Context-Aware GUI Testing for Mobile Applications

Raffaele Martone, Michele Russo, Giuseppe Monteasi
s324807, s326522, s319203

06/07/2025

All the material, experimental results, and detailed data from this project are available at the following repository:
`https://github.com/Martons00/Project_LLM_A3`

1 Introduction

In recent years, testing Graphical User Interfaces (GUIs) for mobile applications has faced increasing challenges due to the diversity and complexity of mobile app interactions. Traditional automated testing tools often lack the adaptability to handle context-specific user interactions, dynamic content, and the wide variety of screen sizes and resolutions in mobile devices. In this context, the application of Large Language Models (LLMs) has emerged as a promising approach for enhancing GUI testing by making it more context-aware and adaptable to real-world scenarios.

LLMs, trained on extensive datasets of language and user interactions, offer the potential to understand user intent and app behavior within different contexts. This ability allows LLMs to simulate realistic user interactions and predict potential edge cases that are difficult to capture with rule-based automation alone. For example, LLMs can generate test cases that respond to contextual cues, such as location, app permissions, or user settings, and adapt these tests as app states change dynamically.

The main ideas behind using LLMs for context-aware GUI testing include:

Contextual Interaction Simulation LLMs can simulate user interactions that adapt to specific contexts, such as user profiles, permissions, and app state, offering insights into real-life usage patterns.

Dynamic Test Case Generation By generating test cases that are flexible and responsive to changes in app behavior and layout, LLMs can improve coverage and reveal defects that would typically require extensive manual testing.

The objective of this research is to investigate the potential of LLMs to support context-aware GUI testing in mobile applications, with the goal of enhancing test coverage, improving the accuracy of test results, and reducing the time and resources required for manual testing. By exploring the integration of LLMs into mobile testing frameworks, this study aims to offer insights into how AI-driven testing can support the development of more resilient and user-friendly mobile applications.

2 Research questions

RQ1: How effectively can LLMs generate working context-aware test cases for mobile GUI testing?

RQ2: What coverage can be obtained by generating test cases with LLMs?

3 Background and Related Work

3.1 Automated Mobile GUI Testing

3.1.1 Rule-based and Model-based Testing

Mobile GUI testing has traditionally relied on frameworks such as Espresso, which allow the automation of interactions with Android applications through predefined scripts. These tools enable the simulation of user actions (such as clicks, swipes, and text input) and the verification of the correct application behavior in response to such inputs. However, the manual creation of these scripts is often burdensome and not very scalable, especially in the case of complex applications or those subject to frequent updates.

A further limitation of frameworks like Espresso is the need to adopt an open box approach: writing effective tests requires an in-depth knowledge of the application's source code. This requirement implies not only direct access to the code, but also an understanding of its internal structure, dependencies, and logical flows. In real-world scenarios, documentation quality may be insufficient, and the code is often poorly readable or scarcely commented, further aggravating the difficulty of writing and maintaining reliable tests.

To overcome these issues, rule-based and model-based approaches have been developed, aiming to automate the generation of test cases by exploiting static rules or navigation models of the GUI.

3.1.2 DroidBot Framework

DroidBot is an open-source tool for the automatic generation of test inputs on Android applications, designed to be lightweight and easily usable without the need to instrument either the application or the operating system. Its main distinguishing feature is the ability to guide GUI exploration through a state transition model that is dynamically built during the app's execution [1].

Unlike tools such as Monkey, which generate completely random sequences of events, DroidBot analyzes the UI hierarchy and monitors the state of the device and the application in real time. Using this information, it constructs a directed graph in which each node represents a UI state and each edge an input event that led to a state transition. The exploration typically follows a depth-first strategy, which allows it to go deep in search of new edges and states that have not yet been visited.

Moreover, DroidBot records screenshots, the UI tree, process information, and system logs for each state, thus providing a rich basis for post-execution analysis.

Thanks to this approach, DroidBot is able to cover a greater number of states compared to random generators, making it particularly effective in identifying sensitive or malicious behaviors that only emerge under specific usage conditions [1].

3.1.3 Humanoid Framework

Humanoid is an advanced input generator for the automated testing of Android apps, designed to overcome the limitations of traditional random and model-based approaches. Its main innovation consists in integrating a deep learning model, trained on hundreds of thousands of human interaction traces (Rico dataset), to predict which GUI elements are most likely to be selected by real users and with which type of action (touch, swipe, text input, etc.)[2].

From an architectural point of view, Humanoid integrates directly with DroidBot, replacing the input selection logic: while DroidBot explores the GUI by dynamically building a graph of states (UTG), Humanoid uses its neural model to assign a probability to each possible action based on the current UI context. In particular, the Humanoid model leverages a combination of convolutional networks (to extract visual features from the UI structure) and residual LSTM (Long Short-Term Memory) modules, which enable modeling of the sequences of state transitions and capture recurring interaction patterns [2].

The model, trained on human interaction traces, generates inputs during testing. The Humanoid exploration algorithm then selects the action with the highest probability among those not yet explored, or navigates towards states with new actions to explore, thus optimizing the coverage of the most relevant functionalities from the user's perspective.

Thanks to this architecture, Humanoid is able to produce input sequences more similar to those of a real user, reaching GUI states that often elude purely random or static rule-based approaches[2].

3.1.4 Current Limitations in Context Awareness

Despite the significant progress achieved by automated testing frameworks, they still exhibit substantial limitations in their ability to dynamically adapt to the application context. In particular, the handling of textual inputs remains one of the main challenges: most tools, including DroidBot and Humanoid, are limited to inserting generic placeholder values such as "Hello World" or predefined strings, without considering the semantics required by different input fields or the variety of possible combinations. This approach significantly reduces the ability to explore alternative paths in the GUI and to detect bugs or anomalous behaviors that only emerge in the presence of specific values. The lack of context awareness and semantic generation of textual inputs highlights the need for more advanced and flexible solutions, capable of dynamically adapting the inserted values according to the structure and logic of the application under test. [5]

3.2 LLMs for Automated and Context-Aware Testing

3.2.1 Overview of LLM Applications in Software Testing

In recent years, the adoption of Large Language Models (LLMs) has revolutionized the landscape of software testing, offering new opportunities to automate and enhance numerous verification and validation activities. According to the survey [5], LLMs have been successfully employed in a wide range of tasks, including the automatic generation of test cases (both at the unit and system level), test oracle generation, debugging, automated bug repair, bug report analysis, and the generation of test scripts for GUIs and APIs [5]. In particular, test case generation and the preparation of inputs for system testing represent two of the most promising areas, where the generative intelligence of LLMs enables overcoming the limitations of traditional rule-based or heuristic approaches [5].

3.2.2 Prompt Engineering, In-Context Learning, and Advanced LLM Techniques

The effectiveness of LLMs in software testing strongly depends on the interaction techniques adopted. The survey by Wang et al. highlights that, in addition to pre-training and fine-tuning strategies, most recent studies leverage prompt engineering to guide LLM behavior towards desired outcomes [5]. The most common strategies include zero-shot and few-shot learning, where the model receives instructions or examples directly in the prompt to generate test cases, test scripts, or specific inputs. In particular, QTypist [3] demonstrates how the automatic construction of contextual prompts, enriched with local and global information extracted from the GUI, can drastically increase the quality and relevance of the generated inputs, allowing to overcome the limitations of static or heuristic approaches.

3.2.3 Context Awareness and Adaptive Test Generation

One of the main advantages of LLMs over traditional methods is their ability to understand and dynamically adapt to the application context, generating inputs and test scenarios that reflect the user's intent and the specific logic of the application. Studies such as QTypist [3] and GPTDroid [5][4] show that, by formulating input generation as a question answering or fill-in-the-blank task, it is possible to exploit the semantic knowledge and flexibility of LLMs to produce adaptive and realistic test cases. This approach makes it possible to overcome the typical obstacle of automated GUI testing, namely the inability to proceed beyond screens that require specific inputs. Moreover, the ability of LLMs to simulate realistic scenarios and to adapt test case generation based on the feedback received from the application opens up new perspectives for automated testing, both in the mobile domain and in other software domains.

3.3 LLM-Enhanced GUI Testing: State of the Art

3.3.1 GPTDroid: Functionality-Aware Decisions

GPTDroid [4] proposes an innovative approach for automated testing of mobile GUIs by formulating the problem as a Question & Answering (Q&A) task in which an LLM interacts directly with the application. The system extracts detailed semantic information from the GUI and the app, which is converted into linguistic prompts to guide the model in generating executable test scripts. A key feature of GPTDroid is the functionality-aware memory, which allows the model to maintain and reason about the progress of testing at a functional level, thus overcoming the limitations of strategies based only on recent or low-level data.

This functionality-aware memory records the functions already tested, the activities visited, and recent operations, providing the LLM with a global and long-term view of the testing process. Thanks to this, GPTDroid is able to generate more human-like and targeted action sequences, prioritizing the key functionalities of the app and significantly improving activity coverage (up to 75%, with a 32% increase over the best baselines) and bug detection (31% more) on a large set of real apps[4]. Moreover, GPTDroid supports compound actions and valid textual inputs, managing to simulate realistic and complex interactions.

3.3.2 DroidFiller

DroidFiller [6] addresses one of the main challenges in automated testing of mobile GUIs: the generation of semantically relevant and contextualized textual inputs. By integrating an LLM with a sophisticated dynamic context retrieval technique based on "function calling," DroidFiller enables the generation of specific texts for each input field, adapting to the application's domain and the simulated user profile. This approach overcomes the limitations of random or predefined inputs, allowing the insertion of values such as coupon codes or personal information in a way that is consistent with the app's logic.

Thanks to a structured prompt that includes information about the input field, the GUI state, and a reasoning template inspired by "Chain-of-Thought," DroidFiller guides the LLM to produce coherent and valid inputs. Furthermore, the ability to invoke external functions to retrieve specific data further improves the quality of the generated inputs[6].

4 Methodology

4.1 Experimental Design

4.1.1 Research Questions Mapping

Our study focuses on two main research questions that guide the entire experimental framework. The first research question (RQ1) "How effectively can LLMs generate working context-aware test cases for mobile GUI testing?" is evaluated through effectiveness metrics that distinguish between working and non-working test cases, analyzing the ability of LLMs to generate semantically correct textual inputs and valid testing actions. To answer this question, we measure the success rate of the generated tests, the quality of the textual inputs produced, and the ability to overcome screens requiring specific inputs.

The second research question (RQ2) "What coverage can be obtained by generating test cases with LLMs?" focuses on coverage metrics, including the number of GUI pages reached and the average number of interactions per page. The goal is to quantify the coverage increase achievable through LLM integration compared to baseline approaches, evaluating both the breadth (variety of explored states) and the depth (exploration depth) of automated testing.

4.1.2 Hypothesis Formulation

Our main hypotheses are articulated on two levels of analysis. The first hypothesis claims that LLM-enhanced approaches systematically outperform baseline methods in terms of effectiveness and coverage, thanks to the ability to generate contextually appropriate textual inputs and to make more informed navigation decisions. This hypothesis is based on the assumption that the semantic knowledge of LLMs can compensate for the limitations of purely heuristic or random approaches.

The second hypothesis concerns the comparative analysis between text generation and decision-making: we hypothesize that integrating LLMs for text generation has a significant impact on coverage, while using LLMs for decision-making mainly improves effectiveness through more strategic GUI navigation. This distinction is crucial to understand which aspects of automated testing benefit most from the integration of generative artificial intelligence.

4.2 Subject Applications

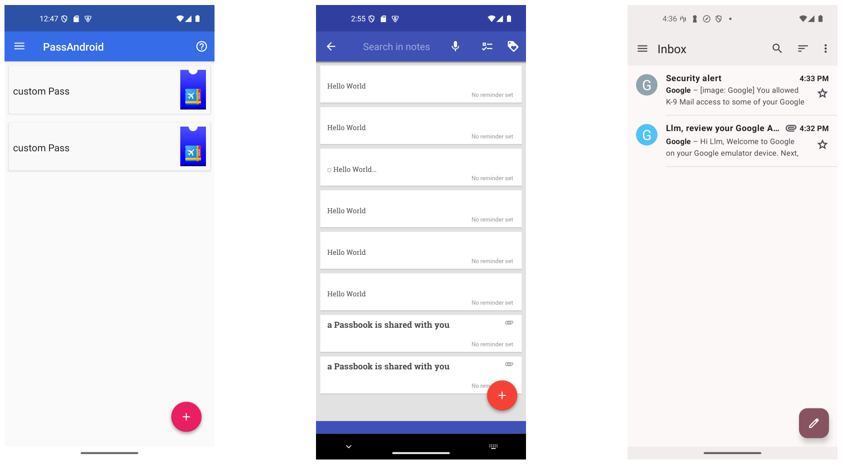
Omni-Notes Omni-Notes[1] is an open-source application for note and task management, designed to offer a simple yet powerful interface. It supports the creation, editing, and organization of textual notes, checklists, reminders, and multimedia attachments. The application includes advanced features such as categorization via tags, quick search, support for Material Design themes, and the ability to lock sensitive notes with a password. The richness of interactive

[1]`https://github.com/federicoiosue/Omni-Notes`

widgets and the presence of numerous navigation flows make Omni-Notes an ideal testbed for evaluating the coverage and quality of inputs generated by automated testing frameworks.

PassAndroid PassAndroid[2] is an open-source application specialized in managing digital passes, such as tickets, loyalty cards, coupons, and boarding passes in Passbook (*.pkpass) and esPass (*.esPass) formats. The app allows importing, viewing, and organizing various types of electronic credentials, offering barcode scanning functionality (QR, PDF417, AZTEC, Code 39, Code 128), full offline access, and an interface focused on ease of use. PassAndroid stands out for its management of dynamic data and the variety of interactions between detail views, filters, and sharing functions, thus representing a challenging context for the automatic generation of contextual tests.

Thunderbird Android (TFA) Thunderbird for Android[3] is an open-source email client focused on privacy and multi-account management. Based on K-9 Mail, it offers advanced features including unified inbox, push notification support, filters, folder and attachment management, and deep customization of settings. The application presents complex usage scenarios, both pre- and post-login, with workflows involving authentication, account navigation, message management, and asynchronous synchronization operations.



Security alert 4:33 PM
Google – [image: Google] You allowed K-9 Mail access to some of your Google

Llm, review your Google A... 4:32 PM
Google – Hi Llm, Welcome to Google on your Google emulator device. Next,

(a) PassAndroid (b) OmniNotes (c) Thunderbird

Figure 1: Blockmaps showing Apps.

[2]`https://github.com/ligi/PassAndroid`
[3]`https://github.com/thunderbird/thunderbird-android`

4.3 Testing Framework Setup

Used LLM: meta-llama/Llama-3.2-1B-Instruct

4.3.1 Environment Configuration

Our methodological approach followed an incremental development path, starting from the analysis of traditional frameworks and then evolving towards LLM-enhanced solutions. The first phase involved an in-depth study of Espresso, in order to understand the underlying mechanisms of manual test case writing and to identify the limitations of open-box approaches. This preliminary phase was fundamental to truly understand how much the subsequent approaches would change the tester's experience.

Subsequently, a testing environment based on *DroidBot* was set up as the baseline, leveraging its ability for automatic GUI exploration and detailed log generation. Modifications were made to the original *DroidBot* code to increase the likelihood of selecting `set_Text` actions, in order to place greater emphasis on our objective.

The integration architecture with *LLAMA* was designed to support both deployment via API and local execution, ensuring flexibility in the management of computational resources and in the configuration of experiments. The environment also supports the systematic collection of performance metrics, execution logs, and screenshots for each tested configuration.

4.3.2 Testing Configurations

- **Baseline Approaches:**
  - DroidBot base: standard configuration for automatic GUI exploration
  - Humanoid base: deep learning-based approach for simulating human interactions
- **Replay-Enhanced Approaches:**
  - DroidBot + Replay output: offline approach that uses LLM to post-process JSON output files, replacing generic textual inputs with semantically appropriate content and then replaying these tests with the latter
  - Humanoid + Replay output: combination of the Humanoid exploration strategy with LLM-based post-processing of textual inputs
- **LLM-Enhanced Approaches:**
  - DroidBot + LLAMA (text generation): online integration of LLAMA into DroidBot's code for dynamic generation of textual inputs during exploration

- DroidBot + LLAMA + Choice (decision making): extension of the previous approach with the use of LLAMA also for the strategic selection of the next actions to perform
- Humanoid + LLAMA: combination of the Humanoid exploration strategy with textual input generation via LLAMA

4.4 LLM Integration Strategy

4.4.1 LLM Input Generation

Ispired by [6] and [4] the LLM integration strategy focused on two complementary approaches: offline and online. In the offline approach (Replay output), the test cases generated by DroidBot are post-processed by analyzing the output JSON files to identify `set_text` events containing generic inputs such as "Hello World". These inputs are replaced with more coherent texts generated via LLM, leveraging the limited but available contextual information in the JSON metadata, such as widget identifiers, hint text, and GUI structure.

The online approach represents a significant evolution, integrating LLAMA directly into DroidBot's code for the dynamic generation of textual inputs during exploration. This integration uses prompt engineering and few-shot learning techniques to guide the model in generating content appropriate to the current context. The prompt includes information extracted in real time from the GUI, such as the type of input field, associated labels, and the application state, allowing for more precise and contextually relevant text generation compared to the offline approach.

The most advanced configuration extends the use of LLAMA beyond text generation, employing the model also for strategic decision-making in the selection of the next actions to perform. This dual-purpose approach leverages the reasoning capability of LLMs to balance the exploration of new GUI states with the in-depth testing of specific functionalities, potentially improving both the coverage and effectiveness of automated testing.

4.4.2 Prompt Engineering

To obtain high-quality textual inputs and contextual decisions, we designed specific prompts to guide LLAMA's behavior during the generation of test events. The base prompt, sent to the model, clearly specifies the assistant's role (to generate only a realistic and appropriate input for a UI element or to select the most correct action) and requires a short response, without explanations or additional formatting.

Text Input Generation For text generation, the prompt provides a representation of the current UI, including details such as the element type (e.g., `EditText`), hint text, and a screenshot of the screen. Few-shot examples are provided, showing concrete cases of expected input, such as entering an email address or filling in a search bar. The model also receives information about the context of the currently focused element (class, text, description) and must produce exclusively the text to be entered, without any additional comments.

```
<s>[INST] <<SYS>>
You are a helpful assistant designed to generate context-aware
     text inputs for UI elements.
Analyze the interface structure, element properties, and
    existing content to suggest realistic values.
<</SYS>>

### Task Description:
1. Given the current UI state and a focus element (marked as
    editable):
2. Suggest a text input that matches the element’s context.
3. The output MUST be few words, without any additional
    explanations or formatting or symbols.
4. The output MUST be a single line of text.
5. The output MUST be different from previous texts.
5. Follow these guidelines:
- Use actual data formats (emails, names, numbers) when
    detectable
- Mirror the style of existing content when applicable
- Prioritize content descriptions over placeholder texts
- Keep inputs minimal but meaningful

### Few-Shot Examples:

Example 1:
[UI Representation]
<input id=0 text=’Email address’ bound_box=10,50,310,100>Enter
     email</input>

Output:
john.doe@example.com

Example 2:
[UI Representation]
<input id=1 text=’Search bar’ bound_box=20,100,300,150>Search
    products...</input>
<button id=2 text=’Search’ bound_box=320,100,620,150></button>

Output:
wireless headphones

### Current Interface:
{full_view_representation}

### Focus Element Details:
{view_details}
```

```
### Instruction:
Based on the above context, suggest appropriate text for the
    focus element.
Provide only the raw text input without explanations, symbols
    or formatting, JUST TEXT. [/INST]

OUTPUT:
```

Action Selection For action selection, the prompt is enriched with the list of available possible actions, each described by event type, description, associated resource, and coordinates. Few-shot examples are provided, illustrating how, given a certain UI state and a list of actions, the model should respond with the index of the most appropriate action. The prompt also includes the history of previous actions, allowing LLAMA to take into account the dynamic context of the interaction.

```
<s>[INST] <<SYS>>
You are an assistant that generates only a realistic, context-
    appropriate text input for a UI element. Respond with a
    single action to take, no explanations, no formatting, no
    extra symbols.
<</SYS>>

### Few-Shot Examples:

Example 1:
[UI Representation]
<input id=0 text='Email address' bound_box=10,50,310,100>Enter
     email</input>
Possible Actions:
0: event_type=touch, desc='Show Calendar List and Settings
    drawer', resource_id=None, class=android.widget.
    ImageButton, bounds=[[11, 147], [165, 301]]
1: event_type=touch, desc='July', resource_id=com.google.
    android.calendar:id/date_picker_button, class=android.
    widget.LinearLayout, bounds=[[166, 147], [650, 301]]
2: event_type=touch, desc='Search', resource_id=com.google.
    android.calendar:id/action_search, class=android.widget.
    Button, bounds=[[661, 158], [793, 290]]
3: event_type=touch, desc='Jump to Today', resource_id=com.
    google.android.calendar:id/action_today, class=android.
    widget.Button, bounds=[[793, 158], [925, 290]]
Output:
1

UI Snapshot:
{full_view_representation}
```

```
Focused Element Details:
{view_details}

Possible Actions:
{view_possible_actions}

History of Actions:
{view_history_actions}

Now, based on the UI Snapshot, Focused Element Details,
    possible actions and History of Actions, provide only the
    appropriate index action.
[/INST]
OUTPUT:
```

Considerations These prompts were optimized to maximize the coherence and relevance of the outputs, leveraging few-shot strategies and always providing a rich and structured context. This approach made it possible to obtain concise and directly usable responses for the generation of test events, both for text input and action selection, reducing the risk of out-of-context or unusable outputs.

5 Results

5.1 PassAndroid

5.1.1 Observed Behavior

During testing on PassAndroid, both DroidBot-based and Humanoid-based configurations demonstrated the ability to simulate a wide range of typical user interactions within the application. In particular, the frameworks attempt to generate a new digital pass by filling in descriptive fields such as title, barcode, and header, leveraging text input features and GUI element selection. Advanced actions are also explored, such as printing the pass, sharing via system options, adding additional custom fields, and setting expiration dates.

The tested configurations do not limit themselves to the initial creation of the pass, but also attempt to modify it afterward, verifying the persistence of changes and the accessibility of editing functionalities. Another observed behavior is the consultation of the app's help menu, demonstrating the frameworks' ability to navigate auxiliary and informational sections of the interface.

Overall, these results highlight how DroidBot and Humanoid are able to interact with complex forms, manage the generation and manipulation of barcodes, and cover advanced functionalities such as printing and sharing, thus offering realistic coverage of the main actions a user might perform on PassAndroid.

5.1.2 Coverage and Effectiveness

Metric	**DB**	**DBR**	**DBL**	**DLC**	**H**	**HR**	**HL**
UTG state	110	110	214	77	170	170	119
UTG edge	200	200	2087	232	360	360	230
Activity coverage	4/15	4/15	4/15	5/15	4/15	4/15	5/15
Valid set text	0/66	0/66	0/31	7/182	0/224	0/224	8/289

Table 1: Coverage and effectiveness metrics for PassAndroid across all configurations.

Legenda: DB: DroidBot base, **DBR**: DroidBot + Replay output, **DBL**: DroidBot + LLAMA (text generation), **DLC**: DroidBot + LLAMA + Choice (decision making), **H**: Humanoid base, **HR**: Humanoid + Replay output, **HL**: Humanoid + LLAMA

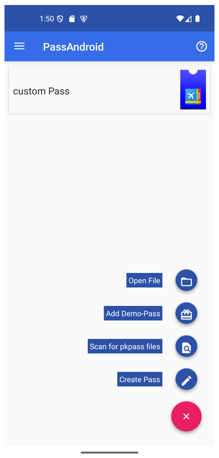
It is observed that the simple execution of DroidBot and Humanoid (both base and with replay output) allows the exploration of a limited number of states (UTG state) and transitions (UTG edge), with activity coverage reaching 4 out of 15 and no valid insertion in textual fields (*Valid set text*).

The integration of LLAMA for text generation in DroidBot shows a significant increase both in the number of explored states (214) and, above all, in transitions (2087), suggesting a greater variety of explored paths thanks to richer and more contextual textual inputs, but in reality only different set-text events are counted. However, the number of *Valid set text* remains low, suggesting that text generation alone is not sufficient to ensure the insertion of valid values in all required fields.

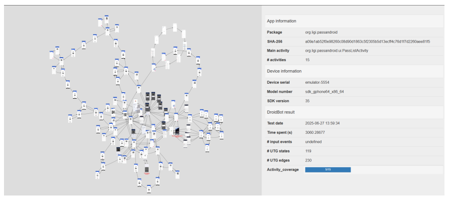
The *DroidBot + LLAMA + Choice* configuration, which also uses the LLM for selecting the next actions, instead shows an improvement in activity coverage (5/15) and in *Valid set text* (7/182), while exploring fewer states compared to text generation alone. A similar trend is observed for *Humanoid + LLAMA*, which also reaches 5 activities out of 15 and an increase in *Valid set text* compared to the baseline configurations.

5.1.3 Difference between different method

Overall, these results suggest that the integration of LLMs, especially when used for both text generation and action selection, helps to increase the depth and quality of exploration, allowing coverage of functional paths that remain inaccessible to traditional approaches. However, the percentage of validly filled textual fields is still a critical point, indicating room for improvement for full automation of semantic testing. Furthermore, UI Automator seems to have difficulty opening the hamburger menu and thus exploring all those precluded activities. Also, the lack of the possibility to load passes from storage or from the camera affects activity coverage.



(a) View of the problematic Toggle



App information	
Package	org.ligi.passandroid
SHA-256	a09a1ab52f0e98260c08d90d1863c5f2305b5d13ecff4c76d1f7d2260aee81f5
Main activity	org.ligi.passandroid.ui.PassListActivity
# activities	15
Device information	
Device serial	emulator-5554
Model number	sdk_gphone64_x86_64
SDK version	35
DroidBot result	
Test date	2025-06-27 13:59:34
Time spent (s)	3060.28877
# input events	undefined
# UTG states	119
# UTG edges	230
Activity_coverage	5/15

(b) UTG Graph of PassAndroid

Figure 2: Blockmaps showing PassAndroid.

5.2 TFA Pre-Login

The division into two phases was necessary for practical reasons: due to the impossibility of automatically bypassing the login, it was decided to distinguish between the *pre-login* phase (before access) and the *post-login* phase (after manual authentication). This made it possible to separately evaluate the exploratory capabilities of the agents in limited and interaction-rich contexts, providing a more detailed and comparable analysis of the techniques used.

5.2.1 Observed Behavior

During the *pre-login* phase, that is, before the authentication screen, a strongly limited behavior was observed for all testing methods. Due to the impossibility of autonomously bypassing the login, exploration was limited to the initial screens of the application. DroidBot and Humanoid were unable to provide valid textual inputs in the fields required for authentication, thus blocking access to deeper states. Integration with LLAMA did not produce significant improvements, as the contextual information offered by the pre-login interface was insufficient to generate coherent inputs.

5.2.2 Coverage and Effectiveness

Legenda: DB: DroidBot base, **DBR**: DroidBot + Replay output, **DBL**: DroidBot + LLAMA (text generation), **DLC**: DroidBot + LLAMA + Choice (decision making), **H**: Humanoid base, **HR**: Humanoid + Replay output, **HL**: Humanoid + LLAMA

The coverage of *activities* was very low (maximum 3/31 for DroidBot), and the

Metric	**DB**	**DBR**	**DBL**	**DLC**	**H**	**HR**	**HL**
UTG state	356	356	15	17	19	19	8
UTG edge	615	615	25	30	34	34	19
Activity coverage	3/31	3/31	2/28	2/28	2/28	2/28	2/28
Valid set text	0/70	12/70	0/10	0/20	0/9	1/9	2/21

Table 2: Coverage and effectiveness metrics for TFA Per-Login across all configurations.

number of valid textual insertions (*valid set_text*) was negligible for almost all methods. Only the replay of DroidBot showed slight effectiveness (`12/70` valid set_text), probably reusing sequences that were already effective. The UTG graphs recorded very few states and transitions: 15/25 with DroidBot+LLAMA and 8/19 with Humanoid+LLAMA, demonstrating almost no navigation.

5.2.3 Difference Between Methods

The replay in DroidBot was the only one to show a tangible improvement over the base execution. However, LLAMA, without a rich context, did not provide evident advantages. Humanoid integrated with LLAMA showed a slight capacity for textual insertion (2/21), but not enough to unlock new screens.

5.3 TFA Post-Login

5.3.1 Observed Behavior

The *post-login* phase made it possible to evaluate the behavior of the methods in realistic usage conditions, once the authentication screen was overcome. In this phase, the exploration of the interface was considerably broader. The agents were able to interact with numerous views and textual fields. Integration with LLAMA enabled more effective contextual generation, thanks to the greater amount of semantically relevant information present in the views (such as *resource_id*, *content_description*, and specific classes).

5.3.2 Coverage and Effectiveness

Metric	**DB**	**DBR**	**DBL**	**DLC**	**H**	**HR**	**HL**
UTG state	226	226	30	54	291	291	37
UTG edge	1112	1112	58	86	642	642	79
Activity coverage	3/31	3/31	2/28	2/28	2/28	2/28	2/28
Valid set text	48/101	39/101	8/19	37/93	102/2100	30/210	27/28

Table 3: Coverage and effectiveness metrics for TFA Post-Login across all configurations.

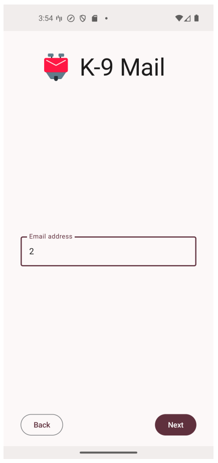
Legenda: DB: DroidBot base, **DBR**: DroidBot + Replay output, **DBL**: DroidBot + LLAMA (text generation), **DLC**: DroidBot + LLAMA + Choice (decision making), **H**: Humanoid base, **HR**: Humanoid + Replay output, **HL**: Humanoid + LLAMA

The metrics show a clear improvement compared to the pre-login phase:

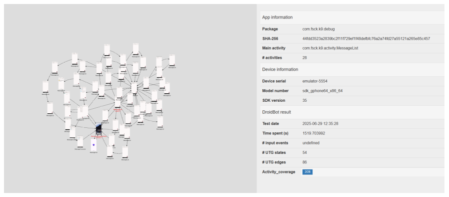
- *UTG states* and *edges* increase drastically (up to 291 states and 1112 transitions).
- *Valid set_text* increase significantly: 48/101 for DroidBot base, 37/93 for DroidBot + LLAMA + Choice, and 27/28 for Humanoid + LLAMA.
- Activity coverage settles at 2/28 for all methods, indicating more homogeneous exploration.

5.3.3 Difference Between Methods

The *LLAMA + Choice* approach, which involves the intervention of an LLM to select the most coherent strings among those generated, showed a clear improvement in the effectiveness of textual inputs, going from 8/19 (pure LLAMA) to 37/93. Replay continues to be useful (39/101), but less effective than adaptive generation. Humanoid + LLAMA achieved the best qualitative performance in terms of *valid set_text*, confirming greater sensitivity to views and greater compatibility with the strings generated by LLMs.



(a) View of the problematic Email-Field



App information

Package	com.fsck.k9.debug
SHA-256	44fdd3523a2839bc2f11f729ef1f48defbfc76a2a74fd27a55121a265e85c457
Main activity	com.fsck.k9.activity.MessageList
# activities	28

Device information

Device serial	emulator-5554
Model number	sdk_gphone64_x86_64
SDK version	35

DroidBot result

Test date	2025-06-29 12:35:28
Time spent (s)	1519.703992
# input events	undefined
# UTG states	54
# UTG edges	86
Activity_coverage	2/28

(b) UTG Graph of Thunderbird

Figure 3: Blockmaps showing PassAndroid.

5.4 Omni-Notes

5.4.1 Observed Behavior

During the experiments on Omni-Notes, the various agents (DroidBot and Humanoid, with and without LLaMA integration) showed different capabilities in exploring the main and secondary functionalities of the app. The tests reveal that all configurations are able to perform basic operations such as creating and editing notes, setting reminders (*Snooze*), managing permissions (camera and microphone), and sharing via system intents.

The advanced versions based on LLaMA, particularly the one with text generation only, enabled the exploration of more complex functionalities such as the use of the internal search bar, activation of voice dictation via *GoogleTTSActivity*, expansion of notes in detailed mode, and interaction with advanced reminder options (such as *Repeat Options*). A more sophisticated categorization of notes was also detected, highlighting greater semantic depth in the understanding of the interface.

The *Humanoid* agent, thanks to its simulation of realistic user behaviors, covered functionalities such as changing the color of categories, adding attachments (photos, files, videos) to notes, and accessing detailed views of notes (*NoteInfoActivity*), up to attempting password protection. The integration with LLaMA in Humanoid mode, however, did not lead to significant variations, maintaining behavior similar to that of the base version.

5.4.2 Coverage and Effectiveness

Metric	**DB**	**DBR**	**DBL**	**DLC**	**H**	**HR**	**HL**
UTG state	73	73	296	18	169	169	52
UTG edge	173	173	787	32	419	419	125
Activity coverage	3/15	3/15	2/15	1/15	5/15	5/15	3/15
Valid set text	50/62	24/62	25/31	31/85	91/172	100/172	74/158

Table 4: Coverage and effectiveness metrics for Omni-Notes across all configurations.

Legenda: DB: DroidBot base, **DBR**: DroidBot + Replay output, **DBL**: DroidBot + LLAMA (text generation), **DLC**: DroidBot + LLAMA + Choice (decision making), **H**: Humanoid base, **HR**: Humanoid + Replay output, **HL**: Humanoid + LLAMA

5.4.3 Difference between different method

Analyzing the quantitative data, it emerges that the *DroidBot + LLAMA (text generation)* configuration achieves the highest number of states and transitions (296 UTG states and 787 UTG edges), indicating greater exploratory depth and the ability to generate alternative paths through different textual inputs.

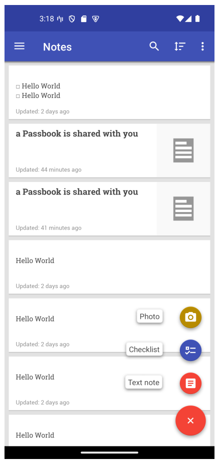
However, activity coverage remains limited (2/15), suggesting that exploration, although broad, is focused on a few repeated contexts.

In contrast, the *Humanoid base* mode (and the one with Replay) shows more extensive functional coverage (5/15 activities), with a consistent percentage of valid texts entered in the fields (*Valid set text* equal to 52.91% and 58.14% respectively), indicating realistic and varied interaction with the interface. Also in this case, integration with LLaMA in the *Humanoid + LLAMA* mode did not generate a significant impact: while maintaining valid input quality (74/158), a decrease in coverage is observed (only 3 activities out of 15).

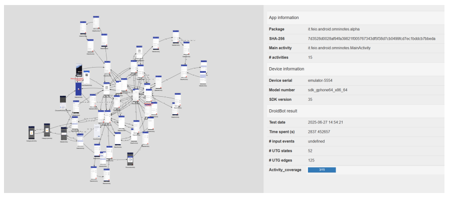
The *DroidBot + LLAMA + Choice* mode, which uses the LLM also for action selection, achieves modest results both in terms of coverage (1/15) and states explored (18 UTG states), but still inserts 31 valid texts out of 85 attempts, indicating potential in balancing exploration and semantic relevance.

In summary, LLM-oriented configurations (DBL and DLC) favor greater variety and depth in textual inputs and transitions, but struggle to extend the coverage of the app's activities. Humanoid agents, on the other hand, manage to touch more different functions, but with less depth in exploration.

A further common limitation is the inability to open some advanced interface functionalities, such as voice dictation or the loading of complex attachments (audio, external documents), due to permission restrictions or UI Automator's limitations in interacting with customized components.



(a) View of the problematic Toggle



App information	
Package	it.feio.android.omninotes.alpha
SHA-256	7d3528d0028af94fa39821f005767343df5f38d7cb0499fcd7ec1bddcb7bbeda
Main activity	it.feio.android.omninotes.MainActivity
# activities	15
Device information	
Device serial	emulator-5554
Model number	sdk_gphone64_x86_64
SDK version	35
DroidBot result	
Test date	2025-06-27 14:54:21
Time spent (s)	2837.452657
# input events	undefined
# UTG states	52
# UTG edges	125
Activity_coverage	3/15

(b) UTG Graph of Thunderbird

Figure 4: Blockmaps showing PassAndroid.

5.5 Discussion: Research Questions

5.5.1 RQ1: Effectiveness in Generating Working Test Cases

Experimental results reveal a heterogeneous picture regarding the effectiveness of LLMs in generating working and context-aware test cases. The cross-sectional analysis on the three target applications (PassAndroid, TFA, Omni-Notes) highlights distinctive patterns that show both the theoretical potential and practical limitations of the approach.

Quantitative results. The *Valid set text* metrics show variable performance across different applications. In PassAndroid and Omni-Notes, the baseline configurations (DroidBot and Humanoid) fail to generate any valid textual input (0% success), while integration with LLAMA leads to modest but significant improvements: DroidBot + LLAMA + Choice reaches 3.8% success and Humanoid + LLAMA 2.8%. Conversely, in TFA Post-Login, much higher performance is observed, with Humanoid + LLAMA achieving 96.4% success in textual inputs, compared to 47.5% for the baseline DroidBot, but this is due to the nature of the application.

Qualitative analysis of effectiveness. Although quantitative results are sometimes modest, qualitative analysis of the observed behaviors reveals advanced capabilities of LLMs. In Omni-Notes, LLM-enhanced configurations show more sophisticated interactions, such as the use of GoogleTTSActivity for voice dictation, advanced categorization with specific textual labels, and the opening of system activities such as AllInOneCalendarActivity. Similarly, in PassAndroid, LLMs guide the generation of digital passes with intelligent completion of descriptive fields such as title, barcode, and header.

Limitations masking theoretical potential. The experimental limitations identified in the document significantly mask the theoretical potential of LLMs. In particular: (1) **hardware and time constraints** limit the extensive exploration that would allow LLMs to develop a deeper understanding of the application context; (2) **poor code descriptiveness** of the apps reduces the quality of contextual information provided to the model; (3) the **limited number of applications** does not allow for assessment of the generalizability of the approach.

Theoretical potential of LLMs. From a theoretical point of view, LLMs possess ideal characteristics for context-aware testing: (1) **semantic understanding** of UI metadata and application structure; (2) **generative capability** to produce contextually appropriate textual inputs; (3) **adaptability** to different application domains without retraining; (4) **strategic reasoning** for the selection of actions based on functional objectives.

5.5.2 RQ2: Coverage of Generated Test Cases

Analysis of coverage metrics reveals complex patterns that highlight both the benefits and challenges of integrating LLMs into automated testing frameworks. The cross-sectional evaluation on multiple coverage dimensions provides significant insights into the impact of generative artificial intelligence.

State exploration metrics. LLM-enhanced configurations show diversified behaviors in exploring the state space. DroidBot + LLAMA reaches the highest number of UTG edges (2087 in PassAndroid and Omni-Notes), indicating greater diversity in explored paths, probably due to the variety of generated textual inputs. However, this does not necessarily translate into greater coverage of unique states, suggesting that the LLM generates variations on existing paths rather than discovering new functional areas.

Activity coverage. The integration of LLMs for decision-making (configurations + Choice) shows consistent improvements in activity coverage: in PassAndroid and Omni-Notes, the DroidBot + LLAMA + Choice and Humanoid + LLAMA configurations reach 5/15 activities (33.3%) compared to 4/15 (26.7%) for the baseline configurations. This suggests that the strategic reasoning capability of LLMs contributes to more effective navigation towards previously inaccessible functionalities.

Theoretical coverage potential. Theoretically, LLMs should offer superior coverage thanks to their ability to: (1) **generate diversified inputs** that unlock previously inaccessible paths; (2) **understand functional relationships** between UI elements to guide strategic exploration; (3) **adapt exploration strategy** based on application feedback; (4) **simulate realistic user scenarios** that reflect real usage patterns.

5.5.3 Performance and Resource Analysis

Analysis of performance and resource usage highlights significant challenges in the practical implementation of LLM-enhanced approaches, while revealing promising directions for future optimizations.

Computational overhead. The integration of LLMs introduces significant computational overhead due to the need to process complex contextual information and generate appropriate responses in real time. Online configurations (DroidBot + LLAMA) require API calls or local model execution for each textual input event, impacting overall execution times.

Scalability and hardware constraints. The hardware constraints identified in the document represent a significant bottleneck for the full expression of LLM capabilities. Limited memory prevents the use of larger and more sophisticated models, while insufficient computing resources limit the depth of contextual analysis.

Efficiency compared to baselines. Despite the overhead, LLM-enhanced configurations show relative efficiency in terms of the quality of tests generated per unit of time. Humanoid + LLAMA in TFA Post-Login achieves 96.4% success in textual inputs, suggesting that the additional computational investment produces significant benefits when the context is rich in semantic information.

5.5.4 Qualitative Analysis

Qualitative analysis of behaviors observed during the experiments reveals sophisticated patterns that highlight the potential of LLMs for intelligent testing, while also pointing out critical areas that require improvement.

LLM behavioral patterns. LLMs show evident contextual adaptation capabilities in different scenarios. In Omni-Notes, integration with LLAMA leads to the discovery of advanced functionalities such as voice dictation and calendar integration, suggesting semantic understanding of relationships between UI components. In PassAndroid, LLMs guide the intelligent generation of digital passes with appropriate completion of domain-specific fields.

Effectiveness in context usage. The quality of context usage varies significantly between applications. In TFA Post-Login, where the context is rich in semantic metadata, LLMs achieve excellent performance (96.4% success). Conversely, in applications with poor documentation or generic UI elements, performance degrades significantly.

Types of errors identified. Qualitative analysis reveals specific error patterns: (1) **semantic understanding errors** when UI metadata are ambiguous or insufficient; (2) **navigation errors** in complex authentication flows; (3) **generation errors** when the provided context does not contain sufficient information; (4) **strategy errors** when the LLM fails to balance exploration and exploitation.

Evidence of qualitative potential. Despite quantitative limitations, qualitative analysis clearly highlights the potential of LLMs to revolutionize automated testing. The ability to generate human-like interactions, adapt to different application domains, and understand complex semantic relationships represents a significant qualitative advancement over traditional rule-based or random approaches.

6 Discussion

In this section, we critically analyze the results of our experimentation, highlighting the main limitations encountered, possible improvements, and the implications of adopting an LLM-based approach for automated test generation.

6.1 Limitations

Despite the promising outcomes, several factors currently limit the effectiveness and generalizability of our approach:

- **Hardware and Resource Constraints:** The experiments were conducted with limited computational resources. Enhanced hardware (e.g., more RAM, faster CPUs/GPUs) could allow for more extensive exploration and faster execution, potentially improving both coverage and test quality.

- **Test Execution Time:** The duration allocated for each test session was restricted. Allowing more time for exploration could enable the tools to discover a greater variety of states and transitions, leading to higher coverage and more robust test cases.

- **App Code Descriptiveness:** Many tested apps featured poorly descriptive component names and limited documentation. Since the context provided to the LLM is derived from the app's code and UI metadata, insufficient or unclear information hampers the model's ability to generate precise and effective test actions.

- **Number of Apps Evaluated:** The current study was limited to a small set of applications. Expanding the evaluation to a broader and more diverse set of apps would help validate the generalizability and robustness of the approach.

6.2 Future Work and Methodological Improvements

To address these limitations, future investigations could focus on:

- Scaling up experiments with more powerful hardware and longer test execution windows.

- Encouraging or enforcing better coding practices and documentation in app development to enhance the quality of context available to LLMs.

- Increasing the sample size by including more apps of varying complexity and domain.

- Exploring advanced context extraction techniques to provide richer and more structured information to the LLM, potentially improving its reasoning and action selection.

6.3 Is the Problem Solved? Manual vs. LLM-based Approaches

While the LLM-based approach demonstrates clear advantages in terms of automation and the ability to generate diverse test cases without manual intervention, it does not fully solve the problem of automated test generation. The quality and coverage of generated tests are still influenced by the aforementioned limitations. However, compared to manual test creation, the LLM-based method offers significant efficiency gains and scalability potential, especially as models and supporting tools continue to improve.

Overall, integrating LLMs into the test generation pipeline represents a promising direction, but further refinement and broader validation are necessary before it can reliably replace or outperform manual approaches in all contexts.

References

[1] Yuanchun Li et al. "DroidBot: a lightweight UI-Guided test input generator for android". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017, pp. 23–26. DOI: 10.1109/ICSE-C.2017.8.

[2] Yuanchun Li et al. "Humanoid: A deep learning-based approach to automated black-box android app testing". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2019, pp. 1070–1073.

[3] Zhe Liu et al. *Fill in the Blank: Context-aware Automated Text Input Generation for Mobile GUI Testing*. 2022. arXiv: `2212.04732 [cs.SE]`. URL: `https://arxiv.org/abs/2212.04732`.

[4] Zhe Liu et al. *Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions*. 2023. arXiv: `2310.15780 [cs.SE]`. URL: `https://arxiv.org/abs/2310.15780`.

[5] Junjie Wang et al. *Software Testing with Large Language Models: Survey, Landscape, and Vision*. 2024. arXiv: `2307.07221 [cs.SE]`. URL: `https://arxiv.org/abs/2307.07221`.

[6] Juyeon Yoon et al. " Integrating LLM-Based Text Generation with Dynamic Context Retrieval for GUI Testing ". In: *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. Los Alamitos, CA, USA: IEEE Computer Society, Apr. 2025, pp. 394–405. DOI: `10.1109/ICST62969.2025.10989041`. URL: `https://doi.ieeecomputersociety.org/10.1109/ICST62969.2025.10989041`.