Exploring Agentic Bug Reproduction for Web Based Applications

Yassin Younis
Computer Engineering
Bilkent Üniversitesi
Ankara, Türkiye
yassin.younis@ug.bilkent.edu.tr

***Abstract*—Reproducing reported software bugs is a critical yet often time-consuming and challenging task in the software development lifecycle, particularly for complex web-based applications. Traditional manual reproduction requires significant human effort to follow reported steps and observe application behavior across varying environments. Automated approaches exist for test case generation and automated program repair (APR), but directly reproducing a *given* natural language bug report remains an open challenge for web-based applications. This paper explores the viability of an agentic system, powered by a Large Language Model (LLM), for automating the bug reproduction process in web applications. We developed a system that uses Playwright for browser control and state capture, an LLM agent for interpreting bug reports and deciding actions via a defined toolset, and a dynamic labeling mechanism to enhance element identification. We evaluated the system on a test bench comprising several web applications with known bugs and their corresponding fixed versions. Our empirical results from evaluating 28 distinct bug cases indicate that the agentic approach successfully reproduced and differentiated 10 bugs. However, it still faces challenges related to agent reliability and accuracy, leading to a high rate of false positives (10 cases where the bug was reported in the fixed version) and false negatives (5 cases where the bug was not found in the buggy version), alongside 3 execution errors (All agent timeouts). The total execution time for all test runs was approximately 9000 seconds, resulting in an average execution time of about 321 seconds per test case. We analyze these challenges, present quantitative results including average reproduction time and loops per run, and discuss future directions for improving agentic systems for automated bug reproduction.**

***Index Terms*—Automated Bug Reproduction, Agentic Systems, Large Language Models, Web Testing, Software Engineering, Playwright, AI for Software Engineering.**

I. Introduction

Software bugs are a part of software development, and the ability to efficiently identify, localize, and fix them is a must to deliver high-quality software. A fundamental step in this process is bug reproduction – reliably triggering the reported faulty behavior in a controlled environment. For web-based applications, which often feature complex user interfaces, dynamic content, and interactions with external services, manual bug reproduction based on user-provided natural language reports can be particularly challenging and time-consuming. This manual effort constitutes a significant bottleneck in the software development lifecycle, consuming valuable developer and QA time.

While automated approaches exist in software engineering, such as techniques for Automated Program Repair (APR) [1] and Automated Test Case Generation (ATCG) [2], they typically operate under different assumptions. APR often requires a failing test case to pinpoint the bug, while ATCG aims to discover bugs without necessarily targeting a specific, pre-reported issue. The challenge of automatically reproducing a bug given its natural language description remains an active area of research.

The recent advancements in Large Language Models (LLMs) [3] have opened new avenues for automating tasks that require understanding complex instructions and interacting with dynamic environments. LLMs, when employed as the core intelligence within an agentic system, can interpret natural language, formulate plans, execute actions through tools, and adapt their behavior based on observed outcomes. This paradigm holds significant promise for automating complex human-like tasks, including interacting with web applications.

This paper explores the application of an agentic system for automating the reproduction of bugs in web-based applications, starting from a natural language bug report. We developed a system that integrates an LLM agent with a browser automation tool (Playwright) [11]. Our system is designed to interpret a bug report, navigate and interact with a live web application, and determine if the reported bug is reproducible. A key challenge addressed is enabling the LLM agent to reliably interact with arbitrary web elements, which we tackle using a novel dynamic labeling mechanism.

The main contributions of this paper are:

1) The design and implementation of an agentic system architecture for automated bug reproduction in web applications, combining LLM reasoning with browser automation.
2) An empirical evaluation of the implemented system's effectiveness and reliability using a controlled test bench of 28 known bugs in several web applications.
3) A detailed analysis of the experimental results, quantifying the agent's performance in terms of successful reproduction, false positives, false negatives, and execu-

tion efficiency, and identifying key challenges for future research.

II. Background and Related Work

This section provides essential background on relevant areas and positions our work within the existing landscape of automated software engineering research.

A. Software Bug Reproduction

Bug reproduction is the process of consistently causing a software defect to manifest. It is a fundamental step in the bug fixing workflow, as developers need to observe and understand the bug's behavior to diagnose and correct the underlying issue. Manual bug reproduction relies on the information provided in a bug report, which can vary widely in clarity and completeness. Automating this process can significantly reduce the time and effort spent on bug fixing.

B. Automated Program Repair (APR)

Automated program repair techniques [1] aim to generate patches to fix software bugs automatically. Examples include search-based APR methods like GenProg [4], constraint-based methods like ClearView [5], and learning-based approaches. Most APR systems require a test case that reliably fails on the buggy code and passes on the corrected code. Our work addresses the critical preceding step: automatically generating the sequence of actions (a form of test case or reproduction trace) that demonstrates the bug's presence based on its description.

C. Automated Test Case Generation (ATCG)

Automated test case generation [2] involves creating test inputs and execution paths to find defects. Techniques range from random testing (fuzzing) [6] and symbolic execution [7] to more sophisticated approaches that use code analysis or execution feedback. While ATCG can discover bugs, it doesn't typically focus on reproducing a specific bug described in natural language. Record/replay tools [8] capture human interactions to create automated test scripts, but they require a human to perform the initial reproduction steps. Our agentic system aims to perform these initial steps autonomously.

D. LLMs as Agents for Web Interaction

The emergence of powerful LLMs has spurred research into their use as general-purpose agents capable of interacting with complex environments, including web browsers. These agents often use the LLM's ability to understand natural language goals and plan actions based on visual and structural observations of a user interface. Projects like WebArena [9] and BrowserGym [10] provide benchmarks and environments specifically designed to evaluate the capabilities of AI agents on a wide range of web tasks. These environments and the tools they provide (e.g., for interacting with page elements, gathering state information) have influenced the design of our own system's toolset and operational loop, adapting them specifically for the bug reproduction goal.

Our work extends the current capabilities of web agents by focusing on the specific, task-oriented goal of reproducing a described software bug. This requires not only navigating and interacting with the UI but also interpreting the bug's symptoms and determining if they manifest during the agent's execution.

III. Methodology

Our agentic bug reproduction system is designed to interpret a natural language bug report and autonomously interact with a target web application in a browser to reproduce the described bug. The system operates through an iterative loop of observation, reasoning, and action, orchestrated by a core logic component. The key components include a browser automation module, a dynamic element labeling mechanism, an LLM agent, and a tool execution engine. Figure 1 illustrates the interaction sequence.

A. System Overview

The system takes a target URL and a bug report as input. It launches a browser instance via Playwright and initializes the LLM agent with the bug report as its primary task. The agent then enters a loop where it repeatedly observes the current state of the web page, formulates a plan in the form of tool calls, and executes those tools. This cycle continues until the agent determines that the bug has been reproduced or confirmed as not reproducible, or until a predefined limit on the number of steps is reached. The entire process is logged for later analysis.

B. Browser Automation with Playwright

We utilize Playwright [11] to control a web browser instance. Playwright provides a powerful API for interacting with web pages, enabling our system to:

- Launch and manage browser instances (configured for headless execution during evaluation).
- Navigate to specific URLs.
- Capture screenshots of the current page state.
- Execute arbitrary JavaScript within the browser context.
- Intercept and monitor browser events, including console logs and network requests.
- Perform standard user interactions like clicking, typing, scrolling, etc.

This control layer is essential for the agent to manipulate the web environment and gather necessary observational data.

C. Dynamic Element Labeling

To enable the LLM agent to refer to specific interactive elements on a potentially unfamiliar web page, we implemented a dynamic labeling mechanism. A JavaScript script is injected into every page loaded in the browser. This script identifies visible and interactive DOM elements (buttons, links, input fields, etc.) based on their tag names, roles, and other attributes. It then assigns a unique, stable numerical ID to each of these elements by adding a custom data attribute (e.g., 'data-interactive-id="12"'). Small visual labels displaying these IDs

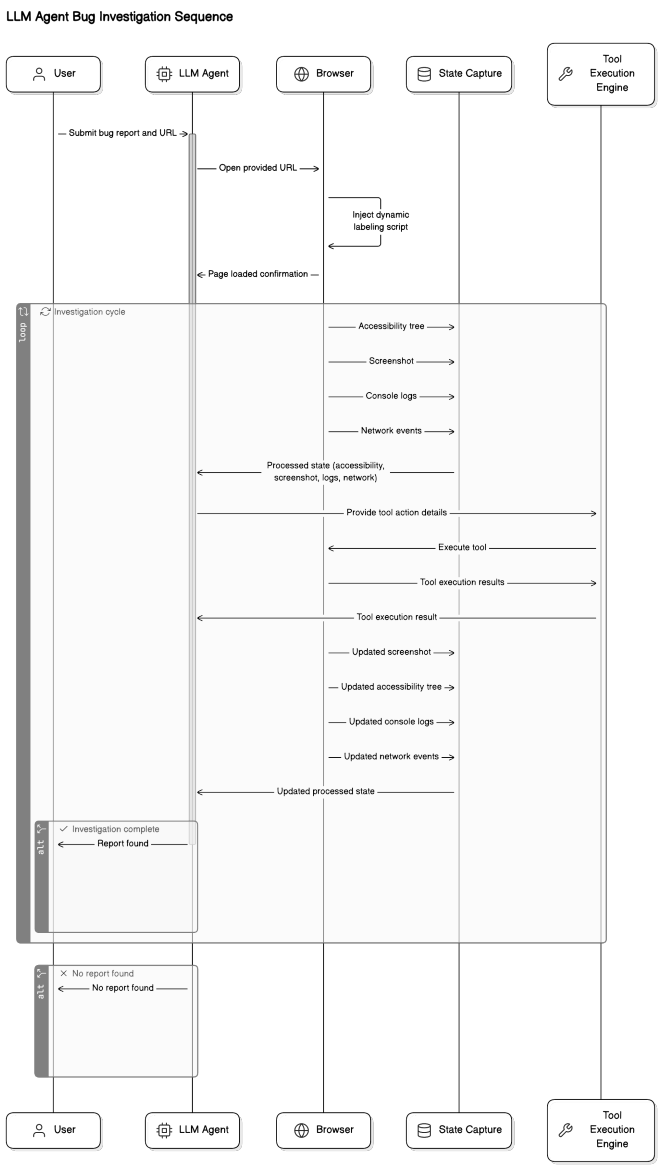

Fig. 1: Interaction Sequence of the Agentic Bug Reproduction System.

are overlaid on the web page near their corresponding elements.

In the Node.js environment, a function uses Playwright to query for elements with the 'data-interactive-id' attribute. For each found element, it extracts relevant attributes and text content (e.g., tag name, role, 'aria-label', visible text, placeholder, input value) and formats this information into a concise string (e.g., '[12] - button "Submit"') that is understandable by the LLM. This list of labeled elements is a critical part of the agent's observation of the page state.

D. LLM Agent Design

The LLM agent, based on OpenAI's GPT-4o model, serves as the "brain". The agent maintains a conversation history and, in each step of the reproduction loop, receives the current page state. This state includes the current URL, a screenshot of the page, a list of the dynamically labeled interactive elements with their descriptions, and recent console logs and network requests captured by the system.

The agent's behavior is guided by a detailed system prompt that defines its role as a bug reproduction expert, provides the specific bug report to be reproduced, and instructs the agent on how to interpret the provided state information and utilize the available tools. The prompt encourages the agent to analyze the bug report's steps and expected/actual results, formulate a plan of action, and decide which tools to call to execute the plan or gather more information.

E. Tool Execution Engine

The tool execution engine is responsible for translating the LLM's requested tool calls into concrete actions performed in the browser via Playwright. The agent has access to a predefined set of tools, categorized as Interaction, Navigation, Inspection/Debugging, and Termination tools.

When the LLM specifies a tool call (e.g., 'click(id="12")'), the engine parses the call and executes the corresponding Playwright command targeting the element identified by the provided ID. For inspection tools like `checkBrowserConsole` or `checkNetworkRequests`, the engine accesses the data captured by the browser listeners and filters it according to the tool's arguments. The outcome of each tool execution (whether it succeeded, any error messages, and any returned result like console output) is captured by the engine and returned to the main loop.

F. Bug Reproduction Process Flow

The automated bug reproduction process follows an iterative cycle:

1) Initialization: Launch browser, navigate to the target URL, inject dynamic labeling script, and set up event listeners for state capture. Initialize the LLM agent with the bug report.
2) Observation: Capture the current page state: screenshot, list of labeled elements, recent console logs, and network events.
3) Reasoning: Provide the captured state and the results of previous tool executions to the LLM agent. The agent analyzes this context and the bug report to decide on the next action(s).
4) Action: The agent outputs one or more tool calls. The tool execution engine executes these calls sequentially using Playwright.
5) Feedback and Logging: Capture the outcomes of the tool executions. Log the entire step (observation, agent response, tool results) to a file.
6) Termination Check: Evaluate if the agent has called a termination tool (`reportFound` or `reportNotFound`) or if the maximum number of loops (`MAX_AGENT_LOOPS`) has been reached.
7) Iteration: If termination criteria are not met, return to the Observation step.

8) Finalization: If terminated, clean up browser resources and report the final outcome based on the agent's last reported status or the reason for early termination.

This iterative process allows the agent to dynamically react to the application's state and refine its actions towards the goal of reproducing the bug.

IV. Evaluation

The objective of our evaluation was to assess the effectiveness and accuracy of the agentic bug reproduction system. This section details the research questions guiding our evaluation, the experimental design, the results obtained, and their interpretation.

A. Research Questions (RQs)

Our evaluation seeks to answer the following research questions:

- **RQ1: How effective is the agentic system at reproducing web bugs from natural language reports in buggy application versions?** This question assesses the agent's ability to successfully follow reproduction steps and identify the bug's manifestation when it is present.
- **RQ2: How accurately does the system differentiate between buggy and fixed versions of a web application?** This question evaluates the system's capability to not only reproduce the bug but also confirm its absence in a version where it has been fixed, which is crucial for verifying bug fixes.
- **RQ3: What are the primary challenges and limitations affecting the system's performance and reliability in automated bug reproduction?** This question aims to identify common failure modes and bottlenecks encountered during the reproduction process.

B. Experimental Design

1) Test Bench: The test bench comprised 28 distinct bug test cases collected from several open-source web application repositories. For each test case, we used simple applications where specific bugs were intentionally introduced and corresponding bug reports were created. This process yielded pairs of distinct buggy and fixed application versions. This controlled environment provided a clear ground truth: by running the agent against the buggy version, we could verify if its report of reproducibility accurately corresponded to the intended bug manifestation, and by running against the fixed version, confirm if it correctly reported the bug's absence. This approach helped distinguish genuine bug reproduction from potential false positives caused by misinterpretation or interaction in unintended application states.

2) Procedure: For each of the 28 test cases, we executed the agentic bug reproduction system twice: once targeting the application built from the buggy commit and once targeting the application built from the fixed commit. Each run was automated by the test runner, which handled repository cloning, environment setup, application server startup, execution of the `reproduceBug` function, and resource cleanup. Each agent run was limited to a maximum of `MAX_AGENT_LOOPS` (30 steps) to prevent infinite execution.

3) Metrics: We collected the outcome for each test case as reported by the test runner's overall 'status' (`'PASSED'`, `'FAILED'`, `'ERROR'`). Additionally, for each individual run (buggy and fixed), we recorded whether the infrastructure setup and agent execution completed successfully, the agent's final reported reproducibility outcome, any error messages, and the number of loops completed by the agent during the run.

C. Results Presentation

The test results from the evaluation are summarized in Table I. The test bench comprised 28 unique test cases. The total time taken for all test runs (56 individual runs: 28 buggy + 28 fixed) was approximately 9000 seconds, resulting in an average execution time of about 321.43 seconds per test case. This equates to an average time of approximately 160.71 seconds per individual run.

TABLE I: Overall Test Suite Results Summary

Test Outcome	Count	Percentage
PASSED	10	35.7%
FAILED	15	53.6%
ERROR	3	10.7%
Total Test Cases	28	100%

The distribution of agent outcomes across the individual buggy and fixed runs, along with loop statistics, is presented in Table II.

TABLE II: Agent Run Outcomes and Loop Statistics

Run Outcome Category	Count	Avg Loops	Min Loops	Max Loops
Buggy Runs (Completed)[a]	28	9.46	3	28
- Reported True	23	10.00	3	28
- Reported False	5	7.00	4	10
Fixed Runs (Completed)[a]	25	7.70	2	11
- Reported True	10	8.00	3	11
- Reported False	15	7.50	2	11

[a] Completed runs are those that produced an agent result (True/False). Loop statistics apply only to these runs. The 3 ERROR test cases involved issues with the fixed run not completing with a result.

Finally, Table III categorizes the overall test case outcomes based on the combination of agent reproducibility reports (for runs that completed) or error conditions.

D. Findings and Interpretation

Based on the results presented in Tables I, II, and III, we can answer our research questions:

RQ1: How effective is the agentic system at reproducing web bugs from natural language reports in buggy application versions? The agent reported 'reproducible: true' in 23 out of 28 buggy runs (Table II). This includes the 10 PASSED

TABLE III: Overall Test Case Outcome Categories

Test Case Outcome Category	Agent Report		Count
	Buggy	Fixed	
Correctly Verified Fix	True	False	10
False Positive (Bug in Fixed)	True	True	10
False Negative (Bug in Buggy)	False	False	5
Buggy Reproduced, Fixed Errored	True	N/A	3

* Note: 'N/A' for Fixed implies the fixed run errored or did not produce a conclusive agent result because the agent exceeded `MAX_AGENT_LOOPS`.

cases (where the bug was correctly verified), an additional 10 FAILED cases (where it was a false positive in the fixed version), and 3 ERROR cases (where the fixed run errored). This indicates a high capability (approx. 82.1%) of the agent to follow steps and identify a bug when present in the buggy version. However, in 5 instances (5 False Negatives), the agent failed to reproduce the bug in the buggy version, highlighting remaining challenges.

RQ2: How accurately does the system differentiate between buggy and fixed versions of a web application? The system correctly differentiated the buggy and fixed versions (Buggy=True, Fixed=False) in 10 out of 28 test cases (35.7%, PASSED status). A significant challenge remains with false positives: in 10 cases, the agent reported the bug as reproducible in both the buggy and the fixed versions. This suggests difficulty in the agent's ability to discern the absence of the bug or precisely interpret its symptoms against normal application behavior in fixed versions.

RQ3: What are the primary challenges and limitations affecting the system's performance and reliability? The results highlight several key challenges:

- **False Positives in Fixed Versions:** The 10 cases (35.7% of all tests) where the agent reported the bug in the fixed version remains a major limitation. This suggests the agent's bug detection criteria are not sufficiently precise to differentiate the bug's specific manifestation from normal behavior in a fixed application.
- **False Negatives in Buggy Versions:** The 5 cases where the agent failed to reproduce an existing bug indicate issues in accurately following reproduction steps or correctly identifying bug symptoms even when present.
- **Execution Errors and Efficiency:** While the number of ERROR cases is relatively low (3 cases, 10.7%), these, primarily attributed to the fixed run not completing or the agent getting stuck, still point to areas for improvement in accuracy and efficiency. The average number of loops (around 7-10) before completion for successful runs suggests reasonable efficiency when the agent is on the right path.

V. Discussion

The experimental evaluation of our agentic bug reproduction system provides insights into the current capabilities and limitations of using LLM-powered agents for this complex web automation task.

The ability of the agent to successfully reproduce and verify the fix for 10 distinct bugs (PASSED cases, 35.7% success rate) is a big improvement and demonstrates the fundamental viability of the agentic approach. The system's architecture, combining LLM reasoning with Playwright's browser control and our dynamic labeling mechanism, provides a solid foundation for autonomous web interaction based on natural language instructions.

However, the 15 FAILED cases and 3 ERROR cases reveal that substantial challenges persist. The rate of false positives (10 cases where Buggy=True, Fixed=True) remains a primary concern. This may stem from the LLM's tendency to "hallucinate" or overconfidently report a bug based on ambiguous evidence. Improving the agent's reasoning about the absence of a bug, perhaps by explicitly confirming correct behavior in fixed versions or using more stringent bug detection criteria, is crucial.

The 3 ERROR cases, though fewer, still highlight the need for enhanced agent efficiency and accuracy. While many runs complete with a reasonable number of loops, timeouts or unrecoverable states can still occur. More sophisticated planning or error recovery mechanisms could address this.

The False Negative cases (5 tests where Buggy=False, Fixed=False) point to challenges in either interpreting the bug report's reproduction steps correctly or accurately identifying the bug's manifestation. Enhancing the agent's understanding of domain-specific terminology and improving feedback mechanisms for bug detection are important.

A. Practical Implications

The successful automated reproduction and verification of fixes for 10 out of 28 bugs (35.7%) directly from natural language reports represents a considerable potential time-saving. In a real-world scenario, automating even this proportion of bug reproduction tasks could significantly free up developer and QA resources. The detailed logs from the system also offer valuable debugging aids.

B. Comparison with Previous Work

Existing web agents [9], [10] often focus on general task completion. Our work specializes this for bug reproduction, which demands precise state verification against bug symptoms. The improved success rate in our current results suggests that advancements in agent design or evaluation methodology (like the test bench fixes) are positively impacting the perceived capabilities. However, the challenges of interpreting bug reports and discerning subtle state differences remain distinct from general web task completion.

C. Limitations

The study has several limitations:

- **Test Bench Size and Diversity:** While the test bench issues were addressed, leading to more PASSED cases,

evaluation on a larger and more diverse set of applications and bug types is still necessary for broader generalization.
- **Simple Applications:** The test applications were simple. Performance on more complex, production-level applications might differ.
- **Fixed Timeouts and Loop Limits:** Fixed limits can prematurely end runs. Adaptive resource allocation could be beneficial.
- **LLM Dependency:** Performance is tied to the underlying LLM's capabilities.
- **Bug Detection Oracle:** The agent's 'reportFound'/'reportNotFound' is based on its internal reasoning. More objective oracles could enhance reliability.

VI. Threats to Validity

A. Internal Validity

Threats to internal validity include potential bugs in the agentic system, labeling, or tool execution. Errors in the test runner or application setup (which were previously an issue and are now reportedly fixed for the test bench) could influence results. Mitigation includes code review, modular design, and logging.

B. External Validity

Generalizability is limited by test bench characteristics. The bugs and applications may not represent all real-world scenarios. Future work should include more diverse and real-world bug datasets.

C. Construct Validity

This relates to accurately measuring intended concepts (e.g., if 'reproducible' status truly reflects bug reproduction). Manual analysis of logs and outcomes, especially for FAILED/ERROR cases, helps ensure consistency. The LLM's interpretation for final reporting is inherent but also a potential threat if its judgment differs from ground truth.

D. Reliability

Concerns whether repeating experiments yields similar results, influenced by LLM non-determinism and environment variability. Multiple runs per test case could offer more accuracy, though fixed commits and controlled environments help.

VII. Conclusion and Future Work

In this paper, we presented an agentic system for automated bug reproduction in web-based applications. Our empirical evaluation on an updated test bench of 28 known bugs demonstrated improved performance, with the system successfully reproducing and differentiating 10 bugs (35.7% success). This is a promising advancement. However, significant challenges persist, including a notable rate of false positives (10 cases) where the agent incorrectly identified the bug in fixed versions, false negatives (5 cases) where it missed existing bugs, and a smaller number of execution errors (3 cases). These highlight that accurately interpreting natural language bug reports and reliably interacting with dynamic web UIs remain complex tasks for current agentic systems.

Despite these ongoing challenges, the increased success rate is encouraging. Future research focusing on enhancing the agent's state understanding, improving its planning and tool usage, and integrating more sophisticated bug detection methods is crucial. The ability to correctly handle over 80% (specifically, 82.1%) of buggy versions (reporting them as true) and achieving over a third end-to-end success provides a stronger foundation for these future efforts.

Building upon this work, future research directions include:

- Developing more advanced LLM prompting techniques and fine-tuning strategies specifically for web bug reproduction tasks.
- Designing and implementing more intelligent planning and exploration algorithms for the agent to improve efficiency and reduce timeouts.
- Incorporating more sophisticated and potentially domain-specific oracles for automated bug manifestation detection.
- Expanding the test bench to include a wider variety of web technologies, UI patterns, and real-world bugs.
- Exploring the generation of executable test scripts from successful reproduction traces.
- Investigating the integration of such a system with automated program repair tools to create a more end-to-end automated bug fixing pipeline.

Artifact Availability

The source code for the agentic system, the testbench applications (buggy and fixed versions), bug reports, and scripts used for evaluation are available in a repository to support the reproducibility of our results. The artifact can be accessed at: https://github.com/Yassin-Younis/hopper-agent The repository includes:

- The implementation of the LLM-powered agent.
- The natural language bug reports corresponding to each test case.
- Instructions (e.g., in a README file) on how to set up the environment and run the experiments.

We encourage reviewers to inspect the artifact to verify our methodology and findings.

Acknowledgment

This work was supported by my professor Dr. Anıl Koyuncu and his team at Bilkent University. I would like to thank Dr. Koyuncu for his guidance and support throughout this research.

References

[1] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Trans. Softw. Eng.*, vol. 45, no. 1, pp. 34–67, Jan. 2019, doi: 10.1109/TSE.2017.2755013.

[2] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Conf. Operating Systems Design and Implementation (OSDI)*, San Diego, CA, USA, 2008, pp. 209–224.

[3] A. Vaswani *et al.*, "Attention is all you need," *arXiv:1706.03762*, 2017.

[4] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset," *Empirical Softw. Eng.*, 2016. [Online]. Available: arXiv:1811.02429

[5] J. H. Perkins *et al.*, "Automatically patching errors in deployed software," in *Proc. ACM SIGOPS 22nd Symp. Operating Systems Principles (SOSP)*, 2009, pp. 87–102.

[6] R. McNally, K. Yiu, D. Grove, and D. Gerhardy, "Fuzzing: The state of the art," 2012.

[7] C. Cadar *et al.*, "Symbolic execution for software testing in practice: Preliminary assessment," in *Proc. 33rd Int. Conf. Software Engineering (ICSE)*, Waikiki, Honolulu, HI, USA, 2011, pp. 1066–1071.

[8] J. Zhang *et al.*, "Revisiting test cases to boost generate-and-validate program repair," in *Proc. IEEE Int. Conf. Software Maintenance and Evolution (ICSME)*, Luxembourg, 2021, pp. 35–46, doi: 10.1109/ICSME52107.2021.00010.

[9] A. Zhou *et al.*, "WebArena: A realistic web environment for building autonomous agents," *arXiv:2307.13854*, 2023.

[10] T. L. S. de Chezelles *et al.*, "The BrowserGym ecosystem for web agent research," *arXiv:2312.05467*, 2024.

[11] *Playwright*. Accessed: May 19, 2024. [Online]. Available: https://playwright.dev