

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | |
|  | | SpendSence | | | | |  | |
|  |  | | | | | | |  |
|  | | | |  |  | | | |
|  | | | | Load Test Report |  | | | |
|  | | | | May, 2025—Viktoria Todorova—First Draft |  | | | |
|  | | |  | | |  | | |

# Introduction

This report presents the results of load testing conducted on the Expense and Budget microservices of my Dockerized application using K6, an open-source performance testing tool. The primary objective of these tests is to evaluate the system’s behavior under various levels of simulated user load, identify performance bottlenecks, and validate the reliability and scalability of critical API endpoints. The tests focus on both GET and POST operations, simulating realistic traffic patterns with increasing user concurrency and duration. This ensures that the application meets expected performance standards before deployment to a production environment.

# Test Environment

The load tests were executed in a controlled development environment using Docker to ensure consistency and isolation. The Expense and Budget microservices, along with their respective dependencies (e.g., databases, authentication services), were deployed in separate Docker containers to mimic a real-world microservices architecture.

Key components of the test environment include:

* Containerization: Docker Compose was used to orchestrate the microservices and dependencies.
* Testing Tool: K6 was utilized for scripting and executing the load tests.

System Resources:

* Host Machine: 16 GB RAM, 8-core CPU, running Ubuntu 22.04
* Docker Version: 24.0+
* K6 Version: 0.48.0

Network Configuration: All containers communicated over a local Docker network to ensure reliable service discovery and consistent latency.

This environment was designed to replicate production conditions as closely as possible while allowing for safe performance validation without affecting live users.

# Tests Conducted

I ran a series of tests on the Expense and Budget microservices to see how well they handle different amounts of users and requests. The goal was to check if these services work smoothly when many people use them at the same time, and to find any problems before going live.

##### Tested Endpoints and Operations

Each microservice exposes a RESTful API with endpoints supporting standard CRUD operations. For the purpose of this test, the following endpoints were selected:

*Expense Microservice:*

* POST /expenses: Simulates the creation of an expense entry with a predefined JSON payload.
* GET /expenses: Fetches a list of all expenses, imitating a dashboard or report view.
* DELETE /expenses/{id}: Removes an expense entry, simulating a user deletion event.

*Budget Microservice:*

* POST /budgets: Adds a new budget record, representing budgeting functionality.
* GET /budgets: Retrieves a list of defined budgets.
* DELETE /budgets/{id}: Deletes a specified budget entry.

Each of these actions was tested by sending many requests, just like real users would do when using the app.

##### Load Profile and Test Configuration

The tests were set up to gradually increase the number of users and requests, then keep the load steady for a while before slowly decreasing it. This illustrates how the system behaves when demand goes up and down. The main settings were:

The tests were set up to gradually increase the number of users and requests, then keep the load steady for a while before slowly decreasing it. This helps us see how the system behaves when demand goes up and down. The main settings were:

* Ramp-Up: Start with a few users and slowly increase to the maximum over 2 minutes.
* Steady Load: Keep the number of users steady for 5 minutes.
* Ramp-Down: Gradually lower the users over 1 minute.
* Maximum Users: Up to 100 users at once.
* Request Rate: Between about 200 to 1000 requests per minute, depending on the test.
* Repeats: Each test was run multiple times to get reliable results.

Some tests focused on just one action at a time, while others combined multiple actions (like adding and getting data) to better reflect real app usage.

# Results

The load tests showed that both the Expense and Budget microservices performed well under increasing user load, but some differences were noted in how each handled the traffic.

During the initial phase with a small number of users (up to 20), response times for all key API endpoints remained very fast, **averaging under 200 milliseconds**, as you can see in *Fig. 1*. This indicates that the services can comfortably handle light usage without any delays.

As the number of users increased to 50 and then 100, response times grew but stayed mostly within acceptable limits. The Expense service’s POST requests (used to add new expenses) showed an average response time increase to about **450 milliseconds** at peak load. GET requests (for fetching expenses) were slightly faster, averaging around **300 milliseconds**. The Budget service showed similar trends, but POST requests occasionally spiked near **500 milliseconds**.

Throughout the tests, as shown in *Fig. 2*, the error rate remained very low, **below 1%**, showing that the system handled requests reliably without many failures. There were no crashes or timeouts observed, which means the system stayed stable under load.

Regarding resource usage, CPU and memory consumption increased predictably with the load but stayed below critical thresholds. CPU usage peaked around **70%** on the host machine, and memory stayed within safe limits, indicating no memory leaks or excessive resource demand during testing.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Phase | Expense POST | Expense GET | Budget POST | Budget GET |
| 10 Users | 150 | 120 | 160 | 130 |
| 50 Users | 350 | 280 | 380 | 300 |
| 100 Users | 450 | 300 | 500 | 320 |

*Fig. 1 - Table: Average Response Times (in milliseconds)*

|  |  |  |
| --- | --- | --- |
| Test Phase | Expense Service | Budget Service |
| 10 Users | 0.1% | 0.2% |
| 50 Users | 0.5% | 0.3% |
| 100 Users | 0.8% | 0.9% |

*Fig. 2 - Table: Error Rates*

# Overall Assessment

The Dockerized Expense and Budget microservices demonstrate solid performance under increasing load, with good response times and low error rates. This suggests that my application is well-optimized and stable for typical user concurrency expected in production environments.

##### Potential Issues

During the load testing, it was observed that the POST requests, which handle the creation of new expense and budget entries, experienced noticeable latency increases under high user load compared to the GET requests. This indicates potential bottlenecks in the write operations, possibly due to database contention, inefficient data processing, or resource constraints during data insertion. Additionally, the Budget microservice showed a slightly higher error rate, as evident in *Fig. 3* —reaching up to 0.9%—compared to the Expense service. While still relatively low, this elevated error rate suggests that the Budget service might be facing specific challenges under load, such as resource contention, concurrency issues, or handling of particular request patterns, warranting further detailed investigation to ensure optimal reliability.

*Fig. 3: Graph – Error Rate vs Number of Users*

##### Recommendations

To improve performance and fix any issues, it’s important to first look closely at how the system handles creating new data. This means checking the POST requests and database actions in both the Expense and Budget services to find slow queries, locking problems, or inefficient data handling. Improving database indexing or adding caching can help speed up writes.

Since the CPU usage reaches about 70%, scaling horizontally by adding more instances of the microservices behind a load balancer can help the system respond faster and handle more users at the same time. It’s also important to review the Docker containers to make sure they have enough CPU and memory and to watch for any memory issues.

Adding caching (like Redis) for the GET requests can make reading data faster and reduce load on the database. Running longer and more realistic load tests will help ensure the system stays stable under heavy use.

Finally, continuously monitoring error logs, especially for the Budget service, will help spot and fix problems early.

# Conclusion

The load tests showed that the Expense and Budget microservices work well and stay stable as more users use the system. Response times were mostly fast, and error rates stayed low, which means the system is ready for normal production use. However, there were some slowdowns during write operations when the load was high, and the Budget service had a slightly higher error rate. These points show where improvements can be made. By checking the database and code for slow spots, making sure the system has enough resources, and adding more instances of the services to share the load, the app can handle more users smoothly. Adding caching and running longer tests will help make the system even stronger.

Overall, the tests show that the app is ready to go live, with clear ideas on how to keep it fast and reliable as it grows.