GenIA-E2ETest: An AI-Based Platform for End-to-End Test Automation

Elvis Júnior
Universidade Federal Fluminense
Niterói, RJ, Brasil
elvisjunior@id.uff.br

Pedro Amaro
Universidade Federal Fluminense
Niterói, RJ, Brasil
ph_amaro@id.uff.br

Allber Ferreira
Universidade Federal Fluminense
Niterói, RJ, Brasil
alferreira@id.uff.br

Vânia de Oliveira Neves
Universidade Federal Fluminense
Niterói, RJ, Brasil
vania@ic.uff.br

Abstract

End-to-end (E2E) testing is essential for validating complete web application workflows, but creating and maintaining automated E2E tests remains costly and technically demanding. This paper presents the GenIA-E2ETest Platform, an AI-based web application that turns a previously proposed generation approach into a usable tool for E2E test automation. The platform implements a multi-stage pipeline that combines Large Language Models (LLMs), web crawling, DOM analysis, and script orchestration to support the creation, execution, and revision of test suites from natural-language requirements. Compared with our earlier work, which introduced and preliminarily validated the core generation method, this paper focuses on the full platform implementation, including the web interface, automated script execution, log inspection, and support for multiple automation frameworks such as Playwright, Cypress, JUnit, Pytest, and Robot Framework. The system also accepts different input formats, including natural-language test cases and Gherkin user stories, and provides configurable prompts, LLM providers, models, and execution flows. We describe the platform architecture, its configurable variables, and the role of each pipeline stage in supporting the end-to-end workflow. Overall, GenIA-E2ETest Platform is designed to reduce the manual effort required for E2E automation while providing a more accessible and flexible environment for researchers and practitioners. The tool is distributed as open-source software under the MIT license.
Demo video: https://doi.org/10.6084/m9.figshare.32559900

Keywords

End-to-End Testing, Generative AI, Software Testing Automation, E2E

1 Introduction

Software testing is essential for ensuring the quality of modern applications, especially in web systems where the graphical interface, business logic, and external integrations must operate in a coordinated manner. Among the different levels of testing, *End-to-End* (E2E) testing occupies a strategic position because it validates complete application flows from the end-to-end users perspective, rather than checking isolated units in abstraction [7].

Despite its importance, E2E test automation is traditionally costly. Implementing and maintaining these tests requires detailed mapping of interface elements, technical mastery of automation frameworks, and continuous adaptation as the application evolves [10]. In addition, teams often need to translate requirements written in natural language, user stories, or test scenarios into executable scripts, which introduces a second barrier: the transformation from specification to automation.

The advancement of Large Language Models (LLMs), such as ChatGPT, Gemini, and Claude, has opened up new opportunities for automating test generation from natural language descriptions. However, most applications of these models in the context of software testing still focus on low-level tasks, such as the generation of unit tests [4, 14, 17]. Although relevant, unit tests alone are not sufficient to ensure the overall quality of complex systems. Wang et al. [14] highlight the need to explore higher levels of testing and to develop strategies capable of transforming textual artifacts, such as user stories and test scenarios, into executable automated tests.

This paper is built on top of the previous GenIA-E2ETest research [8], which proposed and empirically validated the core method for generating E2E test scripts from natural language test cases. That earlier study focused on the methodology and showed promising results, including 77% for both element metrics - precision of element generation and element recall -, 82% for precision of execution, 85% for execution recall, and a minimal 10% manual modification rate. The present work extends that line of research by turning the method into a complete platform, with a RESTful backend API and a browser-based interface exposed to the user.

Commercial AI-driven E2E testing products, such as Testim[1], Functionize[2], and testRigor[3], address part of this problem, but they are proprietary, often expensive, and usually opaque about how their generation logic is implemented. GenIA-E2ETest follows a different path: it is open, configurable, and designed around explicit stages that can be inspected through the interface. This makes the platform more suitable for research, experimentation, and reproducible evaluation.

The platform is intended for QA teams, developers, and researchers who need to accelerate E2E automation without handling

[1] https://www.testim.io/
[2] https://www.functionize.com/
[3] https://testrigor.com/

selectors and browser-level details manually. It is suited for applications with stable structures and conventional navigation flows, but its architecture also makes it possible to extend the generation pipeline to additional frameworks and models. The remainder of the paper first reviews the minimum theoretical background needed to understand the platform, then presents the architecture and configuration variables, and finally details the pipeline and the user-facing views.

2 GenIA-E2ETest Platform

GenIA-E2ETest is the platform implementation of a method that was previously proposed and empirically evaluated in an earlier study [8]. While that work established the feasibility of the approach, the current version turns it into a usable web tool for end-to-end testing. The goal is straightforward: let a user open the system in a browser, provide a test case or user story, select the execution context, and execute an executable E2E test with only a few interactions. The result is not just a script artifact but a complete workflow in which the test is structured, generated, executed, and reviewed.

From the user perspective, the platform reduces E2E automation to a guided flow. The user configures the required variables, triggers generation, and then follows the pipeline through the interface while the backend performs crawling, DOM processing, script generation, validation, execution, and artifact persistence.

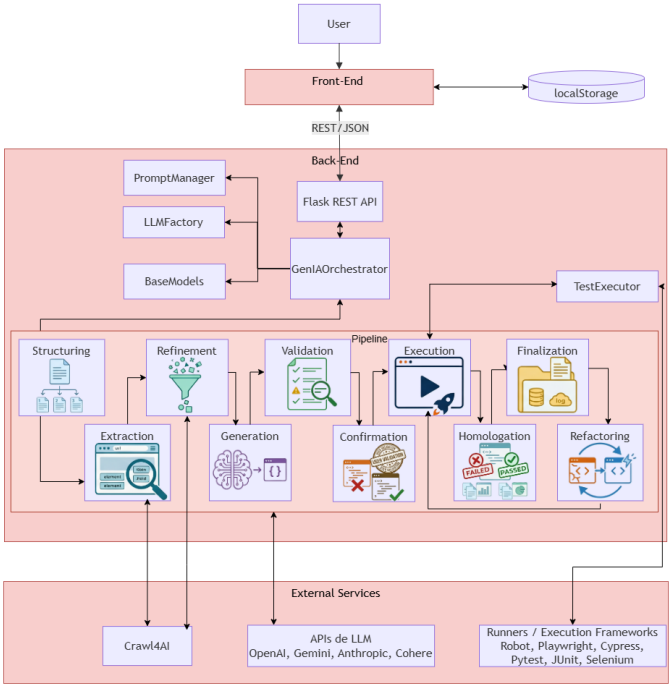

Figure 1: Overview of the GenIA-E2ETest Architecture

2.1 Platform Architecture

The platform is architected as a layered web application, where the user interface serves as the entry point for experiment configuration, while the back-end orchestrates the test-generation pipeline, as illustrated in Figure 1. The solution was designed so that business logic is decoupled into well-defined stages, each producing intermediate artifacts that feed the next phase of the workflow. This decomposition improves traceability, manual inspection, and experimental reproducibility.

The core layer of the solution is the back-end, responsible for coordinating the full transformation cycle from textual scenario to executable test. From the initial request, the back-end creates a pipeline session, maintains execution state in memory, and triggers the stages of structuring, extraction, refinement, generation, validation, confirmation, execution, homologation, finalization, and refactoring. This layer also concentrates LLM integration, stage-specific prompt loading, event and log publication, and the handling of intermediate responses. In practice, it materializes the platform's business logic and connects the remaining components.

To support this workflow, the back-end uses domain models that act as internal contracts between stages, allowing the output of one stage to be consumed by the next without structural loss. In parallel, the architecture relies on an isolated executor for generated scripts, ensuring that test execution occurs in a temporary workspace and under commands compatible with the selected framework.

The web interface occupies the presentation layer and is responsible for collecting user input, displaying pipeline progress, and rendering the artifacts produced at each stage. The front-end does not implement the core generation logic; instead, it acts as an API consumer and an observability layer for the pipeline. Through it, users can configure the project, framework, programming language, LLM provider, model, temperature, and custom prompts, as shown in Section 2.2. Communication with the back-end occurs through REST requests for triggering operations and a real-time event channel for monitoring logs and stage transitions.

2.2 Configuration Variables

The platform configuration is defined by a set of variables that determine how each test run is prepared and executed. These variables are not mere auxiliary settings; they shape the input format, the target project, the automation framework, the language of the generated artifacts, the LLM behavior, and the prompts used at each stage of the pipeline. Together, they make the workflow adaptable to different testing scenarios while keeping the process organized and traceable.

2.2.1 Test Case or User Story Input. The first variable defines the source of the request, which can be either a free-form test case or a user story. The platform accepts these inputs as text files and forwards them to the structuring stage, where they are converted into a normalized representation. This allows the same pipeline to support both direct test descriptions and BDD-style scenarios.

2.2.2 Project. The project variable identifies the application under test and links the execution to the corresponding repository data. In practice, it ensures that generated artifacts remain associated with the correct project context, making it easier to organize runs, inspect results, and preserve traceability across executions.

2.2.3 Framework and Language Selection. The user also selects the target automation framework and the programming language for the generated script. These choices define the syntax and execution style of the output, allowing the same scenario to be adapted to different testing ecosystems without changing the pipeline logic.

2.2.4 LLM Provider, Model, and Temperature. Another group of variables controls the LLM provider, the specific model, and the temperature value. These parameters influence how the platform interprets the input and generates intermediate or final artifacts. Exposing them through the interface gives the user direct control over the trade-off between determinism, creativity, and model capability.

2.2.5 Prompt Bundle and Stage Overrides. Finally, the platform allows the selection of a prompt bundle and, when needed, stage-specific overrides. The bundle defines the default behavior of the pipeline, while overrides let the user refine the instructions used in particular stages. This mechanism makes the platform more flexible without requiring changes to the core implementation.

2.3 Pipeline

After the configuration variables are set and the user clicks the button to generate the test, the platform starts the pipeline. From that moment on, the orchestrator turns the selected configuration into a sequence of staged artifacts, each one consumed by the next stage and exposed to the user through the interface, logs, and history records. Figure 1 summarizes the end-to-end flow, and Table 1 provides a compact view of the inputs and outputs produced by each stage.

2.3.1 Structuring. Structuring is the first computational stage after the user starts the pipeline. Its purpose is to transform the raw natural-language requirement into a normalized test representation that serves as the seed for all subsequent stages. In this context, a *module* represents a logical unit of the test scenario, usually corresponding to a specific page (URL) of the application. Each module groups the actions that belong to the same part of the workflow, helping preserve the relationship between user intent, page transition, and execution context. The *execution_steps* field stores the ordered actions that should be performed within that module, such as filling inputs, clicking buttons, or validating messages. The *extracted_data* field stores the HTML evidence required to execute each step and is populated in Section 2.3.2. Together, these elements provide a structured and traceable representation of the original requirement. The underlying logic follows the earlier study [8]; however, in the current platform, the structured JSON is directly exposed to the user, who can copy or download it, while the graph view provides an immediate visualization of the navigation flow before the rest of the pipeline begins, as illustrated in Figure 2. In this graph representation, each execution step extracted from the test-case JSON is mapped to a node, and the nodes are connected in the same order defined by the scenario, forming the test execution flow (e.g., *Launch browser* → *Navigate to URL* → *Click on the "Signup / Login" button*); in addition, each HTML element identified for a given step is represented as a linked node attached to the corresponding execution_step node, exposing technical details such as *type*, *identifier_type*, and *identifier_tracking*, which allows the user to inspect both the logical flow of the test and the interface evidence supporting each action.

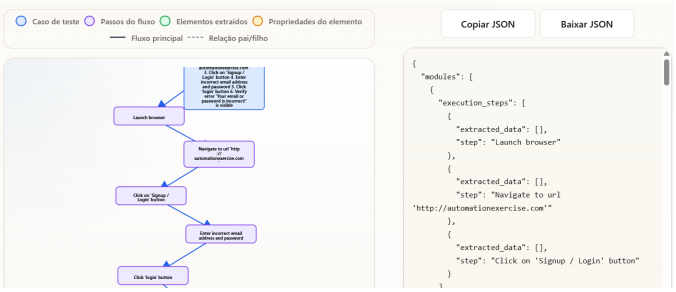

Figure 2: Structuring

2.3.2 Extraction. Extraction processes each structured module by crawling its associated target URL with Crawl4AI [5] to identify the interface elements relevant to each execution step. In practice, this stage combines the structured test context with DOM and HTML evidence to produce a machine-readable representation of the candidate elements, as summarized in Table 1. The extracted result is exposed through JSON and graph views, allowing the user to inspect, copy, or download the generated evidence directly from the interface, as shown in Figure 3.

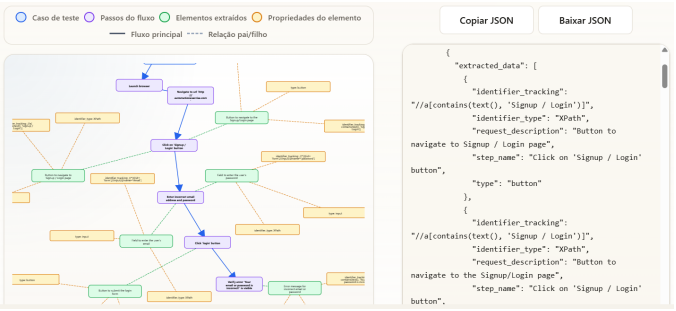

Figure 3: Extraction

2.3.3 Refinement. Refinement consumes the evidence produced during Extraction and transforms it into a cleaner and more stable representation for test generation, as summarized in Table 1. Its main purpose is to eliminate redundant or noisy information, normalize element descriptions, and consolidate the most reliable selectors and page details needed for the final script. As in the earlier version described in [8], the platform uses Crawl4AI to revisit the page data collected during the extraction stage and validate the interface evidence before code generation. The refined JSON is also made available to the user for copying, downloading, and graph-based inspection, same as illustrated in Figure 3.

2.3.4 Generation. Generation takes the refined test case as input and produces the executable script described in Table 1. Because the stage is framework aware, it can align the generated code with the syntax and idioms expected by the target ecosystem; in the

Table 1: Inputs and outputs across the 10 pipeline stages

Stage	Input(s)	Output(s)	Output Description
Structuring	Natural language test case / user story, LLM config, structuring prompt	Structured test case	Converts the free-form requirement into a normalized test model with modules, pages, URLs, and execution steps that becomes the pipeline seed.
Extraction	Structured test case, LLM config, extraction prompt	Extracted HTML elements	Captures the page evidence for each planned step.
Refinement	Structured test case, extracted HTML elements, LLM config, refinement prompt	Refined HTML elements	Removes noise, normalizes candidates, and keeps only the DOM evidence that is relevant for generation and validation.
Generation	Refined test case, LLM config, generation prompt	Generated test script	Produces the executable script that matches the selected framework and encodes the refined test logic.
Validation	Generated test script, refined test case, LLM config, validation prompt	Validation report	Detects variables and consistency issues and turns them into editable validation fields before execution.
Confirmation	Generated test script, user-adjusted validation report, LLM config, confirmation prompt	Confirmed test script	Finalizes the script after user review or reuses the validated version when no manual changes are needed.
Execution	Confirmed test script, execution config	Execution report	Stores the runtime result together with logs, screenshots, traces, and evidence that document the actual run.
Homologation	Validation report, confirmed test script, execution report, LLM config, homologation prompt	Homologation report	Compares the intended flow with the observed run and summarizes matches, deviations, and likely failure causes.
Finalization	Execution report, homologation report, pipeline logs, LLM config, finalization prompt	Finalization report	Consolidates the session into a reusable record with outcomes, events, and logs for history and dashboard views.
Refactoring	Generated test script, execution report, homologation report, finalization report, LLM config, refactoring prompt	Refactoring report	Suggests a revised version of the script or targeted improvements for a future run when the current attempt fails or degrades.

platform, the generated script appears in the result view, where it can be copied and downloaded. Figure 4 presents an example of the generation stage output.

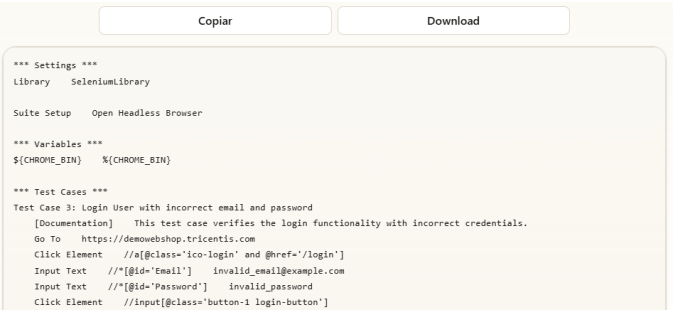

Figure 4: Generation

2.3.5 Validation. Validation compares the generated script with the refined test case and produces the report described in Table 1. This is the first stage that explicitly presents potential user adjustments before execution. If the model detects variables, credentials, or parameterized values, those appear as validation items that the user can confirm or change. The frontend highlights these fields so the user can decide whether the pipeline should continue unchanged or be adjusted before the script is executed. Figure 5 illustrates the validation interface.

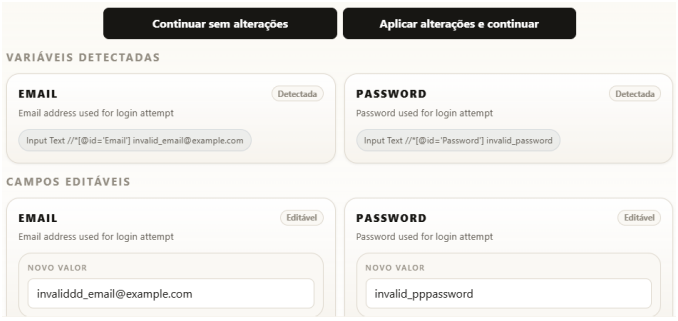

Figure 5: Validation

2.3.6 Confirmation. Confirmation resolves whether the validated script should be reused or rewritten. If the user accepts the generation as-is, the platform reuses the script. If the user edits values or adds notes, the stage can call the LLM again to produce a confirmed version that incorporates those changes. This makes confirmation the bridge between machine-generated code and human approval. Figure 6 presents the confirmation process in the platform.

Figure 6: Confirmation

2.3.7 Execution. Execution receives the confirmed script and the execution context selected earlier. The executor writes the script to a temporary isolated workspace, selects the correct command for the chosen framework, and runs the artifact in a controlled environment. The output is the structured execution report described in Table 1, which includes runtime status and the evidence needed for later review. In the frontend, this result becomes a report view plus downloadable artifacts, so the user can inspect both the pass/fail outcome and the raw evidence.

2.3.8 Homologation. Homologation compares the expected behavior from the refined test case and validation data with the actual behavior observed in execution. Its output is the report summarized in Table 1, which helps interpret whether the observed run matches

the intended business flow. This stage is where the platform summarizes success, deviations, and potential causes of failure. Figure 7 presents an example of the homologation report.

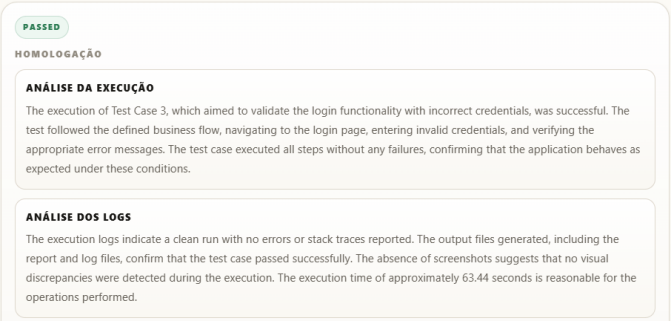

Figure 7: Homologation

2.3.9 Finalization. Finalization aggregates the homologation report, the logs, and the timeline into the report summarized in Table 1. Figure 8 shows the final packaged report generated by the pipeline.

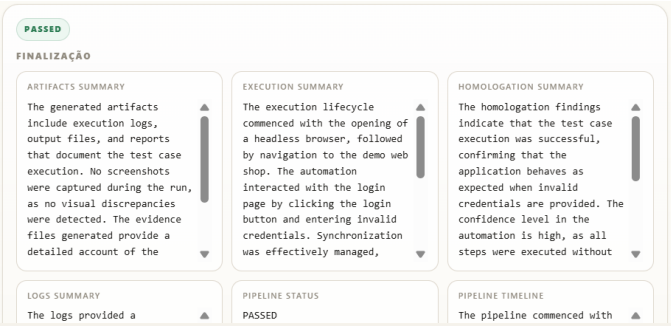

Figure 8: Finalization

2.3.10 Refactoring. Refactoring uses the execution result, the homologation report, the final report, and the test script as input. Its goal is to suggest a revised script when execution fails or to suggest points for improvement even if it has executed successfully. The output is the refactoring report described in Table 1, which can be applied in a new run and supports a lightweight self-healing loop without hiding the fact that human review may still be needed. Figure 9 illustrates an example of a refactoring suggestion generated by the platform.

The validation pauses the pipeline and stores the session state so that the user can inspect the extracted variables and either continue without changes or inject edits. The continuation endpoint can then resume from confirmation, execution, or later stages, which means the pipeline supports both fully automatic and semi-automatic usage modes.

3 Related Work

Research on end-to-end (E2E) testing for web applications has explored several strategies for generating, executing, and maintaining

Figure 9: Refactoring

automated test suites. A recent systematic mapping study identified three major challenges in this domain: test-case generation, test execution efficiency, and regression testing support [2]. Within this landscape, existing solutions can be broadly grouped into model-based approaches, autonomous exploration methods, LLM-based techniques, and commercial AI-driven platforms.

The first group addresses E2E test generation from a different perspective, as it does not target the translation of predefined requirements into E2E scripts. Model-based methods generate tests from navigation models, source-code artifacts, or usage traces [6, 9, 12, 13, 16], whereas autonomous exploration and reinforcement-learning techniques, such as WebExplor [18] and WebQT [3], discover interaction paths by exploring the interface directly. Although useful, these approaches either depend on an intermediate abstraction layer that must be maintained alongside the application or focus on discovering behavior rather than reproducing a specified scenario. the GenIA-E2ETest Platform instead assumes that the intended behavior is already described in natural language, test cases, or BDD scenarios, and focuses on converting those specifications into executable, inspectable workflows, without requiring explicit model construction or dynamic exploration.

The work most closely related to ours applies LLMs to E2E-oriented tasks such as natural-language-to-test conversion, GUI automation, and element identification [1, 4, 11, 14]. VETL [15], for example, uses large vision-language models to generate context-aware textual inputs from visual interface representations, addressing some limitations of prior reinforcement-learning approaches. Ayli et al. [1] explore LLM-assisted element identification through semantic matching between short textual descriptions and DOM elements, enabling non-technical users to issue simplified commands. While these methods reduce the technical burden of test creation, they still tend to address isolated capabilities, such as producing a single input or locating one element, rather than orchestrating a complete E2E workflow from a requirement to executable and inspectable artifact. The GenIA-E2ETest Platform extends this line of work by integrating requirement interpretation, DOM extraction, script generation, validation, execution, and artifact persistence into a single staged pipeline.

Commercial AI-based E2E platforms such as testRigor, Testim, and Functionize illustrate the growing demand for low-code and no-code automation solutions. Testim and Functionize primarily

rely on record-and-playback strategies, which are effective for reproducing existing workflows but less flexible when tests need to be generated from high-level specifications or adapted to evolving requirements. testRigor allows testers to write test cases in plain English, but it remains a proprietary tool, with limited transparency and restricted integration with open-source frameworks such as Robot Framework or Selenium. In addition, its input and workflow assumptions may limit its adoption in multilingual or heterogeneous development environments. GenIA-E2ETest Platform addresses these gaps by providing an open, configurable, and browser-accessible environment that supports multiple input formats, multiple automation frameworks, and guided human review. It is worth clarifying how the platform positions itself with respect to established execution tools. GenIA-E2ETest Platform does not replace frameworks such as Selenium or Robot Framework; rather, it works alongside them, operating at the specification-to-script layer. The platform automates the generation of the test scripts that these frameworks then execute, allowing practitioners to continue using their existing execution stack while delegating the manual and error-prone task of turning requirements into runnable scripts to an LLM-assisted pipeline.

Overall, the related work suggests that there is still a gap between approaches that demonstrate generation or exploration capabilities and platforms that expose a complete, inspectable, and configurable workflow for users. GenIA-E2ETest Platform addresses this gap by operationalizing a previously validated generation method [8] as a complete E2E testing platform, rather than merely as a generation algorithm.

4 Conclusion

This paper presented GenIA-E2ETest as a web-based platform that operationalizes a previously validated method [8] into a browser-accessible workflow for end-to-end test automation. Rather than remaining as a standalone generation algorithm, the approach is now implemented as an interactive system that supports project-aware execution, per-stage configuration, multi-framework generation, multiple input formats such as natural-language test cases and Gherkin user stories, and guided human review through logs, histories, graphs, and downloadable artifacts.

The current platform also extends the original method by integrating web crawling, DOM and HTML analysis, automated script execution, and log inspection into a single pipeline. While these capabilities make the workflow more practical for real usage scenarios, the platform should still be considered an evolving system rather than a final solution. The empirical results reported in our previous work refer to the underlying method, not to the current platform version.

Several limitations remain relevant for end-to-end automation in realistic web applications. The case study reported in this paper intentionally uses a single-page test to make the pipeline stages easy to follow; it should therefore be read as an illustrative walkthrough rather than as evidence that the platform is restricted to one-page scenarios. Nonetheless, more complex scenarios do pose known challenges, which the empirical evaluation of the underlying method already documented [8]. In that evaluation, the scenario involving navigation across multiple pages was the single case with a clearly lower performance and the highest manual-adjustment effort, precisely because context has to be preserved as the flow transitions from one page to the next. Page transitions therefore remain a central open problem: when a workflow spans several pages, the platform must carry forward the execution context, and any loss of that context tends to degrade both the correctness and the completeness of the generated scripts. A related difficulty concerns fragile locators. Modern web frameworks such as React and Angular re-render the DOM dynamically and often generate volatile attributes, so selectors that are valid at generation time may no longer match at execution time. These issues, together with dynamically injected elements, external overlays such as ads and pop-ups, and semantic ambiguity in natural-language requirements, become even more pronounced in multi-step workflows and highly dynamic interfaces, which reinforces the need for stronger context modeling, more resilient locator-generation heuristics, and broader empirical validation on multi-page, real-world applications.

Future work will first focus on implementing a multi-agent architecture, in which specialized agents collaborate across requirement interpretation, DOM analysis, script generation, validation, execution analysis, and repair. This evolution is expected to improve coordination across stages and provide a stronger technical basis for a more robust end-to-end evaluation of the final platform. After that, we plan to expand the evaluation across multiple real-world case studies and different LLM configurations, including local models, to analyze hallucination rates, robustness, and behavior across diverse scenarios. We also intend to study execution costs and trade-offs related to model usage, prompting strategies, and framework selection, as well as conduct a broader validation with third-party participants, especially QA professionals and test automation practitioners, to better understand perceived usefulness, usability, and adoption barriers.

Overall, GenIA-E2ETest contributes a flexible and configurable environment for E2E test automation and outlines a clear path toward further architectural refinement and empirical validation. The platform represents a step toward more practical AI-assisted software testing, but its full assessment depends on the completion of the planned multi-agent evolution and on broader future studies.

Artifact Availability

The authors declare that the research artifacts supporting the findings of this study are available at https://doi.org/10.6084/m9.figshare.33049850. Additional resources can also be accessed on GitHub at https://github.com/uffsoftwaretesting/GenIA-E2ETest-Platform.

Acknowledgements

This paper has been supported by CNPq - *National Council for Scientific and Technological Development* (grant 420025/2023-5); FAPERJ-Fundação Carlos Chagas Filho de Amparo à Pesquisa do Estado do Rio de Janeiro (process E-26/204.379/2025); and undergraduate research scholarships from the PIBIC/UFF and PIBINOVA/UFF programs at Universidade Federal Fluminense (UFF).

We used ChatGPT-4o to assist with the organization and revision of the manuscript and speed up the writing of LaTeX code. We carefully examined and often corrected AI suggestions, whereby we took full responsibility for the form and content of the paper.

References

[1] Maroun Ayli, Youssef Bakouny, Nader Jalloul, and Rima Kilany. 2024. Enhancing the Resiliency of Automated Web Tests with Natural Language. In *Proceedings of the 2024 4th International Conference on Artificial Intelligence, Automation and Algorithms*. 63–69.

[2] Sebastian Balsam and Deepti Mishra. 2024. Web application testing—Challenges and opportunities. *Journal of Systems and Software* (2024), 112186.

[3] Xiaoning Chang, Zheheng Liang, Yifei Zhang, Lei Cui, Zhenyue Long, Guoquan Wu, Yu Gao, Wei Chen, Jun Wei, and Tao Huang. 2023. A reinforcement learning approach to generating test cases for web applications. In *2023 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE, 13–23.

[4] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. ChatUnitest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. ACM, 572–576.

[5] Crawl4AI. 2026. Crawl4AI Documentation. https://github.com/unclecode/crawl4ai.

[6] Yuetang Deng, Phyllis Frankl, and Jiong Wang. 2004. Testing web database applications. *ACM SIGSOFT Software Engineering Notes* 29, 5 (2004), 1–10.

[7] Martin Fowler. 2020. Broad Stack Test. https://martinfowler.com/bliki/BroadStackTest.html. Acesso em: 24 abr. 2025.

[8] Elvis Júnior, Alan Valejo, Jorge Valverde-Rebaza, and Vânia de Oliveira Neves. 2025. GenIA-E2ETest: A Generative AI-Based Approach for End-to-End Test Automation. In *Anais do Simpósio Brasileiro de Engenharia de Software (SBES)*. SBC, Recife, PE, Brazil, 282–292. doi:10.5753/sbes.2025.9927

[9] Chaitanya Kallepalli and Jeff Tian. 2001. Measuring and modeling usage and reliability for statistical web testing. *IEEE transactions on software engineering* 27, 11 (2001), 1023–1036.

[10] Maurizio Leotta, Boni García, Filippo Ricca, and Jim Whitehead. 2023. Challenges of End-to-End Testing with Selenium WebDriver and How to Face Them: A Survey. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 339–350. doi:10.1109/ICST57152.2023.00039

[11] Isela Mendoza, Fernando Silva Filho, Gustavo Medeiros, Aline Paes, and Vânia O Neves. 2024. Comparative Analysis of Large Language Model Tools for Automated Test Data Generation from BDD. In *Simpósio Brasileiro de Engenharia de Software (SBES)*. SBC, 280–290.

[12] Sara E Sprenkle, Lori L Pollock, and Lucy M Simko. 2013. Configuring effective navigation models and abstract test cases for web applications by analysing user behaviour. *Software Testing, Verification and Reliability* 23, 6 (2013), 439–464.

[13] Paolo Tonella and Filippo Ricca. 2004. Statistical testing of web applications. *Journal of Software Maintenance and Evolution: Research and Practice* 16, 1-2 (2004), 103–127.

[14] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing with Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering* 50, 4 (2024), 911–936. doi:10.1109/TSE.2024.3368208

[15] Siyi Wang, Sinan Wang, Yujia Fan, Xiaolei Li, and Yepang Liu. 2024. Leveraging Large Vision-Language Model for Better Automatic Web GUI Testing. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 125–137.

[16] Wenhua Wang, Yu Lei, Sreedevi Sampath, Raghu Kacker, Rick Kuhn, and James Lawrence. 2009. A combinatorial approach to building navigation graphs for dynamic web applications. In *2009 IEEE International Conference on Software Maintenance*. IEEE, 211–220.

[17] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, et al. 2024. On the Evaluation of Large Language Models in Unit Test Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 1607–1619.

[18] Yan Zheng, Yi Liu, Xiaofei Xie, Yepang Liu, Lei Ma, Jianye Hao, and Yang Liu. 2021. Automatic web testing using curiosity-driven reinforcement learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 423–435.