

Empirical Performance of E2E Frameworks in React-Vue SPAs Using DIA

Abdillah Rezeki¹, Setyo Wahyu Saputro², Triando Hamonangan Saragih³,
Radityo Adi Nugroho⁴, Friska Abadi⁵

^{1,2,3,4,5}Department of Computer Science, Lambung Mangkurat University, Indonesia

Article Info

Article history:

Received Feb 01, 2026

Revised Feb 09, 2026

Accepted Feb 21, 2026

Keywords:

Automation Testing;
End-to-End Testing;
Performance Evaluation;
Playwright;
Selenium WebDriver;
Cypress;
Single Page Application.

ABSTRACT

Modern web applications increasingly adopt Single-Page Application (SPA) architectures to enhance the user experience through client-side rendering and dynamic content loading. However, these characteristics introduce significant challenges for automated end-to-end (E2E) testing, including asynchronous DOM manipulation, complex state management, and timing synchronization issues. This study presents a comprehensive empirical comparison of three prominent E2E testing frameworks—Selenium WebDriver, Cypress, and Playwright—across React and Vue-based SPAs. Using a quantitative experimental approach, 25 standardized test cases were executed 15 times each across Chrome, Firefox, and Edge, for a total of 270 testing sessions. Performance evaluation focused on four key metrics: execution time, success rate, CPU usage, and memory consumption. Results demonstrate that Playwright achieved the fastest execution time (56.25 seconds on React-Chrome), while Selenium exhibited superior resource efficiency with the lowest memory consumption (196.59 MB on Vue-Chrome). The Distance to Ideal Alternative (DIA) multi-criteria decision analysis method identified Playwright-Chrome as optimal for React applications (DIA score: 0.886715) and Selenium-Chrome for Vue applications (DIA score: 0.908237), indicating that framework selection should be context-dependent based on application characteristics and deployment requirements. This research supports the conclusion that no universal "best" testing framework exists, underscoring the importance of evidence-based, application-specific tool selection in software quality assurance.

This is an open access article under the [CC BY-SA](#) license.



Corresponding Author:

Setyo Wahyu Saputro,
Faculty of Mathematics and Natural Science,
Department of Computer Science,
Lambung Mangkurat University, Kalimantan, Indonesia
Email: setyo.saputro@ulm.ac.id

1. INTRODUCTION

Modern web technologies drive the Development of increasingly interactive and responsive applications. One prominent approach is the Single Page Application (SPA) architecture, which enables primary data processing on the client side, thereby enhancing the user experience [1]. Unlike traditional multi-page applications that require full-page reloads with each user interaction, SPAs dynamically load only the necessary content, resulting in faster navigation and a more seamless user experience. SPA architecture is widely adopted in enterprise applications such as Gmail, Facebook, and Netflix because it can reduce page load times by 50-70% and significantly improve web application interactivity [2], [3].

Although SPAs offer significant benefits in application performance, their characteristics—such as client-side rendering, dynamic AJAX content loading, and complex state management—introduce unique challenges for automated testing [4], [5]. Key issues include handling

asynchronous DOM updates, developing adaptive wait strategies for dynamic content, and managing event synchronization, which can cause flaky tests—tests that fail intermittently without code changes [6]. This complexity is especially heightened in large-scale projects, where testing efficiency and dependability are vital for ensuring software quality before deployment [6].

In response to these challenges, end-to-end (E2E) testing frameworks such as Selenium, Cypress, and Playwright have undergone significant updates. Selenium WebDriver, as the oldest and most established framework, is known for its extensive support across multiple languages and comprehensive cross-browser testing; however, it faces limitations in execution speed, synchronization complexity, and managing dynamic web elements [4], [5], [6]. Cypress provides a modern approach by executing directly in the browser, offering faster performance and easier debugging with time-travel features. However, it was initially limited to Chromium-based browsers until recent versions added support for Firefox and WebKit [7], [8]. Playwright, developed by Microsoft and the latest framework, features native parallel execution, browser context isolation for independent testing across environments, and broad multi-browser support (Chrome, Firefox, and WebKit) on a single platform [9]. Ref [10], demonstrates that Playwright is more efficient than Selenium, using only 6.25% CPU and 89.73 MB of memory, compared to Selenium's 12.60% CPU and 167.32 MB, underscoring the importance of performance metrics in framework selection.

This study examines two main front-end JavaScript frameworks in the SPA ecosystem: React, which utilizes a Virtual DOM for efficient interface updates through diffing algorithms [1], [11], and Vue.js, which focuses on progressive integration and employs a reactivity system based on dependency tracking for more intuitive data binding [12], [13]. These frameworks were chosen for their distinct architectural styles: React, with its component-based structure and unidirectional data flow, and Vue, with its template-driven approach and two-way data binding. The testing also covers cross-browser compatibility across Chrome (Chromium), Firefox (Gecko), and Microsoft Edge (Chromium), since variations in rendering engines could lead to different behaviors and undetected bugs if only a single browser is tested [14], [15].

Although several comparative studies on E2E testing frameworks are available, they have notable limitations. Ref [12], [13], [7], [10], reveal significant limitations in existing end-to-end testing research. Balsam et al. (2024) identified gaps in web application testing but did not specifically compare modern tools for Single-Page Applications. Adiatma (2023) examined Katalon Studio, Cypress, and WebDriverIO using the Distance to Ideal Alternative (DIA) method; however, it excluded Playwright and did not test across different SPA platforms. Vieira et al. (2025) analyzed Cypress in CI/CD pipelines but did not compare it with other frameworks. Similarly, Wijaya et al. (2025) compared Selenium, Cypress, and Playwright in terms of speed and resource utilization, but limited their evaluation to single-scenario login tests and did not assess different framework architectures (e.g., React vs. Vue) or employ Multi-Criteria Decision Analysis to provide objective framework selection recommendations. Therefore, there is a gap in the literature for a comprehensive evaluation of Selenium, Cypress, and Playwright for React- and Vue-based SPAs that accounts for cross-browser compatibility and employs Multi-Criteria Decision Analysis to provide objective insights.

This research aims to: (1) measure and compare execution time, success rate, CPU consumption, and memory usage of the three frameworks in testing SPA applications based on React and Vue; (2) analyze technical capabilities in handling dynamic content loading, asynchronous operations, and cross-browser compatibility (Chrome, Firefox, Edge); and (3) determine the optimal framework using the Distance to the Ideal Alternative (DIA) method. The methodology employs a quantitative experimental approach, with 25 standardized test cases covering user interaction scenarios, form submission, navigation flows, and data manipulation. The DIA method was selected for its effectiveness in integrating multiple competing criteria—execution performance, resource efficiency, reliability, and maintainability—into a transparent, objective evaluation model [14], [11], [16].

2. RESEARCH METHOD

This research employs an experimental quantitative approach to compare the performance of end-to-end (E2E) testing frameworks, namely Selenium WebDriver, Cypress, and Playwright, on single-page application (SPA) web applications based on React.js and Vue.js. The analysis is conducted comparatively on two web applications with identical source code and features, thereby eliminating implementation bias [17]. The automated testing framework supports the software verification process, speeds up bug detection, and improves testing coverage and overall software quality [4]. Figure 1 presents the research methodology flow, comprising five main stages: preparation of the environment and research objects; test case design; implementation of test scripts and experimental procedures; data collection and evaluation metrics; and data analysis using the Distance to Ideal Alternative (DIA) method.

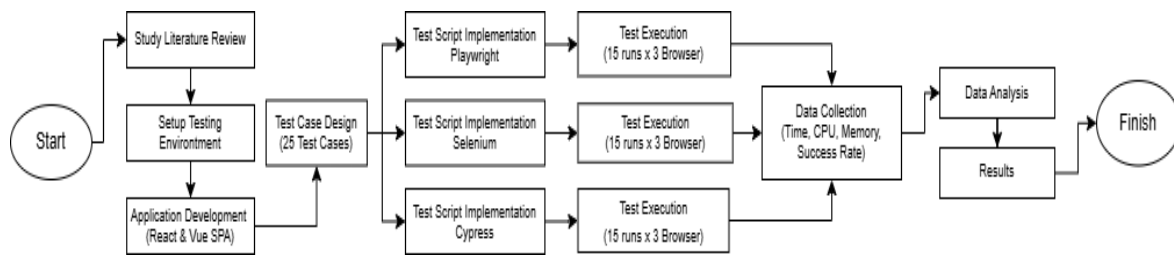


Figure 1. Research Methodology Flowchart

2.1. Objects and Research Environment

The research objects consist of two admin dashboard applications built as Single Page Applications (SPAs) using the CoreUI Free template. The first application is built with React.js version 19.0.0, React Router v6.xx, and Vite v5.4.0 as the build tool, while the second application uses Vue.js version 3.4.0, Vue Router v4.x, and Vite v5.4.0. Testing was conducted on three major browsers to ensure cross-browser compatibility: Google Chrome v141.0.7390.65 (Chromium-based), Mozilla Firefox v143.0.4 (Gecko-based), and Microsoft Edge v141.0.3537.57 (Chromium-based). These versions correspond to the latest stable releases available at the time of data collection in October 2025, and browser auto-updates were disabled during the experiments to preserve consistency across all runs. The experimental environment runs on Windows 11 Pro 64-bit (Version 22H2, Build 22621) with hardware specifications including a 12th Gen Intel Core i3-1215U processor (1.20 GHz, 6 cores/8 threads), 16 GB DDR4 RAM, a 512 GB SSD, and a stable 100 Mbps Wi-Fi network connection. Testing tools used include Selenium WebDriver v4.33.0, Cypress v13.6.6, and Playwright v1.55.1, supported by Node.js v18.20.0 and supporting libraries such as pidusage v3.0.2, pidtree v0.6.0, and ps-list v4.0.0 for process monitoring.

2.2. Test Case Design

This study employs 25 standardized test cases covering functional testing across various application modules, as listed in Table 1. The number of test cases is determined by the functional coverage principle, as specified in ISO/IEC/IEEE 29119-4 [18]. This software testing standard defines test design techniques and recommends that each primary function be covered by at least one test case. A similar approach has been applied in empirical comparative studies. Antończak and Bylina (2024) used 15 test cases to evaluate Selenium and Playwright [19], while Melyawati et al. (2024) used 12 test cases to compare similar frameworks [20]. With 25 test cases covering four functional modules (Login, Category Management, News Management, and Security), this research provides more comprehensive coverage, ensuring the validity of performance comparisons between the React and Vue frameworks' JS-based single-page Applications.

Table 1. List of Test Cases

no	Test Case	Features	No	Test Case	Features
1	TC-01	Login form display	14	TC-14	Edit category with duplicate name
2	TC-02	Valid login credentials	15	TC-15	Category error handling
3	TC-03	Invalid login credentials	16	TC-16	Category boundary testing
4	TC-04	Empty input validation	17	TC-17	Display news list
5	TC-05	API timeout handling			

6	TC-06	Extreme input boundary test	18	TC-18	Add a new news article
7	TC-07	Session expiry and redirect	19	TC-19	Edit news article
8	TC-08	Display category list	20	TC-20	Delete news article
9	TC-09	Add a new category	21	TC-21	News input validation
10	TC-10	Edit category	22	TC-22	News API timeout handling
11	TC-11	Delete category	23	TC-23	News boundary input test
12	TC-12	Category input validation	24	TC-24	Route protection (unauthorized access)
13	TC-13	Duplicate category detection	25	TC-25	Access control validation

2.3. Experimental Procedure

The procedure is conducted as follows [13], [17]:

1. Selenium tests are implemented in Python, while Cypress and Playwright tests are implemented in JavaScript. This choice reflects common adoption patterns in practice: Selenium WebDriver is widely used through multi-language bindings, including Python, in both industrial and research settings, whereas Cypress and Playwright are JavaScript/TypeScript-based end-to-end testing frameworks that are typically integrated directly into modern JavaScript SPA stacks. Using the dominant language binding for each framework preserves ecological validity, ensuring that the evaluation reflects real-world usage rather than enforcing an artificial single-language constraint. The impact of language heterogeneity on internal validity is discussed further in Section 3.8.
2. Repeated Execution: Each combination of framework and browser is run 15 times (15 runs) to validate statistics and reduce test result fluctuations [21].
3. Resource Monitoring: Resource monitoring is performed automatically using pidusage, pidtree, and ps-list, with data collected every 5 seconds to record execution time, CPU, memory usage, and the success rate of child and parent processes during each run. CPU utilization is reported as the aggregate percentage across all CPU cores; on a multi-core processor, this can exceed 100%, where 100% corresponds to one fully utilized core (e.g., 111% $\approx$ 1.11 cores). Non-essential background applications and services are disabled to reduce noise, and the application state is reset before each session, while browser cache, cookies, and local storage are not systematically cleared between repetitions, which may introduce minor caching effects and is acknowledged as a limitation.
4. Execution Mode: All experiments are executed in headless mode for all frameworks and browsers to mimic typical CI/CD execution environments and reduce GUI rendering overhead. Selenium is configured with the --headless option for Chrome, Firefox, and Edge; Cypress tests are run with cypress run --headless; and Playwright launches browsers with the { headless: true } configuration.

2.4. Variables and Evaluation Metrics

The test performance indicators consist of [4], [12], [20]:

- a. Execution Time (T): Total duration (seconds) for each framework-browser combo, indicating how quickly the framework completes all 25 test cases.
- b. Success Rate (S): Percentage of test cases that pass out of the total, reflecting the framework's reliability and stability.
- c. CPU Usage (C): The average CPU load (%) during testing, showing the computational demand on the system.
- d. Memory Usage (MB): The average amount of memory (in MB) used, demonstrating the framework's memory efficiency during operation.

2.5. Data Analysis

The test data results were analyzed descriptively and statistically. The framework was ranked using the Distance to the Ideal Alternative (DIA) method, a multi-criteria approach based on the Manhattan distance [13]. The DIA method was chosen for its ability to evaluate alternatives based on their Distance from the ideal solution while simultaneously considering multiple criteria.

DIA Calculation Steps:

Step 1: Normalisation

Data normalisation is performed using min-max scaling to ensure fair comparison across frameworks and browsers. For benefit criteria (the larger, the better, such as Success Rate), normalization uses Equation (1):

$$X_{norm} = \frac{X_i - X_{min}}{X_{max} - X_{min}} \quad (1)$$

For the cost criteria (the smaller, the better, such as Execution Time, CPU Usage, Memory Usage), normalization uses Equation (2):

$$X_{norm} = \frac{X_{max} - X_i}{X_{max} - X_{min}} \quad (2)$$

where X_i is the metric value for the alternative to- X_{min} is the minimum value of that metric across all alternatives, and X_{max} is the maximum value.

Step 2: Criterion weighting

After normalization, each criterion is assigned a weight that reflects its relative importance in the context of the E2E framework evaluation. The weights used in this study are: Execution Time ($w_1 = 0,30$), Success Rate ($w_2 = 0,25$), CPU Usage ($w_3 = 0,20$), and Memory Usage ($w_4 = 0,25$), with $\sum w_j = 1,00$. Execution Time is given the highest weight because time behaviour is the most frequently studied sub-characteristic of performance efficiency under ISO/IEC 25010, accounting for 48% of studies in a recent systematic literature review [22]. ISO/IEC 25010 defines performance efficiency as a quality characteristic comprising time behaviour, resource utilization, and capacity. Resource utilization (including CPU and memory usage) is the second most frequently addressed sub-characteristic (36%), confirming the relevance of CPU and memory metrics in performance evaluation [22]. Furthermore, higher levels of test automation maturity have been empirically shown to be positively associated with higher product quality and shorter release cycles, which motivates assigning a high weight to Success Rate as a proxy for reliability [23].

The weighted normalized decision matrix $V = [v_{ij}]$ is obtained from the normalized matrix $R = [r_{ij}]$ using:

$$v_{ij} = w_j \cdot r_{ij} \quad (3)$$

where v_{ij} is the weighted normalized value of the alternative i on criterion j , w_j is the weight of the criterion j , and r_{ij} is the normalized value from Step 1.

Step 3: Determination of Positive and Negative Ideal Solutions

After normalization, the ideal solution is positive (A^+) and negative ideal solution (A^-) determined for each criterion using the equation:

$$A^+ = \{v_1^+, v_2^+, \dots, v_n^+\} = \{\max(v_{ij}) \mid j = 1, 2, \dots, n\} \quad (4)$$

$$A^- = \{v_1^-, v_2^-, \dots, v_n^-\} = \{\min(v_{ij}) \mid j = 1, 2, \dots, n\} \quad (5)$$

Where A^+ represents the best positive solution with the highest value for each criterion, while A^- represents the worst solution with the lowest value. The DIA is based on identifying the positive and negative ideal solutions, which are then used to compute the distance between each alternative and these ideal solutions. This method selects the best alternative that not only has the closest distance to the positive ideal solution (best solution) but also the farthest distance from the negative ideal solution (worst solution) [24]. The DIA method creates a straightforward Decision Support System that is easy to understand, with a logically organized concept, and has been proven to deliver optimal accuracy results, where, in certain tests, DIA can show the best alternative with zero distance from the ideal solution (zero distance) [13].

Step 4: Manhattan Distance Calculation

The Manhattan distance from each alternative to the positive ideal solution (D_i^+) and the negative ideal solution (D_i^-) is computed using the equations:

$$D_i^+ = \sum_{j=1}^n |v_{ij} - v_j^+| \quad (6)$$

$$D_i^- = \sum_{j=1}^n |v_{ij} - v_j^-| \quad (7)$$

Manhattan distance, also known as city block distance, calculates the sum of the absolute differences between the coordinates of two points [17]. In the context of DIA, the Manhattan distance measures the distance of alternatives from the positive ideal solution (A^+) and negative ideal solution (A^-) [24]. The use of Manhattan distance in the DIA method allows for the identification of alternatives with the most optimal performance by simultaneously comparing the closest distance to the positive ideal solution and the farthest distance from the negative ideal solution [17]. This approach produces a simpler calculation but remains effective in ranking alternatives based on the Positive Ideal Alternative (PIA) value [24].

Step 5: DIA Score Calculation

The DIA score for each alternative (framework-browser combination) is calculated using the relative closeness coefficient:

$$DIA_i = \frac{D_i^-}{D_i^+ + D_i^-} \quad (8)$$

The DIA score for each alternative (framework-browser combination) is calculated using a relative closeness coefficient ranging from 0 to 1, where values closer to 1 indicate alternatives that are closer to the positive ideal and farther from the negative ideal [25], [26]. The framework with the highest DIA score is considered the most optimal combination [25].

Step 6: Consistency Analysis

The consistency of the evaluation results is verified using the standard deviation (σ) and the coefficient of variation (CV), where CV is defined as the ratio of the standard deviation to the mean to measure relative variability [25].

$$\sigma = \sqrt{\frac{\sum_{i=1}^N (x_i - \bar{x})^2}{N}} \quad (9)$$

$$CV = \frac{\sigma}{\bar{x}} \times 100\% \quad (10)$$

where x_i is the metric value for observation number- i , $\bar{x}$ is the average metric value, and N is the number of observations.

The analysis results are presented in tables and graphs to facilitate the interpretation and comparison of performance across framework-browser combinations.

3. RESULTS AND DISCUSSION

3.1. Execution Summary and Data Collection

The experiment was conducted by running 25 test cases across three E2E testing frameworks (Selenium WebDriver, Cypress, and Playwright) on two single-page applications built with React and Vue. Each framework-browser combination was run 15 times to ensure statistical validity and reduce testing bias [19]. The experiment yielded 270 testing sessions across 18 combinations of three frameworks, three browsers (Chrome, Firefox, Edge), and two applications.

Resource consumption was monitored in the experiment using pidusage v3.0.2, pidtree v0.6.0, and ps-list v4.0.0. Every five seconds, these packages recorded execution time, success rate (%), CPU usage (%), and memory consumption (MB) for each run. Across 270 test sessions, 268 were completed successfully, yielding a success rate of 99.26%. Notably, two failures occurred: one involving Selenium-Chrome and the React application, and another with Playwright-Firefox

and the React application. All collected data were then processed and normalized for comparison using the Distance to Ideal Alternative (DIA) method.

3.2. Performance Test Result React Application

Execution Time: The execution time was measured to evaluate the framework's speed when running 25 test cases on a React application. Previous research indicates that execution time is a critical factor in selecting an automated testing framework. [27]. Figure 2 presents a comprehensive comparison of testing execution times across all combinations tested.

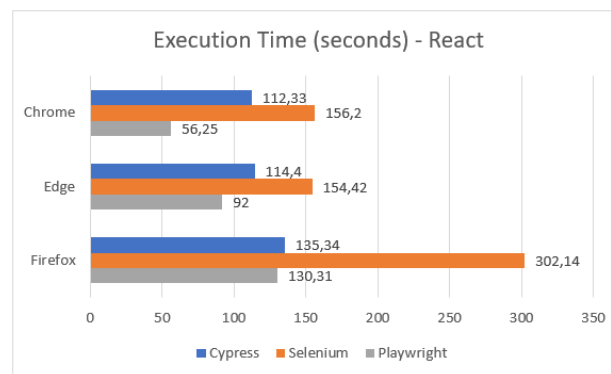


Figure 2. React application execution time comparison across three browsers and testing frameworks.

The test results show that Playwright consistently has the fastest execution time across all browsers, with Chrome averaging 56.25 ± 14.37 seconds, followed by Edge (92.00 ± 1.97 seconds) and Firefox (130.31 ± 15.61 seconds). Cypress demonstrates moderate performance with execution times ranging from 112.33 seconds (Chrome) to 135.34 seconds (Firefox). Selenium records the longest execution times, particularly on Firefox, with 302.14 ± 1.89 seconds—approximately 5.4 times slower than Playwright-Chrome. These findings are consistent with the research by Antończak and Bylina (2024), which reported Playwright's speed advantage of up to 46% on Chrome, 28% on Edge, and 19% on Firefox compared to Selenium WebDriver [19]. The performance anomaly observed in Selenium on Firefox warrants further investigation, as discussed in the Discussion section.

Success Rate: The test results show that Playwright has the fastest execution time, averaging 56.25 seconds on Chrome, followed by Cypress (112.33 seconds) and Selenium (156.20 seconds). This aligns with Antonczak & Bylina (2024), who found that Playwright demonstrates a significant speed advantage, with performance improvements of up to 46% on Google Chrome, 28% on Microsoft Edge, and 19% on Mozilla Firefox compared to Selenium WebDriver [19]. Regarding consistency, Playwright records a standard deviation of 14.37 seconds on Chrome. Selenium shows relatively little variation across Chrome and Edge ($SD < 6$ seconds), while Playwright-Firefox shows the greatest variability ($SD = 15.61$ seconds).

Table 2. Success Rate pada Aplikasi React

Browser	Framework	Success Rate (%)	Passed/Total
Chrome	Cypress	100.0	15/15
Chrome	Selenium	93.3	14/15
Chrome	Playwright	100.0	15/15
Firefox	Cypress	100.0	15/15
Firefox	Selenium	100.0	15/15
Firefox	Playwright	93.3	14/15
Edge	Cypress	100.0	15/15
Edge	Selenium	100.0	15/15
Edge	Playwright	100.0	15/15

The test results show that most framework-browser combinations achieved a 100% success rate. However, two combinations failed: Selenium in Chrome (93.3%, 14/15 successful) and Playwright in Firefox (93.3%, 14/15 successful). Cypress demonstrated perfect consistency with a 100% success rate across all tested browsers. The study by Melyawati et al. (2024) also reported

that test execution stability is an important indicator for evaluating automation testing frameworks, with Playwright showing good consistency across 12 test cases [20].

Resource Consumption: Resource consumption is measured through CPU usage (%) and memory consumption (MB) during test case execution. Resource efficiency is an important consideration, particularly in environments with limited resources or when conducting parallel testing. Figure 3 compares resource consumption for testing React applications.

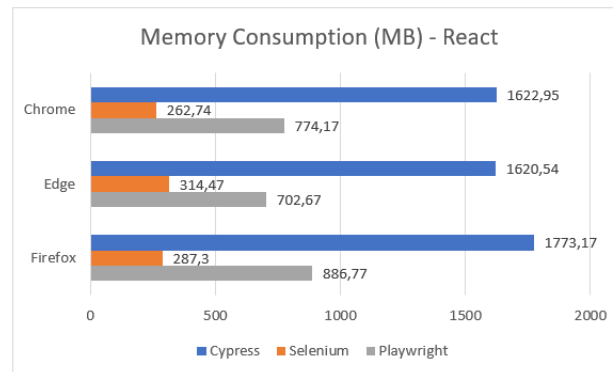


Figure 3. React application memory consumption comparison across three browsers and testing frameworks.

The analysis results show significant variations among frameworks in both metrics. For CPU usage, Playwright-Edge recorded the lowest consumption ($29.05 \pm 1.14\%$), while Cypress-Firefox showed the highest usage ($111.12 \pm 3.66\%$). Selenium demonstrated moderate CPU consumption, ranging from 37.99% to 41.06% across all browsers. In terms of memory consumption, Selenium demonstrated the highest efficiency, using only 262.74 ± 40.72 MB (Chrome) to 314.47 ± 13.52 MB (Firefox). Playwright showed moderate usage, ranging from 702.67 to 886.77 MB, whereas Cypress recorded the highest overhead, reaching 1620.54 to 1773.17 MB—approximately six times that of Selenium. These substantial differences make Selenium the optimal choice for environments with resource limitations or parallel testing scenarios.

3.3. Performance Test Result Vue Application

Execution Time: Measurement of execution time in Vue applications follows the same methodology as in React applications. The characteristics of JavaScript frameworks can affect automated testing performance. [28]. Figure 4 compares execution times for each framework-browser combination on the Vue application.

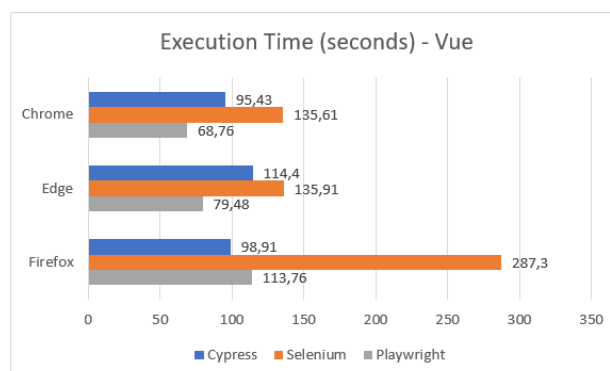


Figure 4. Vue application execution time comparison across three browsers and testing frameworks.

Consistent with the React results, Playwright demonstrates superior performance with Chrome achieving the fastest execution time (68.76 ± 13.09 seconds), followed by Edge (79.48 ± 4.27 seconds) and Firefox (113.76 ± 3.15 seconds). Cypress records moderate performance,

ranging from 95.43 to 137.34 seconds, whereas Selenium again shows the longest times, particularly on Firefox (287.30 ± 1.78 seconds). Compared with React, execution times on Vue are generally faster with Cypress (95.43 vs. 112.33 seconds on Chrome), but slightly slower with Playwright (68.76 vs. 56.25 seconds). These differences indicate the Influence of front-end framework architecture on automation testing performance. These findings align with Antonczak & Bylina (2024), who report that Playwright consistently outperforms in headless mode, with a total time of 2 minutes 4 seconds, compared with Selenium's 2 minutes 38 seconds.

Success Rate: Table 3 shows the success rate for each framework-browser combination on the Vue application.

Table 3. Success Rate pada Aplikasi Vue

Browser	Framework	Success Rate (%)	Passed/Total
Chrome	Cypress	100.0	15/15
Chrome	Selenium	100.0	15/15
Chrome	Playwright	100.0	15/15
Firefox	Cypress	100.0	15/15
Firefox	Selenium	100.0	15/15
Firefox	Playwright	100.0	15/15
Edge	Cypress	100.0	15/15
Edge	Selenium	100.0	15/15
Edge	Playwright	100.0	15/15

Unlike React applications, all framework-browser combinations in Vue applications achieve a 100% success rate (15 out of 15 runs successful). This result indicates that test cases in Vue applications are more stable than those in React. A study by Costa et al. (2023) also shows that the characteristics of the application framework can influence automation testing stability, with differences in rendering architecture and state management playing a significant role [29].

Resource Consumption: Resource consumption in Vue applications is measured using the same methodology as React. Figure 5 compares resource consumption for each framework-browser combination in the Vue application.

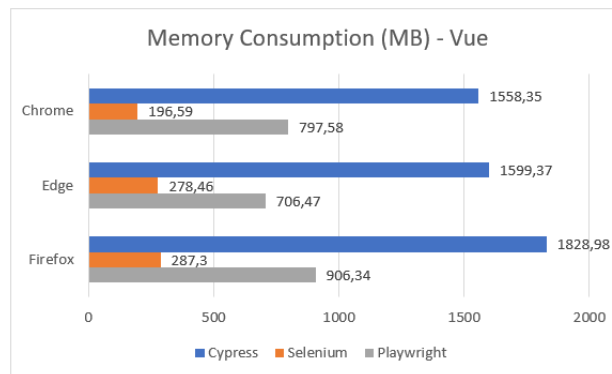


Figure 5. Vue application memory consumption comparison across three browsers and testing frameworks.

The resource consumption pattern in Vue is similar to React's, but with more pronounced differences. For CPU usage, Selenium-Chrome achieves the highest efficiency at $19.67 \pm 9.77\%$, followed by Selenium-Edge at $27.65 \pm 5.35\%$ and Playwright-Edge at $31.87 \pm 1.57\%$. Cypress again shows the highest consumption, reaching $111.28 \pm 4.57\%$ with Firefox. In terms of memory, Selenium maintains an advantage with the lowest consumption across all browsers, ranging from 196.59 ± 27.51 MB (Chrome) to 278.46 ± 16.32 MB (Firefox). Playwright shows moderate usage (706.47–906.34 MB), while Cypress records the highest overhead, reaching 1828.98 ± 26.61 MB on Firefox—about 9.3 times larger than Selenium-Chrome. These findings reinforce Selenium's position as the most resource-efficient option for Vue applications, especially in deployment scenarios with limited resources.

3.4. Cross-Browser Comparative Analysis

Cross-browser analysis was conducted to identify the browser that provides the best performance for each framework. Compatibility and performance across browsers are important factors when selecting an automated testing framework. Table 4 summarizes the average performance of each framework across the three tested browsers.

Table 4a. Framework Performance Comparison Across Browsers – React Application

Framework	Metric	Browser Optimal	Value
Cypress	Fastest Time	Chrome	112.33 seconds
	Lowest CPU	Edge	106.44%
	Lowest Memory	Edge	1620.54 MB
Selenium	Fastest Time	Edge	154.42 seconds
	Lowest CPU	Edge	37.99%
	Lowest Memory	Chrome	262.74 MB
Playwright	Fastest Time	Chrome	56.25 seconds
	Lowest CPU	Edge	29.05%
	Lowest Memory	Edge	702.67 MB

Table 4b. Performance Framework Comparison Across Browsers – Vue Application

Framework	Metric	Browser Optimal	Value
Cypress	Fastest Time	Chrome	95.43 seconds
	Lowest CPU	Edge	102.11%
	Lowest Memory	Chrome	1558.35 MB
Selenium	Fastest Time	Chrome	135.61 seconds
	Lowest CPU	Chrome	19.67%
	Lowest Memory	Chrome	196.59 MB
Playwright	Fastest Time	Chrome	68.76 seconds
	Lowest CPU	Edge	31.87%
	Lowest Memory	Edge	706.47 MB

Cross-browser analysis shows that Chrome consistently delivers the best performance for Playwright and Cypress, with lower execution times in React and Vue. In Selenium, Chrome also performs optimally with Vue applications. Firefox tends to record the longest execution times across all frameworks, especially with Selenium, which exceeds 280 seconds for both applications. These findings align with research by Antonczak & Bylina (2024), indicating significant performance variation among browsers, with Chromium-based browsers generally showing the best results, because Playwright is designed with strong native support for Chrome/Chromium, and Selenium's driver ecosystem is also highly optimized and more mature on Chrome, making test script execution more efficient and stable compared to Firefox [19].

3.5. Cross-Framework Comparative Analysis

Cross-framework analysis compared the performance of the three frameworks across the same applications in the same browser. Table 5 shows the framework rankings based on execution time for each application-browser combination.

Table 5. Ranking Framework Based on Execution Time

Application	Browser	Rank 1	Rank 2	Rank 3
React	Chrome	Playwright (56.25 s)	Cypress (112.33 s)	Selenium (156.20 s)
	Firefox	Playwright (130.31 s)	Cypress (135.34 s)	Selenium (302.14 s)
	Edge	Playwright (92.00 s)	Cypress (114.40 s)	Selenium (154.42 s)
Vue	Chrome	Playwright (68.76 s)	Cypress (95.43 s)	Selenium (135.61 s)
	Firefox	Playwright (113.76 s)	Cypress (137.34 s)	Selenium (287.30 s)
	Edge	Playwright (79.48 s)	Cypress (98.91 s)	Selenium (135.91 s)

Comparison of frameworks shows that Playwright consistently ranks as the fastest across all browser and application combinations. These findings are consistent with Melyawati et al. (2024), who reported that Playwright's total execution time is 4.9 minutes, faster than Selenium's 8.3 minutes when testing 12 test cases [20]. This aligns with Antończak and Bylina (2024), who demonstrated that Playwright outperforms Selenium WebDriver by up to 46% on Chrome, 28% on Edge, and 19% on Firefox [19]. This consistent advantage can be attributed to Playwright's more modern architecture, including native support for parallel execution and browser context isolation,

which allows multiple scenarios to run in a single browser process, as well as deep optimization for Chromium-based browsers, making test script execution more efficient and reducing the synchronization overhead common in Selenium. Selenium tends to be the slowest but offers significant advantages in resource efficiency (CPU and memory). The time difference between the fastest framework (Playwright) and the slowest (Selenium–Firefox) can exceed 4x.

3.6. Data Normalization and DIA Score Calculation

Min-Max Normalization Result: The four metrics (execution time, success rate, CPU usage, and memory consumption) are normalized using min-max normalization, as described in the Method section. Normalization is performed to standardize metrics with different units. [19]. Table 6 shows the normalization results for the React application.

Table 6. React Application Data Normalization

Browser	Framework	Norm Time	Norm Success	Norm CPU	Norm RAM
Chrome	Cypress	0.771924	1	0.015750	0.099452
Chrome	Selenium	0.593536	0	0.888751	1.000000
Chrome	Playwright	1.000000	1	0.856828	0.661399
Firefox	Cypress	0.678365	1	0.000000	0.000000
Firefox	Selenium	0.000000	1	0.853661	0.965756
Firefox	Playwright	0.698803	0	0.778160	0.586850
Edge	Cypress	0.763538	1	0.057062	0.101049
Edge	Selenium	0.600754	1	0.890993	0.986413
Edge	Playwright	0.854630	1	1.000000	0.708740

The normalization results indicate that higher values represent better performance. Playwright-Chrome shows the highest normalized time value (1.0), indicating the fastest execution time. Selenium-Chrome shows the highest normalized RAM value (1.0), reflecting the most efficient memory usage.

Higher normalized values represent better performance. Playwright-Chrome shows the highest normalized time value (1.0), indicating the fastest execution time. Selenium-Chrome shows the highest normalized RAM value (1.0), reflecting the most efficient memory usage.

DIA Score Calculation Ranking: The Distance to Ideal Alternative (DIA) method is used to determine the optimal framework based on four metrics, each normalized using the Manhattan distance. Figures 6 and 7 present the DIA scores and rankings for React and Vue applications.

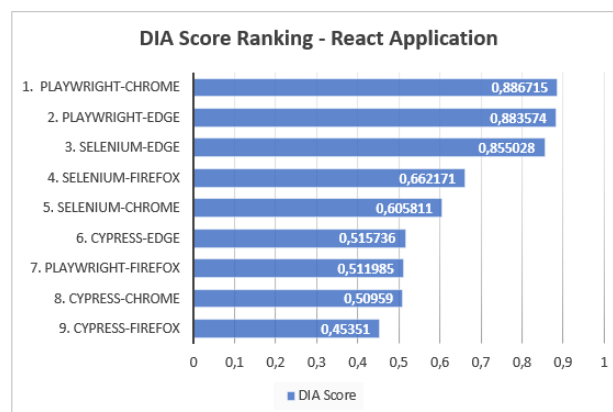


Figure 6. DIA Score React

For React applications, Playwright-Chrome achieved the highest DIA score (0.887), followed by Playwright-Edge (0.884) and Selenium-Edge (0.855). These results indicate that Playwright for React testing is superior when balanced performance is prioritized. The Cypress combination consistently ranks lower (positions 6–9), with Cypress-Firefox receiving the lowest score (0.454) due to high resource overhead.

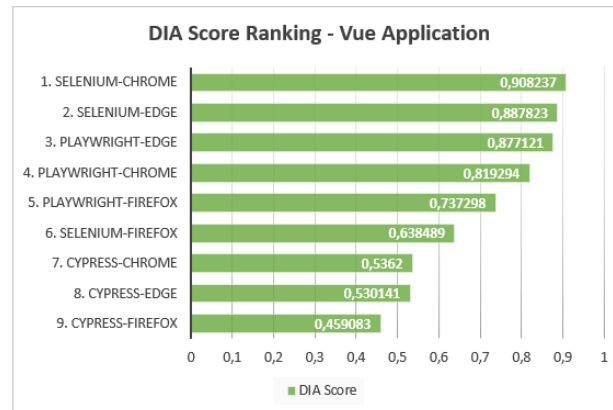


Figure 7. DIA Score Vue

For the Vue application, there was a significant shift in the choice of framework. Selenium-Chrome emerged as the top performer with a DIA score of 0.908—the highest in both applications—followed by Selenium-Edge (0.888) and Playwright-Edge (0.877). In the Vue application, Selenium had the lowest memory consumption across all browser combinations, making it the most efficient option in this experimental scenario. Cypress again ranked the lowest (positions 7–9) with a score of 0.459 for Cypress-Firefox, confirming consistent below-standard performance in both applications.

The difference in optimal framework choice between React (Playwright) and Vue (Selenium) highlights a critical finding: framework choice is context-dependent and must account for the characteristics of the target SPA application.

Framework Optimal: Based on the DIA score calculation that considers all metrics simultaneously, the recommended optimal framework is:

1. Application React: Playwright di Chrome (DIA Score: 0.886715)
 - a. Execution time: 56.25 ± 14.37 seconds
 - b. Success rate: 100.0%
 - c. CPU usage: 40.80%
 - d. Memory: 774.17 MB
2. Application Vue: Selenium di Chrome (DIA Score: 0.908237)
 - a. Execution time: 135.61 ± 4.21 seconds
 - b. Success rate: 100.0%
 - c. CPU usage: 19.67%
 - d. Memory: 196.59 MB

The differences in optimal frameworks between React and Vue indicate that application characteristics influence the choice of testing framework. Playwright excels for React due to its very high execution speed and moderate resource usage, while Selenium is optimal for Vue because it offers the best balance between speed, a perfect success rate, and very high resource efficiency.

3.7. Additional Findings

- a. Stability and Performance Consistency: Standard deviation analysis shows variations in performance consistency across different framework-browser combinations. Playwright-Edge on the React application demonstrates high consistency, with an SD of 1.97 seconds in execution time, while Playwright-Chrome on React shows greater variability, with an SD of 14.37 seconds. This pattern aligns with previous empirical evaluations that report Playwright generally achieves stable execution times in complex scenarios, although variability still depends on the browser and workload configuration. [30], [31], [32].
- b. Difference in Success Rate: Two failures in the React application (Selenium-Chrome and Playwright-Firefox) resulted in a success rate of 93.3%, while the Vue application achieved 100% across all combinations. This difference suggests that the architecture of front-end

frameworks influences the stability of automated testing, with factors such as virtual DOM rendering, reactivity models, and state management playing important roles. Similar observations are reported in comparative studies where differences in DOM update strategies and event handling between React-like and Vue-like architectures lead to distinct patterns of flakiness and robustness in E2E tests [30], [31].

- c. **Trade-off Between Speed and Resource Efficiency:** Playwright offers the highest speed but moderate resource consumption, whereas Selenium provides the best resource efficiency at a slower speed. Cypress is in the middle in terms of speed but has a very high memory overhead (5–6× that of Selenium). These findings are consistent with previous work showing that Playwright typically delivers the shortest execution times, while Selenium remains competitive in terms of CPU and memory efficiency, and Cypress incurs higher memory usage despite its developer-friendly tooling [30], [31], [32]. Furthermore, broader evaluations of UI automation frameworks indicate that such performance differences also translate into significant variation in energy usage, reinforcing the need to balance speed, resource consumption, and environmental impact when selecting an E2E testing framework for specific deployment requirements [33].
- d. **Scope of SPA Workflows and Flakiness:** The 25 test cases in this study primarily cover common business workflows in admin dashboard SPAs—such as authentication, CRUD operations on categories and news, and access control—rather than advanced asynchronous interaction patterns. As a consequence, scenarios that are typically associated with high flakiness, such as concurrent API calls, streaming or WebSocket-driven updates, delayed hydration of lazy-loaded components, and long chains of dependent network requests, are not explicitly represented in the test suite. The results should therefore be interpreted as a baseline for React- and Vue-based dashboard-style SPAs, rather than a comprehensive characterization of all SPA behaviours.

3.8. Threats to Validity

Internal validity: The experiments in this study were conducted on a single Windows 11 machine with non-essential background applications and services disabled to reduce noise, and the application state was reset before each session. Selenium tests were implemented in Python, while Cypress and Playwright tests were implemented in JavaScript/TypeScript, which may introduce minor differences due to runtime overhead and idiomatic wait patterns. Even so, most of the measured execution time is dominated by WebDriver and browser processes, so language-induced variance is likely small relative to the architectural differences between frameworks. CPU and memory usage were sampled every 5 seconds using process-level monitoring, and CPU utilization was reported as the aggregate across all CPU cores; on multi-core processors, this can exceed 100%. Browser cache, cookies, and local storage were not systematically cleared before each run, potentially causing limited caching effects across repetitions. These factors may slightly affect the absolute performance values, but are unlikely to overturn the relative differences driven by the frameworks' architectural characteristics (WebDriver protocol vs. CDP-based vs. in-process execution).

Construct validity: The performance model in this study is based on four metrics—execution time, success rate, CPU usage, and memory usage—which are widely used in performance efficiency and automation testing research. These metrics were normalized using min–max scaling and aggregated using the Distance to Ideal Alternative (DIA) method, with weights that prioritize time behaviour and reliability while accounting for resource utilization. Although this weighting scheme is justified by ISO/IEC 25010 and prior literature, it reflects a particular assumption about practitioner priorities; teams that emphasize extreme reliability or strict resource limits might adopt different weight configurations and obtain different rankings. Future work should perform a systematic sensitivity analysis of the DIA results under alternative weighting scenarios.

External validity: The systems under test are React- and Vue-based admin dashboard SPAs with 25 test cases representing common business workflows (authentication, CRUD operations, access control), rather than all possible SPA interaction patterns. Flakiness-prone

behaviours such as concurrent API calls, streaming or WebSocket-driven updates, delayed hydration of lazy-loaded components, and long chains of dependent network requests are not explicitly modelled. Consequently, the findings should be interpreted as a baseline for dashboard-style SPAs on desktop browsers (Chrome 141, Edge 141, Firefox 143) executed in headless mode on a single Windows environment, and may not generalize to mobile browsers, WebKit/Safari, or CI/CD pipelines with container-based runners and parallel execution. Replicating and extending this study in diverse environments and with more demanding SPA scenarios is an important direction for future research.

4. CONCLUSION

This study conducted a comprehensive evaluation of three end-to-end testing frameworks—Selenium WebDriver, Cypress, and Playwright—on React- and Vue-based single-page applications implementing admin dashboard workflows, using a systematic experimental design across 270 testing sessions. The evaluation was based on four critical metrics—execution time, success rate, CPU consumption, and memory usage—which were then analyzed using the Distance to Ideal Alternative method to determine the optimal framework in each application context.

The main findings indicate that Playwright consistently delivers the best execution speed across all browser–application combinations, with an average time of 56.25 seconds for the React application and 68.76 seconds for the Vue application in Chrome, in line with the findings of Melyawati et al. (2024) and Antończak and Bylina (2024), who also reported that Playwright is faster than Selenium in comparable functional testing scenarios [20], [19]. However, in terms of resource efficiency, Selenium performs better, with the lowest CPU consumption (19.67%) and memory usage (196.59 MB) on the Vue application, making it the optimal choice for environments with limited resources or for containerized deployments with memory constraints, consistent with the results of Wijaya et al. (2025) that highlight resource efficiency in Selenium [10]. These results should be interpreted as a baseline under the tested workload and environment, and may vary for more complex SPA behaviours or different deployment settings.

Multi-criteria analysis using the DIA method reveals that the optimal framework varies across applications. Playwright–Chrome achieved the highest DIA score (0.886715) for the React application, whereas Selenium–Chrome achieved 0.908237 for the Vue application, supporting the hypothesis of context-dependent tool selection, as also shown by Adiatma & Darmayantie (2023), who found that DIA-based framework selection depends on the prioritized criteria [13]. These differences are linked to the architectural characteristics of the front-end frameworks and the tested workflows: Vue’s deterministic reactive system yields a 100% success rate across all combinations in the evaluated admin dashboards, whereas React exhibits two failures and a 93.3% success rate in certain combinations, which is consistent with its asynchronous rendering and state update model.

The academic contribution of this research includes providing a systematic, reproducible baseline comparison of modern end-to-end frameworks for single-page application (SPA) development in the context of React- and Vue-based admin dashboards, applying multi-criteria decision-making methods to select tools based on empirical data, and empirically demonstrating that no single framework is optimal across all application contexts. These findings offer practical guidelines for software testing practitioners for choosing frameworks based on specific priorities: Playwright for continuous integration/continuous deployment (CI/CD) pipelines with tight time constraints, Selenium for resource-limited environments, and special considerations for the application framework being tested.

The practical implications of this research highlight the importance of data-driven evaluation when selecting automation testing tools, with a focus on balancing execution speed and resource efficiency for specific application architectures and deployment environments. Future work should replicate and extend these experiments in container-based CI/CD environments with parallel execution, incorporate WebKit/Safari to fully assess Playwright’s cross-engine capabilities, and enrich the test suite with flakiness-prone SPA scenarios such as concurrent API calls, streaming updates, delayed hydration, and long network request chains. In addition, future studies

could compare the evaluated frameworks with popular codeless or cloud-based automation platforms such as Katalon Studio, BrowserStack, and Sauce Labs, which are widely used in industry for functional and cross-browser testing.

ACKNOWLEDGEMENTS

The authors would like to thank the Informatics Study Program at Universitas Lambung Mangkurat for providing the laboratory facilities and infrastructure used to conduct the experiments in this research. The authors also express their gratitude to colleagues and reviewers who provided valuable feedback on the research design, data analysis, and the manuscript's clarity. The authors appreciate the assistance of Perplexity AI and Grammarly in improving the manuscript's language quality and readability; the authors remain fully responsible for the final content of this publication.

REFERENCES

- [1] A. Bai, "Million.js: A Fast Compiler-Augmented Virtual DOM for the Web," *Proc. ACM Symp. Appl. Comput.*, pp. 1813–1820, 2023, doi: 10.1145/3555776.3577683.
- [2] M. Siahaan *et al.*, "Comparative Analysis Study of Front-End JavaScript Frameworks Performance Using Lighthouse Tool," *J. Mantik*, vol. 6, no. 3, pp. 2685–4236, 2021.
- [3] Octa Selsa Is Anggraeni, Lilik Sugiarto, and Tinuk Agustin, "Studi Komparatif Performa Framework Javascript Modern dalam Pengembangan Aplikasi Web," *Modem J. Inform. dan Sains Teknol.*, vol. 2, no. 4, pp. 162–177, 2024, doi: 10.62951/modem.v2i4.239.
- [4] A. M. Syahputra, "Analisis perbandingan performansi framework Selenium dan Cypress dalam pengujian otomatisasi perangkat lunak pada aplikasi berbasis website," *J. Pengemb. Teknol. Inf. dan Ilmu Komput.*, vol. 8, no. 3, pp. 3959–3966, 2024.
- [5] H. A. Thooriqoh, T. N. Annisa, and U. L. Yuhana, "Selenium Framework for Web Automation Testing: A Systematic Literature ReviewKerangka Selenium untuk Pengujian Otomatis Berbasis Web: Tinjauan Literatur Sistematis," *J. Ilm. Teknol. Inf.*, vol. 19, no. 2, pp. 65–76, 2021, [Online]. Available: <http://dx.doi.org/10.12962/j24068535.v19i2.a1021>
- [6] M. Leotta, B. Garcia, F. Ricca, and J. Whitehead, "Challenges of End-to-End Testing with Selenium WebDriver and How to Face Them: A Survey," *Proc. - 2023 IEEE 16th Int. Conf. Softw. Testing, Verif. Validation, ICST 2023*, pp. 339–350, 2023, doi: 10.1109/ICST57152.2023.00039.
- [7] M. Vieira, V. M. de Lima, W. Viana, M. Bonfim, and P. Rego, "Enhancing Continuous Integration Workflows: End-to-End Testing Automation with Cypress," vol. 2, no. Iceis, pp. 160–167, 2025, doi: 10.5220/0013230200003929.
- [8] S. B. Nettur, S. Karpurapu, U. Nettur, and L. S. Gajja, "Cypress Copilot: Development of an AI Assistant for Boosting Productivity and Transforming Web Application Testing," *IEEE Access*, vol. 13, no. November 2024, pp. 3215–3229, 2025, doi: 10.1109/ACCESS.2024.3521407.
- [9] S. Talakola, "Automated end to end testing with Playwright for React applications," *Int. J. Emerg. Res. Eng. Technol.*, vol. 5, no. 1, pp. 38–47, 2024, doi: 10.63282/3050-922x.ijeret-v5i1p106.
- [10] S. P. Wijaya, M. A. Al Abdilah, M. Farhan, and A. Firmansyah, "ANALISIS PERBANDINGAN EFISIENSI DAN PERFORMA PENGUJIAN SELENIUM, CYPRESS, DAN PLAYWRIGHT SEBAGAI ALAT AUTOMATION TESTING UNTUK APLIKASI WEB," *J. Sains dan Teknol.*, vol. 8, no. 10, pp. 91–100, 2025, doi: 10.2238/rq4d6a48.
- [11] M. Ayli, Y. Bakouny, N. Jalloul, and R. Kilany, "Enhancing the Resiliency of Automated Web Tests with Natural Language," *ACM Int. Conf. Proceeding Ser.*, no. M1, pp. 63–69, 2024, doi: 10.1145/3700523.3700536.
- [12] S. Balsam and D. Mishra, "Web application testing—Challenges and opportunities," *J. Syst. Softw.*, vol. 219, no. August 2024, p. 112186, 2025, doi: 10.1016/j.jss.2024.112186.
- [13] S. Adiatma and A. Darmayantie, "Implementation and Comparative Analysis of Test Automation Framework Performance for Functional Testing of Web-Based Applications using the Distance to the Ideal Alternative (DIA) Method," *Widya Tek.*, vol. 22, no. 2, pp. 36–41, 2023, doi: 10.33508/wt.v22i1.5027.

- [14] M. Cinelli, M. Kadziński, G. Miebs, M. Gonzalez, and R. Słowiński, "Recommending multiple criteria decision analysis methods with a new taxonomy-based decision support system," *Eur. J. Oper. Res.*, vol. 302, no. 2, pp. 633–651, 2022, doi: 10.1016/j.ejor.2022.01.011.
- [15] C. Merina, N. Anggraini, and N. Hakiem, "A comparative analysis of test automation frameworks performance for functional testing in android-based applications using the distance to the ideal alternative method," *Proc. 3rd Int. Conf. Informatics Comput. ICIC 2018*, pp. 1–6, 2018, doi: 10.1109/IAC.2018.8780548.
- [16] Y. P. Cardenas, "Application of multi-criteria decision analysis to the selection of software measures," *Comput. y Sist.*, vol. 22, no. 1, pp. 203–213, 2018, doi: 10.13053/CyS-22-1-2765.
- [17] G. A. Prasetyo and N. Setiani, "Perbandingan Kinerja Framework Automation UI Testing Menggunakan The Distance to The Ideal Alternative," *JIPPI (Jurnal Ilm. Penelit. dan Pembelajaran Inform.)*, vol. 10, no. 1, pp. 224–237, 2025, doi: 10.29100/jipi.v10i1.5760.
- [18] ISO; IEC; IEEE, "Software and systems engineering — Software testing — Part 4: Test techniques," Geneva, 2022. [Online]. Available: <https://www.iso.org/obp/ui/en/#iso:std:79430:en>
- [19] B. B. Agnieszka Antonczak, "Analysis of end-to-end test automation tools based on the examples of Selenium WebDriver and Playwright," *Position Pap. 19th Conf. Comput. Sci. Intell. Syst.*, vol. 40, no. FedCSIS Conference Proceedings, pp. 1–8, 2024, doi: DOI: 10.15439/2024F3747.
- [20] N. L. P. Melyawati, I. M. D. P. Asana, N. N. S. Putri, K. J. Atmaja, and I. G. I. Sudipa, "Comparison of Automation Testing On Card Printer Project Using Playwright And Selenium Tools," *J. Comput. Networks, Archit. High Perform. Comput.*, vol. 6, no. 3, pp. 1309–1320, 2024, doi: 10.47709/cnahpc.v6i3.4362.
- [21] N. Arifin, N. Agustin, and T. Anggoro, "Evaluation Execution Time Features of Simpati Web-Based Monitoring and Evaluation Application Using Automation Testing," *J. Tek. Inform.*, vol. 5, no. 4, pp. 283–289, 2024, doi: 10.52436/1.jutif.2024.5.4.2158.
- [22] H. Rojas, R. Renteria, and V. Martinez, "Understanding Performance Efficiency in ISO / IEC 25000 : A Systematic Literature Review," vol. 9, no. November, pp. 2405–2417, 2025.
- [23] Y. Wang, A. M. Menzies, M. Mäntylä, and M. Nagappan, "Test automation maturity improves product quality," *arXiv preprint arXiv:2202.04068*, 2022.
- [24] A. P. Putra, D. Sartika, and D. Lianda, "Decision Support System For The Best Junior High School Using The Distance To The Ideal Alternative (Dia) Method At The Education And Culture Office Of The City Of Bengkulu Sistem Pendukung Keputusan Sekolah Terbaik Tingkat Menengah Pertama Menggunakan Metode Distance To The Ideal Alternative (Dia) Pada Dinas Pendidikan Dan Kebudayaan Kota Bengkulu," vol. 3, no. 2, pp. 159–166, 2024.
- [25] F. R. B. A. N. R. Setiabudi, "COMPARATIVE ANALYSIS OF AUTOMATION FUNCTIONAL TESTING," vol. 21, no. 1, pp. 9–14, 2024.
- [26] F. S. D. Silva, M. P. S. Lima, D. Corujo, S. Member, A. J. V. Neto, and S. Member, "A Comprehensive Step-Wise Survey of Multiple Attribute Decision-Making Mobility Approaches," vol. 12, no. July, 2024.
- [27] B. Sani and S. Jan, "EAI Endorsed Transactions Empirical Analysis of Widely Used Website Automated Testing Tools," vol. 3, pp. 1–11, doi: 10.4108/airo.7285.
- [28] S. Di Meglio, L. L. L. Starace, V. Pontillo, R. Opdebeeck, C. De Roover, and S. Di Martino, "E2EGit: A Dataset of End-to-End Web Tests in Open Source Projects," *Proc. - 2025 IEEE/ACM 22nd Int. Conf. Min. Softw. Repos. MSR 2025*, pp. 836–840, 2025, doi: 10.1109/MSR66628.2025.00121.
- [29] G. C. B. Miranda, "A Comparative Analysis of Mobile UI Testing Frameworks in Continuous Integration Environments," *ACM Int. Conf. Mob. Softw. Eng. Syst. (MOBILESoft 2023)*, pp. 102–109, 2023, doi: 10.1145/3624032.3624047.
- [30] A. Tymoshchuk, "The Evolution of Test Automation : From Selenium to Playwright . A

- Comparison of Automation Tools : Selenium vs . Playwright vs . Cypress,” vol. 4523, pp. 37–45.
- [31] M. Moń and B. Pańczyk, “A comparative analysis of web application test automation tools Analiza porównawcza narzędzi do automatyzacji testów aplikacji internetowych,” vol. 35, no. January, pp. 159–165, 2025.
- [32] P. Gosik and M. Miłosz, “Comparative analysis of Cypress and Playwright frameworks in end-to-end testing for applications based on Angular Analiza porównawcza frameworków Cypress i Playwright w testach end-to-end dla aplikacji opartych na Angularze,” vol. 36, no. May, pp. 320–327, 2025.
- [33] T. M. R. Lagermann, K. S. Carter, S. Mei, G. Ho, and K. Eder, *Framework Matters : Energy Efficiency of UI Automation Testing Frameworks*, vol. 1, no. 1. arXiv.