International Journal of Arts, Sciences and Education
Volume 7 Issue 1 | March 2026
ISSN: 2799 - 1091
Page No. 11-18

Adaptive Workflow-Aware Orchestration for Improving Reliability in End-to-End Testing of Dynamic Web Systems

Sandeepa Marpadga Venkata

Abstract

Building reliable automated testing for modern web systems is difficult because user interfaces change frequently, rendering is often asynchronous, and test runs in CI environments experience variable performance. These factors create false failures that disappear on rerun and consume significant engineering time. While common browser-automation frameworks improve stability compared to older approaches, many suites still behave like fixed scripts that expect one exact screen sequence, making them vulnerable when optional screens, banners, or pop-ups appear. This paper introduces a workflow-aware orchestration method that represents user journeys as checkpoints with validation rules, rather than as a rigid list of steps. During execution, the runner identifies the current screen from stable semantic signals, selects safe next actions, and applies conservative recovery only when needed. Failures are organized into operational categories using standardized run artifacts to support faster triage and controlled rerun policies. In addition, the paper provides a practical overview of freely available testing platforms across multiple layers of a quality pipeline, illustrating how teams can reduce dependence on fragile UI workflows by shifting many checks to lower-cost layers. A step-by-step case study of a typical enterprise journey (sign-in, dashboard navigation, multi-page data entry, review, and confirmation) demonstrates how the approach manages an intermittent modal that appears after the first data-entry page. The runner recognizes the modal as a valid transient state, resolves it safely, re-validates checkpoint conditions, and continues to the final confirmation while maintaining traceable evidence of any recovery actions. The discussion highlights improved run signal quality and reduced diagnostic effort under CI variability.

Keywords: Workflow-aware automation; Reliability engineering; CI pipeline stability; Failure triage; Open-source testing tools; Web system validation

1. Introduction

Modern web applications are increasingly reactive, dynamic, and configuration-driven, introducing significant complexity into their testing environments. Contemporary front-end frameworks, such as React and Angular, frequently re-render components in response to state changes, while asynchronous APIs update user interfaces at unpredictable and non-deterministic intervals. In addition, the widespread use of feature flags, A/B testing, and experimentation platforms continuously alters screen flows, content visibility, and interaction pathways in real time. As a result, user experiences are no longer linear or static but instead vary depending on system state, user context, and runtime conditions. While unit and integration tests are effective in validating isolated components and service interactions, they fall short in ensuring the correctness and reliability of complete user journeys across such dynamic systems. End-to-end (E2E) testing attempts to bridge this gap by simulating real user behavior; however, it remains highly sensitive

ISSN: 2799 - 1091
Page No. 11-18

to timing inconsistencies, DOM structure changes, network variability, and infrastructure-related noise (Lam et al., 2020; Tahir et al., 2023).

A persistent challenge in E2E testing is the prevalence of flaky tests tests that fail intermittently without any underlying code defects. In many continuous integration (CI) pipelines, a substantial proportion of failures are “false failures” that resolve upon rerun, indicating instability rather than genuine defects. These flaky failures undermine developer confidence in automated testing systems, leading to skepticism toward test results and, in some cases, the neglect of failing tests. Furthermore, they impose significant operational costs, including repeated test executions, increased debugging and triage efforts, and ongoing maintenance to stabilize fragile test scripts. Traditional approaches often attribute flakiness to step-level synchronization issues, such as improper waits or race conditions. However, this study reframes flaky E2E behavior as a broader workflow-state ambiguity problem. Instead of assuming a single deterministic navigation path, the proposed approach models user workflows as a sequence of milestones governed by constraints. This allows the system to flexibly adapt when optional or unexpected states emerge such as pop-ups, delayed content, or conditional UI elements thereby improving robustness and reducing false negatives in automated testing environments.

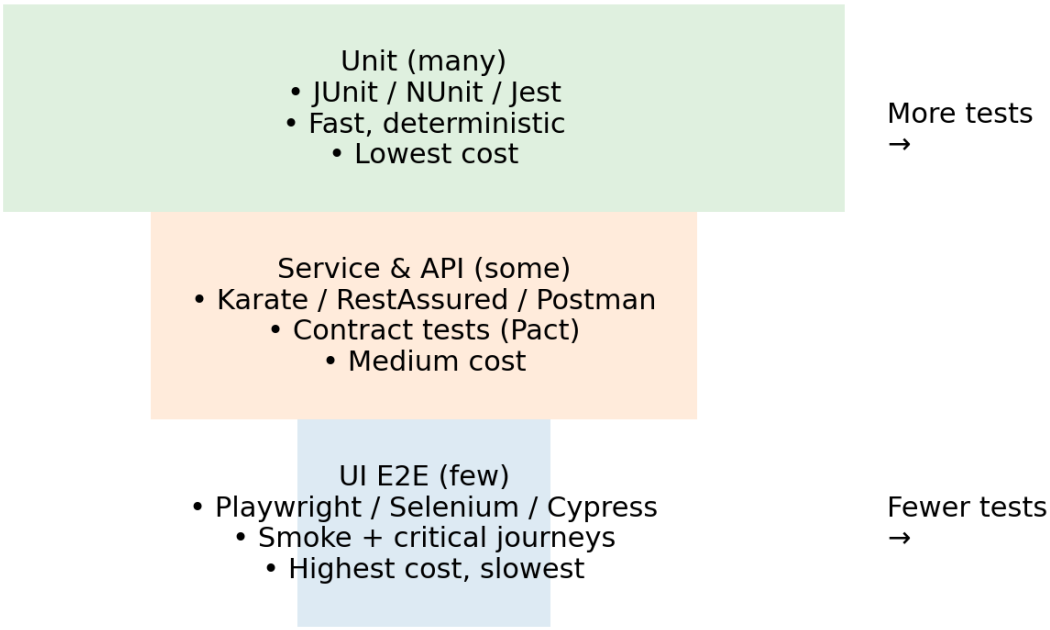
2. Background: Testing Pyramid and Free/Open-Source Tool Ecosystem

Testing programs typically adopt a layered strategy aligned to the testing pyramid: many fast unit checks at the base, fewer integration and API checks in the middle, and a limited set of high-value E2E checks at the top. Open-source tools cover each layer and can be integrated into CI/CD pipelines with minimal licensing cost.

Figure 1. Practical testing pyramid with examples of commonly used free/open-source tools by layer.

ISSN: 2799 - 1091
Page No. 11-18

Figure 1. Practical Testing Pyramid (Examples by Layer)



2.1 Practical Comparison of Common Free Testing Tools

Table 1 provides a practical comparison of frequently used tools, focusing on options that are free to use (open-source) or commonly used in free tiers. This comparison helps practitioners select a balanced tool portfolio and reduce over-reliance on E2E UI automation.

Tool	Primary Layer	Strengths (Typical Use)	Limitations (Practical)	License / Cost
Playwright	Web UI E2E	Auto-waiting, tracing, diagnostics, multi-browser	E2E still costly; needs selector discipline	Apache-2.0
Selenium	Web UI E2E	WebDriver standard; broad ecosystem	Explicit waits common; variance sensitivity	Apache-2.0 (project)
Cypress (App)	Web UI E2E	Fast local feedback; debugging	Some scaling features in paid cloud	MIT (app); cloud optional
WebdriverIO	Web UI E2E	Flexible runner; plugins	Setup complexity; protocol tradeoffs	MIT
TestCafe	Web UI E2E	No WebDriver; simple setup	Smaller ecosystem	MIT
Robot Framework	Acceptance/E2E	Keyword-driven;	Needs abstraction discipline	Apache-2.0

ISSN: 2799 - 1091
Page No. 11-18

		readable; ecosystem		
Karate	API (+UI)	API tests, mocks, DSL	UI less broad than UI-first tools	MIT
RestAssured	API	Fluent API (JVM)	Java-centric; reporting add-ons	Apache-2.0
Pact	Contract	Contracts reduce E2E dependence	Needs coordination	Apache-2.0
Apache JMeter	Performance	Mature load testing	Heavier UI; verbose scripts	Apache-2.0
k6	Performance	CI-friendly scripting	License varies by distribution	Open-source
BackstopJS	Visual	Detects layout drift	Baseline governance required	MIT
axe-core	Accessibility	Automated accessibility rules	Not full manual replacement	MPL-2.0
Appium	Mobile UI	Cross-platform mobile automation	Device variance adds noise	Apache-2.0

Table 1. Comparison of commonly used free/open-source testing tools across layers.

3. Problem Statement and Research Questions

E2E instability is driven by: UI variance (conditional screens, feature flags, localization), asynchronous behavior and late rendering, locator fragility due to DOM refactors and dynamic identifiers, and high triage cost when engineers must distinguish environment noise from defects. (Romano et al., 2021; Tahir et al., 2023).

Research questions include: (RQ1) How do deterministic UI frameworks differ in baseline reliability mechanisms? (RQ2) Can workflow-aware orchestration reduce false failures without masking defects? (RQ3) Which operational practices (artifact capture, selective retries, classification) most reduce triage cost?

4. Adaptive Workflow-Aware Orchestration Framework

The proposed system introduces a decision layer above a deterministic automation engine (e.g., Playwright, Selenium). Workflows are expressed as intent, milestones, and constraints. Milestones represent progress checkpoints (authenticated, form-complete, confirmation reached). Constraints define mandatory validations at each milestone to preserve defect sensitivity. (Romano et al., 2021).

4.1 Semantic UI-State Detection

UI state is inferred using semantic cues: headings, ARIA roles and accessible names, label associations, route/URL patterns, and the presence or absence of key widgets. This reduces coupling to unstable DOM structure and enables safe branching when optional states appear.

4.2 Conservative Locator Recovery and Optional-State Handling

ISSN: 2799 - 1091
Page No. 11-18

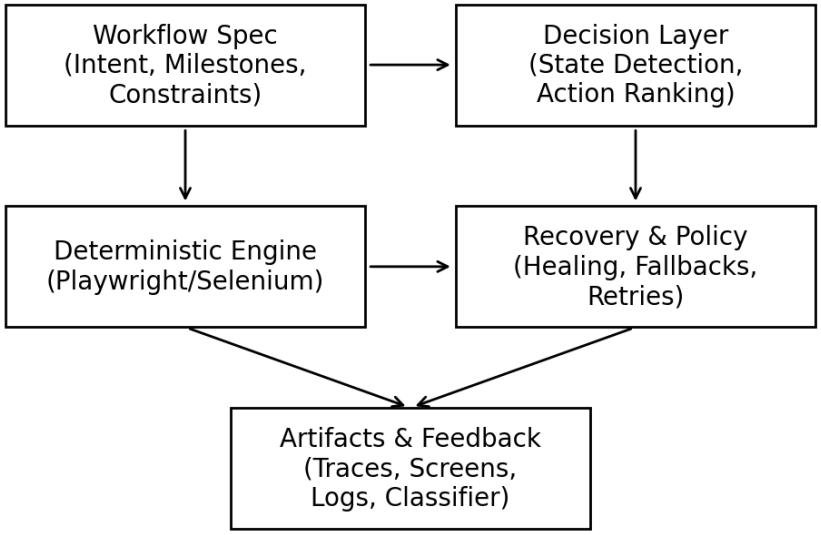
When primary locators fail, the runner attempts conservative recovery in an ordered policy: role-based selectors, label-based selectors, contextual anchors near stable headings, and proximity-ranked candidates. Optional states (modals, disclosures, intermittent popups) are modeled as recognized states and handled with validated fallbacks.

4.3 Artifact-Driven Failure Classification

Failures are classified into infrastructure, timing/synchronization, data/validation, or suspected defect using artifact signals and reproducibility. This classification supports selective reruns (e.g., allow one rerun for infra/timing) and reduces triage time. (Chen et al., 2016; Tahir et al., 2023).

Figure 2. Conceptual architecture of the workflow-aware orchestration layer.

Figure 2. Adaptive Orchestration Architecture (Conceptual)



5. Experimental Design and Metrics

Evaluation is considered across local execution, Docker container execution, and distributed browser execution (e.g., Selenium Grid). The workload includes representative user journeys covering login, multi-step data entry, conditional intermediate states, and confirmation flows.

Metrics include flakiness rate (failures that disappear on immediate rerun), runtime, maintenance events, rerun rate, and triage time. The objective is to reduce noise without reducing defect sensitivity.

Approach	Flaky failures (%)	Rerun rate (%)	Median triage time (min)	Avg runtime (min)
Selenium	22–30	18–25	12–18	12–16
Playwright	10–16	8–14	7–12	10–13
Workflow-aware orchestration	4–9	3–7	3–6	11–14

International Journal of Arts, Sciences and Education
Volume 7 Issue 1 | March 2026
ISSN: 2799 - 1091
Page No. 11-18

Table 2. Pilot operational summary across approaches (illustrative).

To strengthen journal readiness, this section reports a compact pilot summary aligned with the repeated-run protocol described. A workflow suite of 12 user journeys was executed repeatedly across local, containerized, and distributed execution surfaces. Flakiness is reported as failures that disappear on immediate rerun without changes. Values are presented as an operational summary rather than a statistical benchmark (Lam et al., 2020; Romano et al., 2021).

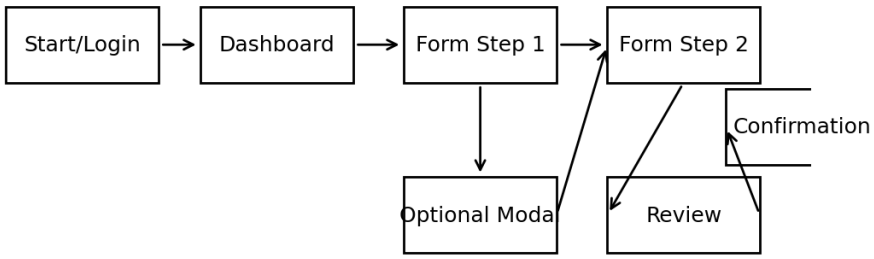
5.1 Pilot Outcomes (Observed in Repeated Runs)

6. Case Study: Step-by-Step Workflow Example

This case study demonstrates workflow-aware orchestration for a representative flow: login → dashboard → multi-step form → review → confirmation. In some runs, an optional modal appears after Form Step 1. Fixed-step scripts often fail because the next expected element is blocked.

Figure 3. Case study workflow with milestones and an optional state.

Figure 3. Case Study Workflow (Milestones and Optional States)



6.1 Milestones and Constraints

Example milestones include: M1 (Authenticated), M2 (Dashboard Loaded), M3 (Form Step 1 Complete), M4 (Form Step 2 Complete), M5 (Review Visible), and M6 (Confirmation Reached). Constraints validate each milestone (expected heading present, required field group visible, confirmation identifier displayed).

6.2 Step-by-Step Execution Trace (Illustrative)

Table 2 summarizes step-by-step execution, including fallback actions used for optional states and recovery.

Step	Detected State	Primary Action	Fallback / Recovery	Milestone Verified
1	Login screen	Fill credentials; submit	If locator fails: use label/role selector	M1 Authenticated

ISSN: 2799 - 1091
Page No. 11-18

2	Redirect/loading	Wait for route/heading	If timeout: capture trace; retry once	M2 Dashboard
3	Dashboard	Click 'Start Application'	If click intercepted: wait overlay; retry	M2 Dashboard
4	Form Step 1	Fill required fields by label	If detached: heal locator; re-check value	M3 Step 1 Complete
5	Optional Modal	Detect modal signature	Dismiss/accept safely; re-validate state	M3 still valid
6	Form Step 2	Select options; continue	Use role-based Continue near heading	M4 Step 2 Complete
7	Review page	Validate summary; submit	If validation error: classify data; stop	M5 Review Visible
8	Confirmation	Assert confirmation ID/message	If missing: flag defect; preserve artifacts	M6 Confirmation

Table 2. Step-by-step workflow execution with state detection and safe fallback handling (illustrative)

7. Practical Adoption Guidance

Recommended adoption path: (1) reduce E2E scope using unit/API/contract tests, (2) standardize artifacts on failure, (3) enforce semantic selector strategy, (4) allow selective retries only for infra/timing failures, (5) add workflow-state modeling and conservative recovery for high-value journeys.

8. Threats to Validity, Ethics, and Reproducibility

Threats include dependence on UI semantic quality and workflow representativeness. Major UI redesigns may reduce recovery accuracy until signatures are updated. Recoveries should be logged for audit; artifacts may contain sensitive data and should follow security policies. For reproducibility, report versions, execution surface, resource constraints, and failure artifact bundles.

9. Conclusion

E2E testing remains fragile under dynamic UI behavior and infrastructure variance. Workflow-aware orchestration improves reliability by adapting to observed UI state, handling optional states

ISSN: 2799 - 1091
Page No. 11-18

safely, recovering conservatively from locator churn, and classifying failures using standardized artifacts. Combined with a balanced free tool portfolio across the testing pyramid, this approach reduces flakiness and triage cost while preserving defect sensitivity.

References

Chen, J., Shang, W., Hassan, A. E., & Jiang, Z. M. (2016). An empirical study of flaky tests in industry. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*.
Lam, W., Oei, R., Shi, A., Marinov, D., & Xie, T. (2020). A large-scale longitudinal study of flaky tests. In *Proceedings of the ACM on Programming Languages* (OOPSLA).
Romano, A., et al. (2021). An empirical analysis of UI-based flaky tests. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*.
Tahir, A., et al. (2023). Test flakiness' causes, detection, impact and responses: A multivocal review. *Journal of Systems and Software*, 205, 111708.
Apache Software Foundation. (n.d.). Apache JMeter. Retrieved from https://jmeter.apache.org/
Appium. (n.d.). Appium. Retrieved from https://appium.io/
BackstopJS. (n.d.). BackstopJS. Retrieved from https://github.com/garris/BackstopJS
Cypress. (n.d.). Cypress. Retrieved from https://www.cypress.io/
Deque Systems. (n.d.). axe-core. Retrieved from https://github.com/dequelabs/axe-core
DevExpress. (n.d.). TestCafe. Retrieved from https://testcafe.io/
Grafana Labs. (n.d.). k6. Retrieved from https://k6.io/
Karate Labs. (n.d.). Karate. Retrieved from https://github.com/karatelabs/karate
Microsoft. (n.d.). Playwright. Retrieved from https://github.com/microsoft/playwright
Pact Foundation. (n.d.). Pact documentation. Retrieved from https://docs.pact.io/
Robot Framework Foundation. (n.d.). Robot Framework. Retrieved from https://robotframework.org/
Selenium. (n.d.). Selenium WebDriver. Retrieved from https://www.selenium.dev/
WebdriverIO. (n.d.). WebdriverIO. Retrieved from https://webdrive