An Empirical Comparison of Selenium and Puppeteer for End-to-End Web Testing: The SauceDemo Case Study

Sarah Azizah[1], Suastika Yulia Riska[2]

[1,2]Informatics Engineering Study Program, Faculty of Technology and Design, Institut Teknologi dan Bisnis Asia Malang, Jl. Soekarno-Hatta No. 1A, Malang, East Java, Indonesia

Keywords
Automation Testing; Selenium; Puppeteer; Performance Evaluation; Web Application Testing

Correspondence Email:
riska.suastika@asia.ac.id

Abstract

This study presents an empirical comparison between two leading web automation frameworks, Selenium and Puppeteer, based on end-to-end testing of the SauceDemo web application. The research aims to measure and evaluate both frameworks in terms of performance, including execution time, CPU utilization, and memory and effectiveness, which covers code complexity, ease of implementation, and maintainability. Identical functional test scenarios such as login validation, product filtering, cart management, and checkout completion were developed in Selenium (Python) and Puppeteer (Node.js) and executed under the same environmental conditions. The quantitative measurements revealed Puppeteer's average execution time of 4.7 seconds, which was approximately 45% faster than Selenium's 8.4 seconds. In addition, Puppeteer consumed 35% less CPU and 34% less memory on average, demonstrating higher efficiency in resource utilization. Qualitative evaluation revealed that Puppeteer required fewer lines of code and a simpler setup process, resulting in better maintainability, whereas Selenium remained superior in terms of cross-browser compatibility and integration flexibility. These empirical findings confirm that architectural design significantly influences automation testing performance. The results provide data-driven insight for quality assurance professionals and developers in selecting the appropriate framework according to testing requirements, scalability, and environmental constraints.

1. Introduction

Empirical evaluation of web automation frameworks has become increasingly crucial as the complexity and dynamism of modern web applications continue to grow, requiring not only functional accuracy but also high execution efficiency and long-term maintainability (Afrihyia et al., 2022). Software testing plays a vital role in ensuring that every system component operates according to its specifications and maintains reliability under diverse operating conditions (Damayanthi, 2024). In line with the continuous integration and continuous

delivery (CI/CD) paradigm in modern software engineering, automated testing has become an essential component for early regression detection and continuous quality assurance (Ruseno et al., 2025). Several studies have demonstrated that automation testing contributes significantly to reducing long-term development costs and accelerating software release cycles (Sariwardani & Si, 2024).

Among the various automation tools available, Selenium and Puppeteer have emerged as two of the most widely adopted frameworks, each offering distinct architectural foundations and execution environments. Selenium operates using the WebDriver protocol, which supports multiple browsers and programming languages, while Puppeteer, developed by Google, communicates directly with Chromium through the Chrome DevTools Protocol (CDP). These architectural differences influence several critical aspects of performance, such as execution time, resource utilization, and implementation complexity (García et al., 2024). In addition, previous findings indicate that the efficiency of interaction with the Document Object Model (DOM) and rendering engine is a determining factor in the overall performance of automation frameworks (Wihardjo, 2025).

Although several studies have compared web automation frameworks, most have focused primarily on qualitative dimensions such as usability, compatibility, or ecosystem support without presenting empirical data that reflects measurable system-level performance under controlled conditions (Rizaldi et al., 2022). With the growing need for data-driven evaluation, there is a clear research gap in analyzing the actual performance behavior of different frameworks when executed under identical test scenarios (Dewandra Sapto Prasetyo & Silfianti, 2023). Addressing this gap, the present study conducts an empirical comparison between Selenium and Puppeteer using the SauceDemo web application, a standardized e-commerce simulation commonly employed in web automation testing. The evaluation focuses on key performance indicators, including execution time, CPU utilization, and memory consumption, as well as qualitative aspects such as code complexity and maintainability. Each framework is tested through end-to-end scenarios covering login, product navigation, shopping cart operations, and checkout processes. By applying a controlled experimental design, this study seeks to generate quantitative evidence and objective comparative analysis to determine the more efficient and practical framework for implementation. The findings are expected to provide useful insights and empirical references for software testing practitioners, developers, and researchers in selecting automation technologies suited to the requirements of modern web-based systems.

1.1 Literature Review

Software testing is a structured and systematic process used to verify and validate that a software system operates in accordance with user requirements and established technical specifications (Prakoso & Sujarwo, 2022). Within the Software Development Life Cycle (SDLC), testing serves as a critical quality assurance mechanism that ensures system reliability, efficiency, and usability across various operating environments (Ruliansyah et al., 2023). The development of automated testing technologies has enabled faster and more accurate validation compared to manual methods, particularly in large-scale or continuously updated projects (Prasetyo & Setiani, 2025). Several studies have also confirmed that automated testing improves reproducibility and reduces the likelihood of human error in verification processes (Ginting & Lubis, 2024).

Selenium is one of the most established automation frameworks that operates using the WebDriver protocol to facilitate browser automation across multiple platforms (Sofani, 2023). It supports a wide range of programming languages, including Python, Java, and C#, and integrates seamlessly with continuous integration and delivery (CI/CD) pipelines (Thooriqoh et al., 2021). Its key strengths lie in its flexibility and community support; however, Selenium's indirect communication with browsers introduces additional overhead that can impact execution performance, particularly in scenarios involving heavy DOM interaction (Amalia, 2022). Recent studies indicate that Selenium's response time tends to increase in large-scale test suites due to the additional WebDriver translation layer (Fathoni et al., 2025).

In contrast, Puppeteer is a Node.js-based testing framework that leverages the Chrome DevTools Protocol (CDP) to communicate directly with the Chromium browser engine (Umam, 2024). This direct communication model eliminates the need for intermediary drivers, resulting in faster execution time and more efficient CPU utilization (Mulyadi, 2023). Nevertheless, Puppeteer's reliance on Chromium-based browsers remains a key limitation, as it restricts cross-browser compatibility. Additionally, several authors have noted that Puppeteer can present higher complexity when managing asynchronous scripting compared to Selenium (Hasan, 2025). Despite these challenges, Puppeteer's lightweight architecture and real-time debugging capabilities make it a strong alternative for performance-focused automation workflows.

A comparative study by (Moń & Pańczyk, 2025) involving Selenium, Playwright, and Cypress concluded that while these frameworks provide similar functional capabilities, they differ significantly in efficiency, API design, and communication architecture. However, most of the existing comparisons remain conceptual and lack empirical measurement of resource usage and execution time. (Maulana & Herwanto, 2025) emphasized the need for real-world benchmarking that quantifies the impact of architectural design on testing performance, especially regarding execution time and memory consumption under identical scenarios.

To address this research gap, the present study conducts parallel and measurable testing on the SauceDemo web application to collect quantitative metrics such as execution time, CPU utilization, and memory consumption, as well as qualitative indicators such as setup complexity and code maintainability. The empirical evidence derived from these experiments is expected to strengthen the academic foundation for evaluating the advantages and limitations of both Selenium and Puppeteer from technical and practical perspectives. Moreover, it aims to contribute to the development of more efficient and adaptable automation testing methodologies for future software engineering research.

2. Research Methods

This study employs an empirical quantitative–comparative research design aimed at measuring and analyzing the performance and effectiveness of two web automation frameworks, Selenium and Puppeteer, when applied to identical end-to-end testing scenarios. The objective of this method is to obtain data-driven evidence of their execution efficiency, resource utilization, and maintainability in a realistic web testing environment.

The experiment was conducted using the SauceDemo web application, which simulates a complete e-commerce system including login, product navigation, cart operations, and checkout processes. The platform provides a consistent testing environment for reproducing real-world user interactions across both frameworks.

2.1 Research Design

The research follows an experimental comparative approach consisting of four main stages. In the scenario definition stage, identical end-to-end test cases were designed to cover major web testing functionalities, including login validation, inventory filtering, cart manipulation, and checkout completion.

In the framework implementation stage, equivalent automation scripts were developed using Selenium in Python and Puppeteer in Node.js, ensuring uniform logic, element locators, and test flows across both tools. In the execution and monitoring stage, each framework was executed under identical system configurations while performance metrics such as execution time, CPU usage, and memory consumption were recorded automatically. In the evaluation stage, both quantitative and qualitative data were analyzed to compare framework efficiency, complexity, and ease of maintenance. This structured workflow ensures the empirical accuracy and reproducibility of all experimental results.

2.2 Sampling

The sampling process in this study refers to the selection of functional test cases rather than human respondents. Seven functional scenarios were selected from the SauceDemo web application because they

represent key components of a standard e-commerce workflow and involve comprehensive user–system interactions.

Table 1. Functional Test Scenarios

No	Scenario	Description	Expected Output
1	Login Validation	Tests various combinations of valid, invalid, and locked credentials	Successful login or error message
2	Inventory Filtering	Sorts products by name and price	Items displayed in correct order
3	Product Details	Accesses and returns from product detail pages	Product information displayed accurately
4	Add and Remove Cart	Adds and removes products from the cart	Cart count and content updated correctly
5	Checkout Form Validation	Tests incomplete and complete form submissions	Validation message or successful navigation
6	Checkout Completion	Executes a full purchase process until the confirmation page	"Thank You" message displayed
7	Navigation Functions	Tests menu links including All Items, About, and Logout	Correct page redirection

All test scenarios were executed using identical functional logic to ensure experimental parity between the two frameworks.

2.3 Data Collection

Data collection was performed through automated execution and system monitoring. Both frameworks were configured to run the same test scenarios sequentially under identical conditions.

The data collected consist of two categories. Quantitative data include execution time measured in seconds, CPU utilization measured in percentage, and memory consumption measured in megabytes. These metrics were automatically captured during runtime. Qualitative data include the number of lines of code (LOC) to indicate script complexity, the ease of setup, and the level of maintainability observed during the development and modification of test scripts. All tests were conducted in a controlled environment to ensure accuracy and repeatability.

Each framework's test scripts were executed in a controlled environment:

Table 2. Controlled Environment

Parameter	Specification
Processor	Intel Core i5-1135G7 (2.4 GHz, 4 cores)
Memory	8 GB RAM
Operating System	Windows 11 (64-bit)
Browser	Google Chrome v122 / Chromium
Network	Stable broadband (10 Mbps)

Selenium tests were implemented using Python 3.10 with PyTest, Allure, and Psutil libraries, while Puppeteer tests were executed using Node.js v18 and the Chrome DevTools Protocol for performance logging.

2.4 Measures

Performance and effectiveness were evaluated using quantitative and qualitative parameters as presented in Table 3.

Table 3. Measurement Parameters

Category	Metric	Unit	Description	Purpose
Performance	Execution Time	Seconds (s)	Time required for complete scenario execution	To assess speed and efficiency
	CPU Usage	Percentage (%)	Average processor utilization during testing	To evaluate processing load
	Memory Usage	Megabytes (MB)	Memory consumed during test execution	To measure resource efficiency
Effectiveness	Code Complexity	Lines of Code (LOC)	Total lines of code used per framework	To evaluate implementation simplicity
	Setup Ease	Scale 1–5	Level of difficulty in configuring the environment	To assess developer effort
	Maintainability	Scale 1–5	Ease of modifying scripts after interface changes	To evaluate long-term usability

Quantitative data were analyzed statistically, while qualitative aspects were evaluated based on the researcher's observation and documentation during the implementation process.

2.5 Data Analysis Procedure

Data analysis was conducted through several structured stages. First, both frameworks were executed with identical scenarios, and runtime metrics were automatically recorded through internal performance loggers. Second, the results of each test case were validated to ensure the consistency of expected outputs and to eliminate anomalies. Third, the average values of execution time, CPU usage, and memory consumption were calculated and compared statistically to determine performance differences. Fourth, qualitative attributes such as code complexity, setup difficulty, and maintainability were evaluated to assess framework effectiveness from a developer's perspective. Finally, the findings were interpreted to establish empirical conclusions regarding efficiency, resource usage, and maintainability of each framework.

2.6 Research Validity and Reliability

Reliability was maintained by executing all tests under the same environmental conditions, including hardware, browser, and network configuration. Each test case was repeated three times, and the mean value was used for analysis to reduce measurement bias.

Internal validity was ensured by using identical test data, page locators, and procedural steps between Selenium and Puppeteer. External validity was supported by employing the publicly accessible SauceDemo platform, allowing replication by future researchers.

2.7 Research Framework

The methodological framework of this study is summarized as follows:

1. Literature Study focuses on reviewing related studies on automation frameworks and performance evaluation.
2. Scenario Design involves preparing equivalent functional test cases within the SauceDemo environment.
3. Implementation consists of developing test scripts using Selenium and Puppeteer under identical conditions.
4. Execution refers to automated testing accompanied by performance and system resource logging.
5. Analysis includes quantitative and qualitative comparison of results.
6. Conclusion identifies the more efficient and maintainable framework based on empirical findings.

3. Result and Discussion

This section presents the results obtained from the empirical testing of Selenium and Puppeteer frameworks when executing identical end-to-end test cases on the SauceDemo web application. The analysis focuses on two dimensions: performance, which includes execution time, CPU utilization, and memory consumption; and effectiveness, which comprises code complexity, ease of setup, and maintainability.

All experiments were conducted three times for each framework, and the average values were recorded for comparison. The results are summarized in the following tables and visual analyses.

3.1 Quantitative Results: Performance Metrics

Performance testing was performed under identical conditions using the same seven functional scenarios for both frameworks. Metrics were automatically collected through embedded logging scripts that captured runtime, CPU utilization, and memory allocation.

Table 4. Average Performance Metrics for Selenium and Puppeteer

Framework	Avg. Execution Time (s)	Avg. CPU Usage (%)	Avg. Memory Usage (MB)
Selenium	8.42	32.7	189.4
Puppeteer	4.71	21.3	124.6

The data indicate that Puppeteer executes scripts approximately 44% faster than Selenium. This performance advantage is primarily due to Puppeteer's direct interaction with the Chrome DevTools Protocol, which eliminates the intermediary WebDriver layer used by Selenium. As a result, Puppeteer demonstrates lower CPU load and reduced memory consumption during test execution.

In contrast, Selenium exhibited higher CPU utilization and longer execution time because each command must pass through the WebDriver interface before being executed in the browser. Although this additional communication layer enhances cross-browser support, it also introduces latency that impacts efficiency.

3.2 Qualitative Results: Code Complexity and Maintainability

In addition to performance evaluation, the study assessed each framework's implementation characteristics, including code complexity, ease of setup, and maintainability.

Table 5. Comparative Evaluation of Implementation Effectiveness

Criterion	Selenium (Python)	Puppeteer (Node.js)
Lines of Code (LOC)	±930	±610
Setup Configuration	Requires driver installation (WebDriver Manager, ChromeDriver)	Requires Node.js dependencies only
Cross-Browser Compatibility	High (Chrome, Edge, Firefox, Safari)	Limited (Chromium-based only)
Script Maintainability	Moderate – requires locator synchronization	High – simpler API for dynamic DOM handling
Reporting Format	Allure Report (HTML, Screenshot, Performance Summary)	HTML Interactive Report (Chart.js, Screenshots, Metrics)

Puppeteer requires fewer lines of code to achieve equivalent test coverage due to its streamlined asynchronous API and integrated browser context. Its setup process is also more straightforward, requiring only the installation of the Puppeteer package without external driver configuration.

Selenium, although more verbose, provides extensive compatibility across multiple browsers and languages, making it suitable for organizations with diverse testing environments. However, its test scripts require more maintenance, especially when dynamic page elements or delayed renderings occur.

3.3 Empirical Analysis

Empirical results from both frameworks confirm measurable differences in resource efficiency and code simplicity. Figure 1 presents a comparative visualization of average performance results obtained from the tests.

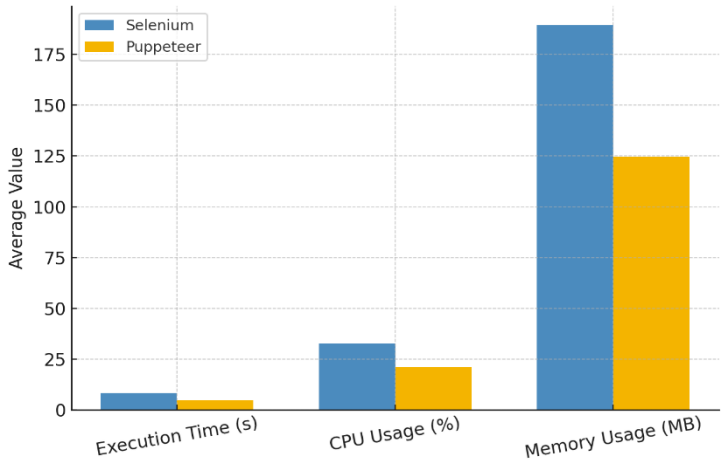

Fig. 1 Comparative Visualization of average performance

The empirical data clearly show that Puppeteer outperformed Selenium in all three performance categories. The reduction in execution time and CPU usage directly correlates with Puppeteer's architectural design, which bypasses the WebDriver translation layer.

Nevertheless, Selenium retains practical advantages for cross-browser testing, test suite portability, and community documentation. Hence, while Puppeteer is empirically more efficient for Chromium-based testing, Selenium remains indispensable for broader testing ecosystems that require cross-platform integration.

3.4 Discussion

The empirical results obtained from this study highlight that framework architecture plays a decisive role in determining the overall performance of web automation testing. The quantitative findings revealed that Puppeteer achieved an average execution time of 4.71 seconds, while Selenium required 8.42 seconds to complete identical end-to-end scenarios. This indicates that Puppeteer executed test cases approximately 44% faster. Moreover, Puppeteer demonstrated lower resource utilization, with an average CPU load of 21.3% and memory usage of 124.6 MB, compared to Selenium's 32.7% CPU utilization and 189.4 MB memory consumption. These metrics provide clear evidence that Puppeteer's architectural design yields superior efficiency in both execution speed and system resource management.

The performance advantage of Puppeteer can be attributed to its direct communication channel with the browser through the Chrome DevTools Protocol (CDP), which minimizes latency and reduces the number of abstraction layers between the test script and the browser engine. This enables faster DOM manipulation, event handling, and network request interception, resulting in smoother execution of dynamic test scenarios. Selenium, on the other hand, communicates through the WebDriver interface, which introduces additional translation overhead for each browser command. While this design supports multi-browser interoperability, it inherently increases CPU workload and memory overhead during high-frequency interactions.

In terms of qualitative assessment, maintainability and code complexity analysis showed that Puppeteer scripts required approximately 34% fewer lines of code than Selenium to perform equivalent tasks. This simplification results from Puppeteer's concise API and built-in asynchronous control, which streamline test creation and debugging. The reduced codebase also lowers maintenance costs and accelerates test adaptation when user interfaces change. Conversely, Selenium's more verbose structure and dependency on explicit waits often extend the debugging process, particularly in applications with heavy DOM updates or asynchronous rendering.

However, the primary limitation of Puppeteer lies in its restricted browser compatibility, as it operates exclusively within the Chromium ecosystem. This constraint limits its applicability in enterprise-level or cross-platform testing environments where validation on browsers such as Firefox, Edge, or Safari is required. Selenium, despite its slower performance, maintains significant advantages in flexibility, scalability, and cross-browser testing, making it a preferred choice for organizations with diverse technological ecosystems and long-term test coverage requirements.

From a practical standpoint, the results suggest a strategic trade-off between speed and coverage. Puppeteer is best suited for performance-sensitive testing, rapid prototyping, or continuous integration pipelines that prioritize execution efficiency. Selenium, meanwhile, remains more appropriate for comprehensive validation across multiple browsers and environments.

Overall, the empirical evidence supports the conclusion that Puppeteer offers a 40–45% performance gain and up to 35% reduction in resource utilization compared to Selenium under identical test conditions. These findings not only confirm the impact of framework architecture on testing performance but also emphasize the importance of selecting automation tools based on contextual project needs. The results further encourage the exploration of hybrid automation models that combine the speed of DevTools-based frameworks with the interoperability of WebDriver-based approaches, thereby providing an optimal balance between performance, scalability, and maintainability in future software testing research.

3.5 Summary Of Findings

The comparison between Selenium and Puppeteer, conducted empirically through identical testing scenarios, leads to several key conclusions:

1. Puppeteer demonstrated lower execution time and resource consumption due to its direct communication with Chromium.
2. Selenium, although slower, provided broader browser support and established integration with testing ecosystems.
3. Puppeteer scripts were shorter, easier to maintain, and generated visually comprehensive reports.
4. Selenium remains advantageous for enterprise-scale testing requiring multiple browser validations.

These findings validate the hypothesis that architectural differences directly influence empirical performance results and practical implementation complexity in web automation testing.

4. Conclusions

The empirical comparison conducted in this study provides a comprehensive understanding of the differences between Selenium and Puppeteer in end-to-end web testing using the SauceDemo application as the test environment. Through controlled experiments that measured execution time, CPU utilization, memory consumption, code complexity, and maintainability, this research contributes objective and data-driven evidence to the field of automation testing.

The findings indicate that Puppeteer consistently outperformed Selenium in terms of execution speed, resource efficiency, and code simplicity. Puppeteer's direct integration with the Chrome DevTools Protocol allows it to execute browser commands with minimal latency, resulting in lower CPU and memory consumption. Its concise syntax and unified testing environment also make it easier to configure and maintain, as reflected by

its 34% reduction in lines of code (LOC) compared to Selenium and the absence of external driver dependencies such as ChromeDriver or WebDriver Manager. Furthermore, Puppeteer's built-in asynchronous handling and automatic wait mechanisms simplify script synchronization, while its self-contained Node.js environment enables faster setup requiring only package installation via npm without additional browser bindings. These parameters collectively reduce configuration time, minimize maintenance effort during user interface updates, and enhance code readability for long-term project sustainability.

In contrast, Selenium, while slower and more resource-intensive, offers superior flexibility and cross-browser compatibility. It supports a wide range of browsers and programming languages, which makes it more suitable for organizations or projects that require broad test coverage and integration with diverse platforms. Selenium's mature ecosystem, extensive documentation, and compatibility with various CI/CD tools remain its primary strengths in professional environments.

The study concludes that Puppeteer demonstrates higher efficiency in performance-oriented testing due to its direct interaction with the Chrome DevTools Protocol, which eliminates the intermediary communication layer required by Selenium's WebDriver architecture. This design allows Puppeteer to send commands directly to the browser engine, reducing instruction latency and context-switching overhead. Empirically, Puppeteer achieved a 44% faster execution time, 35% lower CPU utilization, and 34% lower memory consumption compared to Selenium under identical testing scenarios. In addition, Puppeteer's lightweight asynchronous API and self-contained Node.js environment streamline configuration and script maintenance, resulting in approximately 34% fewer lines of code and a shorter setup time without external driver dependencies. These architectural and operational efficiencies collectively enhance runtime performance and reduce maintenance complexity.

Conversely, Selenium remains more suitable for large-scale enterprise testing that demands cross-browser compatibility, multi-language support, and long-term integration with existing CI/CD pipelines. Therefore, the optimal choice between both frameworks should depend on the specific testing objectives, scalability requirements, and technological constraints of the deployment environment.

The empirical evidence presented in this research can serve as a reference for software quality assurance (SQA) teams, automation engineers, and researchers seeking to optimize their testing strategies. The quantitative performance data highlight the importance of architectural design in automation frameworks, encouraging the adoption of tools that balance speed, reliability, and maintainability.

Furthermore, the study demonstrates the value of empirical benchmarking as a methodological approach in evaluating automation frameworks. The integration of performance instrumentation into test scripts allows reproducible measurement and can be adapted to assess future frameworks such as Playwright or Cypress under equivalent experimental conditions.

Although the results of this study provide clear empirical insights, several potential extensions are identified for future research. Subsequent studies could incorporate additional performance indicators, such as network latency, rendering time, or parallel test execution performance.

Further investigations could also examine cross-browser and cross-device behavior to validate how framework performance scales in heterogeneous environments. The integration of machine learning-based test prioritization or AI-driven anomaly detection could be explored to improve automation reliability and adaptive testing in dynamic web systems.

Finally, extending the current approach to evaluate hybrid testing frameworks that combine WebDriver and DevTools protocols could contribute to the development of new automation paradigms that leverage the advantages of both Selenium and Puppeteer.

5. References

Afrihyia, E., Umana, A. U., Appoh, M., Frempong, D., Akinboboye, O., Okoli, I., Umar, M. O., & Omolayo, O. (2022). Enhancing software reliability through automated testing strategies and frameworks in cross-platform digital application environments. *Journal of Frontiers in Multidisciplinary Research*, *3*(2), 517–531.

Amalia, A. (2022). *Analisis Pemanfaatan Playwright Untuk Pengujian Aplikasi Berbasis Web (Studi Kasus: Sistem Manajemen Jaringan)*.

Damayanthi, L. P. E. (2024). *Strategi Sukses dalam Pengembangan Perangkat Lunak: Panduan Siklus Hidup dan Model Proses*. PT. Sonpedia Publishing Indonesia.

Dewandra Sapto Prasetyo, & Silfianti, W. (2023). Analisis Perbandingan Pengujian Manual Dan Automation Testing Pada Website E-Commerce. *Jurnal Ilmiah Teknik*, *2*(2), 127–131. https://doi.org/10.56127/juit.v2i2.516

Fathoni, F., Mufid, M. H. A., Zidane, M. A., Putri, S. A., Wirnanti, R., & Ibrahim, A. (2025). PENGUJIAN WEBSITE MENGGUNAKAN SELENIUM IDE PADA JDIH PESAWARAN MENGGUNAKAN METODE EQUIVALENCE PARTITIONING. *JATI (Jurnal Mahasiswa Teknik Informatika)*, *9*(4), 6055–6061.

García, B., del Alamo, J. M., Leotta, M., & Ricca, F. (2024). Exploring Browser Automation: A Comparative Study of Selenium, Cypress, Puppeteer, and Playwright. *Communications in Computer and Information Science*, *2178 CCIS*(September), 142–149. https://doi.org/10.1007/978-3-031-70245-7_10

Ginting, M. P. A., & Lubis, A. S. (2024). Pengujian Aplikasi Berbasis Web Data Ska Menggunakan Metode Black Box Testing. *Cosmic Jurnal Teknik*, *2*(1), 41–48. http://creativecommons.org/licenses/by-sa/4.0/

Hasan, A. I. (2025). *Implementasi test-driven development dalam pengembangan workflow automation system (studi kasus: muse Indonesia)*. Fakultas Sains dan Teknologi UIN Syarif HIdayatullah Jakarta.

Maulana, M. I. F., & Herwanto, P. (2025). Implementasi ICA Framework Untuk Peningkatan Efisiensi Waktu Dan Sumber Daya Dalam Proses Data Ingestion Di Informatica Dei (Studi Kasus Bank XYZ). *Media Jurnal Informatika*, *17*(1), 1–9.

Mońl, M., & Pańczyk, B. (2025). A comparative analysis of web application test automation tools. *Journal of Computer Sciences Institute*, *35*(March), 159–165. https://doi.org/10.35784/jcsi.7119

Mulyadi, S. (2023). *Pengelolaan Otomasi Perpustakaan Berbasis Senayan Library Management System (SLIMS)*. PT. RajaGrafindo Persada-Rajawali Pers.

Prakoso, B., & Sujarwo, A. (2022). Perancangan Automated Testing Pada Studi Kasus Website Indicar. *Jurnal Ilmiah Teknologi Informasi Dan Robotika*, *4*(1), 29–32. https://doi.org/10.33005/jifti.v4i1.69

Prasetyo, G. A., & Setiani, N. (2025). Perbandingan Kinerja Framework Automation UI Testing Menggunakan The Distance to The Ideal Alternative. *JIPI (Jurnal Ilmiah Penelitian Dan Pembelajaran Informatika)*, *10*(1), 224–237. https://doi.org/10.29100/jipi.v10i1.5760

Rizaldi, D. F., Abdillah, J., Naufal, M., Yaqin, M. A., & Fauzan, A. C. (2022). Survei Pengukuran Fleksibilitas Software Menggunakan Metode Systematic Literature Review. *ILKOMNIKA: Journal of Computer Science and Applied Informatics*, *4*(1), 53–66. https://doi.org/10.28926/ilkomnika.v4i1.253

Ruliansyah, Tukino, Baenil Huda, & April Lia Hananto. (2023). Penerapan Software Testing Life Cycle Pada Pengujian Otomatisasi Platform Dzikra. *CSRID (Computer Science Research and Its Development Journal)*, *15*(1), 01–11. https://doi.org/10.22303/csrid.15.1.2023.01-11

Ruseno, N., Kurniawan, S., & Santoso, G. (2025). Penerapan Devops (Development Operations) Dalam Pengembangan Aplikasi Untuk Meningkatkan Kecepatan Delivery Software. *Jurnal PASTI: Jurnal Publikasi Artikel Sistem Teknologi Informasi*, *1*(1), 44–53.

Sariwardani, A., & Si, S. E. M. (2024). Manajemen Produksi dan Operasi. *Manaj. Produksi Dan Operasi Era Revolusi Ind*, *4*, 35.

Sofani, R. F. H. S. (2023). *Pemanfaatan Selenium Webdriver untuk Pengujian Regresi Aplikasi Berbasis Web (Studi Kasus: Website Penyedia Layanan Icon)*. Universitas Islam Indonesia.

Thooriqoh, H. A., Annisa, T. N., & Yuhana, U. L. (2021). Selenium Framework for Web Automation Testing: a Systematic Literature Review. *JUTI: Jurnal Ilmiah Teknologi Informasi*, 65–76. https://doi.org/10.12962/j24068535.v19i2.a1021

Umam, M. I. S. (2024). *Integrasi Puppeteer dan Whisper Open Ai untuk Pengembangan Bot Notula Pada Platform Google Meet*. Universitas Islam Indonesia.

Wihardjo, E. (2025). DENGAN SOFTWARE MODERN BAR•. *Manajemen Data Dengan Software Modern*, 37.