor-appointment application.