*degradation of the anthropogenic agricultural landscape and the restoration of natural vegetation cover. The information system, implemented on the basis of the developed technology, is integrated with Google Earth Engine (GEE) and enables classification of fields as cultivated or abandoned in near real time, with the results displayed on an interactive map. The technology uses satellite imagery and does not require a training dataset, which makes it possible to monitor agricultural areas even in war zones where ground surveys are difficult or impossible. Validation on a sample of croplands in Ukraine demonstrated an accuracy of up to 92% (F1-score = 0.896).*

***Keywords:*** *information technology, information system, classification, NDVI, Sentinel-2, monitoring, abandoned cropland*

UDC 004.05
DOI https://doi.org/10.36059/978-966-397-538-2-12

## ANALYSIS OF PERFORMANCE METRICS FOR LOAD TESTING TOOLS

**Ph.D. S. Khlamov**[1[0000-0001-9434-1081]], **M. Mendielieva**[1[0009-0002-4282-3147]], **Ph.D. O. Vovk**[3[0000-0001-9072-1634]], **Ph.D. Zh. Deineko**[4[0000-0001-6747-9130]], **S. Lytvynenko**[5[0009-0003-2632-9082]]

*Kharkiv National University of Radio Electronics, Ukraine*
*EMAIL: [1]sergii.khlamov@gmail.com, [2]mariia.mendielieva@nure.ua,*
*[3]oleksandr.vovk@nure.ua, [4]zhanna.deineko@nure.ua,*
*[5]serhii.lytvynenko@nure.ua*

***Abstract.*** *This study presents a detailed comparative analysis of two widely used tools, JMeter and Postman, for performance testing of application programming interfaces (APIs). As APIs form the backbone of modern distributed applications and cloud-based services, ensuring their reliability and efficiency under different traffic conditions is of significant importance. To address this, performance metrics, including average, minimum, and maximum response times, as well as error rates, were systematically collected and evaluated from five publicly available APIs representing diverse functional domains. The experimental results demonstrate apparent differences in the behavior of the two tools depending on the load intensity. Postman exhibits better stability and efficiency under low and moderate load conditions, which makes it suitable for steady-load testing and routine validation of API endpoints during development. In contrast, JMeter demonstrates superior performance in high-load and peak-load scenarios, highlighting its capability to simulate concurrent user actions and stress-test applications at scale. Furthermore, the performance deltas observed during the experiments indicate that JMeter provides a more accurate model of applications in cases where user interactions introduce delays, making it particularly useful for event-driven or session-based systems. These findings emphasize that no single tool is universally*

*optimal, and the selection should depend on the specific testing goals and operational context. By providing empirical evidence of the strengths and limitations of both tools, this study offers practical guidance for developers and QA engineers in choosing not only the appropriate tool but also designing effective testing strategies to ensure system robustness and scalability.*

***Keywords:*** *Performance testing, API testing, load testing, system stability, Postman, JMeter, response time analysis, percentile metrics, performance evaluation, decision-making, software testing*

## 1 Introduction

Performance testing of a web Application Programming Interface (API) involves assessing its response time, reliability, scalability, and resource utilization to evaluate its overall performance. API performance tests evaluate how effectively an API handles high traffic and transactions while maintaining performance standards, helping identify bottlenecks and potential issues in its design and execution [1].

Representational State Transfer (REST) APIs are widely used to build modern microservices and cloud applications [2]. They are available in every programming language, and their interface enables a more efficient and effective development process. Numerous performance tools are available for API performance testing, each offering unique features and capabilities [3].

Effective microservices performance testing requires a combination of appropriate tooling and methodological approaches that account for the distributed nature of request processing in microservices architectures [4, 5].

Several levels of performance testing maturity have been identified [6]. Depending on a company's size, project capabilities (resources allocated for testing, budget, etc.), and the maturity of the performance testing process, different outcomes can result. For example, some organizations might discover only a small number of performance defects after deployment (5%). In comparison, others may find a significant number of performance defects in production (30%) due to the limited time allocated for performance testing, which is often performed late in the development cycle. In contrast, some performance defects may only be addressed in the live environment, exposing the application to serious risk.

Software applications that utilize APIs vary significantly, ranging from very simple to large and complex systems, with different numbers of end users located in various regions, and supporting varying levels of concurrent usage. Therefore, depending on these factors and the available resources for testing, different performance testing tools may be employed. Each tool has its own strengths and weaknesses, which must be considered when integrating it into the performance testing process. When selecting a suitable performance testing tool, consider the project's performance testing goals and the tool's capabilities.

Traditional performance testing processes often rely on synthetic scenarios and lengthy testing procedures, making them challenging to adopt when testing needs to be realistic and efficient [7]. Moreover, during the organization of performance testing, some issues related to tools can be observed, such as tool installation, the

tool's flexibility in the application, and the response time generated by the tool [8]. Thus, there is a need to understand the real effectiveness of performance test tools.

Furthermore, with the widespread adoption of agile methodologies, development processes can now run faster and iteratively [9]. Continuous Integration and Continuous Delivery (CI/CD) are methods used in Agile development to automate and speed up the software development and testing process. Hence, the speed or performance of a test tool for API testing can significantly impact its effectiveness.

Studies [3, 10] indicate that Postman and JMeter are highly popular tools for API and performance testing, with usage rates of 45% and 20%, respectively. Additionally, these test tools have different learning curves, with JMeter's being particularly steep [11], whereas Postman is designed to simplify the testing process of APIs [12].

Both test tools have different adoption rates in development teams, with Postman having a higher rate (78%) and JMeter a slightly lower one (59%) [10].

Postman has a very friendly interface, and it is very popular for API testing. Postman provides built-in performance testing features, including load testing, to evaluate API performance under simulated traffic. This helps identify bottlenecks and ensures APIs can handle loads using load profiles. [13]. The core functionality of Postman is built around the Hypertext Transfer Protocol (HTTP), HTTPS, and WebSocket protocols. Postman is highly effective for functional API testing, seamlessly integrating with CI/CD pipelines via the Newman CLI tool, and providing excellent opportunities for team collaboration. In the context of performance testing, it provides a basic test setup and is suitable for small to medium-sized projects.

Apache JMeter is a performance testing tool that supports various APIs, including Simple Object Access Protocol (SOAP) and GraphQL [14]. It is well-known for its realistic load testing features and integration with CI/CD tools. It's suitable for quickly identifying performance issues in high-load systems [15]. JMeter offers flexible configuration, and it is effective for more complex projects that require precise performance metrics. With the help of JMeter, distributed load testing with more than 10,000 VUs can be conducted, making it a successful tool for large projects with high requirements for accuracy and detailed reporting.

Additionally, as shown in the work [16], Postman outperformed JMeter and Robot Framework in various data environments. The relevance of this study is undeniable, as it provides valuable insights into the performance testing capabilities of modern tools in varied environments.

Studies [17, 18, 19] show that Postman and JMeter are commonly used in conjunction with each other. Still, for different types of tests, Postman is typically used for API functional testing, while JMeter is primarily used for performance testing of APIs. However, researchers rarely provide direct comparisons between these performance testing tools, which could assist in selecting the suitable tool for various projects and testing scenarios.

This work aims to compare Postman and JMeter as tools for API performance testing, using performance testing metrics such as average response time, minimum and maximum values, and error rate (%). By evaluating their strengths and limitations across four common load types (load, stress, spike, and soak tests) for create, update, delete (CRUD) public API calls, this work aims to determine which tool is more effective in different testing scenarios. This comparison will help Information Technology practitioners make informed decisions when selecting the most suitable tool for their API performance testing needs in real software development projects.

The chapter is structured to provide a comprehensive examination of methodologies, tools, and experimental findings. The introduction highlights the importance of load testing in modern software engineering, followed by a literature review that discusses existing studies and industry practices.

The methodology section presents the selected tools, test environment, and experimental design. Next, the chapter outlines the performance metrics under investigation, including response time, throughput, and error rate, along with their relevance. Finally, the results, discussion, and conclusions are provided to summarize the findings and offer practical recommendations.

## 2 Literature review

Performance testing is crucial for assessing whether web applications deliver a satisfactory user experience under various workloads [20]. A workload reproduces the interactions of multiple concurrent users with the system to observe its actual behavior under stress. Defining meaningful workloads is a key challenge in performance testing [21].

Robust application programming interfaces (APIs) and web applications are critical for the seamless operation of enterprise systems [3]. Robustness in web applications has become increasingly important due to the complexity of software systems and the heightened demand for security, performance, and user satisfaction. Various studies emphasize different facets of robustness, focusing on testing methodologies, performance metrics, and security measures.

Performance testing ensures that the system under test (SUT) behaves as expected under different workloads. For web applications, a workload is equivalent to multiple concurrent users making requests to a web server [22, 23]. By simulating these workloads, testers can analyze system performance and identify potential issues, such as slow response times, resource bottlenecks, or failures under high-stress conditions.

There are several popular methodologies or types of performance tests [24]. Load testing involves testing a software system under various levels of user load to evaluate its response times and overall performance across different usage scenarios. Stress testing involves applying extreme loads to an application under test to identify potential failure points and assess system recovery mechanisms.

Scalability testing aims to determine how well a software system can handle increased loads by adding more resources, such as servers or virtual machines, to

meet the demands of larger user bases. Endurance testing, also known as soak testing, involves subjecting an application to a sustained load for an extended period to identify performance degradation issues.

Spike testing checks the response of an application to sudden and extreme increases in user load (e.g., user traffic spikes unexpectedly). Volume testing focuses on evaluating the system's performance when handling large volumes of data, identifying problems with data handling and database performance. Concurrency testing evaluates a system's ability to handle multiple simultaneous users or transactions efficiently.

Although the types of performance tests described above aim to achieve different goals, their design, execution, and analysis are fundamentally similar [21, 23, 25]. Firstly, during the test design phase, a workload is created. This can be done in two approaches [21, 23]. The first one focuses on making requests according to a specific target rate (e.g., simulating a certain number of immediate requests in a given time interval). The second approach simulates user interactions more realistically through a sequence of requests that reflects typical user behavior, including periods of inactivity.

Secondly, test execution is conducted. It can be automated through a load generation in a test tool. JMeter enables manual workload configuration through a graphical user interface. Its load generation is performed using a component called Thread Group, which represents one or more concurrent users that perform the same actions. Thread Group properties include the number of concurrent users and the frequency or duration of each user's actions. Alternatively, Postman allows users to configure and run performance (load) tests on APIs using the Load Profiles or Collection Runner.

The workload configuration involves settings as load profiles (fixed, ramp-up, spike, peak), virtual users or VUs (specifies the number of concurrent users to simulate during the test, with each virtual user executing requests from the selected collection in a repeating loop, where system resources and collection complexity determine the maximum number of virtual users), test duration, and using data file to supply custom values for each virtual user, allowing for more realistic simulations with varied input data.

Finally, during the test analysis phase, performance test metrics obtained after test execution are analyzed to detect threshold violations or anomalies by comparing observed behavior against expected norms.

To conduct effective API performance testing, it is crucial to utilize tools that not only simulate load but also provide detailed metrics that help identify bottlenecks within the system [26, 27]. Service speed requirements are primarily determined using throughput and response time measurements [28].

Meanwhile, web service workload performance is more easily demonstrated through CPU load and memory usage [29]. Performance metrics can be used as BPIs (Basic Performance Indicators) and are vital for defining SLA (Service Level Agreement). While customers often choose BPI metrics to increase system or

service adoption rates, SLA metrics were developed to govern contractual agreements between service providers and end-users [30].

Performance testing teams measure various metrics during test execution, including average response time, minimum response time, maximum response time, and percentiles (p90, p95, p99) [31]. Response time is the total time taken to process a client request to a web service, including both the request transmission and the server's response time [32]. The average response time is the mean time for all requests made by virtual users during a performance test, measured in milliseconds.

The Error % metric indicates the percentage of failed requests during server connection attempts. Throughput measures the number of requests served per unit of time, reflecting application availability, and it has a linear relationship with average response time.

Performance testing tools verify the system or application before delivering it to customers, and their efficiency builds confidence in users, ensuring they always view accurate statistical results [33]. Existing studies primarily focus on analyzing the key features of Postman, JMeter, and other performance tools based on their technical documentation and specifications, as well as their user-friendliness level, especially in terms of setting up and running tests [3, 10, 34-39].

These tools offer distinct capabilities: JMeter is designed primarily for load and performance testing, and supports a wide variety of protocols. At the same time, Postman focuses on simplifying API performance and functional testing with an intuitive user interface.

A variety of research works compare performance testing tools using quantitative metrics such as average response time, system load, resource consumption, and the complexity involved in test setup and operation [26, 29, 40, 41, 42].

According to authors [43, 44, 45, 46], Apache JMeter is a performance testing tool that enables load testing on various protocols and technologies. It is one of the most widely used open-source tools for performance testing, particularly in the domains of API testing and web applications. JMeter runs on any Java-compatible OS (including Linux, Microsoft Windows, macOS, etc.).

The ability to create and execute complex testing scenarios is one of JMeter's most essential features. The JMeter tool is multithreaded and can simulate a large number of VUs, enabling the simulation of a heavy load by distributing tests across multiple machines.

This tool may be used to test the performance of both static and dynamic resources. Moreover, it is capable of managing both Load and Performance Testing techniques for static and dynamic resources. The results [16] show that JMeter efficiently performs performance testing for quality enhancement compared to other open-source software.

Despite its many advantages, JMeter also has certain limitations, as noted by authors [48-50]. One of the main disadvantages is its relatively high memory consumption, especially when running large-scale tests or simulating a large number of users.

Additionally, JMeter also lacks advanced features, such as real-time monitoring. It has a high learning curve for setting up and configuring distributed tests. Additionally, JMeter lacks a scalable GUI and can be slow when managing complex test plans with a high volume of data.

The effectiveness of using Postman was also established by the authors [50-53] in studies on API performance testing. Postman is a platform for API development and testing that has emerged as a leading tool for API development, offering a very user-friendly interface. It can be used in two forms: as a downloadable client and as a web application. Postman is not an open-source tool. It provides both a free version and a paid version with additional features.

Postman tests can be executed manually using the GUI. Additionally, tests can be run automatically on a schedule using the Collection Runner, or they can be run using the command-line tool companion Newman, which enables the automated execution of Postman Collections directly or as part of a CI/CD pipeline.

Postman supports a wide array of testing functionalities, including request chaining, parameterization, and environmental variable management, which allow for the creation of complex test scenarios. Additionally, Postman enables you to collaborate with teammates by organizing, sharing, and communicating work with APIs.

According to authors [51-54], Postman can be used for API performance testing with a desktop application. Performance tests can be run for a collection of API requests using 1 of 4 load profiles:

1. Fixed, where a constant number of VUs are running tests in parallel;
2. Ramp up, when the number of VUs slowly increases from the initial load to the maximum;
3. Spike, where the number of virtual users increases from the base load to the maximum, then decreases back to the base load;
4. Peak, during which the number of virtual users increases from the base load to the maximum, holds steady, then decreases back to the base load.

Postman provides the option to reuse existing API collections for performance testing with minimal scripting effort. The data file feature enables testers to use the dataset file required to load-test the API with different datasets in each iteration.

Additionally, the number of VUs and test duration should be configured before running a performance test. It is essential to highlight that during performance test execution in Postman, each virtual user runs the requests in the specified order in a repeating loop. All of the virtual users operate in parallel to simulate real-world load on the API in a collection.

Performance test execution can be monitored in real-time through the Postman Summary tab. A summary of performance metrics is available in both tabular and graphical forms, including test duration, VUs count, total request count, requests/second, average response time, and API errors. Furthermore, percentile performance metrics are provided in the report (P90, P95, P99). Test reports, including results and details of each performance metric, are available after test execution. They can be downloaded in PDF or HTML formats and shared via a link.

Thus, analyzing the research results in the reviewed authors' works [51, 52], it is worth noting that there are some limitations to running performance tests in Postman. Firstly, a limited number of performance runs are available each month at no additional cost. If the performance run number is reached, the month's limit is then met, and the paid Postman plan or purchased add-ons can be used.

Secondly, the number of VUs in a performance test depends on available system resources and the collection used for the test. According to the Postman 2024 guidelines, a host with 8 CPU cores and 16 GB of RAM can simulate up to 250 VUs, and a host with 16 CPU cores and 32 GB of RAM can simulate up to 500 VUs.

Attempting to simulate a higher number of virtual users may cause inaccurate metrics and reduced throughput. Additionally, one area for improvement in Postman is that timer features for managing the frequency of requests and a sleep time option to introduce a delay between requests, emulating real-world scenarios, are unavailable, unlike in other load-testing tools. Also, a performance test scenario can have only one data file, which is an unlikely scenario in load testing.

Although both JMeter and Postman have their respective advantages and drawbacks, the selection of the appropriate tool depends on the specific performance goals of the project. However, studies comparing these tools across various scenarios remain limited, and many aspects, such as the performance of the tools, require further investigation.

## 3 Methodology

This work focuses on a comparative approach to evaluate the performance of JMeter and Postman under controlled conditions. To perform performance testing, a set of public APIs was selected for testing, as this approach does not violate their ethical use.

The methodology focuses on testing create, update, and delete (CRUD) endpoints of public APIs, such as GET, POST, PUT, and DELETE, because this helps to simulate real-world interactions with them. These HTTP methods represent the typical operations that users or systems perform when interacting with an API. Each request type was used under different types of load. When using different types of requests with public APIs, it is essential to note that public APIs often return simulated responses (mocked data instead of real data).

However, those public APIs often have different performance characteristics depending on the type of request. Although the responses may be mocked, they offer insight into how the API processes requests and handles various loads. This can help assess whether the API's response times are efficient and stable under load.

The endpoints selected for testing include APIs from five public resources, featuring both simulated and real data, allowing for quick testing without the need to develop a custom backend, such as ReqRes.in, DummyJSON, SampleAPIs, JsonPlaceholder, FakeStoreAPI.

CRUD operations were tested in the form of sequential requests, with a request payload similar to the one shown in Figure 1. It should be noted that the *{postId}*

variable shown in Figure 1 is a generated ID of a new user by the service in the POST request.

This *{postId}* variable was saved after executing the POST request using the *pm.environment.set()* function in Postman collections, as well as by using the Regular Expression Extractor in JMeter tests. After that, *{postId}* was passed to the PUT and DELETE requests for subsequent API calls. Such an approach was implemented in tests for ReqRes.in, SampleAPIs, FakeStoreAPI services. For the DummyJSON and JsonPlaceholder services, existing resource values were used for testing due to limitations of those services.

Different resources were used for public APIs:

- 'users' resource https://reqres.in/api/users (Figure 1);
- 'products' resource of DummyJSON service https://dummyjson.com/products;
- 'codingResources' resource https://api.sampleapis.com/codingresources/codingResources;
- 'posts' resource https://jsonplaceholder.typicode.com/posts;
- 'products' resource of FakeStoreAPI service https://fakestoreapi.com/products.

```
curl --location 'https://reqres.in/api/users?page=1' --header 'x-api-key: reqres-free-v1'

curl --location 'https://reqres.in/api/users' \
--header 'x-api-key: reqres-free-v1' \
--header 'Content-Type: application/json' \
--data-raw '{
            "email": "pikachu@reqres.in",
            "first_name": "Pika",
            "last_name": "Chu",
            "avatar": "https://reqres.in/img/faces/12-image.jpg"
        }'

curl --location --globoff --request PUT 'https://reqres.in/api/users/{{postId}}' \
--header 'x-api-key: reqres-free-v1' \
--header 'Content-Type: application/json' \
--data '{
    "name": "Anders",
    "job": "Anderson"
}'

curl --location --globoff --request DELETE 'https://reqres.in/api/users/{{postId}}' \
--header 'x-api-key: reqres-free-v1'
```

**Figure 1.** Requests to ReqRes.in 'users' resource

For both test tools, JMeter and Postman, the test scenarios involved sequential execution of GET, POST, PUT, and DELETE requests to the public APIs. All requests were identical between the tools and were directed to the same public APIs test servers.

It's important to get objective comparison results of Postman and JMeter test tools. For this reason, it is critical to conduct performance testing under identical

conditions, considering the following factors: number of VU, Think Time or Delay, and test duration. VUs simulate the behavior of real users interacting with the system. The number of VUs directly impacts the load on the system being tested.

If the number of VUs is too low, the results may not accurately reflect how the system performs under high traffic. Conversely, if the number of VUs is too high, the system might experience excessive strain, potentially leading to bottlenecks that don't align with standard usage patterns.

Think Time, also known as Delay, refers to the pause between user actions (or requests). In real-world scenarios, users don't send requests continuously without any pause; they typically take a brief moment to think or interact with the system.

The Test Duration specifies how long the performance test will run, and it can impact the stability and consistency of the results. Short tests may not provide sufficient data to accurately measure the system's performance under sustained load.

In contrast, longer tests can identify performance degradation, memory leaks, or other issues that emerge over time. A combination of different values for the factors mentioned above can be used to design test cases that closely resemble realistic user behavior. Performance test cases should be performed for all load profiles (Ramp Up, Spike, Peak, and Fixed) in all five public APIs using both tools, JMeter and Postman.

Think Time can be implemented in JMeter using the Constant Timer element, as shown in Figure 2a. Similarly, a GET Delay request can be used in Postman for this purpose - GET https://postman-echo.com/delay/X, where X is the number of seconds to pause, to emulate realistic user behavior.

In general, the structure of all tests for all five public APIs in JMeter is similar to Figure 2a (using Loop Controller with GET, POST, PUT, DELETE requests and listeners inside it). Still, the Thread Group type should be varied depending on the load type (Thread Group, Ultimate Thread Group, or Concurrent Thread Group).

In Postman, all CRUD requests and Delay requests after each of those requests were organized into a collection for all five public APIs, as shown in Figure 2b.


a) b)

**Figure 1.** Test cases structure: a) JMeter Loop Controller with requests; b) Postman collection of requests

The load parameters of test cases should be designed to have similar parameters for different load scenarios in both test tools.

For this reason, different Thread Group types can be used in JMeter, including Thread Group, Ultimate Thread Group, and Concurrent Thread Group.

Performance tests were conducted under identical conditions:

- Number of VUs: 10, 20, 30, 40, or 50, depending on the scenario;
- Think Time was ranged: 1000 ms (peak load), 3000-5000 ms (realistic load), 10000 ms (low load);
- Test duration: 5 or 10 minutes, depending on the scenario.

To evaluate Postman and JMeter behavior under different user scenarios, four load profiles were used: Ramp Up, Spike, Fixed Load, and Peak Load.

In each of these profiles, the values of VUs and Think Time intervals were varied depending on the test goal (see test cases in Table 1).

Table 1

**Performance Test Cases for JMeter and Postman**

| TC No | Load Profile | VU, N | Think Time, sec | Duration, min | Comment |
|---|---|---|---|---|---|
| 1 | Ramp Up | 0-10 | 10 | 10 | Increase in load |
| 2 | Ramp Up | 10-30 | 3 | 10 | Increase in load |
| 3 | Spike | 1-10-1 | 10 | 10 | Users attack simulation |
| 4 | Spike | 5-50-5 | 2 | 5 | Users attack simulation |
| 5 | Peak | 2-10-2 | 10 | 10 | Check of requests max |
| 6 | Peak | 8-40-8 | 1 | 5 | Check of requests max |
| 7 | Fixed Load | 10 | 10 | 10 | Stable request flow |
| 8 | Fixed Load | 20 | 5 | 10 | Stable request flow |

The baseline scenario was 10 VU with a Think Time of 10 seconds, during which all five public APIs functioned without errors. It was done to ensure consistent conditions and simplify the results.

Additionally, stress testing scenarios were applied with increased request frequency (think time ranging from 1 to 5 seconds) and peak load values (up to 80 VUs) to obtain results that closely mimic realistic user behavior. Test cases are described in Table 1, and each test case (TC) was run on five public APIs.

The experiment was conducted in a controlled environment (laptop) with consistent configurations to ensure reliable results, including Windows 11 x64 24H2, an 8-core CPU, and 16GB RAM. During the experiment, the following tools were utilized: Postman Desktop application version 11.50.2 and Apache JMeter version 5.6.3.

The primary metrics used to evaluate the performance of test tools included average (avg) response time, minimum (min) and maximum response time values, and error rate (%). The average response time measures the sum of all response times divided by the total number of requests [31].

It represents the typical response time the user will experience. It can be used to evaluate the overall performance efficiency of a system under normal operating conditions. The minimum response time indicates the shortest amount of time it takes for the system to respond to a user request. It represents the best-case scenario and can be used to assess the system's baseline performance under optimal conditions.

Maximum response time refers to the longest period the system takes to respond to a user request. It represents the worst-case scenario and can be used when identifying potential performance bottlenecks or issues under stress or peak load conditions. Error Rate shows the percentage of requests that failed or didn't receive a response.

This metric identifies the issues and bottlenecks that impact the system's performance. In this study, we focused on analyzing the average response time as an indicator of overall test tool performance.

Data was collected using JMeter's built-in test reports and Postman's performance test report. Results were collected for each scenario and metric.

During the performance tests, the difference in average response time metric values (deltas) was calculated to analyze the impact of varying load conditions on response times. Additionally, the aggregated mean values for response times were calculated across all test cases and for each (API, test tool, HTTP method) combination, which helps to provide a representative performance assessment.

The difference in metric values (deltas) can be calculated to analyze which types of requests were slower in Postman or JMeter. This can be done using the formula:

$$\Delta\text{Avg} = \text{Postman}_{\text{Avg}} \quad \text{JMeter}_{\text{Avg}}, \tag{1}$$

where $\Delta\text{Avg}$ is the difference in average response time between Postman and JMeter;

$\text{Postman}_{\text{Avg}}$ is the value of the average response time in Postman;

$\text{JMeter}_{\text{Avg}}$ is the value of the average response time in JMeter.

If the case is a positive value, then Postman had a longer average response time than JMeter. If it is a negative value, then JMeter's average response times exceeded those in Postman, and Postman's performance was faster.

Average values of each performance metric are calculated for 8 test cases for each API and HTTP method:

$$\text{Avg\_metric} = \frac{\sum_{k=1}^{n} m_k}{n}, \tag{2}$$

where $m_k$ – value of performance metric for $test\_case_k$;

$n$ – number of test cases

This should be repeated for 5 APIs using a test tool (such as Postman or JMeter) and a selected HTTP method. Then, mean aggregated values should be calculated for average performance metric values for the selected HTTP method:

$$\text{Mean\_metric} = \frac{\sum_{i=1}^{l} \text{Avg\_metric}_i}{l}, \quad (3)$$

where $\text{Avg\_metric}$ – value of performance metric for selected HTTP method;

$l$ – number of APIs

The final step after analyzing aggregated metrics and their deltas is visualizing them through bar charts to evaluate the differences between the test tools.

It is important to note that although the load parameters were standardized across both test tools, the execution architecture differs between them. In Postman, each VU executes requests sequentially (one after the other). In contrast, in JMeter, each thread is executed in parallel, potentially creating a higher load (higher Requests Per Second (RPS)).

As a result, Loop Controller elements with GET, POST, PUT, and DELETE requests can be added to JMeter test plans within a Thread Group to ensure more accurate and comparable test execution with similar load profiles, similar to Postman.

**4 Results**

Obtained performance test results for JMeter and Postman tools, based on the metric of average response time. A method of comparing aggregated metrics (deltas) was used, along with their visualization through bar charts, to analyze the differences between the tools. This visualization helps to ensure that the differences between the tools are not random fluctuations, but are consistent across all APIs. Both delta values and aggregated mean values for the average response time are calculated and analyzed to provide a comprehensive comparison of the tools' performance. The response times for JMeter and Postman were compared across different test cases.

The test load in TC1 begins with an initial load of 0 VUs and then steadily ramps up to 10 users over 5 minutes, with a total test duration of 10 minutes. Postman tests used the Ramp Up load profile in Postman with 10 VUs, an initial load of 0 VUs, and a test duration of 10 minutes. To obtain an equivalent load in JMeter, a Thread Group with 10 VUs and a ramp-up period of 300 s and a duration of 600 s was used. In this case, the Loop Count was set to 1 in the Loop Controller of JMeter.

The delta between response times in the TC1 ranged from 13 to 87 ms, with Postman showing a slight advantage (negative values) over JMeter (positive values) for the majority of public APIs' services, as illustrated in the bar chart (see Figure 3).

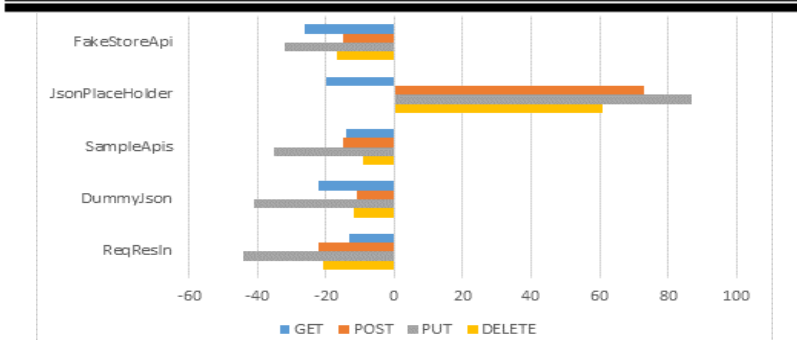


**Figure 3.** Deltas of avg response time, TC1

The load in TC2 is gradually increased from 10 to 30 VUs over 10 minutes. It is generated with a Ramp-Up profile, starting with 10 VUs that run for 2 minutes and 30 seconds, then ramping up to 30 VUs for another 2 minutes and 30 seconds. After that, the load is maintained at 30 VUs for 5 minutes. Postman tests used the Ramp Up load profile with 30 VUs, an initial load of 10 VUs, and a test duration of 10 minutes. To obtain an equivalent load in JMeter, Ultimate Thread Group with two threads was used, as shown in Figure 3b: 1) start threads count = 10, initial delay = 0 sec, startup time = 0 sec, hold load for = 600 sec, shutdown time = 0; 2) start threads count = 20, initial delay = 150 sec, startup time = 150 sec, hold load for = 300 sec, shutdown time = 0. In both cases, the Loop Count was set to 'Infinite' in the Loop Controller of JMeter. Deltas show that the average response time in Postman was greater than in JMeter for 3 out of 5 public APIs. The difference in average ranges was from 1 to 113 ms, indicating an advantage of JMeter over Postman under those load conditions, with a time of 3000 ms (see Figure 4).

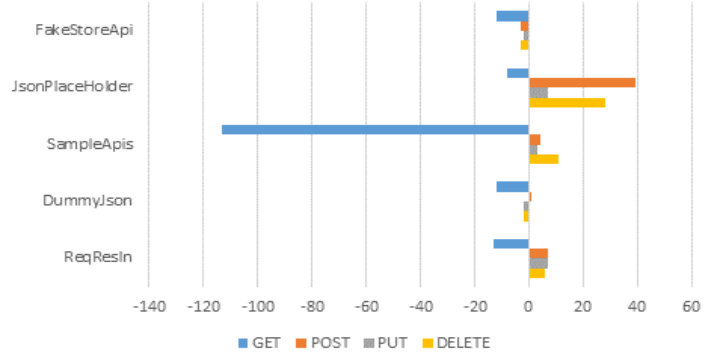


**Figure 4.** Deltas of avg response time, TC2

TC3 used load generation with a sudden spike to 10 VUs, followed by a decline, with a total test duration of 10 minutes and a think time of 10,000 ms. Postman tests

used a Spike load profile with 10 VUs, a base load of 1 VU, and a test duration of 10 minutes.

To obtain an equivalent load in JMeter, Ultimate Thread Group was used: 1) start threads count = 1, initial delay = 0 sec, startup time = 0 sec, hold load for = 600 sec, shutdown time = 0; 2) start threads count = 9, initial delay = 240 sec, startup time = 60 sec, hold load for = 0 sec, shutdown time = 60.

The delta between response times in TC3 was from 16 to 113 ms, with Postman showing a slight advantage over JMeter in 3 out of 5 APIs (see Figure 5).

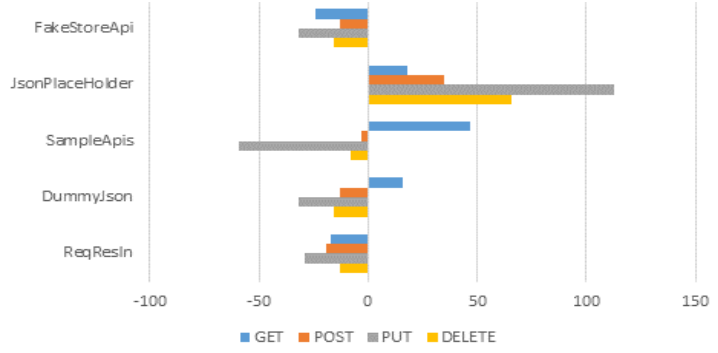


**Figure 5.** Deltas of avg response time, TC3

In TC4, a sudden spike to 50 VUs was applied, followed by a decline in the number of VUs. The test duration was 5 minutes, and the think time was 5000 ms. Postman tests used a Spike load profile with 50 VUs, a base load of 5 VUs, and a test duration of 5 minutes.

To obtain an equivalent load in JMeter, Ultimate Thread Group was used: 1) start threads count = 5, initial delay = 0 sec, startup time = 0 sec, hold load for = 300 sec, shutdown time = 0; 2) start threads count = 45, initial delay = 120 sec, startup time = 30 sec, hold load for = 0 sec, shutdown time = 30.

It's worth mentioning that for the ReqRes.in service, a vast majority of 429 errors (too many requests) were returned in response when the number of VUs started to increase, for instance, significantly in TC4.

For other APIs, 429 and 502 errors were returned in response, but their number was insignificant. The delta between response times in TC4 was from 1 to 75 ms, with Postman showing a slight advantage over JMeter in 3 out of 5 APIs (see Figure 6).

TC5 used a load with a sharp increase to a peak of 10 VUs, followed by a gradual decrease in voltage. The test duration was 10 minutes, and the think time was 10,000 ms. Postman tests used a Peak load profile in Postman with 10 VUs, a base load of 2 VUs, and a test duration of 10 minutes.

To obtain an equivalent load in JMeter, Ultimate Thread Group was used: 1) start threads count = 2, initial delay = 0 sec, startup time = 0 sec, hold load for = 600 sec, shutdown time = 0; 2) start threads count = 8, initial delay = 120 sec, startup time = 120 sec, hold load for = 120 sec, shutdown time = 120.

For most types of requests, Postman demonstrated a faster average response time than JMeter, with deltas ranging from 4 ms to 123 ms. Notably, the most significant positive delta (123 ms) occurred for the JsonPlaceHolder API, where JMeter outperformed Postman. The chart highlights the consistency of Postman's performance across the majority of APIs (see Figure 7).

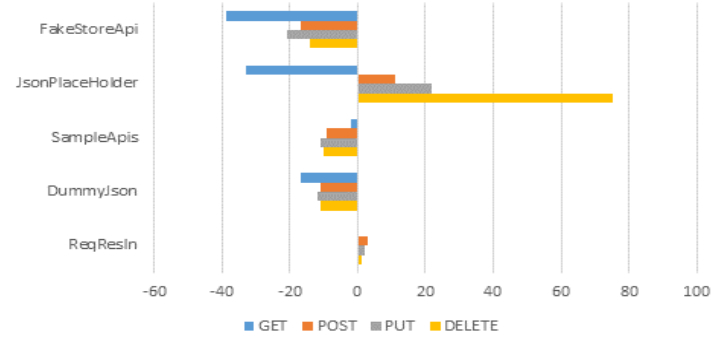


**Figure 6.** Deltas of avg response time, TC4

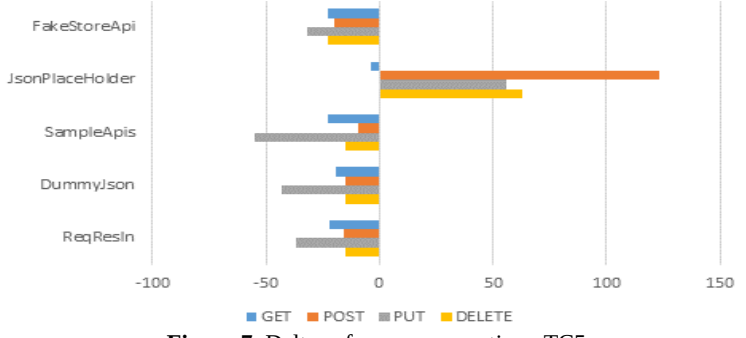


**Figure 7.** Deltas of avg response time, TC5

In TC6, a sharp increase in load to a peak of 40s VUs was used, followed by a gradual decrease. The test duration was 5 minutes, with a think time of 1000 ms. Postman tests used a Peak load profile in Postman with 40 VUs, a base load of 8 VUs, and a test duration of 5 minutes. To obtain an equivalent load in JMeter, Ultimate Thread Group was used, as shown in Figure 8b: 1) start threads count = 8, initial delay = 0 sec, startup time = 0 sec, hold load for = 300 sec, shutdown time = 0; 2) start threads count = 32, initial delay = 60 sec, startup time = 60 sec, hold load

for = 60 sec, shutdown time = 60. Deltas show that JMeter significantly outperformed Postman for 4 out of 5 public APIs. The difference in the average ranges was from 1 to 59 ms. Those results demonstrate a practical advantage of JMeter under load conditions, with a sharp increase in load to a peak of 40 VUs, followed by a gradual decrease and a realistic short think time (see Figure 8).

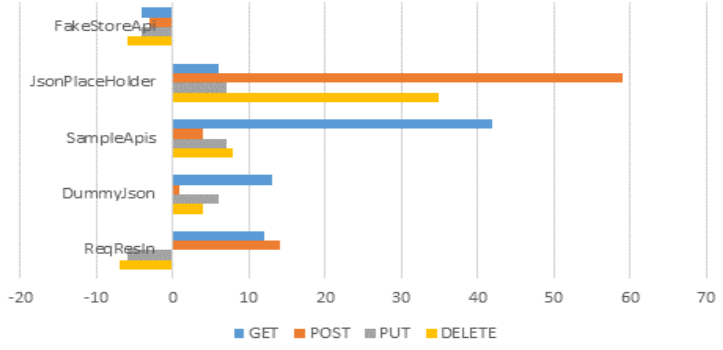


**Figure 8.** Deltas of avg response time, TC6

TC7 generated a load fixedly, maintaining a stable load for 10 VUs over 10 minutes and a think time equal to 10 minutes. Postman tests used a Fixed load profile in Postman with 10 VUs, test duration = 10 minutes. To obtain an equivalent load in JMeter, the Concurrency Thread Group was used, with a target concurrency of 10 and a hold target rate of 10 minutes. Deltas illustrate that the average response time in Postman was less than in JMeter for 4 out of 5 public APIs. The difference in the average ranges was from 6 to 116 ms. Those results demonstrate a practical advantage of Postman under the stable load conditions with few VUs and an ample think time (see Figure 9). In TC8, the load remained stable for 20 VUs over 10 minutes and had a shorter think time of 5,000 ms. Postman tests used a Fixed load profile in Postman with 20 VUs, test duration = 10 mins. To obtain an equivalent load in JMeter, a Concurrency Thread Group was used with the following settings: target concurrency = 20, hold target rate = 20 minutes. Deltas show that the average response time in Postman was much less than in JMeter for most types of requests in TC7 for 3 out of 5 public APIs. Only PUT and DELETE requests were faster in JMeter for 2 APIs. The difference in the average ranges was from 8 to 35 ms. Those results demonstrate a practical advantage of Postman under stable load conditions with a low number of VUs and medium think time (see Figure 10).

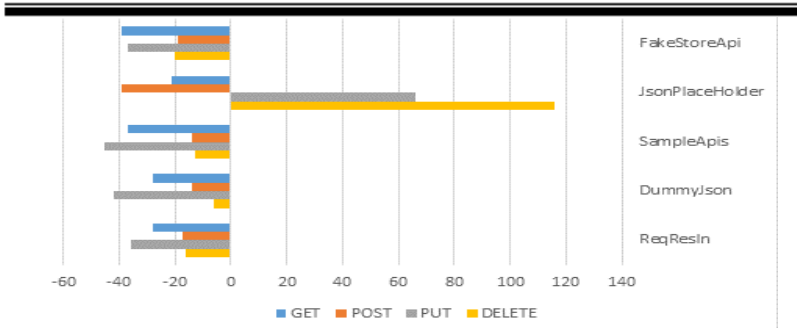


**Figure 9.** Deltas of avg response time, TC7

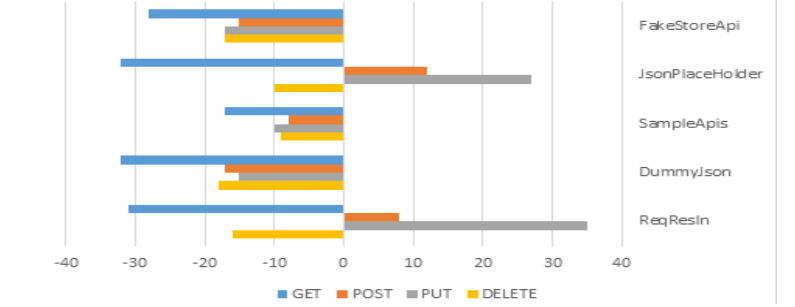


**Figure 10.** Deltas of avg response time, TC8

These values per API were subsequently aggregated across 5 APIs. The final metrics, including mean response time, minimum, maximum, and error percentage values, accurately reflect the tool's performance per HTTP method, as shown in Table 2. The final aggregated values in Table 2 show that, in general, Postman demonstrates better performance compared to JMeter. Although the test was conducted once, aggregated values of metrics were calculated from hundreds of requests, allowing for an objective assessment of system behavior.

Table 2

**Aggregated performance results for Postman and JMeter**

| Method | Test Tool | Mean Avg, ms | Mean Min, ms | Mean Max, ms | Mean Error, % |
|---|---|---|---|---|---|
| GET | Postman | 166,175 | 110,650 | 415,575 | 0,935 |
| GET | JMeter | 181,650 | 115,925 | 499,625 | 0,739 |
| POST | Postman | 148,100 | 117,250 | 877,200 | 7,671 |
| POST | JMeter | 147,175 | 124,700 | 805,525 | 7,999 |
| PUT | Postman | 144,925 | 118,100 | 341,425 | 7,777 |
| PUT | JMeter | 152,075 | 125,525 | 377,275 | 7,936 |
| DELETE | Postman | 166,225 | 136,525 | 315,475 | 7,931 |
| DELETE | JMeter | 163,325 | 143,525 | 374,475 | 8,515 |

These data suggest a preliminary conclusion that Postman exhibits higher performance in low to moderate load conditions, particularly during stable loads with a fixed number of VUs, and in other types of loads where think time exceeds 5 seconds. On the other hand, JMeter demonstrates higher performance under conditions of high load and high request intensity, especially in cases of a sharp increase in load to peak, followed by a gradual decrease with a short think time of 1 second. Additionally, a greater number of samples were generated in JMeter than in Postman in all testing cases.

## 5 Discussions

The analysis is based on a single performance test run for five public APIs, which limits statistical generalizability. However, aggregated values were calculated from hundreds of requests, which allows for a reliable assessment of a tool's behavior at the API level.

Since public open APIs were used, the exact number of active users at the time of the performance test execution remains unknown. However, to minimize this uncertainty, performance tests were conducted at the same time of day during the experiment. Additional experiments with repeated execution of scenarios can be applied to confirm the robustness of the observed effects.

## 6 Conclusions

In conclusion, based on the analysis of performance testing with JMeter and Postman, it is clear that each tool is more efficient in specific load scenarios. Postman demonstrated higher performance in steady-state conditions, particularly with fixed load profiles and gradual ramp-ups, where it excelled in providing stable response times despite increasing load.

Alternatively, JMeter outperformed Postman in high-load and high-intensity scenarios, showing superior performance in rapid peak load conditions, with minimal delays between user actions.

In the context of performance testing, these differences underscore the importance of selecting the appropriate tool for the specific testing scenario. Postman is well-suited for scenarios that require steady performance under controlled conditions, whereas JMeter is more appropriate for high-stress, high-traffic environments. Understanding the capabilities of these tools can have a positive impact on software development processes.

Additionally, calculated deltas show that decreasing think time for the same type of load, such as ramp-up and peak, outperforms JMeter in those types of requests, which can be helpful for testing applications with delays between user actions.

Postman can be more effective in performance tests, which enable the configuration of a gradual increase in load. This is particularly useful when the load is predictable in advance, such as when testing an API with a limited request intensity. These tools can be a good option for small and medium projects,

especially when there are no strict requirements to simulate a realistic user load. The application is quite simple, and resources for testing are limited.

JMeter is better suited for testing with peak loads and evaluating system behavior under high request intensity (e.g., complex applications with high traffic), as well as for identifying the system's performance limits.

By considering factors such as virtual user load, think time, and the complexity of responses, developers and QA engineers can reduce testing time, mitigate the risk of underestimating system performance, and ensure that the system performs effectively under real-world load conditions.

## 7 Acknowledgements

The research was supported by the Ukrainian project of fundamental scientific research, “Development of computational methods for detecting objects with near-zero and locally constant motion by optical-electronic devices” (#0124U000259) from 2024 to 2026.

## References

1. Godinho, A., Rosado, J., Sá, F. A., & Cardoso, F. (2023). Method for evaluating the performance of web-based APIs. *International Conference on Smart Objects and Technologies for Social Good.* https://doi.org/10.1007/978-3-031-52524-7_3

2. Golmohammadi, A., Zhang, M., & Arcuri, A. (2023). Testing RESTful APIs: A survey. *ACM Transactions on Software Engineering and Methodology.* https://doi.org/10.1145/3617175

3. Joshi, N. Y. (2023). Developing robust APIs and web applications for enterprise applications: Automation frameworks and testing strategies. *Journal of Basic Science and Engineering.*

4. Dhandapani, A. (2025). Automation testing in microservices and cloud-native applications: Strategies and innovations. *Journal of Computer Science and Technology Studies, 7*(3), 826–836. https://doi.org/10.32996/jcsts

5. Niedermaier, S., et al. (2019). On observability and monitoring of distributed systems: An industry interview study. In *International Conference on Service-Oriented Computing* (pp. 36–52). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-030-33702-5_3

6. Molyneaux, I. (2014). *The art of application performance testing: From strategy to tools.* O’Reilly Media, Inc.

7. Cooper, Q., Krishnamurthy, D., & Amannejad, Y. (2024). Budget aware performance test selection for microservices. In *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)* (pp. 376–385). Shenzhen, China. https://doi.org/10.1109/CLOUD62652.2024.00049

8. Lenka, R. K., Dey, M. R., Bhanse, P., & Barik, R. K. (2018). Performance and load testing: Tools and challenges. In *2018 International Conference on Recent Innovations in Electrical, Electronics & Communication Engineering (ICRIEECE)*

(pp. 2257–2261). Bhubaneswar, India. https://doi.org/10.1109/ICRIEECE44171.2018.9009338

9. Pratama, M. R., & Sulistiyo Kusumo, D. (2021). Implementation of continuous integration and continuous delivery (CI/CD) on automatic performance testing. In *2021 9th International Conference on Information and Communication Technology (ICoICT)* (pp. 230–235). Yogyakarta, Indonesia. https://doi.org/10.1109/ICoICT52021.2021.9527496

10. Nagineni, S. (2025). Advancing software reliability through systematic API testing: A comparative analysis of modern automation frameworks and methodological implications for distributed systems. *JCSTS, 7*(8), 798–805. https://doi.org/10.32996/jcsts.2025.7.8.94

11. Arif, S., Faisal, M., et al. (2024). Software automation testing: Comparing no-code, low-code, and traditional approaches. *Contemporary Journal of Social Science Review*, 2138–2147. https://doi.org/10.63878/cjssr.v2i04.402

12. Sri, S. D., et al. (2024). Automating REST API Postman test cases using LLM. *arXiv preprint.* https://doi.org/10.48550/arXiv.2404.10678

13. Postman. (2024). Configuring performance tests. Retrieved from https://learning.postman.com/docs/collections/performance-testing/performance-test-configuration

14. Apache JMeter. (2021). Supported protocols. Retrieved from https://jmeter.apache.org/usermanual/component_reference.html

15. Bondi, A. B., & Saremi, R. (2021). Experience with teaching performance measurement and testing in a course on functional testing. In *ICPE 21: Companion of the ACM/SPEC International Conference on Performance Engineering* (pp. 115–120). https://doi.org/10.1145/3447545.3451196

16. Hsieh, C.-H., et al. (2021). Evaluation system for software testing tools in complex data environment. In *2021 4th International Conference on Information Communication and Signal Processing (ICICSP)* (pp. 604–609). Shanghai, China. https://doi.org/10.1109/ICICSP54369.2021.9611846

17. Ball, A., & Ochei, L. C. (2024). Remote performance monitoring system for managed service providers. *International Journal of Applied Information Systems (IJAIS).*

18. Koppanati, P. K. (2021). Automation testing for custom insurance quotation engines using microservices architecture. *Journal of Scientific and Engineering Research*, 326–332. https://doi.org/10.5281/zenodo.14005848

19. Noetzold, D., et al. (2024). JVM optimization: An empirical analysis of JVM configurations for enhanced web application performance. *SoftwareX*. https://doi.org/10.1016/j.softx.2024.101933

20. Thakur, S. S. (2025). Technical review: Performance testing methodologies and implementation strategies. *Sarcouncil Journal of Multidisciplinary.*

21. Battista, E., Martino, S. D., Di Meglio, S., Scippacercola, F., & Lucio Starace, L. L. (2023). E2E-Loader: A framework to support performance testing of web applications. In *2023 IEEE Conference on Software Testing, Verification and*

*Validation (ICST)* (pp. 351–361). Dublin, Ireland. https://doi.org/10.1109/ICST57152.2023.00040

22. Menasce, D. A. (2002). Load testing of web sites. *IEEE Internet Computing, 6*(4), 70–74.

23. Di Meglio, S., et al. (2025). Performance testing in open-source web projects: Adoption, maintenance, and a change taxonomy.

24. Pargaonkar, S. (2023). A comprehensive review of performance testing methodologies and best practices: Software quality engineering. *International Journal of Science and Research (IJSR),* 2008–2014. https://doi.org/10.21275/SR23822111402

25. Jiang, Z. M., & Hassan, A. E. (2015). A survey on load testing of large-scale software systems. *IEEE Transactions on Software Engineering*, 1091–1118. https://doi.org/10.1109/TSE.2015.2445340

26. Hendayun, M., Ginanjar, A., & Ihsan, Y. (2023). Analysis of application performance testing using load testing and stress testing methods in API service. *Jurnal Sisfotek Global***,** 28–34. https://doi.org/10.38101/sisfotek.v13i1.2656

27. Yenugula, M., Kodam, R., & He, D. (2019). Performance and load testing: Tools and challenges. *International Journal of Engineering in Computer Science*, 57–62. https://doi.org/10.33545/26633582.2019.v1.i1a.102

28. Blinowski, G., Ojdowska, A., & Przybyłek, A. (2022). Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access, 10***,** 20357–20374. https://doi.org/10.1109/ACCESS.2022.3152803

29. Lawi, A., Panggabean, B. L. E., & Yoshida, T. (2021). Evaluating GraphQL and REST API services performance in a massive and intensive accessible information system. *Computers.* https://doi.org/10.3390/computers10110138

30. Duman, I., & Eliiyi, U. (2021). Performance metrics and monitoring tools for sustainable network management. *Bilişim Teknolojileri Dergisi***,** 37–51.

31. BlazeMeter. (2020). Key performance metrics. Retrieved from https://help.blazemeter.com/docs/guide/performance-kpis-key-perf-test-metrics.htm

32. Dhalla, H. K. (2021). A performance comparison of RESTful applications implemented in Spring Boot Java and MS .Net Core. *Journal of Physics: Conference Series***.** IOP Publishing. https://doi.org/10.1088/1742-6596/1933/1/012041

33. Jacob, A., & Karthikevan, A. (2018). Scrutiny on various approaches of software performance testing tools. In *2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)* (pp. 509–515). Coimbatore, India. https://doi.org/10.1109/ICECA.2018.8474876

34. Srivastava, N., Kumar, U., & Singh, P. (2021). Software and performance testing tools. *Journal of Information, Electronics and Electronic Engineering***,** 1–12.

35. Ali, A., Maghawry, H. A., & Badr, N. (2022). Automation of performance testing: A review. *International Journal of Intelligent Computing & Information Sciences***.** https://doi.org/10.21608/ijicis.2022.161846.1219

36. Samlı, R., & Orman, Z. (2023). A comprehensive overview of web-based automated testing tools. *İleri Mühendislik Çalışmaları ve Teknolojileri Dergisi***,** 13–28.

37. Ushakova, I., Plokha, O., & Skorin, Y. (2022). Approaches to web application performance testing and real-time visualization of results. *Kharkiv National Automobile and Highway University (KHADU), Ukraine***.** https://doi.org/10.30977/BUL.2219-5548.2022.96.0.71

38. Abbas, R., Sultan, Z., & Bhatti, S. N. (2017). Comparative analysis of automated load testing tools: Apache JMeter, Microsoft Visual Studio (TFS), LoadRunner, Siege. In *2017 International Conference on Communication Technologies (ComTech)* (pp. 39–44). Rawalpindi, Pakistan. https://doi.org/10.1109/COMTECH.2017.8065747

39. Theivendran, P. (2023). Investigating usability and user experience of software testing tools. *Authorea Preprints, TechRxiv.* https://doi.org/10.36227/techrxiv.23251076.v1

40. Lenka, R. K., Mamgain, S., Kumar, S., & Barik, R. K. (2018). Performance analysis of automated testing tools: JMeter and TestComplete. In *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)* (pp. 399–407). Greater Noida, India. https://doi.org/10.1109/ICACCCN.2018.8748521

41. Tiwari, V., Upadhyay, S., Goswami, J. K., & Agrawal, S. (2023). Analytical evaluation of web performance testing tools: Apache JMeter and SoapUI. In *2023 IEEE 12th International Conference on Communication Systems and Network Technologies (CSNT)* (pp. 519–523). Bhopal, India. https://doi.org/10.1109/CSNT57126.2023.10134699

***42.*** Neelapu, M. (2023). Enhancement of software reliability using automatic API testing model. *International Journal of Multidisciplinary Research and Growth Evaluation.*

43. Rodrigues, A. G., Demion, B., & Mouawad, P. (2019). *Master Apache JMeter—From load testing to DevOps: Master performance testing with JMeter*. Packt Publishing Ltd.

44. Akiladevi, R., Vidhupriya, P., & Sudha, V. (2018). A study and analysis on software testing tools. *International Journal of Pure and Applied Mathematics*, 1783–1800.

45. Erinle, B. (2017). *Performance testing with JMeter 3.* Packt Publishing Ltd.

46. Indrianto, I. (2023). Performance testing on web information system using Apache JMeter and BlazeMeter. *Jurnal Ilmiah Ilmu Terapan Universitas Jambi*, 138–149. https://doi.org/10.22437/jiituj.v7i2.28440

47. Shyam Mohan, J. S., & Goswami, K. (2025). Performance analysis and comparison of Node.js and Java Spring Boot in implementation of RESTful applications. *Software: Practice and Experience***,** 1209–1233. https://doi.org/10.1002/spe.3418

48. Siddhant, S., & Prapull, S. B. (2020). Comprehensive review of load testing tools**.** *International Research Journal of Engineering and Technology.*

49. AutomateNow. (2023). Advantages and disadvantages of using JMeter. Retrieved from https://automatenow.io/advantages-and-disadvantages-of-using-jmeter

50. TestSigma. (2025). JMeter vs Postman. Retrieved from https://testsigma.com/blog/jmeter-vs-postman

51. Golian, N., & Tisheninova, V. (2025). Integrated graph-based testing pipeline for modern single-page applications. *Bulletin of National Technical University "KhPI". Series: System Analysis, Control and Information Technologies,* 51–59. https://doi.org/10.20998/2079-0023.2025.01.08

52. NashTech. (2024). Performance testing with Postman: Is it worth? Retrieved from https://blog.nashtechglobal.com/performance-testing-with-postman-is-it-worth

53. Susan Rini, V. S. (2024). When Postman goes that extra mile to deliver performance to APIs. *Software Testing Magazine***.** Retrieved from https://www.softwaretestingmagazine.com/tools/when-postman-goes-that-extra-mile-to-deliver-performance-to-apis

54. Savanevych, V., et al. (2023). Mathematical methods for an accurate navigation of the robotic telescopes. *Mathematics, 11*(10), 2246. https://doi.org/10.3390/math11102246

## АНАЛІЗ ПОКАЗНИКІВ ЕФЕКТИВНОСТІ ІНСТРУМЕНТІВ НАВАНТАЖУВАЛЬНОГО ТЕСТУВАННЯ

**Ph.D. С. Хламов**[1][0000-0001-9434-1081]**, М. Мендієлєва**[2][0009-0002-4282-3147], **Ph.D. О. Вовк** [3][0000-0001-9072-1634]**, Ph.D. Ж. Дейнеко**[4][0000-0001-6747-9130]**, С. Литвиненко**[5][0009-0003-2632-9082]

*Харківський національний університет радіоелектроніки, Україна*
*EMAIL: [1]sergii.khlamov@gmail.com, [2]mariia.mendielieva@nure.ua, [3]oleksandr.vovk@nure.ua, [4]zhanna.deineko@nure.ua, [5]serhii.lytvynenko@nure.ua*

***Анотація.*** *У цьому дослідженні подано детальний порівняльний аналіз двох широко використовуваних інструментів — JMeter та Postman — для тестування продуктивності інтерфейсів прикладного програмування (API). Оскільки API є основою сучасних розподілених застосунків і хмарних сервісів, забезпечення їхньої надійності та ефективності за різних умов навантаження має надзвичайно важливе значення. З цією метою було систематично зібрано й оцінено показники продуктивності, зокрема середній, мінімальний і максимальний час відгуку, а також рівень помилок, із п'яти відкрито доступних API, що репрезентують різні функціональні домени. Експериментальні результати демонструють очевидні відмінності у поведінці двох інструментів залежно від інтенсивності навантаження. Postman показує кращу стабільність і ефективність за умов низького та середнього навантаження, що робить його придатним для тестування стабільних навантажень і рутинної валідації API-ендпойнтів під час*

*розробки. Натомість JMeter демонструє вищу продуктивність у сценаріях високого та пікового навантаження, підкреслюючи його здатність моделювати паралельні дії користувачів і здійснювати стрес-тестування застосунків у масштабі. Крім того, різниця у показниках продуктивності, виявлена під час експериментів, свідчить, що JMeter надає більш точну модель застосунків у випадках, коли взаємодія користувачів спричиняє затримки, що робить його особливо корисним для подієво-орієнтованих або сесійних систем. Отримані результати підкреслюють, що жоден інструмент не є універсально оптимальним, а вибір має залежати від конкретних цілей тестування та операційного контексту. Надаючи емпіричні докази сильних і слабких сторін обох інструментів, це дослідження пропонує практичні рекомендації для розробників і QA-інженерів у виборі не лише відповідного інструмента, але й у проєктуванні ефективних стратегій тестування для забезпечення надійності та масштабованості системи.*

***Ключові слова:*** *тестування продуктивності, тестування API, навантажувальне тестування, стабільність системи, Postman, JMeter, аналіз часу відгуку, перцентильні метрики, оцінка продуктивності, прийняття рішень, тестування програмного забезпечення*

UDC 004.05
DOI https://doi.org/10.36059/978-966-397-538-2-13

# PERFORMANCE PERCENTILE ANALYSIS FOR API-BASED TESTING

**Ph.D. S. Khlamov**[1][0000-0001-9434-1081], **M. Mendielieva** [2][0009-0002-4282-3147],
**Ph.D. O. Vovk**[3][0000-0001-9072-1634], **T. Trunova**[4][ 0000-0003-2689-2679],
**Yu. Teslenko**[5][0009-0009-8349-3683]
*Kharkiv National University of Radio Electronics, Ukraine*
*EMAIL: [1]sergii.khlamov@gmail.com, [2]mariia.mendielieva@nure.ua,*
*[3]oleksandr.vovk@nure.ua, [4]tetiana.trunova@nure.ua, [5]yuliia.teslenko@nure.ua*

***Abstract.*** *This paper focuses on the systematic analysis of performance testing metrics, with particular attention to comparing the behavior of two widely used tools, Postman and JMeter, in the context of public application programming interfaces (APIs). API performance testing plays a critical role in evaluating the responsiveness and reliability of modern software systems; however, one persistent challenge lies in relying solely on average response times. Mean values can be easily skewed by anomalous outliers, which often mask significant response delays and lead to an incomplete picture of system performance. To overcome this limitation, the present study emphasizes the use of percentile-based analysis, which provides a more accurate and user-centered indicator of performance for the majority of requests. The methodology involved comparing average response time*