Performance Test Summary

- **Start Time:** 26/03/2025, 19:25
- **End Time:** 26/03/2025, 19:30

◇ Key Performance Metrics

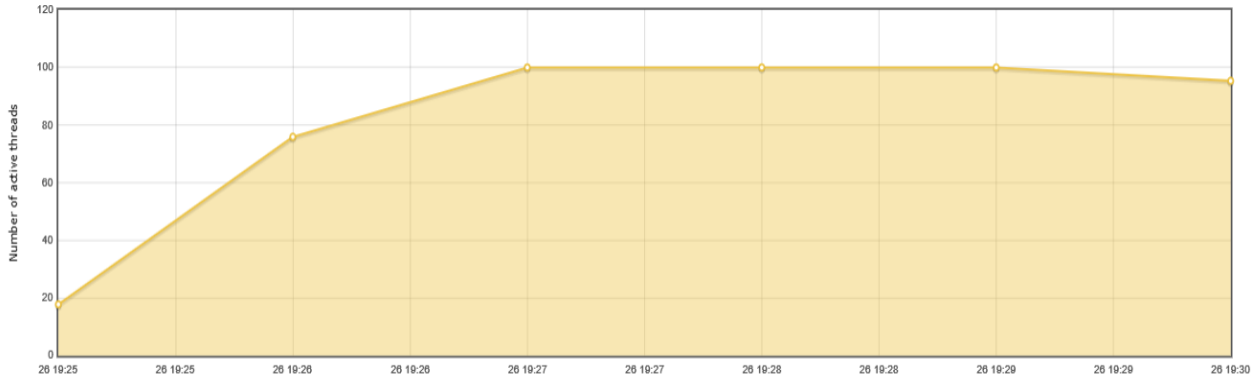
Metric	Value
Total Requests	6,226
Failed Requests	519 (8.34%)
95th Percentile Response Time	8,890 ms (8.89 seconds)
Maximum Response Time	16,897 ms (16.9 seconds)
Average Transactions per Second	20.49
Apdex	0.205

📌 APDEX (Application Performance Index) Analysis

APDEX indicates how users perceive application response times:

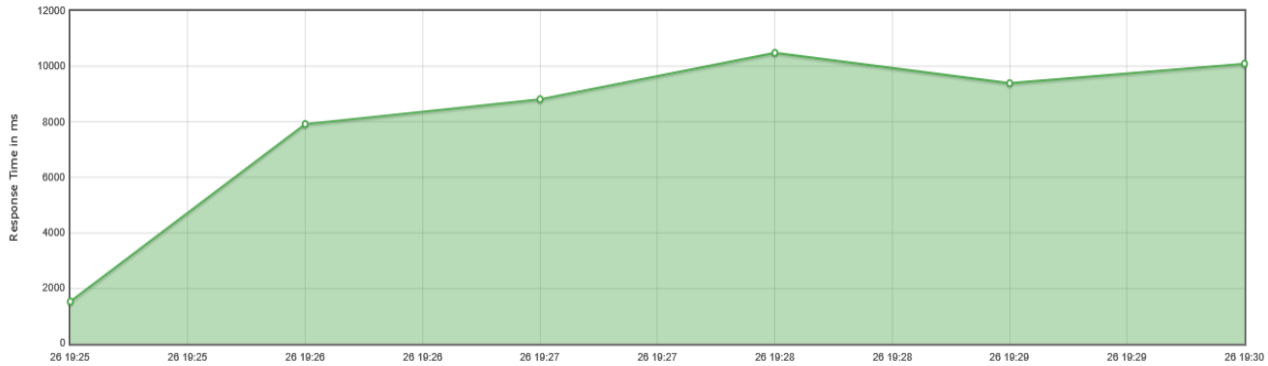
- **APDEX Total: 0.205** (very low, indicating user dissatisfaction)
- **Worst Values (0.000 - critical):**
  - `TC11 - GET /rest/saveLoginIp`
  - `TC09 - POST /rest/basket/checkout`
  - `TC08 - POST /api/Cards/`
  - **Reason:** Extremely long response times or frequent failures
- **Best Values (1.000 - excellent):**
  - `TC02 - GET /rest/admin/application-configuration`
  - `TC09 - GET /rest/track-order/`
  - `TC02 - GET /api/Quantitys/`
  - **Reason:** Very fast server response

Number of Users vs Time



On the chart “Number of Users vs Time” we can see how the number of users changed during the test.

Response Times vs Number of Users



The chart illustrates how response times of the tested application behave as the number of concurrent users increases over time.

At the beginning of the test, we observe a steady increase in response times. This rise corresponds with the ramp-up phase, where new users are gradually added to the system. As more users start interacting with the application simultaneously, the server requires more time to handle the increasing load, resulting in longer response times.

After this ramp-up period, the number of users stabilizes, and so do the response times. Although the response times remain consistently higher than at the start, they plateau and do not continue to increase significantly. This suggests that the system has reached a performance threshold where it can handle the current user load without further degradation in speed

Key Insights from the 95th Percentile:

- `TC09 - POST /rest/basket/checkout`
  has a **95th percentile of 14,204 ms**, meaning that 5% of users are waiting over **14 seconds** for checkout – unacceptable for an e-commerce application.
- `TC05 - DELETE /api/BasketItems`
  shows **10,613 ms**, which indicates serious performance bottlenecks in basket operations.
- `TC02 - GET /rest/products/search?q=`
  Even "quick" request has a **95th percentile of 3,477 ms**, which is still high.

📌 The **95th percentile (95th pct)** represents the response time that **95% of requests were completed within**. This is a better indicator of user experience than the **average response time** because:

Why is the 95th percentile better than the average?

- **More representative of worst-case performance:** The average response time can be skewed by a few very fast responses, making it seem lower than it really is.
- **Better for SLA compliance:** Many performance SLAs are defined based on the **95th percentile**, ensuring that the majority of users have a good experience.
- **Identifies high-latency issues:** Since it ignores the fastest 5% of responses, it helps pinpoint slow outliers affecting users.

📉 Error Analysis

- **400 Bad Request:** 304 cases (58.57%)
  - **Cause:** This error occurs primarily in **`TC03 - POST /api/BasketItems/`**, where the test script randomly selects products from the store to add to the cart. Some of these products might be **out of stock**, leading to a failed request when attempting to add them.
- **404 Not Found:** 181 cases (34.87%)
  - **Cause:** This error is mostly related to **`TC05 - DELETE /api/BasketItems`**, where the script randomly removes a product from the cart. However, if a product failed to be added in **`TC03`**, but is still displayed in the UI due to an application issue, the script might attempt to delete a product that was never successfully added—resulting in a **404 error**.
- **500 Internal Server Error:** 34 cases (6.55%)
  - **Cause:** While a small number of **500 errors** exist, they are likely a consequence of the **unexpected API behaviour** related to adding and removing products.

💡 **Conclusion:** These errors are not a sign of application instability but rather a result of the test script's **random product selection logic**. The application itself **remains stable**, and these errors can be disregarded in the overall performance evaluation.

◇ Key Issues

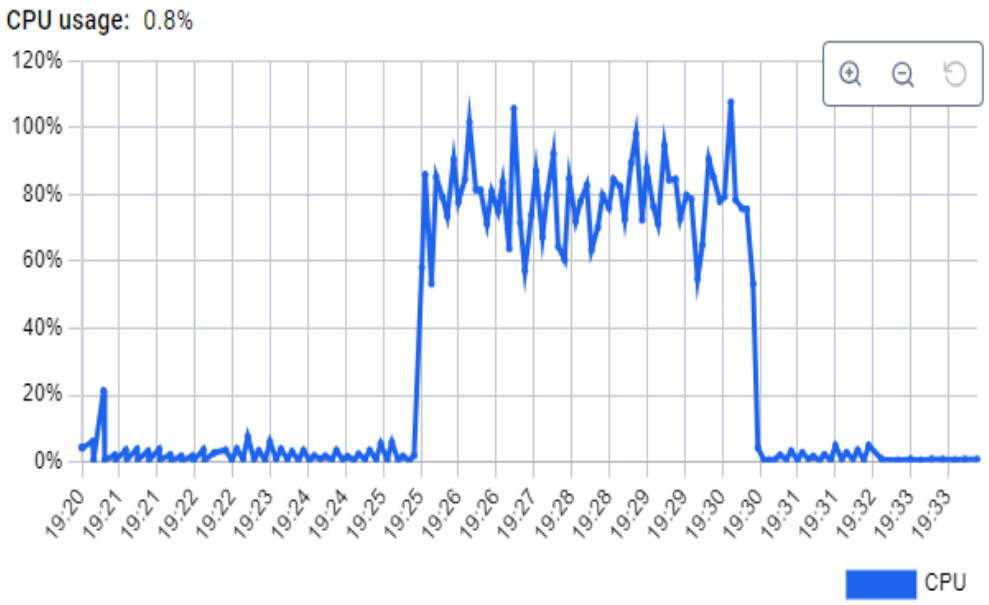
High 95th Percentile Response Times

- The **95th percentile for checkout (`/rest/basket/checkout`) is over 14 seconds**, which is extremely poor.
- Basket operations (`DELETE /api/BasketItems`) have a **95th percentile above 10 seconds**, making the checkout experience frustrating.

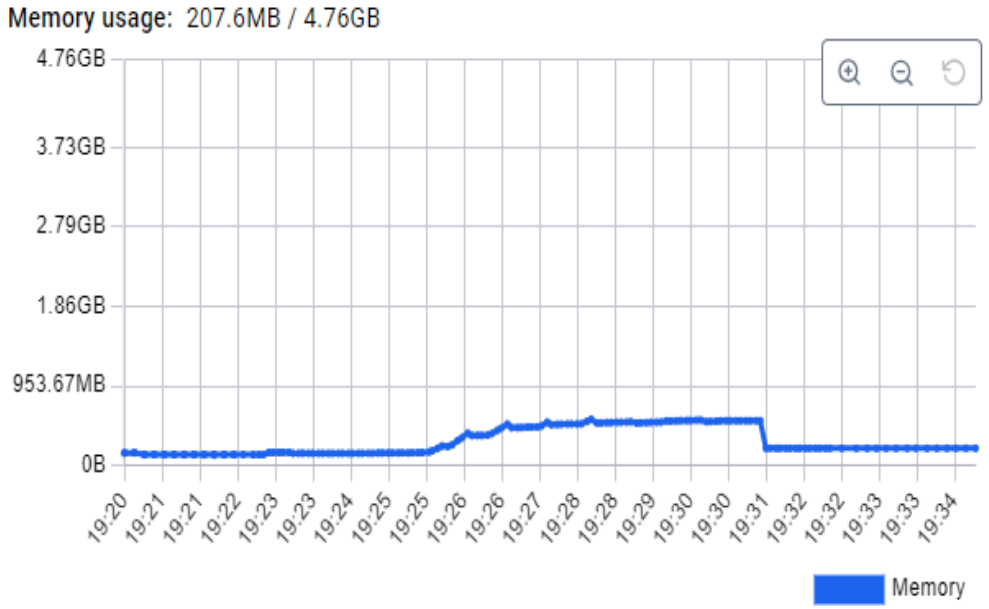
◇ Resource Utilization Analysis

During the performance test, the following resource usage was observed on the Docker container hosting the OWASP Juice Shop application:

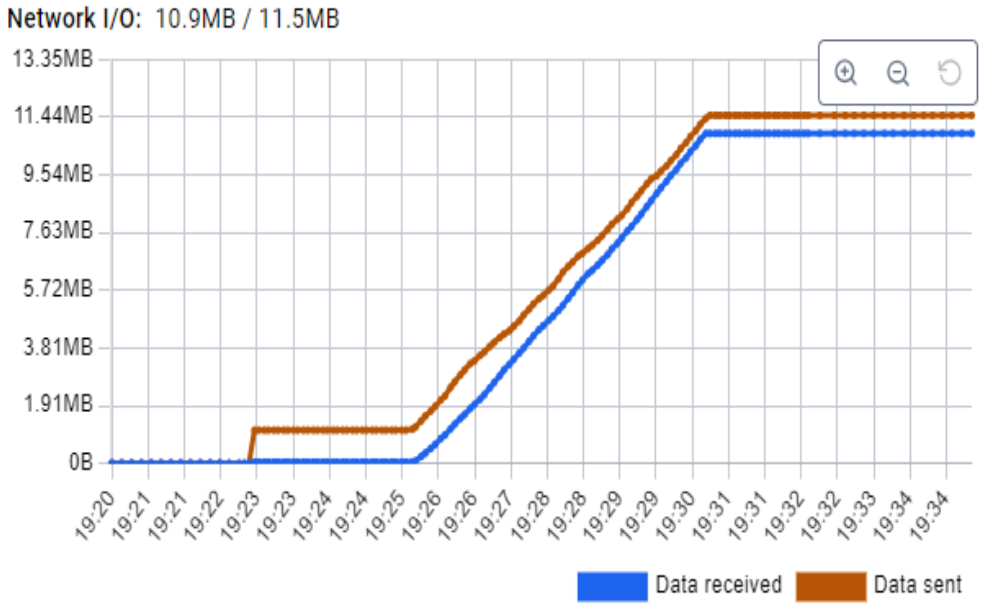
- **CPU Usage: 80-100%** (high utilization, indicating possible CPU bottlenecks).



- **Memory Usage: ~500 MB** (moderate, but could increase under higher load).



- **Network I/O: 10-11 MB** (indicating consistent data transfer during test execution).



💡 **Impact on Performance:**

The high CPU utilization likely contributed to **long response times** and **test failures**. Since CPU was nearly maxed out, the application may have struggled to process requests efficiently, causing bottlenecks and queueing delays.

📌 Summary

✅ What Works Well:

- Some endpoints (`/admin/application-configuration`, `/rest/track-order/`) perform very fast.
- Low number of 500 errors, meaning the backend rarely crashes completely.

⚠️ What Needs Improvement:

- 95th percentile response times are too high (> 8 seconds for key transactions).

🔹 Reccomendations

To improve test results and ensure more stable performance under load, it is recommended to:

1. **Allocate More Resources to the Container:**
    a. Increase **CPU and memory limits** to prevent throttling and improve request handling.
    b. Example: Assign **more vCPUs** or increase **RAM allocation** if running in a resource-constrained environment.
2. **Optimize Application Performance:**
    a. Investigate **CPU-intensive** operations in the backend.
    b. Optimize **database queries** and **API response handling** to reduce server load.

By improving resource allocation and optimizing backend efficiency, the application can handle a higher number of concurrent users without performance degradation.