I Will Try to Fix You: Large Language Models for Mobile GUI Test Repair

I Will Try to Fix You: Large Language Models for Mobile GUI Test Repair

***Abstract*—Mobile applications evolve rapidly, forcing development teams to deliver updates at high frequency while maintaining reliable testing processes. In this context, GUI testing is essential for validating user-facing behavior; however, it is highly affected by fragility, as tests break across application versions not due to functional defects, but rather because of changes in the GUI structure, appearance, or properties. This leads to substantial manual effort to diagnose failures and repair outdated tests.**

This study aims to: (i) identify the most common causes of mobile GUI test breakages across application versions; (ii) assess how Large Language Models (LLMs) can reduce the effort required for repairing broken tests; and (iii) compare an LLM-based repair strategy with a state-of-the-art automated repair tool, *Healenium-Appium*.

A total of 61 broken GUI tests from 19 real-world Android applications were analyzed to identify the underlying causes of breakage. We then developed an LLM-based repair approach, in which the model and user iteratively review test failures and progressively update test scripts until the repaired test passes. The approach was experimentally evaluated against Healenium-Appium.

The LLM-based method successfully repaired 45 out of 61 tests (73.8%) after a single interaction and 56 out of 61 tests (91.8%) after multiple interactions, outperforming *Healenium-Appium* by 50.8% and 68.8%, respectively.

These results show that LLM-based repair effectively mitigates GUI test fragility, reducing maintenance effort while achieving higher repair success rates than existing automated tools.

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

The accelerating pace of mobile app releases has intensified the need for continuous and reliable GUI testing. Automated tests are essential for regression detection. However, GUI tests are notoriously fragile [1]. Updates to an app's UI—such as widget renaming, layout changes, or design modifications—often cause test failures even when no functional defect exists. Developers must determine whether a failing test indicates a true bug or if it is merely an outdated locator. Manual repair of the test is then typically required.

Recent advancements in Large Language Models (LLMs) provide new opportunities. These models can interpret UI structure, reason about semantics, and generate code, making them suitable for tasks such as widget identification, test migration, and automated repair. Prior works—such as GPTDroid [2], MACDroid [3], ScenGen [4], and DroidAgent [5]—demonstrate the feasibility of LLMs for exploration and test generation; yet, their applicability to targeted test repair remains underexplored.

Building on these insights, this work investigates whether LLMs can help mitigate the fragility of mobile GUI tests by assisting in their automated evolution. Specifically, we examine LLMs' performance in repairing real-world test scripts whose locators or interaction flows no longer align with updated applications. While previous research focuses mainly on LLM-driven test generation or exploratory interaction, incrementally repairing existing tests presents unique challenges.

To address this gap, we conduct an empirical evaluation on 19 real-world Android applications comprising 61 broken GUI tests. We first characterize the root causes of test breakage when applications evolve, identifying both structural and semantic factors that contribute to fragility. We then introduce a multi-step LLM-based repair workflow and assess its effectiveness in restoring test executability. Finally, we compare our approach against *Healenium-Appium*, a state-of-the-art locator repair tool. Our findings position LLM-based repair as a promising new direction for enhancing the robustness of scripted automated mobile test suites. We consistently observe more tests fixed by the LLM than by the baseline, suggesting that this technology can be successfully integrated into the Android test repair pipeline.

The remainder of this manuscript is organized as follows: in section II, we describe related work in the field of LLMs used for test repair and for GUI testing; in section III we describe our evaluation approach; in section IV we report the quantitative results of the study; in section V we discuss the results and we analyze the possible limitations of the study; in section VI we conclude the paper and we outline possible future work.

II. Background and Related Work

In this section we report works on test fragility, on the use of LLMs for test repair, maintenance, and GUI testing.

A. Fragility of test cases

Android GUI tests rely on widget locators - IDs, text attributes, XPath structures, or contextual descriptors. As applications evolve, GUI elements may change, and such changes may break test cases without implying defects. This problem, discussed extensively in software testing literature, is referred to as GUI testing fragility [6]. Empirical studies about GUI testing frameworks for Android applications found that up to 10% of modified LOCs in open-source projects belong to test cases, and that a relevant portion of such modifications (up to 60%) are induced by test fragility [1]. Several techniques have been developed in recent years to predict and address fragility, especially in the mobile and web domains. Di Meglio and Starace developed a metric (the *Fragility Score*) to estimate the extent to which a web test is prone to breakage [7]. Many techniques have been developed to fix test case fragility using static approaches based on structural locator analysis on consecutive versions of software repositories such as SIMILO

[8] or COSER [9], visual analysis of the appearance of the rendered GUIs such as the work by Ardito et al. [10] or Pan et al. [11], or a combination of the approaches as in GUIDER by Xu et al. [12].

B. Large Language Models for test development and maintenance

Recent work has applied Large Language Models to diverse tasks related to test development and maintenance. Yaraghi et al. presented TaRGET, an approach that leverages pre-trained code-language models for automated test case repair, treating test case repair as a language translation task. Evaluated on a set of 59 Java projects, the tool obtained an effectiveness of 66.1% match accuracy in fixing test cases [13]. Fatima et al. presented FlakyFix, a tool that leverages Large Language Models to predict test case flakiness and suggest repairs to test code, achieving a high percentage of repairs (up to 80%) to pass [14]. Rahman et al. presented UTfix, a tool for repairing Python tests when their corresponding methods change in software repositories, capable of repairing up to 60% of assertion failures [15].

C. Large Language Models for GUI testing

In GUI testing, LLMs have primarily been used to generate test cases from existing code and/or specifications. Liu et al. developed GPTDroid [16], an LLM-based system for automated GUI testing of Android applications. It follows a Q&A approach, asking the LLM to play the role of a human tester and enabling the interaction between the LLM and the application under test.

AssistGUI [17] by Gao et al. is a benchmark designed for Desktop GUI Automation. While previous similar studies focused on web or mobile applications, AssistGUI is designed for use in desktop productivity software.

GERALLT [18] by Rosenbach, Heidrich, and Weinert is a system that leverages LLMs for exploratory GUI testing of real-life engineering software, aiming to identify unintuitive and inconsistent sections of the user interface. Zimmermann and Koziolek [19] presented an approach to automated GUI testing in web applications that integrates the GPT-4 model with Selenium. The adoption of the GPT-4 model allows for bypassing the hassle of fine-tuning models with pre-recorded user interaction data from human testers.

TEMdroid [20] by Zhang et al. is a learning-based widget matching approach for GUI test migration. While existing migration approaches use manually defined matching functions, TEMdroid uses BERT to capture contextual information and learns a matching model for widgets, enabling it to handle complex matching relations and adapt to diverse scenarios in real-world apps. AutoDroid [21] by Wen et al. is a task automation system for Android applications based on LLMs. The key to AutoDroid is to combine LLMs' commonsense knowledge with app-specific knowledge through dynamic app analysis.

MACdroid [22] by Zhang et al. is an approach to test migration in which LLMs are used first to extract a general test logic from multiple existing test cases, and then to generate concrete test cases for a target app based on the extracted test logic.

DroidAgent [5] by Yoon et al. is an autonomous Android GUI testing agent for semantic, intent-driven GUI testing automation based on LLMs. DroidAgent generates natural-language descriptions of specific tasks that can be performed using the given Application Under Test (AUT), and then attempts to interact with the AUT's GUI to achieve those tasks.

However, there is a gap in the literature regarding the application of LLMs to repair GUI test cases for mobile applications. The current state of the art includes Healenium [23], an open-source library that leverages machine learning to repair test cases written in Selenium (for web applications) or Appium (mobile applications). Fedriga et al. presented a preliminary analysis of zero-shot test repairing activities performed with GitHub Copilot with Claude 3.7 Sonnet [24].

III. Methodology

The objective of this study is to examine the frequency and underlying causes of GUI test breakage in Android applications as their user interfaces evolve. Furthermore, we aim to evaluate the effectiveness of a Large Language Model–based repair approach for restoring outdated tests and compare its performance with established, non-LLM repair techniques.

The Research Questions (RQs) that drive this research are the following:

- **RQ1:** What is the incidence and what are the causes of Android GUI test fragilities?
- **RQ2:** What is the performance of an LLM-based approach in Android GUI test repair?
- **RQ3:** How does our LLM-based approach for Android GUI tests perform compared to the *Healenium-Appium* baseline?

To answer RQ1, we first ran tests for each application on the base version to understand their behavior and verify that they ran without issues in our environment. Then, each test was executed on the updated versions of the applications. For each test, we determined whether it passes on the updated version, and for all failures, we carefully annotated where the test fails and the causes. Since a test can have multiple causes of failure, but only the first is evident during execution, a further analysis was conducted using Appium Inspector to identify all possible causes of failure for the tests.

For RQ2, we designed an LLM-based approach for mobile GUI test repair. This approach is applied to all the tests whose execution failed on the updated version of their application, with the aim of generating a repaired version of each test, i.e., a test that passes when executed on the updated version of its application, while also maintaining its intended behavior in testing widgets and functionalities. The repair process is described in detail in section III-C.

Finally, in RQ3, we want to compare the performance of our LLM-based approach for mobile GUI test repair with that of *Healenium-Appium*. Similarly to RQ2, the *Healenium-Appium* repair approach was applied to each test whose execution failed on the updated version of its application. The results are annotated and compared to those of the LLM-based approach.

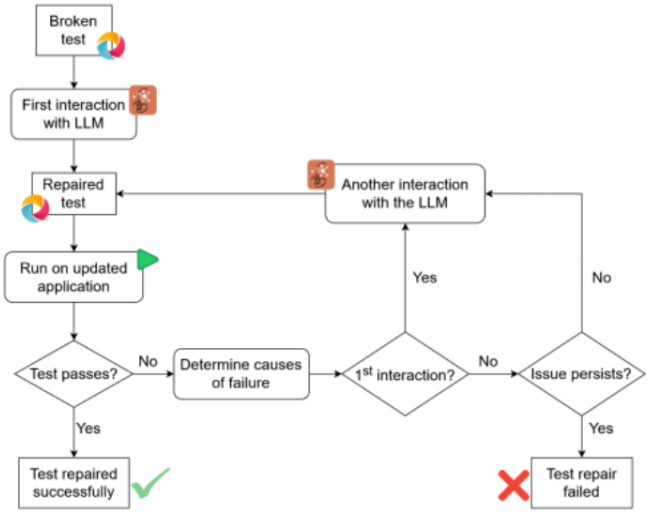

Fig. 1. Repair process

TABLE I
LIST OF THE APPLICATIONS USED IN THE RESEARCH.

Application	Base Version	Updated Version
AnkiDroid	v2.6	v2.13.0
AntennaPod	v2.1.3	v2.5.0
AppLaunch	v2.0.0	v3.1.1
BookDash	v2.8.0	v2.9.2
Counter	v13	v18
Currency	v1.0	v1.35
DIDA	v5.7.0.0	v5.8.8.1
DNS66	v0.4.1	v0.6.8
DumbphoneAssistant	v0.4	v0.5
Fwknop	v1.0.0	v1.2.3
Mgit	v1.5.1	v1.6.1
Now in Android	v0.0.5	main branch
OpenBikeSharing	v1.0	v1.10.0
RedReader	v1.14.0	v1.23.1
Sgit	v1.3.0	v1.3.2
SimpleCalculator	v2.0.0	v3.1.2
SimpleWeather	v4.1.0	v5.3.2
SoundRecorder	v1.2.5	v1.3.0
SunTimesWidget	v0.11.0	v0.15.13

A. Operational Settings

To perform our analysis, we experimented with GitHub Copilot instrumented with Claude Sonnet 3.7 Thinking in September 2025. Unfortunately, at the time of writing, the used version was retired by GitHub Copilot[1], so all the results may vary, thereby reducing the replicability of our results. All the test cases were written in Python 3.13 and required to be consistent with Appium 2.0 style.

The applications were deployed on two Android emulators provided by Android Studio: the Google Nexus 5X running Android 6.0 was employed for all applications from the ExtRep dataset, to maintain consistency with the sampling environment, whereas the Pixel_3a_API_34_extension_level_7_x86_64 device with Android 14.0 was designated for Now in Android (to support the newer features integrated in Android). The Android Debug Bridge (ADB) was employed to install the applications on the emulated devices using the APK files.

Appium Inspector was utilized to examine the applications under evaluation. Appium Inspector is a graphical user interface (GUI) tool for Appium that facilitates a visual inspection of the application being tested. It displays a screenshot of the current application page, along with its page source, and offers various features for interacting with the app. Additionally, Appium Inspector was employed to extract the view hierarchy files and capture screenshots of the applications.

B. Project selection

The set of applications used in this research comes from the repository used in ExtRep [25][2]. Each application comes with the APK files for both the base and updated versions, along with Appium GUI tests for the base version; in total, the ExtRep dataset includes 40 applications with 112 tests. Some applications from this dataset had to be excluded: some applications could not be installed on the emulator in use; one application had visualization problems on Appium Inspector, which hindered an accurate analysis of the application; one application had unresolvable performance issues with the Appium engine; some applications did not have any tests that needed to be repaired. In the end, 18 applications from the ExtRep dataset were used, totaling 87 tests.

In addition – for continuity with our previous work [**?**][3] – we also used an open-source application, Now in Android, for which we wrote 26 Appium tests that follow the logic of the application's GUI tests in its GitHub repository.

The names and versions of the applications used in this research are listed in Table I.

C. Repair process

The applications drawn from the ExtRep dataset included GUI tests originally implemented using Appium 1.0. Since Appium 1.0 has been deprecated and is no longer maintained, we first devoted effort to migrating these tests to Appium 2.0, the current and actively supported version. All additional tests authored as part of this study were written directly using Appium 2.0.

In contrast, *Now in Android* did not provide existing Appium-based tests. Instead, it contained a set of GUI tests implemented using the Jetpack Compose UI Test framework. Based on the testing logic encoded in these Compose tests, we developed a corresponding suite of 26 Appium test cases, ensuring that the Appium tests exercised the same functional behavior.

All tests for the ExtRep applications were implemented in Python, whereas the Appium tests developed for *Now in Android* were written in Java to align with the host project's conventions.

After preparing all applications and their associated test suites, we initiated the repair process (represented graphically in Figure 1, which proceeded according to the following steps:

[1]https://github.com/copilot
[2]https://github.com/anonymousproject25/ExtRep)
[3]citation suppressed for double-blind review policy

1) **Baseline test execution.** Each test was first executed on the base version of its corresponding application. This allowed us to understand the intended behavior of tests originally written by others and ensured that all tests passed on the version for which they were designed.
2) **Execution on the updated version and failure identification.** The tests were then run on the updated versions of each application to identify which required repair. A test was considered broken if its execution failed or if its behavior differed from the expected behavior observed on the base version. For every failing test, we carefully documented the underlying cause.
3) **LLM-based repair attempt.** Each broken test was subsequently repaired individually using an LLM-driven procedure. For every test, the LLM was provided with: (i) the file containing the original test script, and (ii) the view hierarchy files in XML format corresponding to the GUI screens exercised by the test, extracted from both the base and updated versions of the application. A uniform prompt template was used for all tests to ensure methodological consistency.
   The prompt used in this phase is the following:

```
"The file <file_name> contains an Appium test
written for the older version (referred to as v1)
of an Android application. Your goal is to repair
the test for the newer version of the application
(referred to as v2). You are provided with the
view hierarchy files of the tested screen, both
for v1 and v2. Screens v1: <list_of_screens>.
Screens v2: <list_of_screens_v2>."
```

4) **Evaluation of the repaired test and iterative refinement.** The LLM produces a repaired version of the test, which is then executed against the updated application. If the test passes while preserving its intended behavior, the repair is deemed successful. Otherwise, the LLM is queried again to produce a revised repair. In this second interaction, the LLM is provided with all artifacts from the initial attempt, including the repaired test generated during the first interaction; when beneficial, screenshots of the most relevant screens are also included. The prompt also includes the error log from the failed execution and/or a concise description of the observed issue, followed by a request to repair the test. The LLM then outputs a new repaired version of the test. As before, the repair is considered successful only if the test passes and maintains its intended behavior when executed on the updated version. If the same issue persists after this second interaction, the repair attempt is considered unsuccessful. If the original issue is resolved but a different problem emerges, this step is repeated.

IV. Results

In this section, we report the results of our analysis, grouped by research question.

A. RQ1: Incidence and causes of fragilities

The first step of our analysis is to assess the main cause of the fragility of our GUI test cases. Upon running the original tests developed for the base version on the updated versions of each application, a total of 102 causes of failure were identified (a test can have multiple causes of failure). We manually inspected the causes of failure for each test and classified them. We adopted the Grounded Theory Open Coding technique [26], applying it to the test reports and to code inspections to extract the different categories. The cause of failure and their occurrences are shown in Table II. Of all the causes of failure, 78 are due to an outdated locator for the widget being searched (resource-id, text, class name, xpath, or content description), 11 to a widget moved to another screen, and 13 to functionality removed or to steps that differ.

TABLE II
Test failure causes and repair results.

Fail Cause	Fails	pass @ 1st	pass @ 2nd
resource-id outdated	40	37	38
text outdated	24	24	24
functionality changed/removed	13	5	10
widget moved	11	11	11
class name outdated	9	4	8
content-desc outdated	3	3	3
xpath outdated	2	2	2

Based on these data, outdated locators are the primary cause of broken tests across different versions of the same application. More specifically, 40 times the property *resource-id* of a widget was outdated, 24 times the *text*, 9 times the *class name*, and 3 times the *content-desc*, while on 2 occasions the tests were not able to identify a widget using the *xpath*. Despite the low number of failures, using *xpath* to locate a widget is usually discouraged in GUI-specific guidelines [27] due to its fragility and maintenance difficulty.

In the tests that were analyzed, the *resource-id* emerged as the most commonly used property for widget identification. There are two primary reasons why this locator may fail: firstly, the *resource-id* of a widget may be updated between different versions or may be removed in an updated version of the application; secondly, the *resource-id* may not always be unique among the widgets present on a screen. In screens that display lists of elements, it is typical for the items in the list to have the same auto-generated *resource-id* and rely on indexes to differentiate one item from another. This situation creates the potential for instability, as altering the order of items or introducing a new element into the list can render previous indices invalid.

The same problem occurs with the *class name* locator. This locator is inherently non-unique within a screen, requiring indexes to pinpoint a specific element. This introduces a level of fragility, and it is quite common for this type of locator to fail whenever an update changes the screen.

B. RQ2: performance of the LLM-based approach

To evaluate LLM performance in mobile GUI test repair, we measured repair success rates after the first interaction and after 2 or more interactions. Every new interaction with the LLM, the context was enriched with new feedback, i.e., the reason for the failure.

TABLE III
APPLICATIONS TESTED AND REPAIR RESULTS COMPARISON BETWEEN LLM-BASED AND *Healenium-Appium* APPROACHES.

Application	# of tests	broken tests	LLM 1 iter.	LLM 2+ iter.	Healenium
AnkiDroid	7	4	2	3	2
AntennaPod	5	3	1	3	0
AppLaunch	5	2	2	2	2
BookDash	3	1	1	1	0
Counter	6	2	1	1	0
Currency	5	2	1	2	0
DIDA	8	8	4	7	0
DNS66	6	5	5	5	0
DumbphoneAssistant	2	2	2	2	2
Fwknop	2	1	1	1	1
Mgit	2	2	2	2	0
Now in Android	26	4	4	4	3
OpenBikeSharing	5	5	5	5	0
RedReader	6	3	2	3	0
Sgit	2	2	1	2	0
SimpleCalculator	3	1	1	1	1
SimpleWeather	3	2	2	2	0
SoundRecorder	4	1	1	1	0
SunTimesWidget	13	11	7	9	3

Following our approach, 45 of the 61 broken tests were successfully repaired after the first interaction with the LLM; with two or more interactions, the number of successfully repaired tests rose to 56 out of 61, corresponding to repair success rates of 73.8% and 91.8%, respectively.

The third and fourth columns of Table II report the corresponding fix produced after each step, divided by category of fix. Table III reports instead the number of tests repaired by the LLM for each Android app.

C. RQ3: comparison with Healenium

To compare the results with a leading baseline tool for test repair, we assess the repair success rate relative to *Healenium-Appium*. The baseline tool successfully repaired 14 of 61 broken tests, for a success rate of 23.0%. It is worth noting that only one test was successfully repaired by *Healenium-Appium* and not by the LLM-based approach. The results for RQ3 are shown in Table III.

V. DISCUSSION

The results of our experimental evaluation highlight the substantial potential of integrating LLMs into the mobile GUI test repair process. Overall, the proposed LLM-based approach repaired the vast majority of broken tests, markedly outperforming the *Healenium-Appium* baseline. These findings suggest that LLMs can play a significant role in reducing the manual maintenance burden typically associated with evolving GUI test suites.

Our analysis of test failures revealed three dominant causes: (i) outdated widget locators, (ii) widgets relocated to different screens, and (iii) changes or removal of the tested functionality. Outdated locators constituted by far the most prevalent category, accounting for 76.5% of all failures, while relocated widgets and modified functionalities accounted for 10.8% and 12.7% respectively. This distribution underscores the extent to which GUI evolution impacts locator reliability, affirming prior observations on the fragility of GUI-based automated tests.

The LLM demonstrated strong capabilities in repairing tests affected by outdated locators and in correctly identifying widgets that had been moved to different screens. In most such cases, a single interaction with the LLM was sufficient for the model to infer the intended behavior and update the locators accordingly. This effectiveness appears to stem from the model's ability to reason over the view hierarchy files and align the structure of the outdated and updated GUI representations.

Healenium-Appium successfully repaired the locators of widgets displayed on the same screen in the latest version, albeit with mixed outcomes; when a test fails to locate a widget on the screen, *Healenium* attempts to identify the corresponding widget in the updated screen, but the results are not consistently satisfactory, particularly when the correspondence is complex. *Healenium* is unable to address other types of test failures, such as those involving moved widgets and altered functionalities, as it can only substitute a malfunctioning locator with a new one and cannot modify the test structure in any manner, such as interacting with another widget to navigate to a new screen or altering the sequence of operations.

Although the LLM-based method has demonstrated greater effectiveness in producing successful test repairs, it is notably slower than the baseline. The duration required to repair a test using *Healenium-Appium* can essentially be minimized to the execution of two test cases: one on the base version and another on the updated version of the application, with only a few seconds of delay when a broken locator is encountered. Conversely, the LLM-based approach necessitates time for each test to gather the XML view hierarchy files from both the base and updated versions of the application. Additionally, for every interaction with the LLM, it requires time to compose the prompt, supply the files, and execute the repaired test on the updated version of the application to ascertain whether the repair was successful. Even though some operations can be automated, the two versions of the application must be in operation to collect the corresponding XML, or it must be captured beforehand. As a rough estimate, it can be stated that a test repair using the LLM-based approach takes approximately two to three times longer than the *Healenium-Appium* baseline, with multiple interactions further exacerbating this time difference.

Nevertheless, several challenges emerged. First, locator updates involving non-unique attributes or index-based access patterns often require additional interaction. In these cases, providing the LLM with explicit descriptions of the issue and, when appropriate, screenshots of the relevant screens helped guide the model toward a correct repair. Second, tests whose failures stemmed from changes in the underlying functionality proved substantially more difficult to repair. Since the LLM only had access to view hierarchy files and screenshots—not source code or dynamic execution traces—it struggled to infer how a functionality had evolved or how a test should adapt to the new workflow. This limitation led to more multi-interaction repairs and, in some cases, unsuccessful outcomes.

Despite these challenges, the LLM-based approach sig-

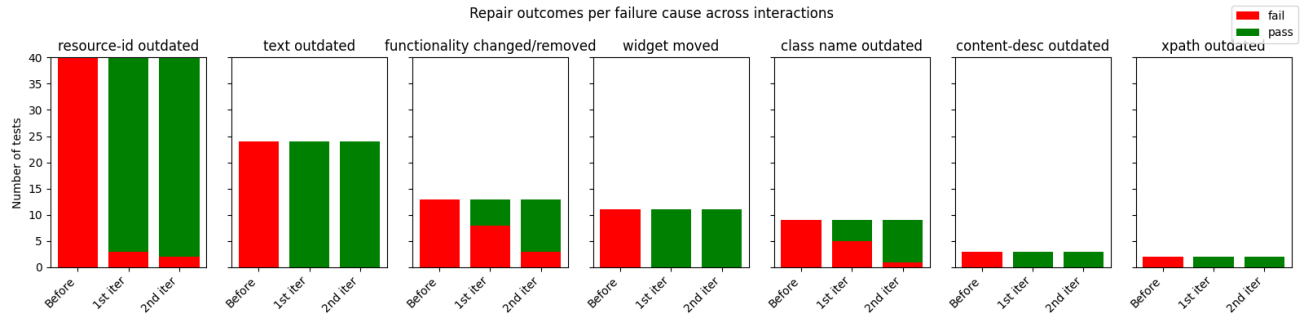

Fig. 2. Causes of failure of the tests

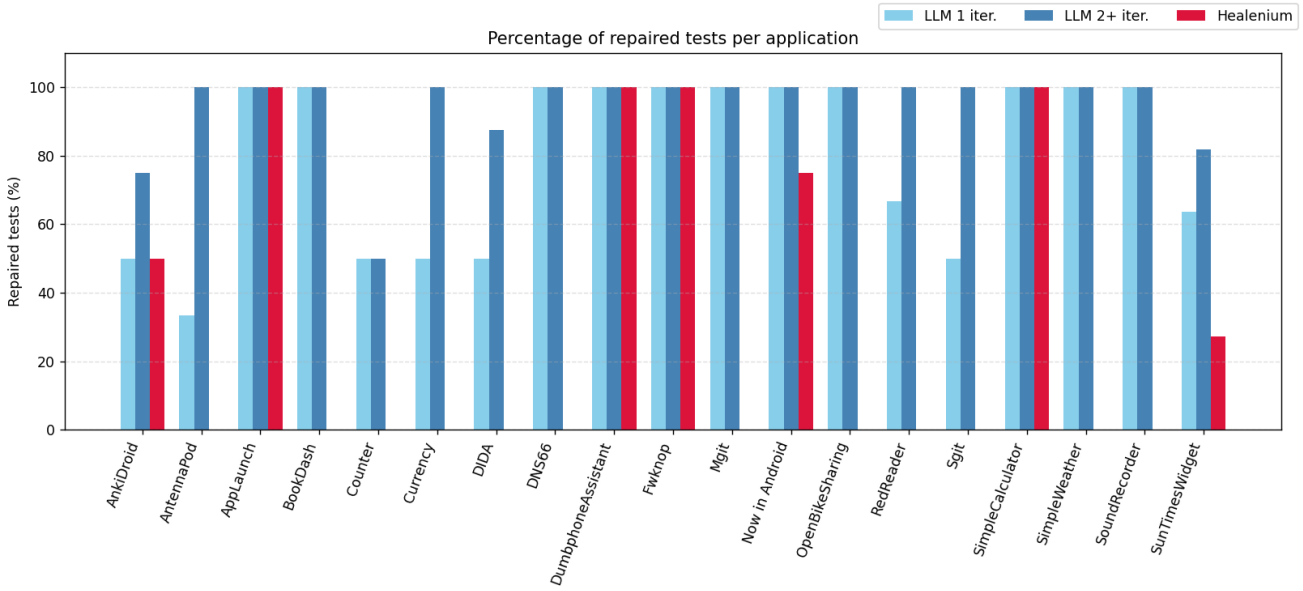

Fig. 3. Comparison of the repairs performed with LLM and Healenium

nificantly outperformed the baseline. The LLM successfully repaired 73.8% of tests after a single interaction and 91.8% after multiple interactions, whereas *Healenium-Appium* repaired only 23.0% of tests. These results clearly indicate that LLMs provide a more flexible and semantically-aware repair mechanism than traditional locator-healing techniques, particularly in scenarios involving substantial UI evolution.

A. Threats to Validity

We report the validity threats of our analysis following the criteria by Feldt et al. [28].

Internal validity threats concern whether the treatment used in the experiment is actually accountable for the observed results rather than other factors. In our case, internal validity refers to whether the LLM-based repair approach actually produces repaired tests based on the provided information, without external interference, such as previous conversations with the LLM or information extracted from other files. To mitigate this problem, we started a new conversation with the LLM and reset the model's default memory for each test to avoid interference from prior exchanges. Additionally, we relied on the web version of Copilot and provided it with only the relevant files. Another concern regarding internal validity is whether the new tests we introduced may have introduced some errors or misleading tests. To mitigate this issue, we based the test on the provided documentation and ran all tests on the base version of their application to ensure they pass on that version and maintain the application's functionality.

Construct validity threats concern whether the treatment and the outcome actually correspond to the cause and effect we are interested in. In this research, this aspect pertains to the accuracy with which we measure the LLM's effectiveness in repairing mobile GUI tests. As a mitigation, we carefully analyzed all tests to determine which were broken by running them on both versions of their application and annotating the failure causes. After the repair process, each test was executed on the updated application and carefully inspected to ensure the test's intended behavior was preserved.

External validity threats concern whether the results can be generalized outside of the scope of our study. In this research, this relates to the possibility of generalizing the findings to other Android applications or to other mobile platform contexts. To mitigate this issue, we selected multiple real-world applications for our experiment, each one with

exhaustive testing (either from the beginning or after adding the tests written by us based on the provided documentation).

VI. Conclusion and Future Work

In this work, we investigated the fragility of Android GUI tests and evaluated the effectiveness of LLMs in repairing tests that become outdated solely through graphical changes, with no functional modifications. Using a dataset of 19 real-world Android applications—each consisting of a base version, a corresponding GUI test suite, and an updated version—we first analyzed the causes of test breakage across versions. Three dominant causes emerged: outdated widget locators, widgets relocated to different screens, and changes or removals of the tested functionality, with outdated locators being the most frequent source of failure.

To address these issues, we designed an LLM-based repair approach that operates through one or more interactions. The first interaction attempts to generate a repaired test using the failed test and the XML view hierarchy files of the relevant screens. When necessary, subsequent interactions supplement this information with descriptions of the observed issues and screenshots. This iterative strategy proved highly effective, enabling the LLM to repair the large majority of failing tests.

We compared our method against *Healenium-Appium*, a state-of-the-art locator-healing tool, and found that our approach substantially outperformed the baseline in the number of tests successfully repaired. Overall, the results demonstrate that LLMs provide a powerful, flexible foundation for improving the maintainability of mobile GUI test suites.

Several avenues exist for enhancing the effectiveness and scalability of our approach. First, further prompt refinement may improve repair accuracy and reduce the number of interactions required, enabling the LLM to more reliably infer intended behavior from the available artefacts.

Second, the most challenging cases in our study involved tests whose interaction sequences had changed significantly. Because the LLM could only view hierarchy files and screenshots, it sometimes lacked sufficient information to infer the updated behavior. Incorporating richer contextual data—such as execution traces, documentation, or extended UI metadata—could help overcome this limitation.

Finally, automating the repair pipeline represents an important improvement. Our current workflow involves substantial manual effort, from test execution and artifact collection to LLM prompting and result validation, which also makes the process slower than traditional automated approaches. Introducing automation would greatly improve efficiency and move LLM-based repair closer to practical integration in continuous testing environments.

References

[1] R. Coppola, M. Morisio, and M. Torchiano, "Scripted gui testing of android apps: A study on diffusion, evolution and fragility," in *Proceedings of the 13th international conference on predictive models and data analytics in software engineering*, pp. 22–32, 2017.

[2] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, X. Che, D. Wang, and Q. Wang, "Chatting with gpt-3 for zero-shot human-like mobile automated gui testing," *arXiv preprint arXiv:2305.09434*, 2023.

[3] Y. Zhang, C. Liu, X. Xie, Y. Lin, J. S. Dong, D. Hao, and L. Zhang, "Llm-based abstraction and concretization for gui test migration," *arXiv preprint arXiv:2409.05028*, 2024.

[4] S. Yu, Y. Ling, C. Fang, Q. Zhou, Y. Zhao, C. Chen, S. Zhu, and Z. Chen, "Llm-guided scenario-based gui testing," *arXiv preprint arXiv:2506.05079*, 2025.

[5] J. Yoon, R. Feldt, and S. Yoo, "Autonomous large language model agents enabling intent-driven mobile gui testing," 2023.

[6] R. Coppola, M. Morisio, and M. Torchiano, "Mobile gui testing fragility: a study on open-source android applications," *IEEE Transactions on Reliability*, vol. 68, no. 1, pp. 67–90, 2018.

[7] S. Di Meglio and L. L. L. Starace, "Towards predicting fragility in end-to-end web tests," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pp. 387–392, 2024.

[8] M. Nass, E. Alégroth, R. Feldt, and R. Coppola, "Robust web element identification for evolving applications by considering visual overlaps," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 258–268, IEEE, 2023.

[9] S. Cao, M. Pan, Y. Pei, W. Yang, T. Zhang, L. Wang, and X. Li, "Comprehensive semantic repair of obsolete gui test scripts for mobile applications," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, (New York, NY, USA), Association for Computing Machinery, 2024.

[10] L. Ardito, A. Bottino, R. Coppola, F. Lamberti, F. Manigrasso, L. Morra, and M. Torchiano, "Feature matching-based approaches to improve the robustness of android visual gui testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–32, 2021.

[11] M. Pan, T. Xu, Y. Pei, Z. Li, T. Zhang, and X. Li, "Gui-guided test script repair for mobile apps," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 910–929, 2022.

[12] T. Xu, M. Pan, Y. Pei, G. Li, X. Zeng, T. Zhang, Y. Deng, and X. Li, "Guider: Gui structure and vision co-guided test script repair for android apps," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, (New York, NY, USA), pp. 191–203, Association for Computing Machinery, 2021.

[13] A. S. Yaraghi, D. Holden, N. Kahani, and L. Briand, "Automated test case repair using language models," *IEEE Transactions on Software Engineering*, 2025.

[14] S. Fatima, H. Hemmati, and L. C. Briand, "Flakyfix: Using large language models for predicting flaky test fix categories and test code repair," *IEEE Transactions on Software Engineering*, vol. 50, no. 12, pp. 3146–3171, 2024.

[15] S. Rahman, S. Kuhar, B. Cirisci, P. Garg, S. Wang, X. Ma, A. Deoras, and B. Ray, "Utfix: Change aware unit test repairing using llm," *Proceedings of the ACM on Programming Languages*, vol. 9, no. OOPSLA1, pp. 143–168, 2025.

[16] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, X. Che, D. Wang, and Q. Wang, "Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, (New York, NY, USA), Association for Computing Machinery, 2024.

[17] D. Gao, L. Ji, Z. Bai, M. Ouyang, P. Li, D. Mao, Q. Wu, W. Zhang, P. Wang, X. Guo, H. Wang, L. Zhou, and M. Z. Shou, "Assistgui: Task-oriented desktop graphical user interface automation," 2024.

[18] T. Rosenbach, D. Heidrich, and A. Weinert, "Automated testing of the gui of a real-life engineering software using large language models," in *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 103–110, IEEE, Mar. 2025.

[19] D. Zimmermann and A. Koziolek, "Gui-based software testing: An automated approach using gpt-4 and selenium webdriver," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pp. 171–174, 2023.

[20] Y. Zhang, W. Zhang, D. Ran, Q. Zhu, C. Dou, D. Hao, T. Xie, and L. Zhang, "Learning-based widget matching for migrating gui test cases," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, (New York, NY, USA), Association for Computing Machinery, 2024.

[21] H. Wen, Y. Li, G. Liu, S. Zhao, T. Yu, T. J.-J. Li, S. Jiang, Y. Liu, Y. Zhang, and Y. Liu, "Autodroid: Llm-powered task automation in android," in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, ACM MobiCom '24, (New York, NY, USA), pp. 543–557, Association for Computing Machinery, 2024.

[22] Y. Zhang, C. Liu, X. Xie, Y. Lin, J. S. Dong, D. Hao, and L. Zhang, "Gui test migration via abstraction and concretization," *ACM Transactions on Software Engineering and Methodology*, Apr. 2025.

[23] Y. Kim and R. Kouatly, "Self-healing automation testing with selenium and chatgpt api integration to improve the efficiency of the maintenance," in *2024 2nd International Conference on Foundation and Large Language Models (FLLM)*, pp. 339–344, IEEE, 2024.

[24] A. Fedriga, T. Fulcini, R. Coppola, D. Amalfitano, D. Distante, F. Ricca, *et al.*, "An analysis of the test repair capability of llms in android gui testing," in *Quality of Information and Communications Technology*, Springer, 2025.

[25] Y. Long, Y. Chen, C. Zeng, X. Chen, X. Chen, X. Zhou, J. Yang, G. Huang, and Z. Zheng, "Extrep: a gui test repair method for mobile applications based on test-extension," *Automated Software Engineering*, vol. 32, 04 2025.

[26] K.-J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research: a critical review and guidelines," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 120–131, 2016.

[27] T. Fulcini, G. Garaccione, R. Coppola, L. Ardito, and M. Torchiano, "Guidelines for gui testing maintenance: a linter for test smell detection," in *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation*, A-TEST 2022, (New York, NY, USA), p. 17–24, Association for Computing Machinery, 2022.

[28] R. Feldt and A. Magazinius, "Validity threats in empirical software engineering research - an initial survey.," pp. 374–379, 01 2010.