

Interaction-Level Test Script Generation for Web and Mobile Applications

Authors

Jia Wei Tan^{1*}, Aisyah Rahman², Arun Nair³

Affiliations

Department of Computer Science, School of Computing, National University of Singapore, COM1, 13 Computing Drive, Singapore 117417, Republic of Singapore

*Corresponding author: tan.jiawei@comp.nus.edu.sg

Abstract

Automated user-interface testing is essential for modern web and mobile applications, but writing stable test scripts is labor-intensive. Generated test scripts often fail because they depend on fragile selectors, ignore asynchronous loading, miss important user paths, or contain weak test oracles. This study examines interaction-level test script generation for web and mobile applications. We propose TestNav-Agent, a multi-agent model that includes a screen-state analysis agent, a user-intent planning agent, a test script generation agent, an oracle construction agent, and a flakiness repair agent. The screen-state analysis agent builds page and component representations from DOM trees, accessibility labels, screenshots, and event traces. The user-intent planning agent converts functional requirements into realistic interaction paths. The test script generation agent produces Playwright, Selenium, and Appium scripts. The oracle construction agent checks expected page transitions, data changes, form validation messages, and error states. The flakiness repair agent replaces unstable selectors, adds wait conditions, and removes timing-sensitive assertions. The study was conducted on 52 production web and mobile interfaces from e-commerce, online education, healthcare appointment, and financial service applications. The evaluation material included 68,000 recorded user actions, 9,400 screen states, 1,260 bug reports, and 740 manually written regression tests. TestNav-Agent generated scripts that covered 76.8% of critical user paths identified by product teams, compared with 54.2% for a direct script-generation model. Across 100 repeated test runs, the flaky failure rate decreased from 13.5% to 4.1%. The generated tests detected 37 previously documented regression bugs and 11 unreleased interface defects during staging evaluation. Selector repair was especially effective in dynamic pages, where element-location failures decreased by 62.4%. The average maintenance effort after interface changes was reduced from 19.6 minutes to 8.3 minutes per test case. These results suggest that interaction-level planning and flakiness repair can improve the practical value of automated UI test generation.

Keywords: UI testing; test script generation; web applications; mobile applications; software testing; flakiness repair; multi-agent systems

1. Introduction

Web and mobile applications are updated frequently, making automated interaction testing difficult to maintain over time. A test script must locate the correct interface element, wait for the required page state, follow the intended user interaction path, and verify that the resulting state is correct. Changes in DOM structure, accessibility attributes, element labels, rendering time, network latency, or response data can cause previously valid scripts to fail even when the application itself remains functionally correct. Such flaky tests increase maintenance costs, consume debugging effort, and reduce developers' confidence in regression testing results [1]. Existing approaches mainly focus on repairing test cases after failures have already occurred, for example by replacing invalid selectors, identifying similar elements, or reconstructing broken interaction sequences [2,3]. Meanwhile, large language models and multi-agent systems have demonstrated considerable potential for generating, refining, and optimizing executable code through coordinated planning, implementation, evaluation, and revision processes [4]. These advances suggest that intelligent agents may also support automated test generation. However, directly generated test scripts may contain invalid selectors, omit important user actions, use insufficient synchronization mechanisms, or rely on weak assertions that fail to verify the intended application behavior [5,6].

Recent studies have extended language-model-based automation to mobile graphical user interface testing. Language models have been used to generate context-sensitive text inputs, interpret screen content, explore Android interfaces, construct unusual or adversarial inputs, reproduce application crashes, improve interface coverage, and migrate existing test cases between application versions [7,8]. These methods reduce the dependence on manually written interaction scripts and allow testing systems to adapt their actions according to the observed interface state. Related research in web testing has mapped natural-language instructions to browser operations and used page structure, interface semantics, or screen-transition information to construct end-to-end interaction sequences [9,10]. Benchmark environments such as WebArena further show that autonomous agents can complete multi-step browser tasks involving navigation, form input, search, and information retrieval [11,12]. Autonomous web agents have also been evaluated in software-testing scenarios, indicating that language models can participate in test planning, action execution, and result interpretation [13,14]. Despite this progress, reliable interaction-level testing remains challenging. Dynamic interfaces frequently contain asynchronous components, hidden elements, delayed network responses, modal dialogs, dynamically generated identifiers, and controls whose visible labels change according to application state [15,16]. A generated script may therefore succeed during one execution but fail during another because an element appears later than expected or because the underlying DOM representation has changed. Selector robustness is another important concern. Absolute XPath expressions, long CSS paths, dynamically generated class names, and index-based selectors may work temporarily but are sensitive to even minor interface modifications. Semantic selectors based on roles, accessible

names, stable attributes, and surrounding context are generally more resistant to these changes, yet generated tests do not always select them appropriately [17,18]. Test correctness also depends on the quality of the test oracle rather than successful action execution alone. A script that clicks the expected button and reaches a new page cannot necessarily determine whether the underlying function is correct [19,20]. Effective interaction tests should verify page transitions, content changes, database-related updates visible through the interface, validation messages, error states, and other observable effects associated with the user requirement [21,22]. Weak assertions, such as checking only whether a page loads or an element exists, may allow functional defects to remain undetected. This problem becomes more important in end-to-end testing because a single user path can involve several application states, asynchronous operations, and intermediate responses [23,24]. Existing studies often evaluate generated tests using a limited set of applications or a single execution of each test case. Consequently, repeated-run stability, selector degradation after interface updates, oracle quality, repair effectiveness, and long-term maintenance effort remain comparatively underexplored.

This study presents TestNav-Agent, an interaction-level framework for generating and maintaining automated test scripts for web and mobile applications. The proposed method represents interface states using complementary information from DOM trees, accessibility labels, screenshots, and recorded event traces, allowing the model to interpret both structural and visual characteristics of the current screen. Functional requirements are transformed into executable user paths that describe required actions, expected transitions, and verification points. Based on these paths, TestNav-Agent generates test scripts for Playwright, Selenium, or Appium and introduces assertions for page transitions, data changes, validation feedback, successful operations, and expected error conditions. A repair mechanism further analyzes execution failures, replaces unstable selectors with more reliable alternatives, and introduces suitable waiting strategies when failures are associated with asynchronous loading or delayed interface updates. The study evaluates the framework across web and mobile applications through repeated executions and interface-change scenarios. The evaluation focuses on critical-path coverage, execution stability, flaky-test frequency, defect-detection capability, selector robustness, repair success, and maintenance effort. By integrating test generation, state-aware verification, repeated-run evaluation, and automatic repair within a unified workflow, this work aims to move language-model-based testing beyond one-time script generation toward more reliable and maintainable interaction testing. The results are expected to clarify whether multimodal interface understanding and adaptive repair can reduce the gap between automatically generated tests and the stability required for practical regression-testing environments.

2. Materials and Methods

2.1. Sample and Study Setting

The study used 52 real web and mobile interfaces from e-commerce, online education, healthcare appointment, and financial service applications. An interface was included when it supported at least one complete user task in a test environment. The dataset contained 68,000 recorded user actions, 9,400 screen states, 1,260 past bug reports, and 740 manually written regression tests. Test accounts contained artificial records only. Traces with personal data, missing steps, or service failures that could not be repeated were removed. Each sample was a requirement–interface task pair. A task represented a complete user flow, such as login, product search, appointment booking, payment confirmation, or form submission.

2.2. Experimental Design and Control Conditions

TestNav-Agent was compared with a single-agent script generator. TestNav-Agent used separate agents for screen analysis, user-path planning, script writing, result checking, and test repair. The control model received the same task requirements and interface data, but wrote the test script in one step. Both systems used the same language model, model version, prompt length, test data, and execution environment. For each task, the two scripts ran in the same browser version or mobile-emulator image. No script was changed by a human before the first run. Each script ran 100 times to measure execution stability. This design allowed a direct comparison between the multi-agent process and direct script generation.

2.3. Measurement Procedures and Quality Control

Screen information was collected from DOM trees, accessibility labels, screenshots, visible text, and user-event records. The expected result for each task was defined from the written requirement and the related manual regression test. The result checker examined page changes, displayed messages, form values, saved records, and error states. Each run used a new browser profile or a clean mobile-emulator state. Test data were reset before execution, and the network setting was kept the same. Failures were marked as functional failures, selector failures, timing failures, or environment failures. Environment failures were recorded separately and rerun after the test service recovered. Two test engineers checked selected execution records independently. Any disagreement was resolved by reviewing the original requirement and screen record.

2.4. Data Processing and Model Formulation

User actions were arranged by time and linked to the related screen state. Repeated events were removed before analysis. Temporary IDs, timestamps, and page values that changed between runs were masked before selector generation. Critical-path coverage was calculated as:

$$CPC = \frac{1}{N_{path}} \sum_{i=1}^{N_{path}} I(C_i=1)$$

where N_{path} is the number of critical user paths, and $C_i=1$ means that script i completed the required path and passed the result checks. The flaky failure rate was calculated as:

$$\text{FFR} = \frac{1}{N_{\text{test}}R} \sum_{i=1}^{N_{\text{test}}} \sum_{r=1}^R I(F_{i,r}=1)$$

where N_{test} is the number of generated scripts, $R=100$ is the number of repeated runs, and $F_{i,r}=1$ means that script i failed inconsistently in run r . A lower FFR value indicates more stable test execution.

2.5. Outcome Measures and Statistical Analysis

The study measured critical-path coverage, flaky failure rate, detected past bugs, newly found interface defects, selector failures, and maintenance time after an interface change. A past bug was counted when a generated script reproduced a recorded regression issue. A new defect was counted only after staging review confirmed that it was caused by the application. Maintenance time was measured from the interface change to the first successful run of the updated script. TestNav-Agent and the control model were compared for each requirement–interface task pair. The results were reported as mean values, percentage changes, and 95% bootstrap confidence intervals from 1,000 resamples.

3. Results and Discussion

3.1. Critical-Path Coverage and Script Execution

TestNav-Agent completed 76.8% of the critical user paths. The direct script-generation model completed 54.2%. The difference was 22.6 percentage points. The main reason was the task-planning step. TestNav-Agent used the current screen state and the task requirement to choose the next action. The direct model wrote the script from the requirement in one step. TestNav-Agent completed more multi-step tasks, including login, search, booking, payment, and form submission [25,26]. Fig. 1 shows a related web-testing process that extracts page information before script execution. TestNav-Agent also used accessibility labels, screenshots, event records, and result checks. These inputs helped when DOM data did not fully describe the user task.

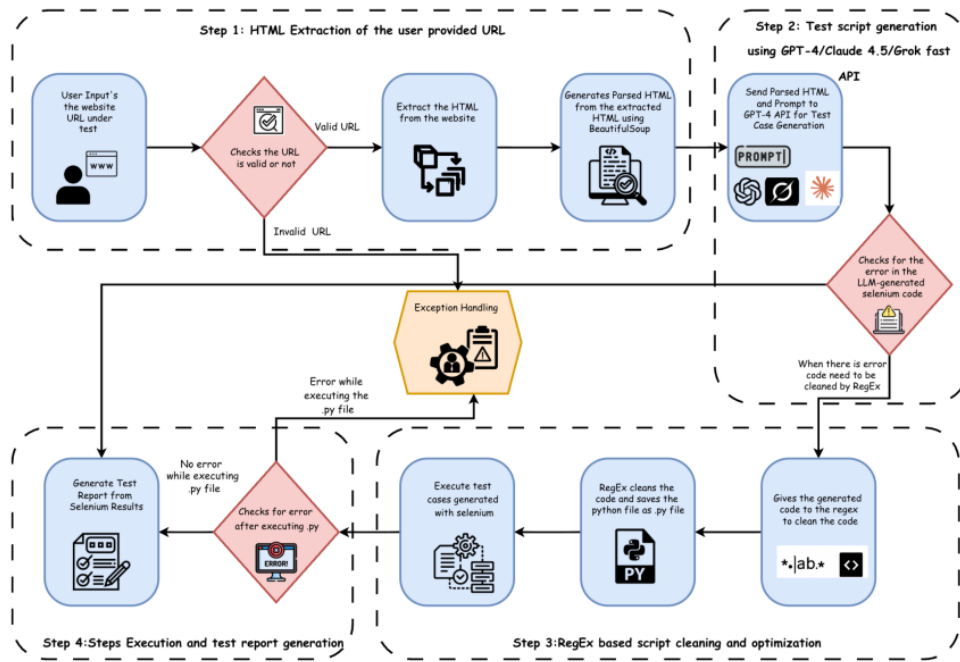


Fig. 1. Steps for collecting page data, creating test scripts, and running web UI tests.

3.2. Repeated-Run Stability and Selector Repair

The flaky failure rate was 4.1% for TestNav-Agent and 13.5% for the direct model across 100 runs. This was a 69.6% relative decrease. The repair agent changed unstable selectors and added wait conditions when page elements appeared late or changed after data updates. In dynamic interfaces, element-location failures decreased by 62.4%. Many test-generation methods report whether a script passes once. This result shows that a single successful run is not enough to judge script quality [27,28]. Repeated execution is needed because timing, page loading, and changing elements can cause later failures.

3.3. Bug Detection and Result Checks

TestNav-Agent detected 37 recorded regression bugs and 11 new interface defects during staging tests. The result checker did not only confirm that an element could be clicked. It also checked page changes, saved values, validation messages, and error states. Fig. 2 shows how GUI actions can be linked to expected screen states during test generation. This type of checking can find cases in which a page opens but shows the wrong value or moves to the wrong state. Scripts that only check page loading or element presence may miss these problems. The added checks were useful for healthcare booking and financial service tasks, where an incorrect page state may affect an appointment or transaction.

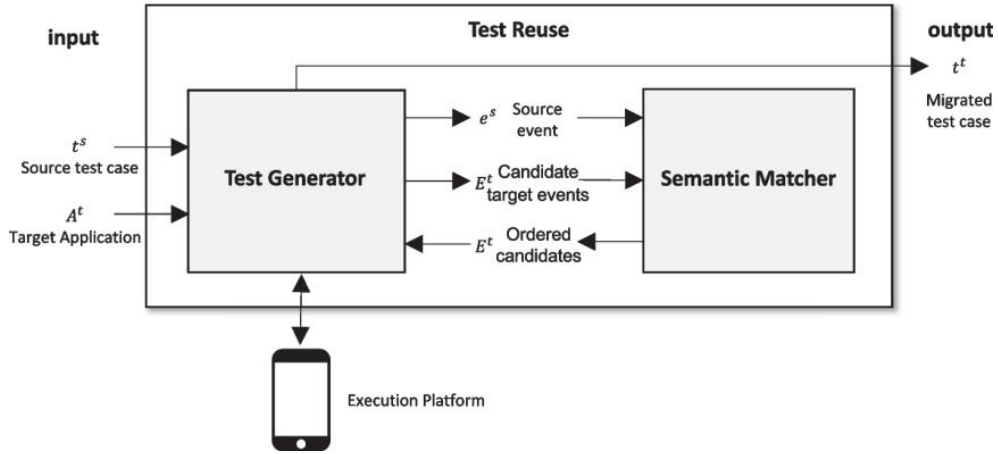


Fig. 2. Process for reusing GUI tests by matching similar actions and creating new tests.

3.4. Maintenance Time and Study Limits

After interface changes, the mean maintenance time decreased from 19.6 min to 8.3 min per test case. This was a 57.7% reduction. TestNav-Agent kept the original task goal and searched for a new element when the old selector no longer worked. Earlier GUI test-reuse methods mainly transfer a test from one application to another application with similar functions. TestNav-Agent repaired a test after a new version of the same interface was released. The study included 52 interfaces in a controlled test setting. The results may differ for desktop software, unstable networks, large continuous-integration systems, or device-specific mobile functions [29,30]. Later studies should test more applications and longer update periods.

4. Conclusion

This study tested TestNav-Agent for generating web and mobile UI test scripts. The system used screen analysis, task planning, script writing, result checks, and script repair. It covered 76.8% of critical user paths, while the direct model covered 54.2%. The flaky failure rate decreased from 13.5% to 4.1%. Selector-location failures in dynamic pages decreased by 62.4%. TestNav-Agent also found 37 recorded regression bugs and 11 new interface defects. The mean maintenance time after an interface change decreased from 19.6 min to 8.3 min per test case. The findings show that UI test generation should not be judged by one successful run. It should also be judged by path coverage, repeated-run stability, result checks, and maintenance time. The method can support regression testing in web and mobile development. This study included 52 interfaces in a controlled test setting. Later work should test more applications, devices, network conditions, and continuous-integration environments.

References

1. Bao, Y., & Wang, H. (2026). Orion: A High-Throughput, Fault-Tolerant Event Routing Architecture for Hyperscale Data Streams. *Fault-Tolerant Event Routing Architecture for Hyperscale Data Streams* (January 01, 2026).
2. Yaraghi, A. S., Holden, D., Kahani, N., & Briand, L. (2025). Automated test case repair using language models. *IEEE Transactions on Software Engineering*, 51(4), 1104-1133.
3. Wang, S., Feng, Y., & Fang, X. (2026, May). A Large Language Model-Enabled Multi-Agent Collaboration Method for Complex Task Solving. In *2026 6th International Symposium on Computer Technology and Information Science (ISCTIS)* (pp. 253-256). IEEE.
4. Zhang, Z., Wang, J., Li, Z., Wang, Y., & Zheng, J. (2025). Annocoder: A mti-agent-based code generation and optimization model. *Symmetry*, 17(7), 1087.
5. Hasan, M. M., Li, H., Fallahzadeh, E., Rajbahadur, G. K., Adams, B., & Hassan, A. E. (2026). An empirical study of testing practices in open source AI agent frameworks and agentic applications. *Empirical Software Engineering*, 31(5), 124.
6. Yang, Q., & Du, Y. (2026). Predicting Rollback Risks and Controlling Gradual Rollout Traffic for Online Ranking Feature Deployments.
7. Zhao, Z., & Welsch, R. E. (2026). Point-in-Time Financial RAG with Frozen LLMs and Market-Feedback Adaptive Retrieval. *arXiv preprint arXiv:2605.31201*.
8. Yuan, Y., Huang, J., Yin, J., & Xu, T. (2026). Confidence Calibration and Semantic Error Analysis for Ambiguous Coreference Resolution in User-generated Social Media Text. Available at SSRN 7393238.
9. Trong, M. P., Son, N. T., Giang, V. T., Viet, B. H., Hai, L. N., & Tung, D. T. (2026). A process-centric review of large language models in graphical user interface testing: architectures, lifecycle impact, and challenges. *PeerJ Computer Science*, 12, e3695.
10. Zhang, Z. (2026). A Study on the Impact of Cost Presentation on Decision-making Bias and Cancellation Behavior in Online Subscriptions. Available at SSRN 7349839.
11. Qi, C., & Qiao, X. (2026). Building and Operating a Large Scale Multi-Agent System: A Case Study from Industry. Available at SSRN 6795198.
12. Du, Y., Liu, Q., Dong, Y., & Chen, Z. (2026). Cross-Domain Transfer and Few-Shot Recognition of Defect Images in Complex Industrial Settings.
13. Fischer, S., & Kloihofer, W. (2026). LLM Agents for Autonomous System Testing: A Semi-structured Literature Review. In *International Conference on Software Quality* (pp. 147-167). Springer, Cham.
14. Xiong, W., Guo, Y., & Zeng, Y. (2026). A Study on the Energy Efficiency of MEP Systems and Coordinated Dispatch Mechanisms for Multi-energy Systems in High-density Urban Buildings. Available at SSRN 7120518.
15. Hong, Z., Liang, S., Xu, T., & Chen, H. (2026). A Study on Failover Verification and Recovery Objective Prediction for Cross-Region Cloud Services.

16. Huang, J., Xu, T., Yin, J., & Yang, J. (2026). Data Quality Monitoring and Intelligent Scheduling Optimization in Financial Data Workflow Automation. Available at SSRN 7179338.
17. Sulpovar, M., Konsynski, B. R., Kanchwala, Q., & Goodhart, G. (2026). ContextNest: Verifiable Context Governance for Autonomous AI Agent. arXiv preprint arXiv:2607.02116.
18. Jiao, Y., Shi, T., Zhao, B., & Wang, A. (2026). Retrieval-Guided Structured Reasoning and Interpretable Representation Learning for Large-Scale Video–Language Models.
19. Liang, S., Du, Y., Chen, W., & Liu, Z. (2026). Order Allocation and Emergency Transportation Decisions Amid Multi-Source Procurement Disruptions.
20. Bao, Y., & Wang, H. (2026). Large-scale Metrics Migration: A Systematic Approach to Rebuilding Production Measurement Infrastructure. Available at SSRN 7185660.
21. Muhamad, H., & Taha, M. (2026). Optimizing the Performance of Web Applications in Dynamic Network Environment: A Systematic and Comprehensive Analytical Survey. *Concurrency and Computation: Practice and Experience*, 38(6), e70641.
22. Yuan, Y., Yin, J., Huang, J., & Xu, T. (2026). Weakly Supervised Anomaly Detection and Privacy Risk Scoring in Community Message Threads.
23. Liang, S., Hsu, K. H., & Liu, Z. (2026). Rolling Scheduling and Capacity Balancing for Server Assembly Under Engineering Change Disturbances.
24. Mabotha, E., Mabunda, N. E., & Ali, A. (2025). A dockerized approach to dynamic endpoint management for RESTful application programming interfaces in internet of things ecosystems. *Sensors*, 25(10), 2993.
25. Yao, G., Zheng, J., Wang, Z., Zhang, W., Han, R., Zhao, C., ... & Liu, R. (2026, March). V-pruner: A fast and globally-informed token pruning framework for vision transformer. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 40, No. 40, pp. 34396-34404).
26. Gu, X. (2026). Identifying Causal Effects and Analyzing Heterogeneity of User Growth Interventions on Digital Platforms: Evidence from Large-Scale Behavioral Data. Available at SSRN 6809181.
27. Aggarwal, P., Neubig, G., & Welleck, S. (2026, April). Gym-anything: Turn any software into an agent environment. In *COLM 2026 The 2nd Workshop on Lifelong Agents: Learning, Aligning, and Evolving*.
28. Zhao, J., Fan, J., & Li, L. (2026). A Study on an Explainable Causal-Enhanced LLM Agent for Predicting the Forming Quality of Automotive Component Materials.
29. Cajas Ordóñez, S. A., Samanta, J., Suárez-Cetrulo, A. L., & Carbajo, R. S. (2025). Intelligent edge computing and machine learning: A survey of optimization and applications. *Future Internet*, 17(9), 417.
30. You, S. (2026). Verifiable Audit Mechanisms in AI Compliance Automation: Scalability. Available at SSRN 6547458.