

Original Research

Received 06 June 2026

Revised 17 July 2026

Accepted 01 Aug 2026

Natural Resources for Human Health 2026, Vol. 6 (15s): 519–528

KEYWORDS:

Selenium WebDriver, adaptive wait, flaky tests, web automation, Core Web Vitals, performance testing, explicit wait, end-to-end testing

Performance-Aware Adaptive Smart Wait Strategy for Reliable and Efficient Selenium Web Automation

¹P. Sathish Kumar, ²Dr. M.Chandran

¹ Ph.D Research Scholar, ²Associate Professor

1sathishkumarp3011@gmail.com, 2 onchandran@gmail.com

^{1, 2}Dept. of Computer Science,

^{1, 2}Sri Ramakrishna Mission Vidyalaya College of Arts and Science, Coimbatore, India

ABSTRACT: The web automation is vulnerable to timing uncertainty created by asynchronous JavaScript, network variability, dynamic rendering, server load, and browser scheduling. In Selenium WebDriver, fixed sleeps and statically configured timeouts can either fail when a page is slower than expected or waste execution time when the page is faster than the chosen delay. This paper proposes a Performance-Aware Adaptive Smart Wait (PASW) strategy that dynamically estimates an element-specific wait budget from two complementary information sources: runtime page-performance signals and historical element-readiness behaviour. The framework combines navigation and rendering indicators, long-task/blocking behaviour, prior element readiness times, variability, and recent failures to calculate a bounded adaptive timeout before applying condition-aware WebDriver waits. A preliminary baseline obtained from the supplied GTmetrix assessment reports a Grade C, Performance score of 67%, Structure score of 89%, Largest Contentful Paint (LCP) of 2.7 s, Total Blocking Time (TBT) of 133 ms, and Cumulative Layout Shift (CLS) of 0. This baseline indicates that the test environment is suitable for studying the relationship between page responsiveness and automation synchronization. The proposed evaluation compares no-wait/fixed-sleep, implicit wait, explicit wait, fluent wait, and PASW under controlled network and server-load conditions. Primary outcomes are test success rate, flaky-failure rate, timeout frequency, element-wait duration, and total execution time. The study is designed to determine whether performance-aware adaptation can reduce unnecessary waiting while retaining or improving automation reliability.

1. INTRODUCTION

Web applications increasingly rely on asynchronous operations such as API requests, deferred JavaScript execution, client-side rendering, lazy loading, animations, and background resource fetching. These mechanisms improve user experience but create timing uncertainty for browser automation. A Selenium script can locate or interact with a target before that target exists, becomes visible, becomes enabled, or becomes unobstructed. Selenium documentation explicitly notes that a test can attempt an interaction before a dynamically created or revealed element is ready, and it warns that fixed sleep values can be either too short for reliability or excessively long for efficiency [1].

WebDriver standardizes browser automation and includes configurable timeout behaviour, but timeout values remain application- and environment-dependent [2]. In practice, automation engineers frequently compensate for timing uncertainty by increasing sleep durations or timeout limits. This can suppress some intermittent failures, yet it creates a fundamental trade-off: larger waits improve tolerance of slow executions but increase test-suite duration, whereas smaller waits reduce runtime but may amplify flakiness. Presler-Marshall et al. demonstrated that Selenium configuration and timeout strategies affect both test stability and runtime performance [3].

Flaky tests are tests whose outcomes vary without a corresponding change in the software or test logic. UI automation is particularly exposed because browser rendering, network activity, event ordering, and resource availability are inherently asynchronous. Romano et al. analysed 235 flaky UI tests and found asynchronous-wait issues to be a major root cause, accounting for 105 cases in their dataset [4]. Subsequent research has introduced automated approaches for replacing sleeps, tuning waiting times, and synthesizing wait oracles [5]-[9]. These studies establish that synchronization is not merely a coding convenience; it is a measurable software-testing problem with direct implications for reliability and cost.

The central idea of this study is that the wait decision should incorporate information about how the web page is actually behaving. Traditional waits are primarily driven by a timeout chosen in test code. The proposed Performance-Aware Adaptive Smart Wait (PASW) instead treats wait time as a dynamic estimate. It combines element-level history with runtime page-performance signals and then delegates final synchronization to a condition-aware explicit wait. This creates a feedback loop in which each execution contributes evidence for later executions.

The contribution is therefore not a new replacement for Selenium WebDriver, but a performance-aware synchronization layer above it. The framework is intended for dynamic institutional portals, online examination systems, dashboards, admission systems, learning platforms, and similar web applications in which response time can vary with network quality and concurrent server load.

2. BACKGROUND AND RELATED WORK

2.1 Selenium Synchronization

Selenium provides implicit waits and explicit condition-based waits as synchronization mechanisms [1]. An implicit wait is a session-wide timeout applied to element location; the default is zero. Explicit waits repeatedly evaluate a condition until it succeeds or reaches a timeout, and FluentWait allows the polling interval and ignored exception classes to be configured. Selenium cautions against mixing implicit and explicit waits because their interaction can produce unpredictable timeout duration [1]. The W3C WebDriver specification similarly defines timeout configuration and the conditions under which element interaction can fail [2].

A fixed sleep differs conceptually from a condition-based wait. Sleep pauses for the entire programmed duration, even when the element becomes ready earlier. A condition-based wait can return as soon as its predicate is satisfied. However, both still require an upper bound, and choosing that bound is the central issue addressed in this paper.

2.2 Flakiness and Asynchronous Waits

Flaky tests undermine trust because a failure no longer reliably indicates a regression. The survey by Parry et al. synthesizes a broad literature on causes, impacts, detection, and mitigation of flaky tests [10]. For UI systems, the asynchronous nature of browser and network events is particularly important. Romano et al. reported that 45.1% of the 235 UI flaky-test samples they studied were caused by asynchronous-wait issues, with the proportion higher in web projects than mobile projects [4]. Hashemi et al. likewise found concurrency-related issues, including asynchronous waits and race conditions, to be a dominant cause of flakiness in JavaScript projects [11].

2.3 Automated and Adaptive Waiting Research

Presler-Marshall et al. compared Selenium configurations, browsers, and timeout strategies, showing that wait configuration influences stability and runtime [3]. Olanas et al. proposed SleepReplacer, which automatically replaces Thread.sleep statements with explicit waits and demonstrated the practical value of migrating from unconditional delay to state-aware synchronization [5]. Pei et al. studied 49 reproducible asynchronous-wait flaky tests from 26 open-source projects and found that developers changed wait times in about 63% of cases. Their TRaf approach uses code similarity and change history to recommend more efficient waiting times [6].

Liu et al. introduced WEFix, which uses browser UI changes to predict client-side execution and generate explicit wait oracles. On 122 web end-to-end flaky tests, WEFix reported high correctness and substantially lower overhead than a repeated-rerun baseline [7]. Zhang et al. presented an industrial study of a robust waiting strategy for web GUI testing, reinforcing the practical importance of synchronization design in large systems [8]. More recently, Olanas, Leotta, and Ricca proposed SleepReplacer-GPT to replace thread sleeps in Selenium WebDriver tests using an AI-assisted approach [9].

The distinction of PASW is the explicit inclusion of web-performance measurements in the timeout-estimation loop. The reviewed approaches emphasize configuration, code replacement, change history, UI-change signals, or industrial waiting patterns. PASW instead asks whether observable performance characteristics such as navigation delay, long-task pressure, prior LCP/TBT behaviour, and element readiness history can jointly predict a suitable timeout budget at runtime.

Table 1. Positioning of the proposed PASW framework against representative related work

Approach	Primary idea	Condition-aware	Learns/history	Explicit page-performance input
Selenium waits [1]	Implicit/explicit/fluent synchronization	Yes (explicit)	No	No
Presler-Marshall et al. [3]	Analyse imeout/configuration effects	Varies	No	No
SleepReplacer [5]	Replace sleeps with explicit waits	Yes	No	No
TRaf [6]	Infer wait values from code similarity/history	Time-based	Yes	No
WEFix [7]	Generate waits from UI-change signals	Yes	Yes/signals	Indirect
Robust industrial waiting [8]	Industrial waiting strategy	Yes	Application-specific	Not central
PASW (this work)	Runtime performance + element history + explicit condition	Yes	Yes	Yes

2.4 Web Performance Signals

Core Web Vitals currently focus on loading, responsiveness, and visual stability through Largest Contentful Paint (LCP), Interaction to Next Paint (INP), and Cumulative Layout Shift (CLS) [12]. LCP reflects when the largest content element in the viewport is rendered; web.dev recommends 2.5 seconds or less for a good LCP at the 75th percentile [13]. Lighthouse additionally reports Total Blocking Time (TBT), the sum of main-thread blocking portions of long tasks between First Contentful Paint and Time to Interactive. Chrome documentation classifies desktop TBT of 0-150 ms as fast and mobile TBT of 0-200 ms as fast [14].

These metrics were designed primarily to characterize user experience rather than Selenium reliability. Nevertheless, they describe the same browser and main-thread environment in which automation interactions occur. A page experiencing slow resource delivery or prolonged main-thread blocking may also delay the point at which an element becomes reliably interactable. PASW therefore uses page-performance measurements as predictive context, not as a replacement for WebDriver conditions.

3. PRELIMINARY PERFORMANCE BASELINE

The supplied GTmetrix assessment is used as the initial performance baseline for the planned experiment. It reports Grade C, a Performance score of 67%, a Structure score of 89%, LCP of 2.7 s, TBT of 133 ms, and CLS of 0. These values are recorded as observed inputs from the supplied test report and are not generated by the proposed framework.

Table 2. Preliminary web-performance baseline from the supplied GTmetrix result

Metric	Observed value	Interpretation for the study
GTmetrix Grade	C	Moderate overall baseline suitable for improvement and stress experiments
Performance	67%	Indicates measurable performance headroom
Structure	89%	Relatively strong structural score compared with overall performance
LCP	2.7 s	Slightly above the 2.5 s good threshold; useful loading-latency signal [13]
TBT	133 ms	Within Lighthouse fast range for both desktop and mobile thresholds [14]
CLS	0	Excellent visual stability; layout shift is unlikely to dominate this baseline

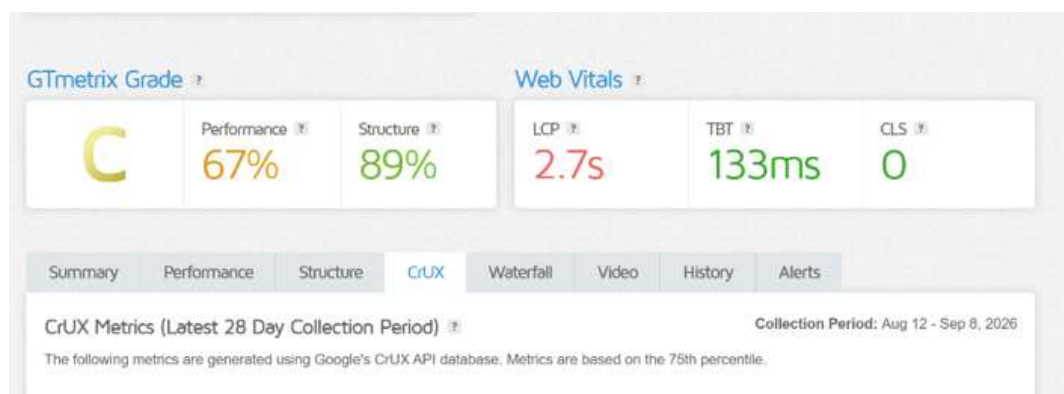


Figure 1. Supplied GTmetrix baseline used to motivate the performance-aware waiting study

The LCP result is particularly relevant because it indicates that the page is not consistently within the recommended "good" loading range, whereas TBT and CLS are already strong. This combination creates an informative starting point: the system can be tested to determine whether Selenium failures correlate more closely with loading latency, blocking behaviour, server/network delay, or element-specific readiness rather than assuming that a single global performance score explains automation reliability.

4. PROPOSED PERFORMANCE-AWARE ADAPTIVE SMART WAIT (PASW)

4.1 Design Principle

PASW separates two concerns. First, it estimates a timeout budget that is large enough for the current context but not unnecessarily conservative. Second, it uses Selenium conditions to decide the exact moment when execution can continue. The timeout is therefore a safety boundary, not a mandatory pause. If an element becomes ready immediately, the wait returns immediately; if the environment is slow, the timeout expands within configured limits.

Performance-Aware Adaptive Smart Wait Architecture

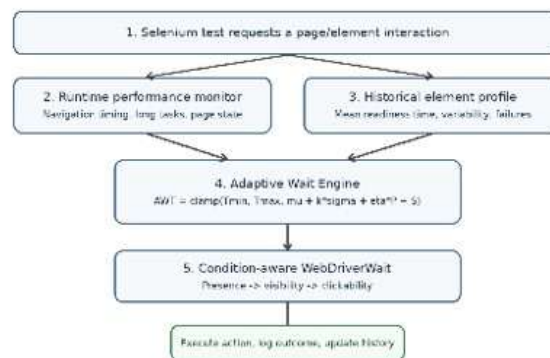


Figure 2. Proposed PASW architecture and feedback loop y

4.2 Input Signals

PASW uses signals that are available at different points in a page lifecycle. Current-run navigation timing (for example, response and DOMContentLoaded timing) and recent long-task information can be available before all Core Web Vitals finalize. LCP and TBT can be used when available in the current run or as historical page-profile values from prior runs. This distinction avoids the methodological mistake of waiting for LCP merely to decide how long to wait for an earlier interaction.

Table 3. Candidate input signals for PASW timeout estimation

Signal group	Examples	Availability	Purpose
Element history	mean readiness time, variance, recent timeout/retry count	Across repeated runs	Captures element-specific behaviour
Navigation timing	TTFB/response time, DOMContentLoaded elapsed time	Early/current run	Captures server/network/page startup delay
Main-thread activity	long tasks, recent blocking duration	Current run	Captures JavaScript/rendering pressure
Page profile	historical/current LCP, TBT, CLS	Current when finalized or historical	Captures broader performance context
Environment	network profile, server-load level, browser/device class	Experiment metadata	Controls and stratifies adaptation

4.3 Adaptive Timeout Model

Each target element e , the framework maintains the historical mean readiness time μ_e and a dispersion estimate σ_e . A normalized page-performance penalty P summarizes the currently available performance context. The proposed bounded timeout is:

$$AWT_e = \text{clamp}(T_{\min}, T_{\max}, \mu_e + k \cdot \sigma_e + \eta \cdot P + S)$$

where AWT_e is the adaptive wait-time ceiling for element e ; $T_{\min}$ and $T_{\max}$ prevent unrealistically small or excessive values; k controls tolerance for element-time variability; η scales the page-performance penalty; and S is a small safety margin.

P is computed from normalized runtime indicators such as navigation delay and recent blocking activity, with historical LCP/TBT values contributing when current values are not yet finalized. The weights and constants are not assumed to be universally optimal; they are hyperparameters to be tuned on a training subset and frozen before final evaluation.

An exponentially weighted update can give greater importance to recent executions while preserving history. If x_t is the observed readiness time in the current run, the mean may be updated as $\mu_t = \alpha * x_t + (1 - \alpha) * \mu_{(t-1)}$, with $0 < \alpha \leq 1$. A similar exponentially weighted variance can be maintained. This allows the profile to respond to changing server or application behaviour without discarding earlier observations.

4.4 Condition Selection

Timeout adaptation alone is insufficient. PASW pairs the estimated budget with the condition appropriate to the action. A text assertion may require presence or visibility; a click should require visibility and clickability; an AJAX-updated value may require a text/value predicate; navigation may require URL or document-state change. This design follows the principle that a wait should express the state the test truly needs rather than merely delaying execution.

4.5 Retry and Failure Policy

PASW permits a limited, observable recovery rather than unlimited retries. If the first adaptive wait expires, the framework records the timeout context and may perform one bounded retry with an expanded ceiling if the page still shows progress. Repeated failure is reported as a genuine test failure. Logging the first timeout, retry decision, final outcome, and actual readiness time provides data for research analysis and prevents silent masking of defects.

4.6 Algorithm

Algorithm 1: Performance-Aware Adaptive Smart Wait

Input: driver, element locator e , required condition c

Output: element/action result or timeout failure

1. Read current runtime performance signals R
2. Load historical profile H_e for element e
3. Compute normalized performance penalty $P(R, \text{page history})$
4. Estimate $AWT_e = \text{clamp}(T_{\min}, T_{\max}, \mu_e + k * \sigma_e + \eta * P + S)$
5. Start condition-aware `WebDriverWait(AWT_e)` for condition c
6. If c becomes true:
 - record actual readiness time x_t
 - update H_e using exponentially weighted statistics
 - execute action and return success
7. Else:
 - record timeout and current progress signals
 - if one bounded retry is allowed and progress is observed:
 - increase ceiling within $T_{\max}$ and retry once
 - else:
 - return timeout failure

8. Persist metrics for experimental analysis

4.7 Illustrative Selenium Implementation Pattern

```
Duration adaptiveTimeout = model.timeoutFor(pageId, elementId, runtimeSignals);
WebDriverWait wait = new WebDriverWait(driver, adaptiveTimeout);
long start = System.nanoTime();
try {
    WebElement element = wait.until(
        ExpectedConditions.elementToBeClickable(By.id("submit"))
    );
    long readyMs = (System.nanoTime() - start) / 1_000_000;
    model.recordSuccess(pageId, elementId, readyMs, runtimeSignals);
    element.click();
} catch (TimeoutException ex) {
    model.recordTimeout(pageId, elementId, adaptiveTimeout, runtimeSignals);
    throw ex;
}
```

This pattern is intentionally compatible with standard Selenium APIs. PASW changes how the timeout is estimated and how observations are logged; it does not require a proprietary browser driver. A Java prototype can store profiles in JSON, SQLite, MySQL, or a simple CSV during early experiments, with a database preferred for large test suites.

5. EXPERIMENTAL METHODOLOGY

5.1 System Under Test and Test Scenarios

The evaluation should use a real dynamic web application containing representative interactions such as login, dashboard loading, menu navigation, dynamic tables, AJAX-driven forms, file or content loading, and submission confirmation. Twenty to thirty stable functional actions should be selected. Each action must have a deterministic functional oracle so that synchronization failure can be separated from genuine application failure.

For high-load experiments, Selenium should be used as the functional browser probe rather than as the sole load generator. A dedicated load-testing tool such as JMeter or k6 can generate 25, 50, 100, or 150 concurrent backend users while a controlled number of Selenium sessions measure end-user automation behaviour. This separation avoids conflating browser-process overhead with server-load effects.

5.2 Experimental Factors

Table 4. Experimental factors and suggested levels

Factor	Suggested levels	Purpose
Network condition	Normal, 100 ms latency, 300 ms latency, 600 ms latency, constrained throughput	Tests sensitivity to client/network delay
Server load	1, 25, 50, 100, 150 concurrent virtual users	Tests backend-load sensitivity
Wait strategy	M1-M5	Independent variable
Browser	Chrome stable as primary; optional Edge/Firefox replication	Controls/replicates browser effects
Run repetition	At least 30-50 repeats per principal configuration	Supports flakiness and distribution analysis

5.3 Two-Stage Experimental Plan

1. Stage A - Network sensitivity: compare M1-M5 across five network profiles with server load held constant. Fifty repetitions per combination yield $5 \times 5 \times 50 = 1,250$ test-suite executions.
2. Stage B - Server-load sensitivity: compare M1-M5 across selected concurrent-load levels while network conditions are fixed. Thirty repetitions across five load levels yield $5 \times 5 \times 30 = 750$ additional executions.

The two-stage structure makes causal interpretation easier than simultaneously changing all factors. A later factorial study can investigate interactions after the principal effects are understood. The exact number of repetitions may be adjusted following a pilot variance analysis or power analysis, but the decision should be documented before final data collection.

5.4 Statistical Analysis

Execution-time and readiness-time distributions are expected to be skewed, so the analysis should report median, interquartile range, mean, standard deviation, and 95th percentile. Normality should be assessed before choosing parametric tests. If assumptions are satisfied, ANOVA with suitable post-hoc comparisons can be used; otherwise Kruskal-Wallis followed by corrected pairwise tests is appropriate. Effect sizes such as Cohen's d or Cliff's delta should accompany significance tests.

Binary outcomes such as pass/fail and timeout/no-timeout can be compared using chi-square or Fisher's exact tests and modelled with logistic regression. A regression model can estimate the contribution of wait strategy, network profile, server load, and their interactions to failure probability. Correlation or regression can also test whether element readiness is associated with page-performance indicators. Multiple-comparison correction should be applied where necessary, and confidence intervals should be reported rather than relying only on p-values.

6. EVALUATION CRITERIA AND RESULT REPORTING

The study should not claim improvement until the comparative runs are completed. The decision rule for PASW is therefore defined in advance. PASW will be considered practically successful if it maintains a success rate statistically indistinguishable from, or higher than, the strongest conventional baseline while reducing median or total execution time by a practically meaningful amount. It should also avoid merely shifting failures into retries; first-timeout frequency, recovered retries, and final failures must be reported separately.

Table 5. Result table to be populated after experimental execution

Strategy	Success rate	Median runtime	95th pct. runtime	Timeouts	Retries
Fixed sleep	To measure	To measure	To measure	To measure	To measure
Implicit wait	To measure	To measure	To measure	To measure	To measure
Explicit wait	To measure	To measure	To measure	To measure	To measure
Fluent wait	To measure	To measure	To measure	To measure	To measure
PASW	To measure	To measure	To measure	To measure	To measure

In addition to the aggregate comparison, a per-element analysis is important. Some elements may benefit substantially from adaptation while others are already stable. Reporting only a whole-suite average can hide this difference. The final results section should therefore identify high-variance elements, show how their adaptive timeout evolved, and compare their failure and wait-time distributions across strategies.

The proposed framework reframes waiting from a fixed configuration parameter into a prediction problem. An automation script rarely needs to know exactly how long a page will take; it needs a sufficiently safe upper bound while remaining responsive to early success. Historical readiness data captures the local behaviour of an element, while performance signals explain some of the environmental variation around that behaviour. Combining the two creates a cross-layer view of synchronization.

The preliminary GTmetrix baseline illustrates why a single metric should not be treated as the wait oracle. LCP is slightly outside the recommended good range, while TBT and CLS are strong. A Selenium failure could still occur because an API response, overlay, delayed component, or server-side request is slow even when page-level scores appear acceptable. PASW therefore uses performance context to scale risk but continues to rely on the exact WebDriver condition required by the action.

One expected practical advantage is reduced manual timeout tuning. In conventional suites, engineers often increase a timeout after a failure and leave the larger value permanently. Over time, this can accumulate excessive latency across hundreds of interactions. PASW instead records readiness observations and permits different elements to have different timeout distributions. A stable button can retain a small ceiling, while a network-dependent table or report can receive a larger one.

However, adaptation can also conceal defects if implemented carelessly. A page that becomes progressively slower should not simply receive ever-increasing timeouts. The $T_{\max}$ bound, failure logging, trend monitoring, and one-retry policy are therefore essential. An adaptive framework should improve synchronization, not normalize a deteriorating application.

7. CONCLUSION

Timing uncertainty is a central cause of unreliable end-to-end web automation. Fixed sleeps are simple but inefficient, and static timeout values cannot represent the full range of element and environment behaviour. Prior research has established the importance of asynchronous waits and has demonstrated the value of automated wait repair and explicit wait generation. Building on this foundation, the PASW framework proposes a performance-aware, element-specific adaptive timeout that combines runtime page context with historical readiness behaviour while preserving Selenium's condition-based synchronization semantics.

The supplied GTmetrix result provides a concrete preliminary baseline: Grade C, Performance 67%, Structure 89%, LCP 2.7 s, TBT 133 ms, and CLS 0. The next research step is empirical execution of the predefined comparison matrix. The contribution should be judged not by whether PASW simply waits longer, but by whether it can maintain reliability with less unnecessary waiting across realistic network and server-load variation. The paper therefore provides a transparent research design that can be converted into a full empirical journal manuscript once comparative measurements are collected.

REFERENCES

- [1] Selenium Project, "Waiting Strategies," Selenium WebDriver Documentation, 2025. Accessed Aug. 2026.
- [2] World Wide Web Consortium (W3C), "WebDriver 2," W3C Recommendation/Specification, 2026.
- [3] K. Presler-Marshall, E. Horton, S. Heckman, and K. T. Stolee, "Wait, Wait. No, Tell Me: Analyzing Selenium Configuration Effects on Test Flakiness," in 2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST), 2019, pp. 7-13, doi: 10.1109/AST.2019.000-1.
- [4] Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang, "An Empirical Analysis of UI-based Flaky Tests," in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 1585-1597, doi: 10.1109/ICSE43902.2021.00141.
- [5] D. Olanas, M. Leotta, and F. Ricca, "SleepReplacer: A Novel Tool-Based Approach for Replacing Thread Sleeps in Selenium WebDriver Test Code," *Software Quality Journal*, vol. 30, pp. 1089-1121, 2022, doi: 10.1007/s11219-022-09596-z.
- [6] Y. Pei, J. Sohn, S. Habchi, and M. Papadakis, "Non-Flaky and Nearly Optimal Time-Based Treatment of Asynchronous Wait Web Tests," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 2, Art. 45, 2025, doi: 10.1145/3695989.
- [7] X. Liu, Z. Song, W. Fang, W. Yang, and W. Wang, "WEFix: Intelligent Automatic Generation of Explicit Waits for Efficient Web End-to-End Flaky Tests," in *Proceedings of the ACM Web Conference 2024*, 2024, doi: 10.1145/3589334.3645628.
- [8] H. Zhang, L. Liao, Z. Ding, W. Shang, N. Narula, C. Sporea, A. Toma, and S. Sajedi, "Towards a Robust Waiting Strategy for Web GUI Testing for an Industrial Software System," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2024, pp. 2065-2076, doi: 10.1145/3691620.3695269.
- [9] D. Olanas, M. Leotta, and F. Ricca, "SleepReplacer-GPT: AI-Based Thread Sleep Replacement in Selenium WebDriver Tests," in *16th Workshop on Automated Testing (A-TEST), ICST Workshops 2025*, pp. 111-120, doi: 10.1109/ICSTW64639.2025.10962456.

- [10] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A Survey of Flaky Tests," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 1, pp. 1-74, 2022, doi: 10.1145/3476105.
- [11] N. Hashemi, A. Tahir, and S. Rasheed, "An Empirical Study of Flaky Tests in JavaScript," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022, pp. 24-34, doi: 10.1109/ICSME55016.2022.00011.
- [12] web.dev, "Core Web Vitals," Google/Chrome web performance guidance. Accessed Aug. 2026.
- [13] web.dev, "Largest Contentful Paint (LCP)," Google/Chrome web performance guidance. Accessed Aug. 2026.
- [14] Chrome for Developers, "Total Blocking Time," Lighthouse Documentation. Accessed Aug. 2026.
- [15] M. Kuuttila, M. Mäntylä, and M. Claes, "Benchmarking Web-Testing: Selenium versus Watir and Various Software Platforms," *arXiv:1611.00578*, 2016.