# Context-Aware GUI Testing for Mobile Applications

Bouchari Youness, Bergamini Luca, Bollo Matteo

28/02/2025

**GitHub Repository**

## 1 Introduction

Important challenges in GUI testing pertain to the area of test evolution and maintenance. As applications evolve, test suites must keep up with frequent GUI changes, which can lead to the "oracle problem": determining the expected outcomes or "oracles" of test cases in the face of these updates. This issue is often compounded by ambiguities in user interactions and dynamically changing content, making it difficult to maintain accurate and meaningful test assertions.

In this context, Large Language Models (LLMs) present a promising solution for enhancing GUI test evolution and repair by providing context-aware, adaptable insights into test oracles.

LLMs, with their advanced language understanding and reasoning capabilities, offer the potential to address the oracle problem by generating or refining test oracles based on app context, expected user interactions, and historical data. This enables more accurate detection of changes in app behavior and helps determine whether the observed changes align with intended functionality or represent defects. For instance, LLMs can analyze changes in GUI layout or text and suggest updates to assertions, adapting tests to evolving requirements and preserving test validity over time.

The key aspects of using LLMs for GUI test evolution and repair include:

Oracle Refinement LLMs can adjust expected outcomes in response to GUI changes, helping tests stay accurate and relevant.

Dynamic Assertion Adaptation By observing changes in app behavior and interface, LLMs can adjust assertions based on new layouts or content, minimizing false positives.

Test Repair and Maintenance LLMs support the automatic updating of test cases as applications evolve, reducing the time and manual effort required to maintain reliable tests.

This research aims to explore the use of LLMs in solving the Oracle problem for GUI testing, enhancing the adaptability and resilience of test suites as applications evolve. By investigating how LLMs can contribute to GUI test evolution and repair, this study seeks to advance automated testing frameworks, ultimately supporting the development of robust and reliable applications.

## 2 Research questions

How effectively can LLMs generate accurate oracles for GUI tests in response to application changes?

To what extent can LLMs adapt GUI test cases and assertions to maintain relevance across frequent interface changes?

## 3 Background

LLMs offer a unique combination of semantic understanding, contextual reasoning, and the ability to process multimodal data, making them ideal candidates for detecting and adapting to GUI changes. By integrating visual, textual, and structural features, these models can discern subtle modifications that might otherwise go unnoticed by conventional techniques. This paradigm shift toward learning-based approaches is transforming how automated testing frameworks cope with the evolving nature of modern interfaces. Below, the overall task is divided into two key stages: **recognizing GUI changes** and **migrating test cases**.

### 3.1 Recognizing Widget Changes

A major challenge in automated GUI testing is accurately identifying and interacting with UI elements despite frequent changes in layout, structure, and content. Traditional approaches rely on static locators (e.g., XPath, CSS selectors), which can become unreliable when elements are renamed, moved, or dynamically generated. To address this, recent research has explored the use of Large Language Models (LLMs) and machine learning techniques to make element recognition more adaptive.
Liu et al. [1] introduce a vision-driven multimodal LLM for automated GUI testing in mobile applications. Their approach integrates textual descriptions (e.g., button labels, UI metadata) with visual representations (e.g., screenshots, layout structures) to improve widget recognition. By combining these two modalities, their model can understand GUI elements holistically, making it more robust against UI changes, different screen sizes, and dynamic content. This method enables more reliable interaction with mobile apps, even when UI elements undergo layout modifications or textual changes.
Similarly, Nass et al. [2] focus on web applications and propose an LLM-based element localization technique that improves upon traditional selector-based

methods. Instead of relying solely on static HTML attributes, their approach analyzes the surrounding textual and structural context to determine an element's function. For example, if a button's identifier changes but its purpose remains the same (e.g., "Submit" instead of "Confirm"), the LLM can still accurately locate it based on semantic meaning. This technique makes GUI testing more resilient to changes in the Document Object Model (DOM) and reduces the maintenance burden of constantly updating test scripts.

### 3.2 Migrating Test Cases

Beyond recognizing changes, automated test scripts must also adapt when UI updates occur across different versions, platforms, or applications To tackle this issue, Zhang et al. [3] propose a learning-based widget-matching approach that enables automated test case migration. Their method uses machine learning algorithms to compare old and new UI structures and identify corresponding widgets across different versions. This is achieved through a combination of:

1. **Visual Similarity Analysis:** The system extracts visual features such as *shape, color, size, and position* of UI elements to determine how widgets in different versions correspond to one another.

2. **Text-Based Similarity:** Textual information from UI elements, such as *labels, tooltips, and accessibility descriptions*, is used to match widgets that serve the same function despite minor wording differences.

3. **Structural Relationship Analysis:** The hierarchical structure of the UI (e.g., *parent-child relationships in the UI tree*) is examined to maintain contextual relationships between elements, ensuring that migrated test cases still interact correctly with the new interface.

The extracted features are used to train a matching model that predicts correspondences between old and new UI elements. A similarity score is assigned to each potential widget pair, and high-confidence matches are used to migrate test cases automatically. Once widgets are matched, test scripts are updated dynamically to reference the new UI elements.

## 4 Methodology

Our research began with the selection of a mobile application to serve as the foundation for our experiments. After evaluating several options, we decided to focus on Omni-Notes, an open-source note-taking application, comparing two specific versions: 6.1.0 and 6.3.0. The motivation behind this choice was to analyze how the user interface (UI) evolved across these versions and to assess the impact of these changes on automated testing.
To have full control over an application being able to test the LLM in multiple ways we also developed a toy android application in 2 versions.

## 4.1 Appium for .xml extraction

The first step in our methodology involved extracting the UI structure of both versions by capturing all the XML files corresponding to each screen of the application. To accomplish this, we utilized the Appium framework, which enables automated interaction with mobile applications by connecting to a physical or virtual Android device through an Appium server. By leveraging Appium's capabilities, we were able to systematically navigate through the application and continuously capture both the XML-based UI hierarchy and high-resolution screenshots in JPG format. These artifacts provided a comprehensive representation of the application's interface at different stages of execution, allowing us to document structural changes between the two versions.
Following the extraction of UI elements, the next step was to retrieve all existing UI tests from the older version, Omni-Notes 6.1.0. These tests served as a baseline for our study, enabling us to analyze how well the previously defined test cases aligned with the newer version of the application. This step was crucial in identifying potential test obsolescence due to UI modifications, as well as assessing the extent of adaptation required to maintain test coverage for version 6.3.0.

## 4.2 Testing App

To evaluate the adaptability of our model in recognizing differences and dynamically adjusting test procedures, we developed two toy applications featuring distinct UI layouts. Each application contained a variety of elements, including an image, an input field, a title, and additional interactive components. In the second version, we intentionally altered the order of elements, modified the item IDs, and changed the content while preserving overall functionality. This setup allowed us to systematically analyze the model's ability to detect structural and semantic differences, ensuring its robustness in handling UI variations.

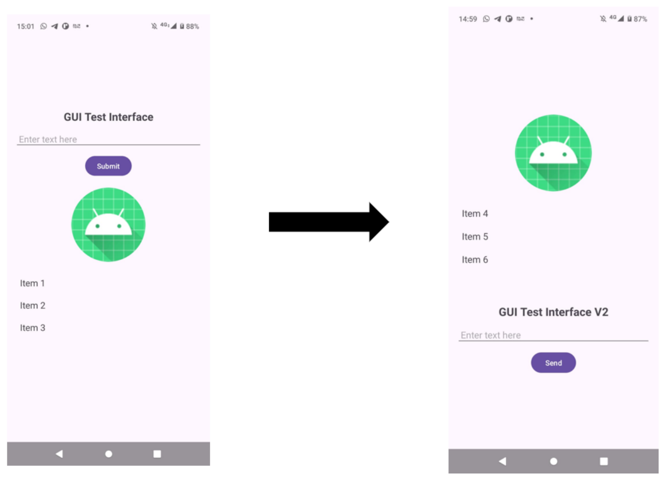

Figure 1: The 2 versions of the app.

## 4.3 LLM

Now we can create the steps that our system will take. To effectively analyze UI differences and adapt the tests, we use two calls of OpenAI's GPT-4o-mini. The first step involved structuring the necessary data for comparison. We extracted and parsed all the XML files that represent the UI structure of both application versions. These files were stored in separate directories and read programmatically to consolidate their contents. To facilitate effective processing, we converted the XML data into structured text formats, making it suitable for input into an LLM.

With these structured data in place, as we have already said, our system executed two LLM calls. The first call was designed to analyze the differences between the UI structures of version 6.1.0 and version 6.3.0. By comparing the XML representations of the two versions, the model was able to identify the modified (structural and non-structural), removed, and newly introduced UI components. This comparison provided crucial information on the necessary adaptations for test migration.

The second call to the LLM focused on generating updated UI tests for version 6.3.0. Using the differences identified in the previous step, along with the existing test cases from version 6.1.0, the model inferred how each test should be modified or rewritten. The output consists of Java test files that adhered to the original testing framework but incorporated the necessary modifications to ensure compatibility with the updated UI.

By automating this process, we minimized the need for manual test updates, significantly reducing the effort required to maintain the consistency of the user interface test across app versions. This approach demonstrated the effectiveness of LLM-based strategies in adjusting and evolving software testing processes ef-

ficiently.
In both calls, the most critical factor influencing the results was the structure and clarity of the prompts provided to the LLM. Prompt engineering played a pivotal role in ensuring that the model correctly identified UI differences and generated accurate, functional test adaptations.
For the first LLM call, the prompt was designed to systematically analyze structural and semantic UI modifications. It explicitly instructed the model to identify added, removed, renamed, and repositioned elements, as well as changes in attributes and text values. The structured approach in the prompt ensured that no critical modifications were overlooked. Additionally, we requested the output in JSON format, working with structured data.
The second LLM call was structured to generate updated UI test cases, leveraging the insights extracted in the previous step. The prompt clearly outlined the necessary modifications, ensuring that the new tests accounted for renamed, removed, or relocated UI elements while preserving the intended functionality of the original tests, passed to the prompt as a template.

# 5 Results

## 5.1 Effectiveness in identifying the changes in the GUI of the application

Evaluating the initial application of the LLM in identifying changes between different versions of an application's GUI, we opted to develop two versions of a custom GUI application rather than using one of the existing linked application repositories (as shown in Figure 1). By designing a controlled environment, we were able to systematically assess the differences detected by the LLM and verify their correctness. This approach ensured a clear and precise understanding of all actual modifications between the two versions.
After getting the detected changes from the first LLM call, we carefully analyzed them to check their accuracy. Our evaluation showed that the LLM correctly identified all modifications, including both structural changes and differences in element attributes.

## 5.2 Effectiveness in generating assertions

To evaluate the reliability of the second LLM call, we first tested our approach using a custom GUI application. This allowed us to easily verify whether the generated test cases were correctly updated based on the identified changes between versions. Using this approach, we found that the second call produced the same tests as the human-written ones.
After confirming the correctness of the approach on our custom GUI, we extended the evaluation to a real-world application by using the Omni-Notes repository. Specifically, we applied our system to versions 6.1.0 and 6.3.0 of the application, generating updated test cases for the newer version. To quanti-

tatively assess the effectiveness of these AI-generated tests, we computed precision and recall, comparing them against the human-written test cases from the repository. Precision measures the proportion of AI-generated elements that match the human-written tests, while recall evaluates how many of the relevant elements in the human tests were successfully identified by the AI.
To perform this evaluation, we tokenized the Java test files, extracting words while ignoring special characters and case differences. We then compared the frequency of common tokens between AI-generated and human-written tests to compute precision and recall for each test case. Finally, we aggregated results across all test cases to obtain overall precision and recall scores, providing a comprehensive measure of the system's ability to adapt test cases accurately.
The achieved results are:

| **Test** | **Precision One shot** | **Recall One shot** |
|---|---|---|
| CategoryLifecycleTest.java | 0.9987 | 0.8588 |
| SearchTagBackArrowTest.java | 0.8596 | 0.4558 |
| NoteLifecycleTest.java | 0.9790 | 0.6619 |
| FabCameraNoteTest.java | 1.0000 | 0.6452 |
| AutoBackupTest.java | 1.0000 | 0.8624 |
| RecurrenceRuleTest.java | 0.9956 | 0.6338 |
| SettingsActivityTest.java | 0.9932 | 0.8420 |
| RemindersLifecycleTest.java | 1.0000 | 0.6254 |
| BaseEspressoTest.java | 0.9914 | 0.7670 |
| MrJingleLifecycleTest.java | 0.9699 | 0.7655 |
| NoteListMenuTest.java | 0.8543 | 0.5244 |
| DrawerMenusEspressoTest.java | 0.9124 | 0.5208 |
| FabLifecycleTest.java | 0.9897 | 0.5884 |

- **Overall Precision One shot** : 0.9824
- **Overall Recall One shot**: 0.7339

| **Test** | **Precision Few Shot** | **Recall Few Shot** |
|---|---|---|
| CategoryLifecycleTest.java | 0.9986 | 0.9893 |
| SearchTagBackArrowTest.java | 0.8793 | 1.0000 |
| NoteLifecycleTest.java | 0.9958 | 0.9958 |
| FabCameraNoteTest.java | 0.9870 | 0.9870 |
| AutoBackupTest.java | 1.0000 | 1.0000 |
| RecurrenceRuleTest.java | 0.9959 | 0.9959 |
| SettingsActivityTest.java | 1.0000 | 1.0000 |
| RemindersLifecycleTest.java | 0.9948 | 0.9095 |
| BaseEspressoTest.java | 0.9485 | 0.9984 |
| MrJingleLifecycleTest.java | 0.9859 | 0.8108 |
| NoteListMenuTest.java | 0.9318 | 0.9919 |
| DrawerMenusEspressoTest.java | 0.9920 | 0.9764 |
| FabLifecycleTest.java | 0.9602 | 0.8977 |

- **Overall Precision Few shot** : 0.9832
- **Overall Recall Few shot**: 0.9670

## 6 Discussion

Our experimental results demonstrate that the simple agent we built performs remarkably well, achieving high accuracy and recall in detecting and adapting to UI variations. This confirms the effectiveness of using an LLM-based approach for automated UI testing, as it significantly reduces manual effort and improves adaptability. However, our approach is not without limitations. One key challenge is the reliance on predefined patterns and textual descriptions, which may struggle with highly dynamic or visually complex interfaces where contextual understanding beyond text is required. Additionally, LLMs may introduce biases or hallucinations, potentially leading to false positives or misinterpretations of UI elements. Further investigations could explore integrating vision-based models or hybrid approaches to improve robustness. While our method successfully addresses the problem within controlled scenarios, real-world applications may require fine-tuning or additional constraints to handle diverse and unpredictable UI designs. Compared to manual testing, an LLM-based approach offers a clear advantage in scalability and efficiency, making it a promising direction for automated UI analysis.

## References

[1] Zhe Liu et al. Vision-driven automated mobile gui testing via multimodal large language model. *arXiv preprint*, arXiv:2407.03037, 2024.

[2] Michel Nass, Emil Alégroth, and Robert Feldt. Improving web element localization by using a large language model. *Software Testing, Verification and Reliability*, page e1893, 2023.

[3] Yakun Zhang et al. Learning-based widget matching for migrating gui test cases. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024.