UDC 004.05
DOI

METHODOLOGICAL FRAMEWORK FOR PERFORMANCE TESTING AND EVALUATION OF RESTFUL API USING AUTOMATED TESTING TOOLS

Ph.D. Khlamov S. [0000-0001-9434-1081], Mendielieva M. [0009-0002-4282-3147],
Ph.D. Vovk O. [0000-0001-9072-1634], Trunova T. [0000-0003-2689-2679],
Ph.D. Deineko Zh. [0000-0003-0175-4181], Lytvynenko S. [0009-0003-2632-9082]
Kharkiv National University of Radio Electronics, Ukraine
EMAIL: sergii.khlamov@gmail.com

Abstract. *The performance of REST API load-testing tools varies depending on workload characteristics, testing objectives, and infrastructure constraints. This study compares k6, Apache JMeter, and Postman under a controlled and unified REST API testing environment using consistent workloads and multiple load profiles. Performance and resource utilization were measured to identify differences in tool behavior and practical trade-offs. The results show that the evaluated tools have distinct performance profiles and that no single tool is optimal across all testing scenarios. SWOT analysis was used to summarize the experimental findings and identify key tool characteristics. Based on these results, a scenario-based approach was developed for four testing scenarios: latency critical, CPU constrained, memory constrained, and stress testing. The study demonstrates that load-testing tool selection should be based on the primary objective and constraints of a specific testing scenario.*

Keywords: *REST API, load testing, performance testing, resource utilization, load-testing tools, tool selection, scenario-based selection.*

Introduction

Performance testing is essential for assessing whether software systems can sustain realistic workloads and meet performance requirements [1]. Its importance has increased with the widespread of microservices, cloud platforms, and DevOps (Development and Operations), where REST (Representational State Transfer) APIs (Application Programming Interfaces) play a central role in distributed system interactions [2].

Numerous open-source and commercial tools are available for REST API load testing, differing in workload generation, scripting, resource consumption, reporting, and CI/CD (Continuous Integration/ Continuous Delivery) integration [3]. These differences can lead to different tool behavior across workload scenarios: a tool with favorable response-time performance may consume more computational resources or behave less predictably as the workload increases. The resulting research problem is to determine which load-testing tool is most appropriate for a given testing context, considering workload characteristics, REST API requirements, and infrastructure constraints. Although numerous studies have evaluated and compared performance testing tools [4], the existing literature largely focuses on specific tool combinations, individual testing scenarios, or isolated performance characteristics. Comparisons conducted under different experimental conditions also make it difficult to determine whether the observed differences are attributable to the testing tool itself, the workload profile, the characteristics of the tested API, or the underlying

infrastructure. This motivates the development of an experimentally grounded approach for scenario-specific REST API load-testing tool selection.

This monograph addresses this gap through a controlled comparison of k6, JMeter, and Postman tools using a unified REST API testing environment, consistent workloads, and multiple load profiles. The study analyzes their performance, resource utilization, and practical trade-offs to develop an approach to tool selection under different testing objectives and infrastructure constraints.

Literature review

Performance testing is essential for developing software products that meet user requirements. Web server performance directly affects the speed, efficiency, reliability, and overall user experience of web applications [3]. As demand for high-performance web services increases, evaluating web servers and applications under different workload conditions becomes increasingly important. However, scalability and maintenance challenges can limit the ability of web application architectures to satisfy growing performance and load-testing requirements, particularly in high-load environments [5]. Due to the time-consuming nature of performance testing, its integration into DevOps CI/CD pipelines has become important for accelerating software delivery and reducing testing delays [6]. Several open-source and widely used tools support web and API performance testing. Grafana k6 is an open-source load-testing tool developed in Go that combines high execution performance with ease of use and supports JavaScript-based test scripting [7, 8]. Its CLI-oriented (Command Line Interface) design, Docker support, and relatively low resource consumption make it suitable for automated and high-load testing scenarios [9, 10]. Apache JMeter is an open-source performance-testing tool supporting multiple protocols and workload configurations [11]. Apache JMeter architecture is based on the concept of emulated users. JMeter employs XML-based test plans with thread-based load generation. Test scenarios in JMeter are structured into Thread Groups, which define the number of concurrent users and the execution behavior of the workload Each emulated user is represented by a separate thread responsible for generating requests to the system under test according to the specified configuration. JMeter built-in components and extensions can be combined to define different workload patterns and reproduce complex testing scenarios. However, JMeter may exhibit high memory consumption during large-scale tests and has limitations related to real-time monitoring and the complexity of distributed test configuration [12, 13, 14]. Postman is a widely used API development and testing platform that also supports performance testing through predefined load profiles and automated JavaScript-based test scripts [15, 16, 17]. Its user-friendly interface and API-oriented workflow can simplify test development and monitoring, although limitations include resource-dependent virtual-user capacity, restricted test-data configuration, and limited support for introducing request delays in performance scenarios. Command-line execution is another important characteristic of performance-testing tools because it simplifies automated, repeatable, and CI/CD-integrated testing. CLI-based execution allows test configurations and runtime parameters to be explicitly defined and version-controlled while avoiding the additional resource overhead associated with graphical interfaces. Apache JMeter provides a dedicated command-line mode for load testing, while k6 is primarily designed around CLI-based execution. Postman

also supports automated execution through the Postman CLI, enabling collections and performance tests to be integrated into CI/CD workflows [18]. Previous studies have compared performance-testing tools under different workloads and testing environments. Study [4] evaluated Apache JMeter, Grafana k6, Gatling, Locust, and Artillery in a Jenkins-based CI/CD pipeline using representative, maximum, and spike workloads. Grafana k6 demonstrated the highest resource efficiency across the evaluated scenarios. A comparative study [19] of JMeter, k6, Gatling, and Locust using web-game workloads showed that tool performance depends strongly on workload characteristics: k6 performed best in write-intensive scenarios, JMeter in read-intensive workloads, Gatling provided efficient memory utilization under mixed loads, and Locust was suitable for resource-constrained environments. Other research reported that Postman achieved higher overall performance than JMeter and Robot Framework [20], while a comparison of Karate DSL, REST Assured, k6, and JMeter showed differences in test execution time, with JMeter requiring the longest execution time in the evaluated scenarios [3].

Overall, existing studies confirm the importance of performance testing and demonstrate substantial differences between testing tools. However, variations in testing environments, workloads, and evaluation criteria make their results difficult to use as a direct basis for tool selection. Existing research often evaluates tools according to general performance characteristics rather than considering specific workload profiles, API requirements, and infrastructure constraints.

Controlled experiments using a common environment and consistent workloads can provide a more comparable basis for evaluation. Therefore, this monograph combines controlled experimental evaluation with systematic analysis of tool characteristics. SWOT analysis is used to interpret the experimental findings, which are subsequently applied to develop a scenario-based approach for selecting REST API load-testing tools according to the primary objective and constraints of each testing scenario.

Methodology

The experimental methodology was used to compare the performance of three load testing tools under different workload patterns. The experimental environment was deployed on a local physical server to minimize the influence of network latency and external infrastructure variability. All experiments were executed under identical hardware and software conditions to ensure reproducibility and fair comparison between the evaluated load testing tools. Experiments were conducted on a workstation equipped with Windows 11 x64 25H2, equipped with an Intel Core i9-11900H 8-core CPU and 16 GB RAM. The application under test was implemented as a RESTful API using Node.js with the Express framework. Data persistence was provided by the Better-SQLite3 embedded database. Synthetic test data were generated using the Faker library to create realistic datasets while maintaining identical input conditions across test runs (Figure 1a). The supporting services were executed in Docker containers (see Figure 1b). Resource utilization were collected using Prometheus, while Grafana was used for visualization and inspection of CPU and memory (RAM) consumption throughout each experiment.

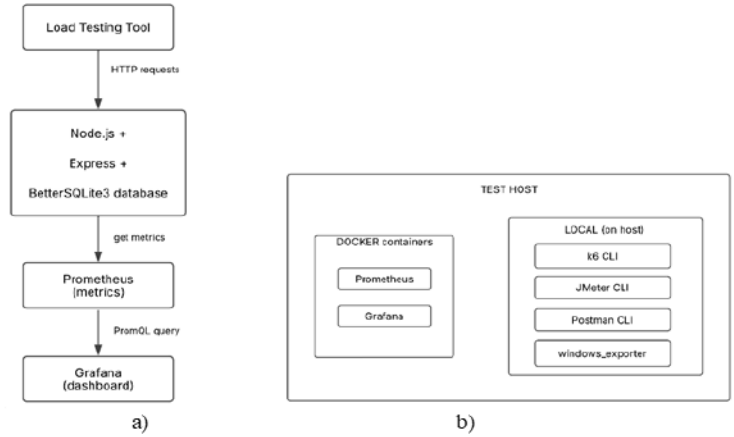


Figure 1. Experimental environment: a) architecture; b) supporting services (source: own elaboration)

The supporting services were executed in Docker containers (see Figure 1b). Resource utilization were collected using Prometheus, while Grafana was used for visualization and inspection of CPU and memory (RAM) consumption throughout each experiment. To ensure consistency, the same application version, database state, workload configuration, and monitoring setup were used for every experimental run. Each Tool × Load combination was executed 10 times under identical conditions: Node.js – v24.16.0; Express – v5.2.1; Better-SQLite3 – v12.11.1; Faker – v5.5.3; Docker – v4.81.0; Prometheus – v3.12.0; Grafana – v13.0.2. The application uses a relational database consisting of four entities: Users, Products, Orders, and Order_Items, as shown in entity-relationship (ER) diagram on Figure 2. Each user could create multiple orders, and each order contained one or more order items linked to products, representing a simplified e-commerce workflow for CRUD performance evaluation. A Better-SQLite3 database was populated with 1000 users, 3000 products, 10000 orders, and 30000 order items. Before each experiment, the database was restored from a predefined dump and the local application server was restarted to ensure identical initial conditions.

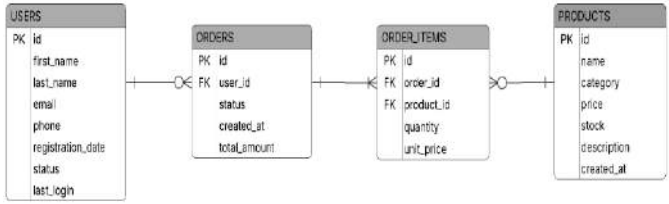


Figure 2. ER diagram of the experimental database schema (source: own elaboration)

Four workload profiles were designed to represent common performance testing scenarios: Ramp, Spike, Peak, and Fixed. Each workload profile was executed independently using all three load testing tools. For every Tool × Load combination,

10 independent experimental runs were performed to reduce the influence of random variation. Each load profile defines a different pattern of virtual user generation over time. The workload patterns used in this study are illustrated in Figure 3.

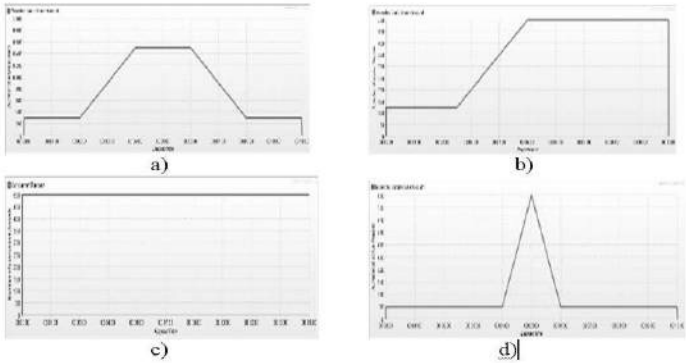


Figure 3. Load type visualization: a) Ramp; b) Spike; c) Peak; d) Fixed. (source: own elaboration)

A common logical test scenario was implemented in k6, Apache JMeter, and Postman for cross-tool comparison. Although the tools use different workload-generation mechanisms, equivalent configurations were selected to reproduce the same workload profiles (e.g., ramping-vus and constant-vus in k6, Ultimate and Concurrent Thread Groups in JMeter, and predefined load profiles in Postman). The endpoint sequence, test conditions, and iteration data selection strategy were kept consistent across all experiments. The configurations are summarized in Table 1.



Figure 4. HTTP requests used in test plan/collection for JMeter and Postman (source: own elaboration)

To ensure realistic and comparable workload generation, the scenario included product and order operations (Figure 4), multi-table database joins, and data aggregation. A fixed 3 second think time was applied between requests across all tools, using the Constant Timer in JMeter, the Postman Echo delay endpoint in Postman, and sleep() in k6.

Table 1

Workload configurations

Workload profile	Tool	Test Duration, min	Configuration
Ramping load	K6	10	ramping-vus; startVUs=125; stages: 125→500 (5 min), 500→500 (5 min)
	JMeter	10	Ultimate Thread Group: 1) start threads count = 125, initial delay =0, startup time = 0, hold load for = 600 s, shutdown time = 0 s 2) start threads count = 325, initial delay = 150 s, startup time = 150 s, hold load for = 300 s, shutdown time = 0 s
	Postman	10	Ramp-up load profile; initial load = 125, Virtual users = 500, Test duration = 10 min
Spike load	K6	10	ramping-vus; startVUs: 80; stages: 80 (4 min), 80→800 (1 min), 800→80 (1 min), 80 (4 min)
	JMeter	10	Ultimate Thread Group: 1) start threads count = 80, initial delay =0, startup time = 0, hold load for = 600 s, shutdown time = 0 s 2) start threads count = 720, initial delay = 240 s, startup time = 60 s, hold load for = 0 s, shutdown time = 60 s
	Postman	10	Spike load profile; initial load = 80, Virtual users = 800, Test duration = 10 min
Peak load	K6	10	ramping-vus; startVUs: 300; stages: 300 (2 min), 300→1500 (2 min), 1500 (2 min), 1500→300 (2 min), 300 (2 min)
	JMeter	10	Ultimate Thread Group: 1) start threads count = 300, initial delay =0, startup time = 0, hold load for = 600 s, shutdown time = 0 s 2) start threads count = 1200, initial delay = 120 s, startup time = 120 s, hold load for = 120 s, shutdown time = 120 s
	Postman	10	Peak load profile; initial load = 300, Virtual users = 1500, Test duration = 10 min
Fixed load	K6	15	constant-vus; vus: 500; duration: 15m
	JMeter	15	Concurrency Thread Group: target concurrency = 500, hold target rate = 15 min
	Postman	15	Fixed load profile; virtual users = 500, Test duration = 15 min

```
  //GET PRODUCTS
  const getProd = http.get("http://localhost:8080/api/products", {
  tags: {
    endpoint: "GET /products"
  }
});
  sleep(3);
  check(getProd, {'products are read successfully': r => r.status === 200});

  //GET ORDERS
  const getOrders = http.get("http://localhost:8080/api/orders", {
```

```
    tags: {
        endpoint: "GET /orders"
    }
});
    sleep(3);
    check(getOrders, {'orders are read successfully': r => r.status === 200});

    //GET ORDER DETAILS
                                    const                getOrderDetails                =
http.get(`http://localhost:8080/api/orders/${orderIdGet}/details`, {
    tags: {
        endpoint: "GET /orders/:id/details"}
});
    sleep(3);
    check(getOrderDetails, {'order details are read successfully': r => r.status === 200});

    //POST - CREATE ORDER
    const createRes = http.post(http://localhost:8080/api/orders',payload,
        {
            headers: {
                'Content-Type': 'application/json'
            },
            tags: {
        endpoint: "POST /orders"}
        },
    );
    sleep(3);

    check(createRes, {
        'order created': r => r.status === 201
    });
    const createdOrder = createRes.json();
    const orderId = createdOrder.orderId;
```

Listing 1. Example of part of HTTP requests sequence used in k6 test scenario

Predefined test data were selected deterministically using the iteration index (__ITER in k6 and equivalent iteration variables in JMeter and Postman) as shown in Listing 2 and Figure 5. Five values were cycled for userId, productId, and orderId to distribute database access and avoid repeatedly targeting the same records [21].

```
const users = [1, 2, 3, 4, 5];
const products = [10, 20, 30, 40, 50];
const orders = [10, 20, 30, 40, 50];

function pick(arr, index) {
    return arr[index % arr.length];
}

//flow logic
export default function() {
    const userId = pick(users, __ITER);
        const productId1 = pick(products,
__ITER);
```

```
def users = [1, 2, 3, 4, 5]
def products = [10, 20, 30, 40, 50]

def threadNum = ctx.getThreadNum()

Integer iteration = vars.getObject("iteration")
if (iteration == null) {
  iteration = 0
}

vars.putObject("iteration", iteration + 1)

int index = threadNum + iteration
```

```
   const productId2 = Number(productId1
+ 1);
      const orderIdGet = pick(orders,
__ITER);

   // PAYLOAD - CREATE ORDER
   const payload = JSON.stringify({
      userId: userId,
      status: "NEW",
      items: [
         {
            productId: productId1,
            quantity: 1
         },
         {
            productId: productId2,
            quantity: 2
         }
      ]
   });
```

a)

```
log.info("iteration = " + vars.get("iteration"))

int productId1 = products[index %
products.size()]
int productId2 = productId1 + 1

vars.put("userId", users[index %
users.size()].toString())
vars.put("productId1",
productId1.toString())
vars.put("productId2",
productId2.toString())
```

b)

Listing 2. Selection of test identifiers in: a) k6; b) JMeter JSR223 POST preProcessor

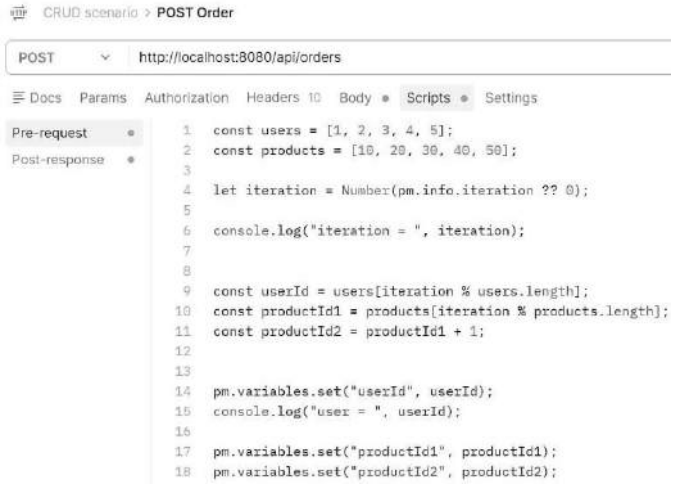


Figure 5. Selection of test identifiers in Postman pre-request script for POST request (source: own elaboration)

To ensure a fair comparison across all evaluated tools, the workload profiles were designed according to the load generation strategies supported by all platforms. In particular, the four workload profiles (Ramp, Spike, Peak, and Fixed) were adopted because they are natively supported by Postman and can be equivalently implemented in k6 and JMeter.

All experiments were executed through the CLI of each evaluated tool to ensure a consistent and reproducible execution procedure. CLI-based execution was used to provide a consistent and reproducible test procedure and to minimize the influence of interactive configuration during test execution. Tool-specific CLI commands were used to initialize the tests, configure the required workload parameters, and export the execution results for subsequent analysis. The example of commands for k6, JMeter, and Postman is provided below:

```
k6 run test-ramp-load.js --summary-export=ramp_run_1.json --out json=ramp_out_1.json

.\jmeter -n -t "D:\Ramp_Ultimate_Thread_Group_mixed.jmx" -l D:\ramp_run_1.jtl

postman login --with-api-key {api_key}
postman performance run {collection_id} --vu-count 500 --duration 10 --load-profile ramp-up
```

Metrics and statistical analysis

During each experimental run, performance metrics (average response time, 95th percentile (p95) response time, CPU utilization, and RAM consumption) together with workload execution metrics (total requests and completed iterations) and error metrics (overall error rate, HTTP errors, connection errors, and request timeouts) were recorded. The analysis was based on 10 independent runs for every Tool × Load combination. Response time metrics characterize both the typical request latency and its variability under different workload conditions, while resource utilization metrics (CPU and RAM) reflect the computational overhead introduced by each tool. To ensure consistent comparison across the evaluated tools, Postman JSONError events were classified as HTTP errors because they represent failures in client-side processing of successfully received HTTP responses rather than connection failures or request timeouts.

Since each experimental run included multiple REST API endpoints with different response times, a single performance value was calculated for every Tool × Load × Run combination. Specifically, the Metric Mean Response Time (M) was calculated as the arithmetic mean of the average response times measured for all evaluated endpoints:

$$M = \frac{1}{N} \sum_{i=1}^{N} R_i , \quad (1)$$

where R_i - denotes the average metric value of the i-th endpoint during a single experimental run; N - is the total number of evaluated endpoints.

Average response time (Avg) was used to evaluate typical system latency, while the p95 metric was analyzed to capture tail latency and identify performance degradation under load. CPU and RAM utilization were monitored on the local host using Prometheus time-series data in Grafana while executing the load tests (see Figure 6 and Figure 7). CPU and RAM utilization reported in this study represent only the resource consumption of the load-testing tool processes (k6, Apache JMeter, and Postman). Host-wide CPU and memory utilization were intentionally excluded from the analysis. Process-level metrics were collected using windows_exporter and retrieved from Prometheus via PromQL queries filtered by the process name (process="k6", process="java", process=~"postman"). During the experiments, no

other Java processes except Apache JMeter were running on the load-generation host. For both CPU and memory metrics, values were aggregated by summing all matching process instances. CPU usage was calculated from the windows_process_cpu_time_total metric using the PromQL function rate(), while memory consumption was obtained from windows_process_working_set_bytes.

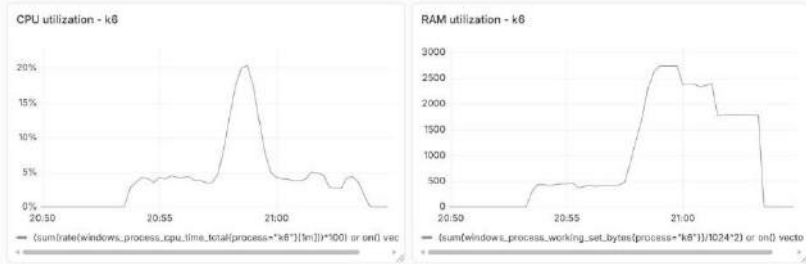
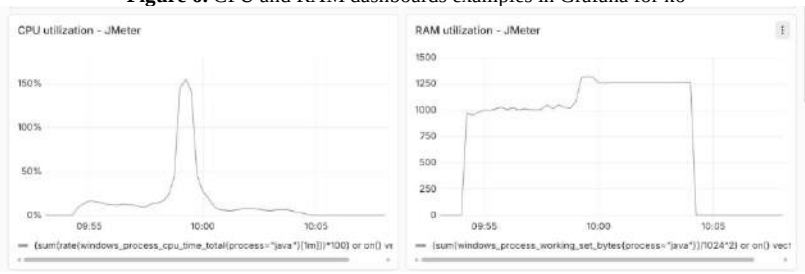
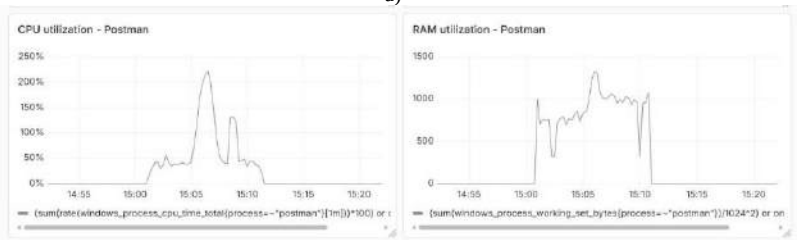


Figure 6. CPU and RAM dashboards examples in Grafana for k6



a)



b)

Figure 7. CPU and RAM dashboards examples in Grafana: a) Jmeter; b) Postman

The collected metrics served as dependent variables in the subsequent statistical analysis. Each experimental run provided raw response time and resource utilization measurements. These metrics were collected during a test run and aggregated into a single value. For statistical analysis, each row in resulting dataset represented an aggregated observation. For overall performance metrics, an observation corresponded to a Tool-Load-Run combination, while endpoint-level observations additionally included the API endpoint identifier. Average response time, p95 response time and errors were obtained directly from the summary reports exported by k6, JMeter, and Postman for each experimental run. Endpoint-level response time

metrics were extracted from the corresponding endpoint-level reports and analyzed separately by applying the same statistical procedure to each endpoint individually. For k6 endpoint-level performance metrics were obtained by post-processing the exported JSON test report with a custom Python script, which aggregated HTTP request measurements by endpoint using only http_req_duration records and calculated the corresponding response time (avg, p95) statistics. CPU and RAM utilization were obtained from Prometheus time-series data in Grafana by first aggregating all matching process instances and then calculating the arithmetic mean of the sampled values over the test duration using the Grafana Mean function on Stat panel of the created test dashboard. Experimental results were exported after each execution and processed using Python.

Scenario-Based Approach for Tool Selection

The SWOT analysis was used to develop a scenario-based approach for REST API performance tool selection under different testing objectives and infrastructure constraints. The approach selects a preferred tool for each testing scenario based on the primary criterion associated with the scenario's main testing objective (Figure 8).

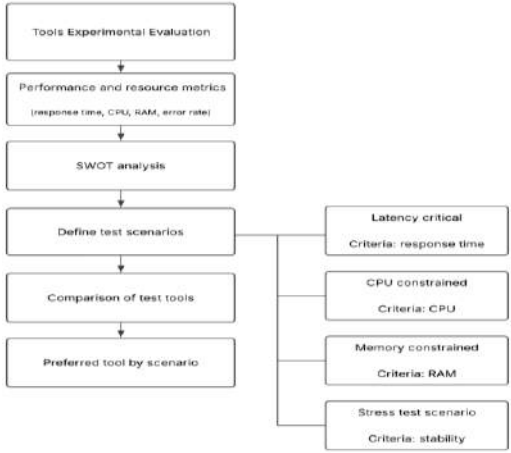


Figure 8. Scenario-Based Approach for Tool Selection

Four representative testing scenarios were defined based on the identified performance characteristics, resource constraints, and practical implications of the SWOT analysis: latency critical testing, CPU constrained testing, memory constrained testing, and stress testing under increasing workload. Each scenario represents a distinct testing objective or infrastructure constraint and therefore prioritizes a different performance characteristic. For the latency critical scenario, response time is the primary criterion. For CPU and memory constrained scenarios, CPU and memory consumption are used as the respective primary criteria. For stress testing, the primary criterion is operational stability under increasing workload. In this study, operational stability refers to the ability of a testing tool to maintain consistent operation and predictable performance as the workload increases without

critical failures. Stability is assessed based on changes in response time, resource consumption, and error rate across increasing workload levels.

For each testing scenario, the relevant performance and resource criteria are first identified according to the primary testing objective. The evaluated tools are compared directly with respect to the criterion (Table 2), and the tool with the most favorable result is identified as the preferred candidate.

Table 2

Scenario-based selection

Scenario	Testing Objective	Criterion
Latency critical	Minimize response time	Avg and p95 response time
CPU constrained	Minimize CPU consumption while maintaining competitive avg or p95	CPU utilization
Memory constrained	Minimize memory consumption while maintaining competitive avg or p95	Memory utilization
Stress testing	Maintain stable operation under increasing workload	Stability, error rate

Results

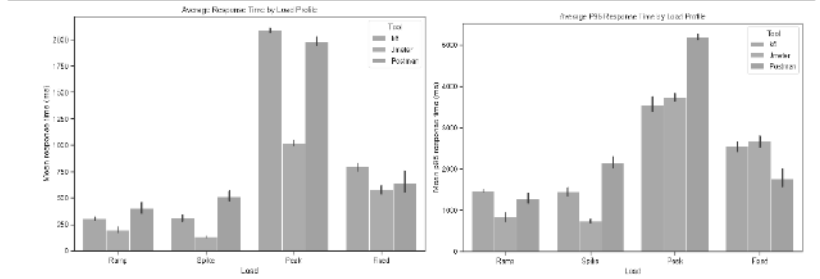
To provide an initial comparison of the evaluated performance testing tools, the average response time (avg) for each workload level and then standard deviation, and 95% confidence interval (95% CI) were calculated. These descriptive statistics summarize the overall behavior of each tool before applying statistical analysis (Table 3).

Table 3

Aggregated Average response time performance metric values

Tool	Load	Mean Avg	Standard Deviation	95% CI lower	95% CI upper	Error Rate
k6	Ramp	305.338	18.502	292.1027	318.573	0
JMeter	Ramp	194.6	45.998	161.695	227.505	0
Postman	Ramp	400.6	89.641	336.475	464.726	0.0302
k6	Spike	312.15	48.403	277.524	346.776	0.0003
JMeter	Spike	132.3	8.460	126.248	138.352	0.017
Postman	Spike	510.383	79.221	453.712	567.054	0.4052
k6	Peak	2090.722	32.580	2067.416	2114.028	15.229
JMeter	Peak	1022.1	38.869	994.295	1049.905	14.5583
Postman	Peak	1984.733	62.897	1939.74	2029.727	13.4913
k6	Fixed	794.197	63.273	748.934	839.460	0.246
JMeter	Fixed	575.6	60.959	531.992	619.208	0.7538
Postman	Fixed	637.4	166.239	518.480	756.321	1.3898

The results in Table 3 and Figure 9a presents the aggregated average response time (avg) for the evaluated performance testing tools under different load profiles. Error bars correspond to the 95% confidence interval. The results show that k6 demonstrated the highest average response time across Peak and Fixed load, while JMeter had the lowest response time across all tested load types. Postman showed higher response time for Ramp and Spike load.

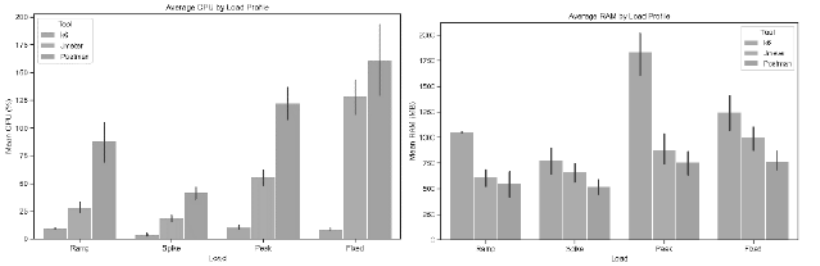


a) b)

Figure 9. Mean response time: a) Average; b) p95 response time (source: own elaboration)

Figure 9b demonstrates the aggregated p95 response time for evaluated performance testing tools under 4 workload profiles. For the Ramp and Spike workload profiles, JMeter tool consistently demonstrates lower p95 values than the other tools. The differences between p95 values of k6 and JMteter are relatively small for the Fixed profile, but become more pronounced under Peak loads for Avg metric. Postman p95 value increases more than the others for the Spike and Peak profiles, but for the Fixed profile, it is the smallest of all test tools.

K6 tool demonstrated lower CPU utilization but higher memory consumption compared with the other tools, as shown in Figure 10a. This indicates a different resource utilization pattern during workload generation by tools. In addition to higher RAM consumption (Figure 10b), k6 had a higher average response time than JMeter. This suggests that JMeter was a more resource-efficient under the evaluated conditions. Postman had the highest CPU utilization during all load types in CLI mode, but lower RAM consumption than JMeter and k6.



a) b)

Figure 10. Average resources consumption by Load profile: a) CPU; b) RAM (source: own elaboration)

Figure 11 and Figure 12 present average and p95 response times across endpoints and workload profiles. Colors denote evaluated tools, and error bars represent 95% confidence intervals. and endpoint-level analysis confirmed that tool performance depends on request type and workload profile. JMeter provided the most stable performance for both read and write operations, while k6 showed increased latency for complex GET requests under Peak load. Postman achieved competitive results only for specific retrieval scenarios under Fixed workload.

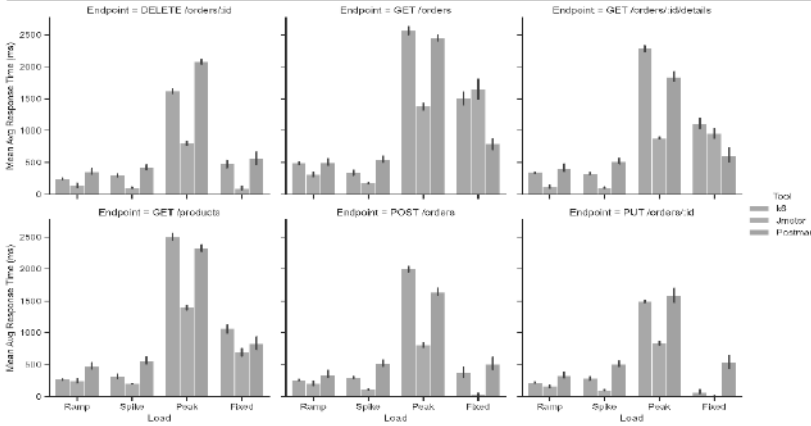


Figure 11. Mean Avg response time across endpoints (source: own elaboration)

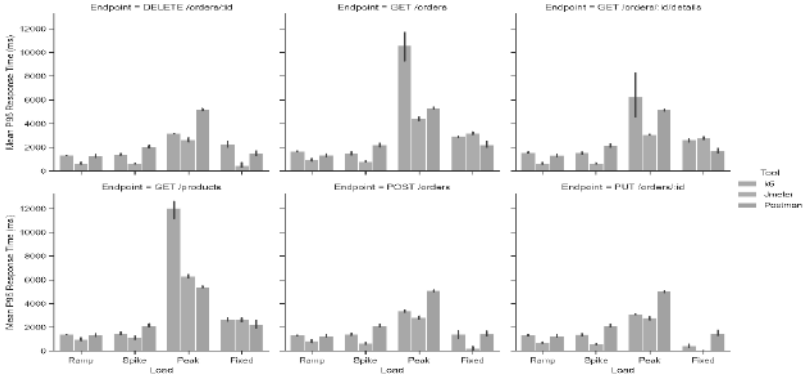


Figure 12. Mean p95 response time across endpoints (source: own elaboration).

To summarize the experimental findings and their practical implications, a SWOT analysis was performed for the evaluated performance testing tools. The analysis summarizes the principal strengths and weaknesses identified from the experimental results and relates them to potential application scenarios. Opportunities represent potential use cases derived from the observed characteristics of each tool, whereas threats reflect contextual conditions that may limit the applicability of these characteristics. Execution environment sensitivity in Table 4 refers to the potential dependence of measured performance on hardware, software, and runtime conditions. Then, based on SWOT matrix the comparative analysis was performed by considering both performance and resource-related criteria across the evaluated workload profiles. Average and p95 response times were interpreted using a lower-is-better direction, whereas stability was evaluated using a higher-is-better direction (or using the lower-is-better direction for error rate).

Table 4

SWOT matrix

	JMeter	K6	Postman
Strengths	Low latency	CPU efficiency	Efficient in read operations for Fixed load
Weaknesses	Higher CPU consumption	Higher RAM consumption	Higher latency for write operations
Opportunities	Sutable for latency-sensitive testing scenarios	Suitable for Resource-constrained testing	Suitable for lightweight testing
Threats	Sensitivity to execution environment	Memory constraints	Potentially insufficient for high-load scenarios

CPU utilization and RAM consumption were also interpreted according to the lower-is-better principle. For latency-critical testing, JMeter demonstrated the best response-time characteristics across most workload profiles, while k6 and Postman showed higher latency under specific high-load conditions. JMeter provided the most stable performance for both read and write operations, while k6 showed increased latency for complex GET requests under Peak load. Postman achieved competitive results only for read Fixed load. Regarding CPU constrained environments, k6 provides the smallest overall CPU consumption. Additionally, Postman and JMeter can be potentially applied in memory constrained test scenarios. Moreover, Postman tool can be considered the most suitable candidate for lightweight testing, where low memory consumption and ease of use are more important than CPU efficiency, for example in moderate-load test runs with simple API. k6 and JMeter can be suitable for stress testing scenario because they maintain stable operation under increasing workload levels, despite not providing the lowest resource consumption.

Conclusions

In conclusion, this monograph presented an approach to selecting REST API load-testing tools under different testing objectives and infrastructure constraints. K6, JMeter, Postman test tools were evaluated under consistent testing conditions and multiple workload profiles with respect to response time, CPU and memory utilization, and performance stability under increasing workload. The findings showed that the evaluated tools have different performance and resource profiles, making a universal ranking inappropriate. SWOT analysis was used to synthesize these differences and identify the characteristics relevant to practical tool selection. Based on the SWOT findings, a scenario-based approach was developed in which a criterion is assigned to each testing objective. Four scenarios were considered as latency critical, CPU constrained, memory constrained, and stress testing. The resulting recommendations show that JMeter might be preferable for latency-critical and stress-testing scenarios, k6 for CPU-constrained environments, and Postman for lightweight testing with moderate load. Thus, the study demonstrates that load-testing tool selection can be treated as a context-dependent decision based on the primary objective of the testing scenario rather than on a universal performance ranking. The proposed approach provides a structured basis for translating experimental measurements into practical tool-selection recommendations. Its applicability is

limited by the evaluated tools, experimental environment, and workload profiles, which provides opportunities for further validation across additional tools, APIs, workload patterns, and infrastructure configurations.

References

[1] Binder, E., Li, X., & Janes, A. (2026). An Agent-Based Approach to Automating Software Performance Testing (Work In Progress Paper). *In Companion of the 17th ACM/SPEC International Conference on Performance Engineering*. (pp. 55-61). https://doi.org/10.1145/3777911.3801107.

[2] Peldszus, S., Rutenkolk, J., Heide, M., Sollmann, J., Klatt, B., Köhne, F., & Berger, T. (2026). Developer Perspectives on REST API Usability: A Study of REST API Guidelines. *In Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering*. (pp. 966-977). https://doi.org/10.1145/3803437.3805267.

[3] Shah, Y. R. Comparative Analysis of API Test Automation Frameworks: An Experimental Study on Performance, Reliability, and CI/CD (Computer science), Ph.D. thesis, Parul University, Vadodara, Gujarat, India, 2026.

[4] Pinyagin, M., & Sadovykh, A. (2026). Automating Performance Testing in CI/CD - Tools Evaluation. In: Bonfanti, S., Papadopoulos, G.A. (eds). *Testing Software and Systems. ICTSS 2025. Lecture Notes in Computer Science*, 2025, vol 16107. Springer, Cham. https://doi.org/10.1007/978-3-032-05188-2_13.

[5] Chaudhary, P., & Banerjee, D. (2024). Continuous Performance Assurance in Cloud-Native Environments: A Testing Framework for Microservices and Serverless Architectures. *International Journal of Advanced Research in Computer Science & Technology (IJARCST)*, 7(6), 11289-11302, https://doi.org/10.15662/ IJARCST. 2024.0706014.

[6] Henning, S., & Hasselbring, W. (2022). A configurable method for benchmarking scalability of cloud-native applications. *Empirical Software Engineering*, 27(6), 143.

[7] Di Meglio, S., & Starace, L. L. L. (2024). Evaluating performance and resource consumption of rest frameworks and execution environments: Insights and guidelines for developers and companies. *IEEE Access*, 12, 161649-161669. https://doi.org/10.1109/ ACCESS.2024.3489892.

[8] Kravchenko, Y., Leshchenko, O., Trush, O., Dakhno, N., & Krasnopyorov, P. (2023). Optimizing and Improvement a Web Application Using Open Source Tools. *In IT&I*. (pp. 368-379).

[9] Hegyi, G. I., Maliosz, M., & Simon, C. (2019). Performance measurements of function as a service platforms. *In Proceedings of the 2019 3rd International Symposium on Computer Science and Intelligent Control*. (pp. 1-6).

[10] Pinastawa, I. W. R., Pradana, M. G., & Maulana, N. (2024). Web server performance evaluation: Comparing nginx and apache using k6 testing methods for load, spike, soak, and performance. *In 2024 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS)*. (pp. 1002-1007). IEEE.

[11] Leonov, S., & Tyrtyshnyi, D. (2026). Performance Optimisation of real-time data ingestion: Enhancing the InfluxDB backend listener for Jmeter. *Вісник Національного технічного університету "ХПІ". Серія: Інформатика і моделювання*, 1(15), 43-67. https://doi.org/10.20998/2411-0558.2026.01.03.

[12] Gupta, V. K., Gupta, A., Jain, P., Singh, S., & Singh, L. K. (2025). Enhancing Productivity in Software Testing Automation: A Comparative Study of Open-Source Tools for Smart Manufacturing. *In 2025 2nd Global AI Summit-International Conference on Artificial Intelligence and Emerging Technology (AI Summit)*. (pp. 1307-1312). IEEE.

[13] AutomateNow. (2023). Advantages and Disadvantages of using JMeter. https://automatenow.io/advantages-and-disadvantages-of-using-jmeter/.

[14] TestSigma. (2025). JMeter vs Postman. https://testsigma.com/blog/jmeter-vs-postman/.

[15] Sapundzhi, F., Jovanova, E., Popstoilov, M., Georgiev, S., Georgiev, I., Stefanova, T., & Todorov, V. (2025). An Integrated Approach to Evaluating E-Commerce Platforms using Postman and Selenium. *In 2025 6th International Conference on Communications, Information, Electronic and Energy Systems (CIEES)*. (pp. 1-6). IEEE.

[16] Postman. (2026). Performance Test Configuration. https://learning.postman.com/docs/tests-and scripts/performance-testing/performance-test-configuration.

[17] Patil, G. (2025). Postman. *In: Django REST APIs Demystified. Apress, Berkeley, CA*. https://doi.org/10.1007/979-8-8688-1850-9_5.

[18] Thooriqoh, H. A., Mulyo, B. M., & Rakhmadi, A. (2024). Advanced RESTful API Testing: Leveraging Newman's Command-Line Capabilities with Postman Collections. *In 2024 IEEE 10th Information Technology International Seminar (ITIS)*. (pp. 188-193). IEEE.

[19] Park, D., Kim, J. H., & Sung, H. (2025). Performance and Resource Efficiency Analysis of Load Testing Tools for Web Game Workloads. *Journal of Computer Games and Contents*, 38(7), 16-26.

[20] Hsieh, C.-H. et al. (2021). Evaluation System for Software Testing Tools in Complex Data Environment. *2021 4th International Conference on Information Communication and Signal Processing (ICICSP)*, Shanghai, China. (pp. 604-609). https://doi.org/10.1109/ICICSP54369.2021.9611846.

[21] Di Meglio, S., Starace, L. L. L., Pontillo, V., Martins, L., Di Nucci, D., & Palomba, F. (2026). A Taxonomy of Bad Practices in Software Performance Testing: Insights from Gray Literature and Practitioner Validation. *ACM Transactions on Software Engineering and Methodology*. https://doi.org/10.1145/3821538.

МЕТОДОЛОГІЧНИЙ ФРЕЙМВОРК ДЛЯ ТЕСТУВАННЯ ПРОДУКТИВНОСТІ ТА ОЦІНКИ RESTFUL API З ВИКОРИСТАННЯМ АВТОМАТИЗОВАНИХ ІНСТРУМЕНТІВ ТЕСТУВАННЯ

Ph.D. Хламов С., Мендєлєва М., Ph.D. Вовк О., Трунова Т., Ph.D. Дейнеко Ж., Литвиненко С.

Харківський національний університет радіоелектроніки, Україна

Анотація. *Продуктивність інструментів для навантажувального тестування REST API залежить від характеристик робочого навантаження, цілей тестування та обмежень інфраструктури. У цьому дослідженні порівнюються k6, Apache JMeter та Postman у контрольованому та уніфікованому середовищі тестування REST API з використанням узгоджених робочих навантажень та кількох профілів навантаження. Продуктивність та використання ресурсів вимірювалися для виявлення відмінностей у поведінці інструментів та практичних компромісів. Результати показують, що оцінювані інструменти мають різні профілі продуктивності, і що жоден інструмент не є оптимальним для всіх сценаріїв тестування. SWOT-аналіз було використано для узагальнення експериментальних результатів та визначення ключових характеристик інструментів. На основі цих результатів було розроблено сценарний підхід для чотирьох сценаріїв тестування: критична затримка, умови обмеженого CPU, умови обмеженої пам'яті та стрес-тестування. Дослідження демонструє, що вибір інструменту для навантажувального тестування повинен базуватися на основній меті та обмеженнях конкретного сценарію тестування.*

Ключові слова: *REST API, навантажувальне тестування, тестування продуктивності, використання ресурсів, інструменти для навантажувального тестування, вибір інструментів, вибір на основі сценаріїв.*