

Comparative Analysis of Jmeter and Postman for API-Based Performance Testing

Sergii Khlamov^{1,*†}, Mariia Mendieliieva^{1,†}, Oleksandr Vovk^{1,†} and Zhanna Deineko^{1,†}

¹ Kharkiv National University of Radio Electronics, Nauki avenue 14, Kharkiv, 61166, Ukraine

Abstract

Testing of user interface (UI) presents numerous challenges, primarily related to labor intensity and the cost of the process, as well as its sensitivity to various factors, such as device type and network conditions, which is especially important when using cloud servers and distributed systems. Unlike UI testing, application programming interface (API) load testing is less dependent on conditions such as hardware or client environments, enabling a more accurate evaluation of backend performance, particularly in cloud infrastructure environments. This paper presents a comparative analysis of the performance of two popular tools for Representational State Transfer (REST) API load testing: Postman and JMeter. The study aimed to identify performance differences between Postman and JMeter when conducting load tests on five public APIs under four types of load. Aggregated response time metrics, including average response time, minimum, maximum, and error percentage values, were collected for each tool. The results demonstrated that Postman shows higher performance in low to moderate load conditions or situations where there is a lower demand for request intensity. On the other hand, JMeter demonstrates better performance under conditions of high load and request intensity. Despite the study's limitations (a single test run on five public APIs), the results suggest that Postman may perform well in scenarios with regular and low load. Alternatively, JMeter is well-suited for high-load scenarios, particularly when high performance is required to handle a large number of requests. We believe the obtained results will have a positive impact on the decision-making process in the development team regarding the choice of test tools, based on the scale and nature of the load. As a result, it can shorten the implementation time and improve the system's overall efficiency.

Keywords

Performance testing, Postman, JMeter, load testing, virtual users, response time, delay

1. Introduction

Test automation involves using software tools to execute tests, verify results, and manage repetitive testing tasks with minimal human intervention [1]. It is often related to continuous integration (CI) in Agile development. The popularity of both CI and test automation is increasing due to market pressure to release product features or updates frequently [2].


There are numerous advantages for companies to use test automation in their projects. Firstly, automation can save a significant amount of time for quality assurance engineers by automating monotonous tasks. A recent report by PractiTest shows that test automation has replaced about 50% of previously manual testing efforts [3]. Secondly, business outcomes, such as the ability to deliver more features with high automation levels and a faster route-to-live that increases the customer base, can be achieved through test automation, as noted by respondents in the 2024 World Quality Report [4]. By using test automation and adopting iterative models in software delivery, it is possible to ensure that software products meet quality standards, reduce the appearance of serious bugs that escape into production [5], and, in the event of an issue or defect arising in the development or test

ICST-2025: Information Control Systems & Technologies, September 24-26, 2025, Odesa, Ukraine

* Corresponding author.

† These authors contributed equally.

✉ sergii.khlamov@gmail.com (S. Khlamov); mariia.mendieliieva@nure.ua (M. Mendieliieva); oleksandr.vovk@nure.ua (O. Vovk); zhanna.deineko@nure.ua (Zh. Deineko)

 0000-0001-9434-1081 (S. Khlamov); 0009-0002-4282-3147 (M. Mendieliieva); 0000-0001-9072-1634 (O. Vovk); 0000-0001-6747-9130 (Zh. Deineko);



© 2025 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

phase, address them efficiently and rapidly using fast feedback. However, there are potential risks associated with using test automation [6], such as inaccurate estimations of time, costs, and effort required to introduce a tool, maintain test scripts, and change test processes. Moreover, one of the significant risks includes unrealistic expectations about the benefits of a tool and using a test tool when manual testing is more appropriate.

Among the different approaches to software testing, User Interface (UI) testing, Application Programming Interface (API) testing, and performance testing are three critical areas that address different aspects of system quality. UI testing is a process of evaluating the correct work and display of UI components of an application on popular types of devices and environments [7]. This type of testing is crucial in user-centric applications, where the end-user experience has a direct impact on the software's success [8]. UI tests based on test frameworks typically simulate real user interactions by locating Document Object Model (DOM) elements, performing browser actions (such as clicking, typing, and scrolling), and inspecting the page state of the UI application.

API testing, on the other hand, evaluates the backend services and communication interfaces that enable interaction, exchanging data and invoking functionalities of different software components or systems [9]. API testing has gained significant importance recently due to the increasing number of microservices, applications, and the dependency of frontend applications on their APIs. For large and complex systems, especially those that rely on a graphical user interface (GUI), UI testing can be inefficient compared to API testing. UI testing is typically performed on the front-end and has a slow execution time [10]. Other challenges of UI automation include the difficulty of performing UI testing at the early stage of the design phase of the project cycle and the flakiness or fragility of UI test scripts due to the complex interactions and events within the DOM of an interface [11]. Moreover, there may be difficulties in maintaining test automation since changes to the UI can be pretty frequent [12]. Therefore, API tests can be executed much faster, as they are not dependent on visual changes in the UI, and can provide more stable and reproducible results. Performance, defined as efficiency and effectiveness of a software application, has become one of the most essential attributes of a product. To achieve a high-quality software product with a growing number of users and increasing data volumes, particularly in cloud applications and microservices architectures, the performance of the user interface and APIs becomes a critical factor [13]. Performance testing is a subset of automated tests used to evaluate an application's behavior under specific conditions. Moreover, performance testing is not executed in all development environments, and it is often implemented within the ecosystem of a CI environment. It can be performed for various devices and services, including mobile devices, websites, online services, and cloud services [14].

There are several popular methodologies or types of performance tests [15]. Load testing involves evaluating a software system's response times and overall performance under various user load 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. Performance issues can appear at the API level, and they can only be identified using specific performance testing tools [16]. 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.

Considering the variety of tools used for performance testing, it is necessary to critically evaluate which one best suits specific needs, particularly when testing APIs. Two commonly used tools for performance testing, such as Postman and JMeter, offer different capabilities and approaches. Postman is known for its simple interface and robust functionality for API testing. At the same time,

JMeter is more focused on load testing and is recognized for its ability to simulate high traffic, providing detailed insights into system performance under load. Moreover, Postman can also run performance tests for APIs in collection and simulate activity of virtual API users, using configurable load durations and profiles [17]. Both JMeter and Postman are powerful tools for test automation; however, their performance under the same conditions may vary significantly.

The purpose of this study is to compare Postman and JMeter as tools for API performance testing. By evaluating their strengths and limitations across four common load types (load, stress, spike, and soak tests) for CRUD public API calls, this research aims to determine which tool is more effective in different testing scenarios, considering performance metrics such as average response time, minimum, maximum values, and error percentage. 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.

2. Literature Review

In the field of performance API testing, various approaches are actively used to simulate real-world traffic patterns and accurately measure system behavior under load. Most of these approaches are based on using performance testing tools to simulate virtual user (VU) requests, as well as performance monitoring and analysis of system stability under load.

Performance testing tools verify the system or application before delivering it to customers, and its efficiency is related to the accuracy of its statistical results [18]. Performance testing tools can be classified into two main categories: cloud-based and on-premise tools. The authors [19, 20] highlight that the cloud-based performance test tools provide scalability, flexibility, and potential cost savings. However, performance testing in the cloud is expensive, as it requires infrastructure spin-up and load generation. On the other hand, on-premise load testing is used in organizations with sensitive data (e.g., healthcare, banking, astronomy [21]) or those that run their application behind a firewall.

The effectiveness of using GUI-based (Graphical User Interface) tools and CLI-based (Command Line Interface) tools was also established by the authors [22-25] in terms of usability, flexibility, and efficiency. GUI-based performance testing tools (e.g., LoadRunner, BlazeMeter, Postman) are widely used due to their user-friendly interfaces, which allow for the design, execution, and analysis of performance tests with minimal technical expertise. However, GUI tools can be slower to run compared to CLI test tools during large-scale load testing. On the other hand, CLI performance tools (JMeter, Locust, Gatling, k6, Artillery, etc.) can generate high loads and simulate thousands of VUs with minimal impact on system performance. Additionally, CLI-based tools are better suited for integration into automated testing workflows and CI/CD pipelines, as they provide immediate feedback without delaying releases. These tools often require a deeper technical understanding, as they rely on script-based configurations and command-line commands to run tests. According to authors [26-28], Apache JMeter is a performance testing tool that allows users to perform load tests 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.

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 performance of both static and dynamic resources. Test scenarios in JMeter can be created with a GUI [29] that allows users to design test plans, configure various types of samplers, and analyze results. JMeter's flexibility is further enhanced by its support for integration with external services and tools such as CI/CD pipelines, monitoring systems (InfluxDB, Prometheus, Grafana, Kibana, Elasticsearch), and third-party performance analysis platforms.

Despite its many advantages, JMeter also has certain limitations, as noted by authors [30, 31]. 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 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 [32, 33] in studies on API performance testing. Postman is a platform for API development and testing, which has emerged as a leading tool for API development with 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. Additionally, Postman enables you to collaborate with teammates by organizing, sharing, and communicating work to APIs. According to authors [34, 35], 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 out of 4 load profiles:

1. Fixed, where a constant number of VUs run 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 an 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 note that during performance test execution in Postman, each virtual user runs the requests in the specified order within 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, which provides a summary of performance metrics available in both tabular and graphical forms.

Thus, analyzing the research results in the reviewed authors' works [36], it is worth noting that there are some limitations to running performance tests in Postman. Firstly, there is a limited number of performance runs that can be used each month at no additional cost. Secondly, the number of VUs in a performance test depends on available system resources and the collection used for the test.

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, performance test scenarios can have only one data file, which is an unlikely scenario in load testing. As shown in the work [37], Postman outperformed JMeter and Robot Framework in various data environments. The relevance of all these studies is undeniable, as modern web services require testing to ensure their reliability and performance.

3. Methodology

To perform performance testing, a set of public APIs should be selected for testing. Using public APIs under various types of load is not considered a DDoS attack or unauthorized use of resources, and this approach does not violate their ethical use.

Test cases should include executing different types of requests, such as GET, POST, PUT, DELETE, because this helps to simulate real-world interactions with public APIs. These HTTP methods represent the typical operations that users or systems perform when interacting with an API. Each request type generates different types of server 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, 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.

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 measure the system's performance under sustained load accurately.

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. Obtained performance test cases should be performed for all load profiles, Ramp Up, Spike, Peak, Fixed, in all five public APIs in both tools, JMeter and Postman.

3.1. API selection

Public websites with open APIs were selected for performance testing. These websites provide ready-made API endpoints with simulated real data, allowing you to quickly start testing without having to develop your backend:

1. *ReqRes.in* website (<https://reqres.in>) was used for performance testing as it provides a simple and accessible way to simulate real API requests and responses, allowing for modeling various load scenarios. This can help to focus on testing the performance of the client application, without the need to configure a complex server infrastructure, and test the system under various conditions.

2. *DummyJSON* is an online service (<https://dummyjson.com>) that provides a range of pre-configured REST APIs, offering mock data for testing, development, and prototyping. DummyJSON provides data that mimics real-world APIs, allowing developers to simulate different levels of data size, from small datasets to larger collections, through pagination and various endpoints. This helps in testing how the application handles large volumes of data and performs under stress conditions.

3. *SampleAPIs* is an online platform (<https://sampleapis.com>) that offers a collection of free, publicly accessible mock APIs. Developers can simulate various operations, including fetching large sets of data, creating, updating, or deleting resources, which are everyday tasks for front-end applications. By testing these actions under heavy load, developers can evaluate the performance and scalability of their applications when performing these CRUD operations.

4. *JsonPlaceholder* (<https://jsonplaceholder.typicode.com>) provides a diverse set of realistic mock data that can be used to simulate real-world interactions, such as loading user profiles, creating posts, or fetching a list of comments. This variety makes it an ideal choice for testing how applications perform under different scenarios, including handling user-generated content, displaying lists, and updating resources. Developers can use JsonPlaceholder to simulate how their applications interact with APIs when under load. The ability to make multiple simultaneous requests to various endpoints enables performance testing, such as simulating the behavior of an application under high traffic, load, or stress.

5. *FakeStoreAPI* is a free online service (<https://fakestoreapi.com>) that provides a set of mock APIs designed to simulate e-commerce store interactions, offering realistic product data in a structured

format. FakeStoreAPI enables developers to simulate a vast product catalog comprising thousands of items. This makes it an ideal tool for testing how an application handles large datasets, such as loading hundreds of products in an e-commerce store. Developers can use FakeStoreAPI to simulate load testing by sending multiple simultaneous requests to various endpoints (e.g., retrieving product lists, adding items to the cart).

3.2. Limitations of comparison

It is important to note that although the load parameters were standardized, the execution architecture differs between the tools. In Postman, each virtual user 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.

Think Time can be implemented in JMeter using the Constant Timer element, which is shown in Figure 1a. 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 1a (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 1b.

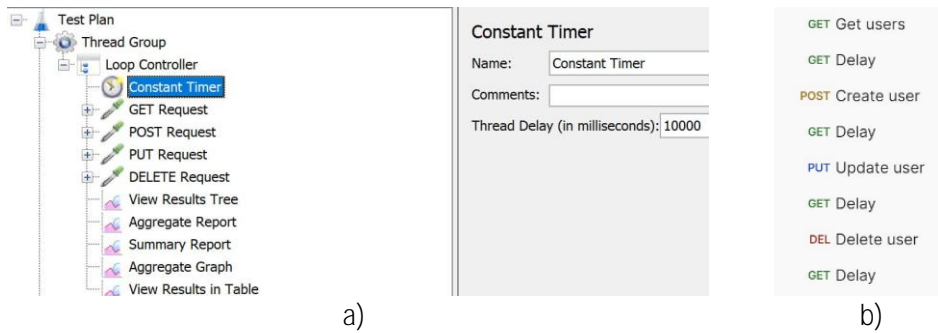


Figure 1: Test cases structure: a) JMeter Loop Controller with requests; b) Postman collection of requests.

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.

Obtained performance test results for five public APIs should be collected, and received performance metrics (average response time, min, max values and error %) should be analyzed. Then, 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:

$$\Delta Avg = Postman_{Avg} - JMeter_{Avg}, \quad (1)$$

where ΔAvg – difference in average response time between Postman and JMeter;

$Postman_{Avg}$ – value of average response time in Postman;

$JMeter_{Avg}$ – value of average response time in JMeter.

In case ΔAvg is a positive value, then Postman had a longer average response time than JMeter. If ΔAvg is a negative value, then JMeter's average response times exceeded those in Postman, and Postman's performance was faster. Calculating the aggregated mean values for performance metrics across all test cases for each (API, test tool, HTTP method) combination helps to provide a

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

$$\text{Avg_metric} = \frac{\sum_{k=1}^n m_k}{n}, \quad (2)$$

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 selected HTTP method:

$$\text{Mean_metric} = \frac{\sum_{i=1}^l \text{Avg_metric}_i}{l}, \quad (3)$$

where **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.

4. Results

The proposed performance testing approach was applied to a set of 5 public APIs using two tools: Postman and JMeter. Response times were measured for various HTTP methods (GET, POST, PUT, DELETE), after which the following metrics were collected: average response time (avg), minimum (min), maximum (max), and error rate (error %). Performance tests were conducted under identical conditions:

- Number of VU: 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.

For evaluating 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). 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.

Table 1
Performance Test Cases for JMeter and Postman

TC	Load Profile	VU	Think Time, s	Duration, min	Comment
1	Ramp Up	0 – 10	10	10	Increase of load
2	Ramp Up	10 – 30	3	10	Increase of load
3	Spike	1 – 10 – 1	10	10	Users attack simulation
4	Spike	5 – 50 – 5	2	5	Users attack simulation
5	Fixed Load	10	10	10	Stable request flow
6	Fixed Load	20	5	10	Stable request flow
7	Peak	2 – 10 – 2	10	10	Check of requests maximum
8	Peak	8 – 40 – 8	1	5	Check of requests maximum

The TC1 Ramp Up load profile in Postman with 10 VUs, an initial load of 0 VUs, and a test duration of 10 minutes enables the following scenario: steadily ramp up to 10 users for 5 minutes, with each user executing all requests sequentially, as shown in Figure 2a. To obtain an equivalent load in JMeter, a Thread Group with 10 VUs and ramp-up period = 300 s and duration = 600 s was used, as shown in Figure 2b. In this case, the Loop Count was set to 1 in the Loop Controller of JMeter.

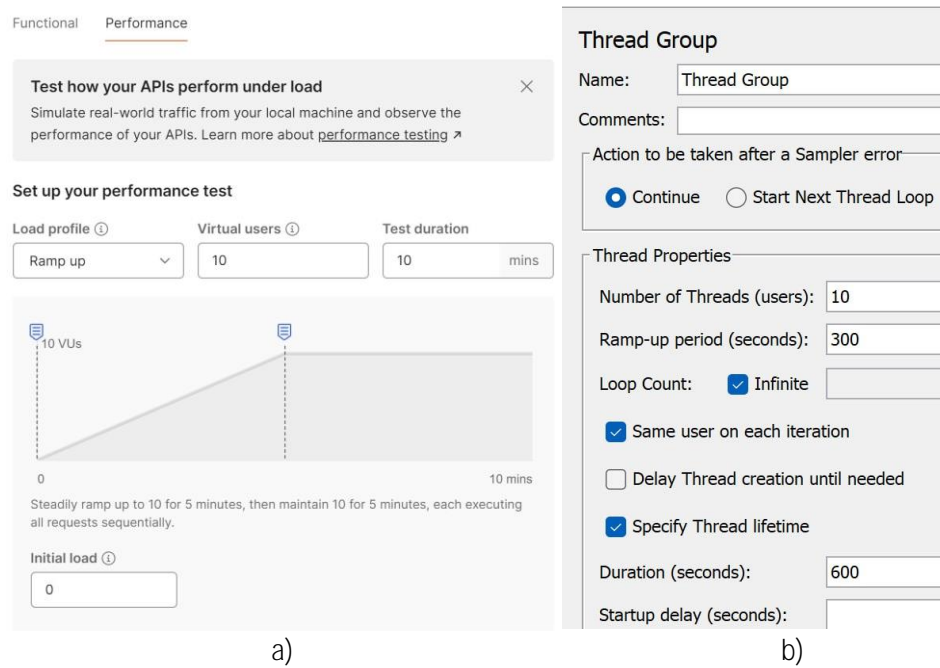


Figure 2: Ramp Up load parameters with 10 VUs and 0 VUs as initial load, test duration 10 mins: a) Postman load profile; b) JMeter Thread Group plugin configuration.

TC2 Ramp Up load profile in Postman with 30 VUs, initial load = 10 VUs, test duration = 10 mins enables the following scenario: 10 VUs run for 2 mins 30 s, then ramp up to 30 VUs for 2 mins 30 s, after that maintain 30 VUs for 5 minutes, each executing all requests sequentially, as shown in Figure 3a. 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 this case and subsequent one, the Loop Count was set to 'Infinite' in the Loop Controller of JMeter.



Figure 3: Ramp Up load parameters with 30 VUs and 10 VUs as initial load, test duration 10 mins: a) Postman load profile; b) Ultimate Thread Group plugin configuration in JMeter.

TC3 Spike load profile in Postman with 10 VUs, base load = 1 VU, test duration = 10 mins enables the following scenario: 1 VU runs for 4 minutes, then spikes to 10 VUs over 1 minute, drops to 1 VU

over 1 minute, maintains 1 VU for 4 minutes, each executing all requests sequentially. 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.

TC4 Spike load profile in Postman with 50 VUs, base load = 5 VUs, test duration = 5 mins enables the following scenario: 5 VUs run for 2 minutes, then spikes to 50 VUs over 30 sec, drops to 5 VUs over 30 sec, maintains 5 VUs for 2 minutes, each executing all requests sequentially. 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.

TC5 Peak load profile in Postman with 10 VUs, base load = 2 VUs, test duration = 10 mins enables the following scenario: 2 VUs run for 2 minutes, ramp up to 10 VUs over 2 min, maintains 10 for 2 min, decrease to 2 over 2 min, maintains two over 2 min, each executing all requests sequentially. 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.

TC6 Peak load profile in Postman with 40 VUs, base load = 8 VUs, test duration = 5 mins enables the following scenario: 8 VUs run for 1 min, ramp up to 40 VUs over 1 min, maintains 40 for 1 min, decrease to 8 over 1 min, maintains eight over 1 min, each executing all requests sequentially, as shown in Figure 8a. 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.

TC7 Fixed load profile in Postman with 10 VUs, test duration = 10 minutes, enabling the following scenario: 10 VUs run for 10 minutes, each executing all requests sequentially. To obtain an equivalent load in JMeter, Concurrency Thread Group was used: 1) target concurrency = 10, hold target rate = 10 minutes. TC8 Fixed load profile in Postman with 20 VUs, test duration = 10 mins enables the following scenario: 20 VUs run for 10 min, each executing all requests sequentially. To obtain an equivalent load in JMeter, a Concurrency Thread Group was used with the following settings: 1) target concurrency = 20, hold target rate = 20 minutes.

Testing of CRUD operations was performed on the following endpoints:

1. For *ReqRes.in* service, GET, POST, PUT, and DELETE requests were used for 'users' resource <https://reqres.in/api/users>, as illustrated in Figure 4:

```
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 4: Requests to ReqRes.in 'users' resource.

It should be noted that `{{postId}}` in Figure 4 is a generated ID for a new user by the service in the POST request. This `{{postId}}` was saved after executing the POST request using variables and the `pm.environment.set()` function in Postman collections, and correspondingly, by using the Regular

Expression Extractor in JMeter tests. After that, $\{\{postId\}\}$ was passed to PUT and DELETE requests. Such an approach was implemented in tests for ReqRes.in, SampleAPIs, FakeStoreAPI services. For DummyJSON and JsonPlaceholder, the existing services' values were used for testing due to the limitations of these services.

2. For *DummyJSON* service, GET, POST, PUT, and DELETE requests were used for '*products*' resource <https://dummyjson.com/products> in a similar manner as shown in Figure 4.

3. For *SampleAPIs* service, GET, POST, PUT, and DELETE requests were used for '*codingResources*' resource <https://api.sampleapis.com/codingresources/codingResources>.

4. For *JsonPlaceholder* service, GET, POST, PUT, and DELETE requests were used for '*posts*' resource <https://jsonplaceholder.typicode.com/posts>.

5. For *FakeStoreAPI* service, GET, POST, PUT, and DELETE requests were used for '*products*' resource <https://fakestoreapi.com/products>.

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.

During the experiment, the following tools were utilized: Postman Desktop application version 11.50.2 and Apache JMeter version 5.6.3. The scripts were executed in an identical environment: Windows 11 x64 24H2, 8-core CPU, 16GB RAM. Obtained test cases (TC) results are presented both in tabular form and through visualization (bar chart) for different load profiles in both tools—gradual increase in load from 0 to 10 VUs over 10 minutes.

Table 2

Performance results for TC1 Postman, ReqRes.in service

Method	Samples, N	Minimum, ms	Maximum, ms	Average, ms	Error, %
GET	112	47	199	64	0
POST	109	98	133	112	0
PUT	106	103	151	115	0
DELETE	104	99	136	109	0

Table 3

Performance results for TC1 JMeter, ReqRes.in service

Method	Samples, N	Minimum, ms	Maximum, ms	Average, ms	Error, %
GET	117	40	157	77	0.00
POST	115	113	564	134	0.00
PUT	113	118	263	159	0.01
DELETE	110	111	185	130	0.01

Difference in average response time values (deltas) were calculated using Formula (1). A positive value indicates a higher Postman value, meaning a longer response time (see Table 4).

Table 4

Difference in average response time values between Postman and JMeter for ReqRes.in service in TC1

Method	Average Postman, ms	Average JMeter, ms	Δ Average, ms
GET	64	77	-13
POST	112	134	-22
PUT	115	159	-44
DELETE	109	130	-21

The results obtained, as shown in Tables 2-4, indicate that the average response time in Postman is less than that in JMeter for all types of requests in TC1.

Similarly, average response time deltas were calculated for other public APIs. The difference between the average response time values is further illustrated in the chart in Figure 5, where it is clear that Postman shows lower average response times for 4 out of 5 APIs.

This visualization helps to ensure that the differences between the tools are not random fluctuations, but are consistent across all APIs.

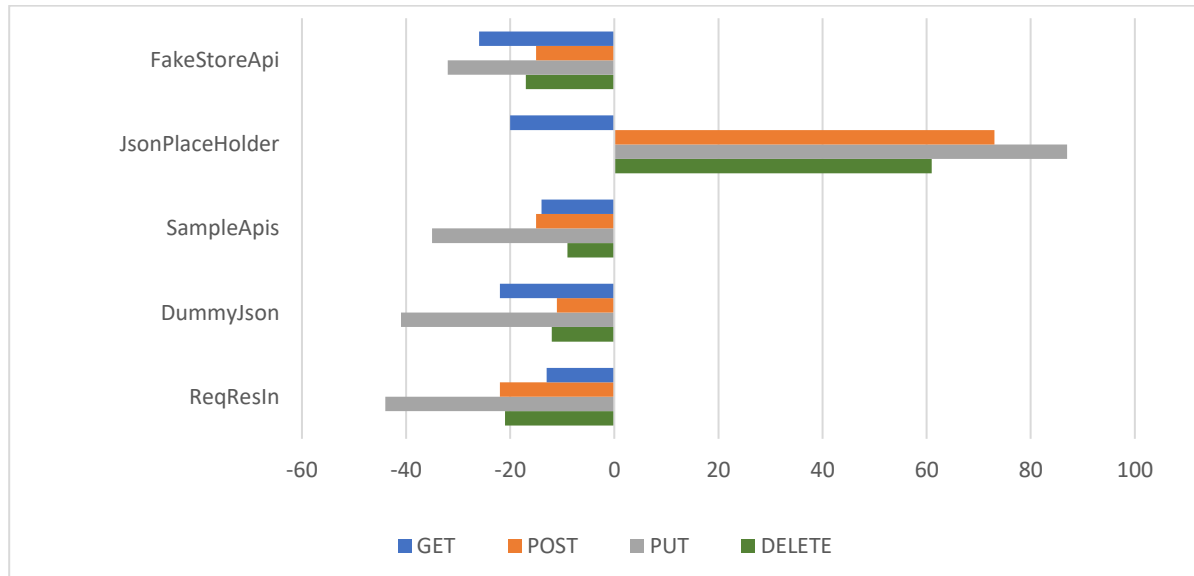


Figure 5: Deltas of Average response time values for public APIs for TC1.

The same results were obtained in other test cases, except in cases where the think time was reduced to 5 seconds or less, or in cases where there was a significant increase in the number of VUs in combination with a short think time. In all these cases, JMeter demonstrated better performance, as evident in the experimental results for TC6 in Figure 6.

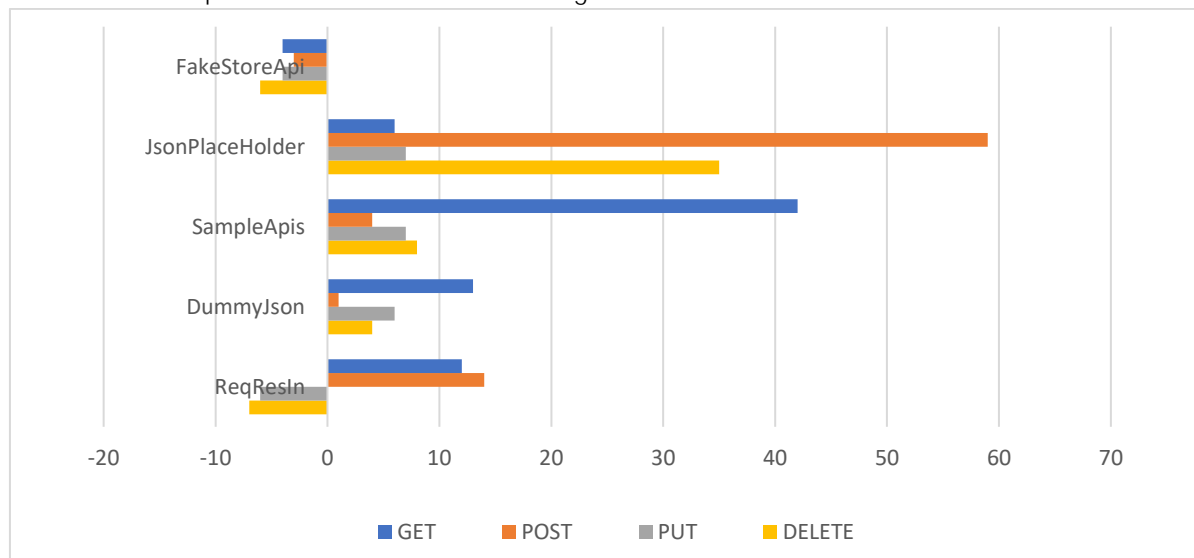


Figure 6: Deltas of Average response time values for public APIs for TC6.

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. To ensure statistical objectivity [38] and reduce case-specific variability, average

response times, min, max values and error % were first collected per test case, then averaged across 8 test cases for each combination of API and HTTP method, as described in Formulas (2) and (3).

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 5.

Table 5
Aggregated performance results for Postman and JMeter

Method	Test Tool	Mean Avg, ms	Mean Min, ms	Mean Max, ms	Mean Error, %
GET	Postman	162,400	104,975	410,200	0,9348
GET	JMeter	181,650	115,925	499,625	0,739
POST	Postman	148,100	117,250	393,035	7,671
POST	JMeter	147,175	124,700	806,050	7,999
PUT	Postman	144,925	118,100	341,425	7,777
PUT	JMeter	151,625	125,525	377,275	7,936
DELETE	Postman	166,189	136,561	307,629	7,931
DELETE	JMeter	161,725	143,525	374,475	8,515

The final aggregated values in Table 5 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 [39]. These data provide a preliminary conclusion that Postman shows higher performance in low to moderate load conditions, particularly during stable loads with a fixed amount of VUs, and in other types of loads where think time is greater than 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 performance testing cases including mathematical methods [40].

5. Discussion

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 in each collection in both test tools, allowing for a reliable assessment of tools' behavior at the API level. The visualization of the differences confirms a consistent trend, where Postman demonstrates better performance across the average response time metric, particularly when there are not many VUs and the think time is more than 5 seconds for all load profiles. In other cases, JMeter shows faster results in performance metrics. Since public open APIs were used, the exact number of active users at the time of the performance test execution remains unknown. This introduces an element of uncertainty, as the actual load on the system could vary based on the number of concurrent users accessing the API during the test period. However, performance tests were conducted at the same time of day during the experiment to minimize this uncertainty. In our opinion, additional experiments with repeated execution of scenarios can be applied to confirm the robustness of the observed effects.

6. Conclusion

In conclusion, the conducted research showed that Postman performs more efficiently under conditions of low to medium user concurrency or when request rates are relatively low. In contrast, JMeter is more appropriate for scenarios involving high concurrency, where the system must sustain substantial request loads.

Postman shows high performance, particularly during cases as stable load with fixed amount of VUs (fixed load profile), with gradual increase in load (ramp up profile with initial load as 0) and in

other types of loads where think time value represents a relatively long delay between user actions, significantly reducing the request rate per user.

JMeter achieves better performance in scenarios with high load and high request intensity. Such performance behavior is most pronounced in high-intensity scenarios involving a rapid ramp-up to peak load, a gradual ramp-down, and minimal delays between user actions.

The obtained data could have a profound impact on improving both the stability and effectiveness of software development, especially in the context of selecting appropriate testing tools and practices. Considering factors such as the complexity of returned responses, planned load parameters (e.g., number of virtual users, think time, and test duration), it is possible to ensure a reduction in testing time and the risk of underestimating the system's performance in high-traffic conditions.

We believe that using Postman as a tool for performance testing in low to moderate load conditions would provide a step forward for small and medium-sized organizations and firms in their efforts to achieve a good software quality level. Additionally, using JMeter could be beneficial for complex cloud servers [41] and distributed systems including decision-making process [42] with high load and request intensity [43].

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

- [1] BrowserStack, Test Automation Benefits, Limitations, Tools, Best Practices 2025. URL: <https://www.browserstack.com/guide/what-is-test-automation>.
- [2] Y. Wang, M. V. Mäntylä, Z. Liu, J. Markkula, Test automation maturity improves product quality – Quantitative study of open source projects using continuous integration, *Journal of Systems and Software* 188 (2022). doi: 10.1016/j.jss.2022.111259.
- [3] PractiTest, The 2024 State of Testing™ Report, 2024. URL: <https://www.practitest.com/assets/pdf/stot-2024.pdf>.
- [4] 16th edition of World Quality Report, 2024, URL: <https://www.capgemini.com/wp-content/uploads/2024/10/WQR-24-MAIN-REPORT-CG.pdf>.
- [5] Cloudzero, Cloud Computing Statistics report, 2025, URL: <https://www.cloudzero.com/blog/cloud-computing-statistics>.
- [6] R. Cerquozzi, W. Decoutere, J.-F. Riverin, ISTQB Certified Tester Foundation Level Syllabus, V4.0.1, 2024.
- [7] BrowserStack, UI Testing Guide, 2025. URL: <https://www.browserstack.com/guide/ui-testing-guide>.
- [8] O. Vovk, I. Chebotarova, M. Mendieliava, Approach to comprehensive website testing: combining usability and functional test methods, in: *Proceedings of the 10th annual conference on Print, Multimedia & Web. Modern Trends, LLC "Drukarnya Madrid", Kharkiv, Ukraine*, vol. 1, (2025) 5-30. doi: 10.30837/PMW.2025.T1.005.
- [9] A. Owen, *Microservices Architecture and API Management: A Comprehensive Study of Integration, Scalability, and Best Practices*, International University of Applied Sciences, 2025.
- [10] TestSigma, API vs UI testing, 2025. URL: <https://testsigma.com/blog/api-vs-ui-testing>.
- [11] Y. Pei, J. Sohn, M. Papadakis, An Empirical Study of Web Flaky Tests: Understanding and Unveiling DOM Event Interaction Challenges, in: *Proceedings of the Conference on Software Testing, Verification and Validation (ICST)*, (2025) 92-102. doi: 10.1109/ICST62969.2025.10989030.
- [12] M. Nass, E. Alégroth, R. Feldt, Why many challenges with GUI test automation (will) remain, *Information and Software Technology* 138 (2021) 106625. doi: 10.1016/j.infsof.2021.106625.
- [13] Akamai, What Is API Performance Testing, 2025. URL: <https://www.akamai.com/glossary/what-is-api-performance-testing>.

- [14] M. Yenugula, R. Kodam, D. He. Performance and load testing: Tools and challenges, *International Journal of Engineering in Computer Science* 1 (2019) 57-62. doi: 10.33545/26633582.2019.v1.i1a.102.
- [15] S. Pargaonkar, A Comprehensive Review of Performance Testing Methodologies and Best Practices: Software Quality Engineering, *International Journal of Science and Research (IJSR)* 12 8 (2014). doi: 10.21275/SR23822111402.
- [16] M. Hendayun, A. Ginanjar, Y. Ihsan, Analysis of application performance testing using load testing and stress testing methods in API service, *Jurnal Sisfotek Global* 13 1 (2023) 28-34. doi: 10.38101/sisfotek.v13i1.2656.
- [17] Postman, Performance Testing, 2024. URL: <https://www.postman.com/templates/collections/performance-testing>.
- [18] A. Jacob, A. Karthikeyan, Scrutiny on Various Approaches of Software Performance Testing Tools, in: *Proceedings of the Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, (2018) 509-515. doi: 10.1109/ICECA.2018.8474876.
- [19] LoadViewTesting, Cloud vs. On-Premise Load Testing: An ROI Comparison, 2025. URL: <https://www.loadview-testing.com/learn/roi-comparison-cloud-vs-premise-load-testing-tools>.
- [20] LoadNinja, Should You Use Cloud-Based or On-Premise Load Testing, 2025. URL: <https://loadninja.com/articles/cloud-based-on-premise-load-testing>.
- [21] V. Troianskyi, V. Kashuba, O. Bazyey, et al., First reported observation of asteroids 2017 AB8, 2017 QX33, and 2017RV12, *Contributions of the Astronomical Observatory Skalnat Pleso* 53 2 (2023) 5-15. doi: 10.31577/caosp.2023.53.2.5.
- [22] D. Westerveld, *API Testing and Development with Postman: API creation, testing, debugging, and management made easy*, Packt Publishing Ltd, 2024.
- [23] N. Srivastava, U. Kumar, P. Singh, Software and performance testing tools, *Journal of Informatics Electrical and Electronics Engineering* 2 1 (2021) 1-12. doi 10.54060/jieee/002.01.001.
- [24] S. Khlamov, et al., The astronomical object recognition and its near-zero motion detection in series of images by in situ modeling, in: *Proceedings of the International Conference on Systems Signals and Image Processing* (2022) 1-4. doi: 10.1109/IWSSIP55020.2022.9854475.
- [25] M. Pirah, H. Tahseen, B. Sania, S. S. Qureshi, Comparative study of testing tools Blazemeter and Apache JMeter, *Journal of Computing and Mathematical Sciences* 2 1 (2018) 70-76. doi: 10.30537/sjcms.v2i1.66.
- [26] F. Okezie, I. Odun-Ayo, S. Bogle, A critical analysis of software testing tools, *Journal of Physics: Conference Series* 1378 4 IOP Publishing, (2019). doi: 10.1088/1742-6596/1378/4/042030.
- [27] I. Indrianto, Performance testing on web information system using apache jmeter and blazemeter, *Jurnal Ilmiah Ilmu Terapan Universitas Jambi* 7 (2023) 138-149. doi: 10.22437/jiituj.v7i2.28440.
- [28] T. Vatsya, U. Sachin, K. G. Jayati, A. Sanjiv, Analytical evaluation of web performance testing tools: Apache JMeter and SoapUI, in: *Proceedings of the 12th International Conference on Communication Systems and Network Technologies*, (2023). doi: 10.1109/CSNT57126.2023.10134699.
- [29] S. Siddhant, S. B. Prapulla, Comprehensive review of load testing tools, *International Research Journal of Engineering and Technology* 7 5 (2020) 3392-3395.
- [30] AutomateNow, Advantages and Disadvantages of using JMeter, 2023. URL: <https://automatenow.io/advantages-and-disadvantages-of-using-jmeter>.
- [31] TestSigma, JMeter vs Postman, 2025. URL: <https://testsigma.com/blog/jmeter-vs-postman>.
- [32] Postman, What is Postman, 2025. URL: <https://www.postman.com/product/what-is-postman>.
- [33] S. D. Sri, M. S. Aadil, S. R. Varshini, R. Raman, G. Rajagopal, S. T. Chan, Automating REST API Postman Test Cases Using LLM, *arXiv preprint:2404.10678* (2024). doi: 10.48550/arXiv.2404.10678.
- [34] H. A. Thooriqoh, B. M. Mulyo, A. Rakhmadi, Advanced RESTful API Testing: Leveraging Newman's Command-Line Capabilities with Postman Collections, in: *Proceedings of the 10th Information Technology International Seminar, Indonesia*, (2024) 188-193, doi: 10.1109/ITIS64716.2024.10845315.

- [35] V. S. Susan Rini, When Postman Goes That Extra Mile to Deliver Performance to APIs, Software Testing Magazine, 2024. URL: <https://www.softwaretestingmagazine.com/tools/when-postman-goes-that-extra-mile-to-deliver-performance-to-apis>.
- [36] NashTech, Performance testing with Postman: Is it worth?, 2024. URL: <https://blog.nashtechglobal.com/performance-testing-with-postman-is-it-worth>.
- [37] C.-H. Hsieh, Z. Wang, Q. Zhang, Y. Song, X. Wu, Z. Wang, Evaluation System for Software Testing Tools in Complex Data Environment, in: Proceedings of the 4th International Conference on Information Communication and Signal Processing, 2021. doi: 10.1109/ICICSP54369.2021.9611846.
- [38] V. Shvedun, et al., Statistical modelling for determination of perspective number of advertising legislation violations, Actual Problems of Economics 184 10 (2016) 389-396.
- [39] S. Khlamov, et al., Machine Vision for Astronomical Images using The Modern Image Processing Algorithms Implemented in the CoLiTec Software, Measurements and Instrumentation for Machine Vision, (2024) 269-310. doi: 10.1201/9781003343783-12.
- [40] V. Savanevych, et al., Mathematical methods for an accurate navigation of the robotic telescopes, Mathematics 11 10 (2023) 2246. doi: 10.3390/math11102246.
- [41] S. Khlamov, et al., Astronomical knowledge discovery in databases by the CoLiTec software, in: Proceedings of the 12th IEEE ACIT 2022, Ruzomberok, Slovakia, September 26th – 28th, (2022) 583-586. doi: 10.1109/ACIT54803.2022.9913188.
- [42] Y. Romanenkov, V. Mukhin, V. Kosenko, et al., Criterion for Ranking Interval Alternatives in a Decision-Making Task, International Journal of Modern Education and Computer Science (IJMECS) 16 2 (2024) 72-82. doi: 10.5815/ijmeecs.2024.02.06.
- [43] S. Khlamov, et al., CoLiTec software for the astronomical data sets processing, in: Proceedings of the IEEE 2nd International Conference on Data Stream Mining and Processing, Lviv, Ukraine, August 21st – 25th, 2018, pp. 227-230. doi: 10.1109/DSMP.2018.8478504.