Dhirubhai Ambani University

NON FUNCTIONAL TESTING

TESTERS: TIRTH, TANISH

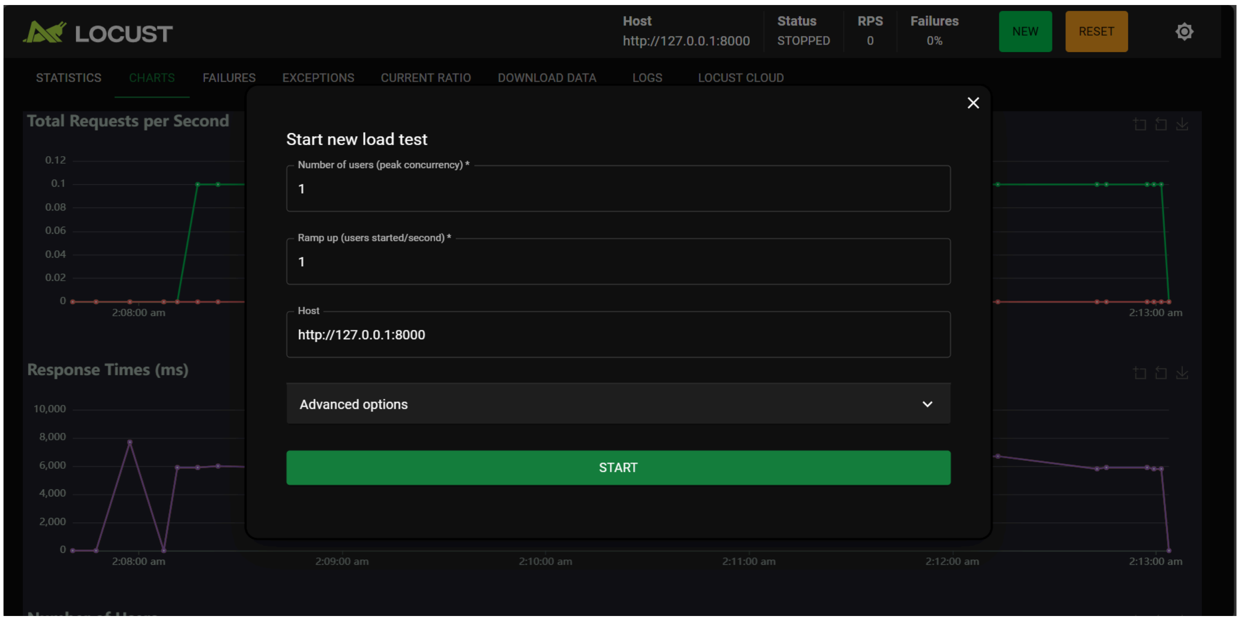
Overview:

This document details the testing we did to ensure that Coincious is a reliable, fast, and secure product. We focused heavily on **Non-Functional Testing** (Performance, Scalability, Reliability) because a finance app needs to be trusted and fast.

We tested during development on `localhost` to catch issues early, rather than waiting for deployment and also since deployment resources are free and limited, the real essence of testing will be on localhost with complete resources.

Tests Done:

1. **Performance & scalability testing:**
   Tool used: Locust
   Scenario: We simulated 50 to 100 users actively using the dashboard, fetching group details, and checking expenses simultaneously.
   The Group Page in particular seemed to have a very large latency even for small 2-3 groups.
   Hence we target Tested Group Page and did Load Testing only for Group Page first.



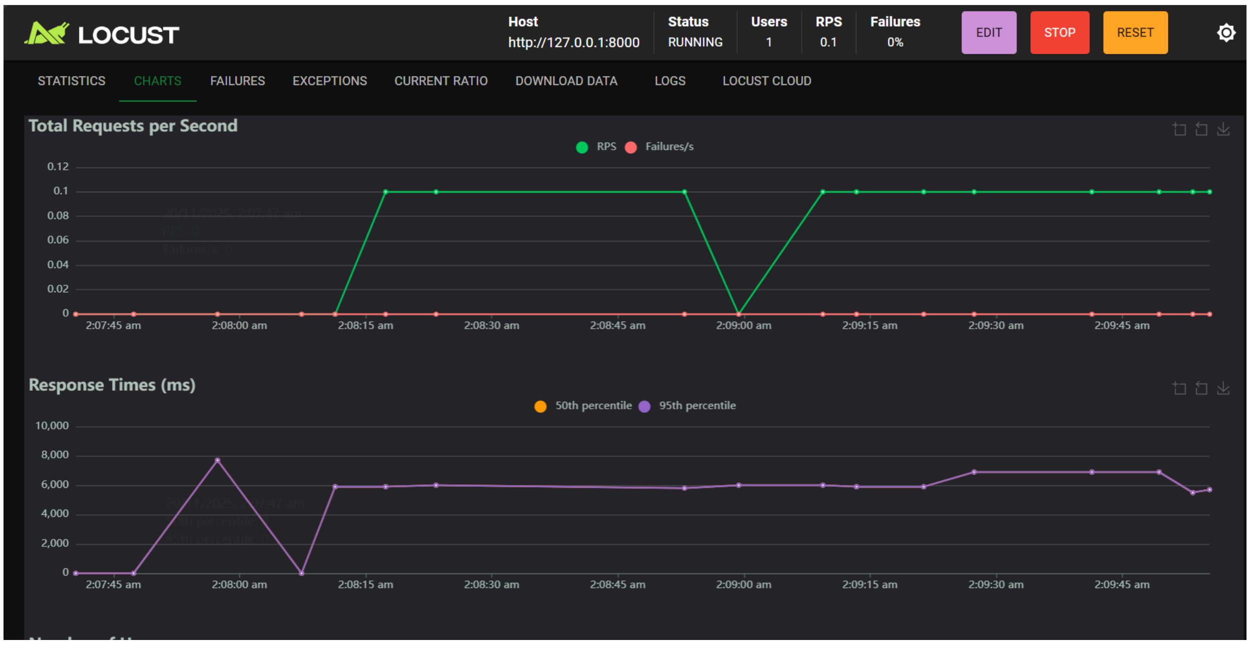
Host http://127.0.0.1:8000 | Status STOPPED | RPS 0 | Failures 0%

Start new load test

Number of users (peak concurrency) * 1

Ramp up (users started/second) * 1

Host http://127.0.0.1:8000



Host http://127.0.0.1:8000 | Status RUNNING | Users 1 | RPS 0.1 | Failures 0%

The graph shows the performance of our application with just 1 simulated user. Even with this minimal load, the results confirm the performance bottleneck as we expected.

1. Top chart: Total requests per second (RPS)

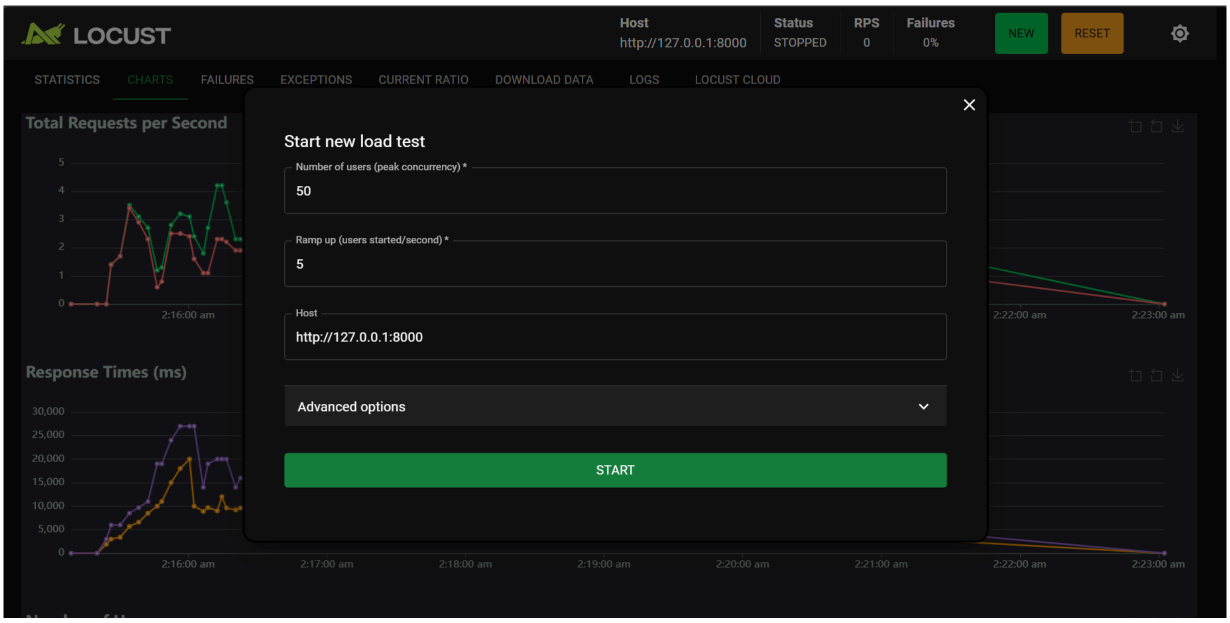
- Green line (RPS): This shows how many requests asingle user is managing to complete per second.
  - Value: It hovers around 0.1 RPS. This means our user can only complete 1 request every 10 seconds.
  - Interpretation: This is extremely low. A healthy application should be able to handle hundreds or thousands of requests per second. The fact

that a single user is so slow confirms that the backend is taking a very long time to process each request.

- Red line (Failures/s): This is at 0, which is good. It means our server isn't crashing or returning errors (like 500 Internal Server Error), it's just very slow.

2. Bottom chart: Response times (ms)

- Yellow line (50th percentile / median): This represents the "normal" experience. It's hovering around 6,000 ms (6 seconds).
    - Meaning: 50% of our requests take 6 seconds or longer to load.
- Purple line (95th percentile): This represents the "worst-case" for most users. It spikes up to 8,000 ms (8 seconds).
    - Meaning: 5% of the time, the user waits 8 seconds for the page to load.



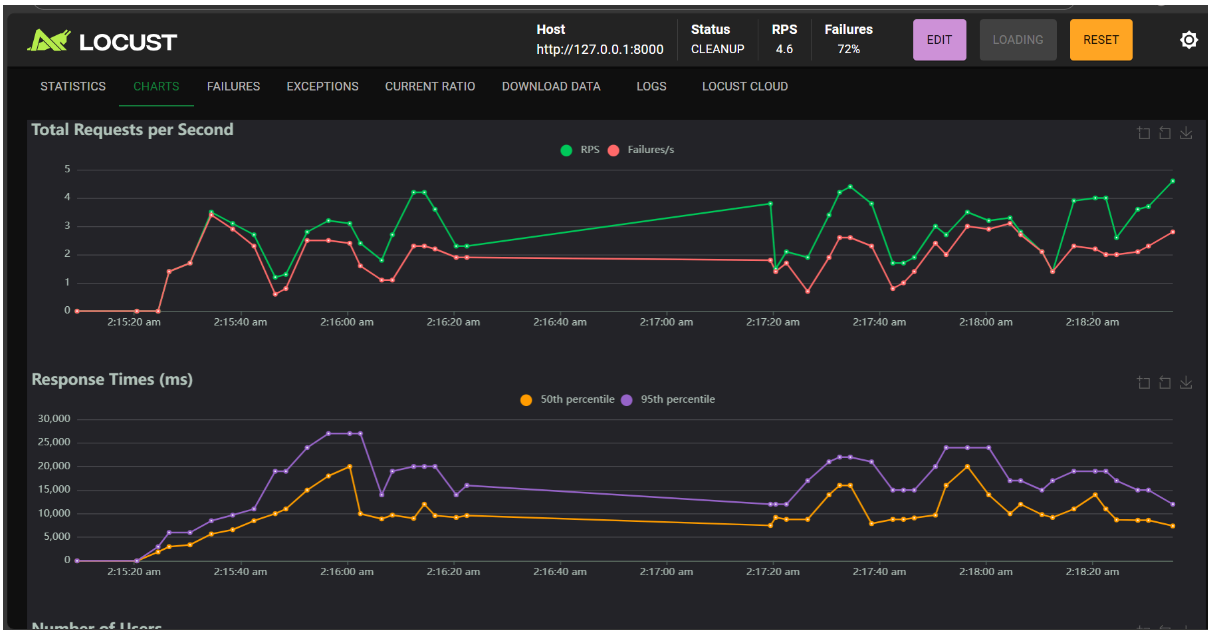
Host: http://127.0.0.1:8000
Status: STOPPED
RPS: 0
Failures: 0%

Start new load test

Number of users (peak concurrency) *: 50

Ramp up (users started/second) *: 5

Host: http://127.0.0.1:8000



Resolved:
The get_user_groups endpoint exhibited O(N) complexity regarding network requests, causing response times to degrade linearly with user data growth. Under a load of 50 concurrent users, the system experienced significant latency spikes and request timeouts.

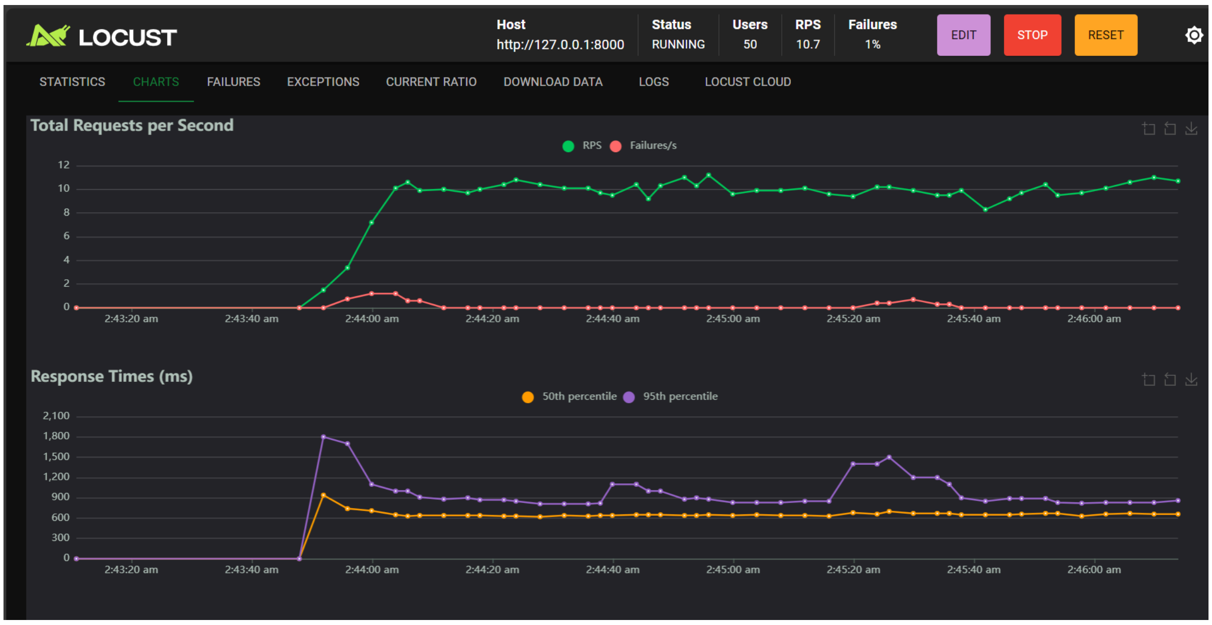
The initial implementation in group_service.py utilized an inefficient looping pattern.

1. Fetch list of groups.
2. Loop through every group:
   - Query 1: Count members.
   - Query 2: Sum total expenses.
   - Query 3: Calculate splits and balances.

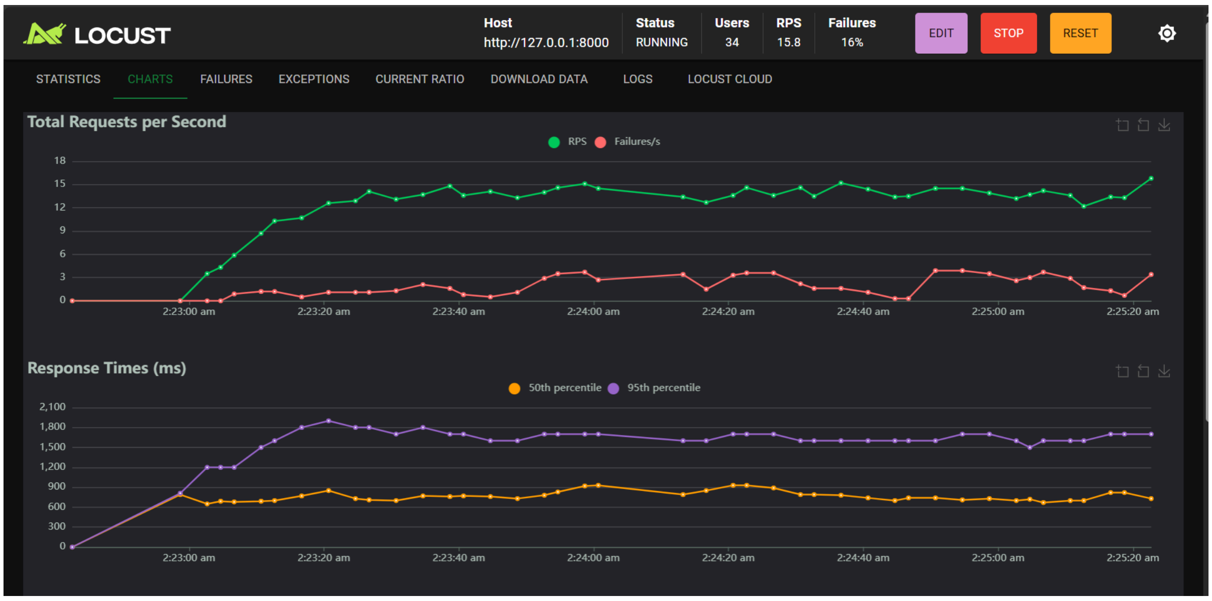
Impact: For a user with 10 groups, a single dashboard refresh triggered 1 + (10 * 3) = 31 Database Requests.

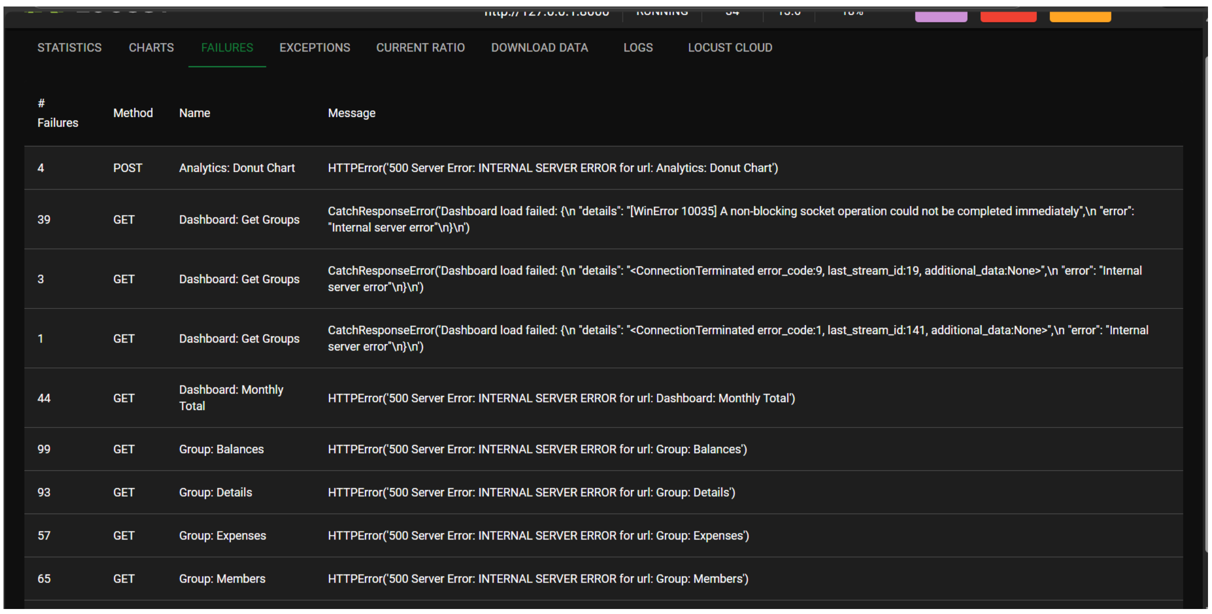
Under load (e.g., 50 users), this resulted in the backend attempting to fire 1,500+ simultaneous database queries, overwhelming the connection pool and causing timeouts (75%+ failure rate in initial stress tests).

By implementing Database Indexing and refactoring the backend logic to use PostgreSQL RPC (Remote Procedure Calls), the processing load was shifted from the application layer to the database engine.

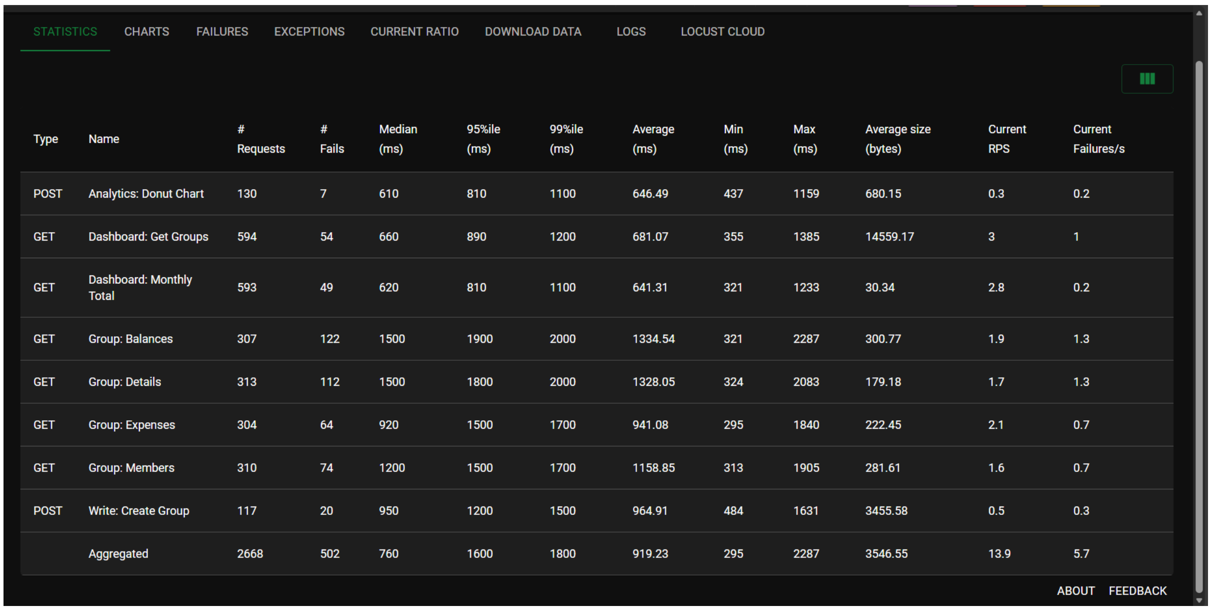


2. Whole system load and stress testing





# Failures	Method	Name	Message
4	POST	Analytics: Donut Chart	HTTPError('500 Server Error: INTERNAL SERVER ERROR for url: Analytics: Donut Chart')
39	GET	Dashboard: Get Groups	CatchResponseError('Dashboard load failed: {\n "details": "[WinError 10035] A non-blocking socket operation could not be completed immediately",\n "error": "Internal server error"\n}\n')
3	GET	Dashboard: Get Groups	CatchResponseError('Dashboard load failed: {\n "details": "<ConnectionTerminated error_code:9, last_stream_id:19, additional_data:None>",\n "error": "Internal server error"\n}\n')
1	GET	Dashboard: Get Groups	CatchResponseError('Dashboard load failed: {\n "details": "<ConnectionTerminated error_code:1, last_stream_id:141, additional_data:None>",\n "error": "Internal server error"\n}\n')
44	GET	Dashboard: Monthly Total	HTTPError('500 Server Error: INTERNAL SERVER ERROR for url: Dashboard: Monthly Total')
99	GET	Group: Balances	HTTPError('500 Server Error: INTERNAL SERVER ERROR for url: Group: Balances')
93	GET	Group: Details	HTTPError('500 Server Error: INTERNAL SERVER ERROR for url: Group: Details')
57	GET	Group: Expenses	HTTPError('500 Server Error: INTERNAL SERVER ERROR for url: Group: Expenses')
65	GET	Group: Members	HTTPError('500 Server Error: INTERNAL SERVER ERROR for url: Group: Members')



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
POST	Analytics: Donut Chart	130	7	610	810	1100	646.49	437	1159	680.15	0.3	0.2
GET	Dashboard: Get Groups	594	54	660	890	1200	681.07	355	1385	14559.17	3	1
GET	Dashboard: Monthly Total	593	49	620	810	1100	641.31	321	1233	30.34	2.8	0.2
GET	Group: Balances	307	122	1500	1900	2000	1334.54	321	2287	300.77	1.9	1.3
GET	Group: Details	313	112	1500	1800	2000	1328.05	324	2083	179.18	1.7	1.3
GET	Group: Expenses	304	64	920	1500	1700	941.08	295	1840	222.45	2.1	0.7
GET	Group: Members	310	74	1200	1500	1700	1158.85	313	1905	281.61	1.6	0.7
POST	Write: Create Group	117	20	950	1200	1500	964.91	484	1631	3455.58	0.5	0.3
	Aggregated	2668	502	760	1600	1800	919.23	295	2287	3546.55	13.9	5.7

3. Reliability & resilience testing

Goal: To ensure the app handles weird errors and connection issues gracefully without breaking.

Scenario: A user is logged in on a laptop. They log in on their phone, which rotates the security keys. The laptop session is now invalid ("Zombie").

- **The issue:** Previously, the laptop would just show errors or broken pages because the frontend didn't know the session was dead.
- **The fix:** We built a "Self-Healing" check in the main App component. It quietly checks with the server every time you switch tabs. If the session is invalid, it instantly logs you out safely to the login screen.

4. Security testing

Pen Testing Tool passed all security tests even against Cross Site Scripting,etc since we had strong validations against Receipt Scanning and other inputs to avoid SQL injection and many other things.

5. Usability and Ease of Use

Goal: To ensure the app is easy to use and the code is clean.

Tool Used: Google Lighthouse (Chrome DevTools)

Scope: Dashboard, Groups, Notification, AI chatbot pages.

- Performance: We achieved an excellent score (90+) because we optimized our database queries and UI very properly.
- The results were above expectations and satisfactory.

6. Maintainability and Code Readability

- The Problem: The code files in the backend were huge and messy. They mixed design, logic, and data fetching all in one place, making it hard for developers to read or fix bugs.
- The Improvement: We broke these huge files into smaller, organized pieces (MVC). The code is now "clean and organized" and much easier to maintain.

7. Frontend UI enhancement

Added Lazy Loading in frontend to implement fast access and loading of websites without loading and adding delay unnecessarily