Evaluating LLMs for Multimodal GUI Test Generation in Android Applications

Nayse Fagundes
Federal University of Pernambuco
Recife, Brazil
nsf@cin.ufpe.br

Leopoldo Teixeira
Federal University of Pernambuco
Recife, Brazil
lmt@cin.ufpe.br

ABSTRACT

Graphical User Interface (GUI) testing is an important task in mobile application development but remains time-consuming when done manually. With the rise of Large Language Models (LLMs), there is growing interest in their potential to automate software development tasks, including GUI test generation. This study investigates the ability of LLMs to generate GUI test intentions and scripts for Android applications using multimodal inputs, such as screenshots and structured UI data. We present an approach that combines visual and textual input from eight open-source Android apps and evaluate the performance of four LLMs. The results show significant variation in the models' ability to generate GUI tests: Claude 3 Sonnet produced the most detailed and complete test sequences, GPT-4o generated simpler test scripts with fewer test intentions and user interactions, focusing on more basic user flows, while Gemini 2.5 and Gemma 3 presented moderate and similar results. These findings indicate that while LLMs can aid GUI test automation, their effectiveness varies significantly across models.

KEYWORDS

Testing, GUI, LLMs

1 Introduction

The Graphical User Interface (GUI) is a fundamental component of any mobile application, as it enables users to directly interact with the software. GUI testing ensures that visual components—such as buttons, forms, and navigation elements—function as expected [6]. Traditionally, GUI testing is performed manually by creating test cases, which is time-consuming and labor-intensive. To reduce this effort, several automated GUI testing tools and techniques have been proposed [1, 5, 22].

The emergence of Large Language Models (LLMs), with their natural language processing capabilities and reasoning [8], offers new opportunities for automation across various domains. LLMs have demonstrated promising results in code generation [20], GUI prototyping [12], and computer vision tasks [23]. These models rely on deep learning architectures, particularly transformers with self-attention mechanisms, and are trained on large-scale datasets that include diverse textual content from the web and GitHub repositories [10]. As such, LLMs have significantly influenced how developers approach automated software testing [14, 33].

Mobile apps are increasingly complex, particularly in their user interfaces, making GUI testing a growing challenge in software development [6]. Given LLMs' capabilities in understanding and generating code, it is reasonable to explore their potential for automating GUI test generation. However, it remains unclear to what extent LLMs can generate meaningful test intentions and scripts, in particular when provided with different forms of input, such as raw screenshots or structured UI metadata. Additionally, there is limited knowledge on how to structure prompts to effectively guide LLMs in such tasks.

This study is motivated by the need for more effective methods for generating GUI tests and by the potential of LLMs to automate this process. Specifically, we explore whether LLMs can leverage both visual (screenshots) and textual (UI descriptions) inputs to generate GUI test intentions and executable scripts for Android applications. Therefore, the main contributions of this work are:

- A multimodal approach that integrates screenshots and UI metadata for GUI test generation using four LLMs;
- An evaluation of the generated tests to assess the feasibility and effectiveness of LLM-based GUI test automation.

This paper is organized as follows. Section 2 presents the context and motivation for the study. Section 3 provides the necessary background. Section 4 describes the methodology. Section 5 presents the experimental setup. Section 6 reports the evaluation results. Section 7 discusses related work. Finally, Section 8 concludes the paper with future directions.

2 Context and Motivation

Mobile applications have become indispensable in daily life, offering users convenient access to services ranging from banking to entertainment. These apps are available in large numbers on online stores such as Google Play and Apple's App Store. According to Statista, as of 2024, the Google Play Store hosted approximately 2.2 million apps,[1] while the Apple App Store featured around 2.0 million apps.[2] This vast and growing ecosystem underscores the importance of ensuring app quality and usability—particularly through effective GUI testing.

Large Language Models (LLMs) have emerged as a powerful technology, driven by advances in transformer-based neural networks with self-attention mechanisms [24]. These models are designed using stacked attention layers and fully connected networks for both encoding and decoding processes, and they have demonstrated strong performance across natural language, vision, and multimodal tasks. Leading examples include GPT (OpenAI), Claude (Anthropic), Gemini (Google DeepMind), LLaMA (Meta), Mistral, and DeepSeek.

Although LLMs are primarily known for processing and generating natural language, their applications extend well beyond text. Previous studies have applied them to various tasks, such as image recognition [2, 23, 26, 31], software development [13], and GUI prototyping [12]. These developments suggest that LLMs

[1] https://www.statista.com/study/117148/google-play-store/
[2] https://www.statista.com/topics/9757/apple-app-store/

could also play a valuable role in automating GUI testing for mobile applications.

Given the increasing complexity of mobile UIs and the growing demand for automation, exploring how LLMs can support GUI test generation is both timely and necessary. This study investigates whether LLMs can reason over multimodal inputs, such as screenshots and structured UI metadata, to generate meaningful test intentions and scripts for Android apps.

3 Background

This section introduces foundational concepts needed for the study, focusing on three main areas: user interfaces in mobile applications, graphical user interfaces (GUIs) and their components, and current practices in GUI testing. These topics provide the technical context for analyzing the effectiveness of large language models in GUI test generation.

3.1 User Interface

The User Interface (UI) is the part of a mobile application through which users interact with the system. In Android applications, the UI, also known as the View Hierarchy, is structured as a tree of visual components, where each UI element is a node with parent-child relationships. These components are implemented using classes like `View` and `ViewGroup` in Android, and `UIView` in iOS.

A key aspect of the UI is the use of widgets, which are interactive components such as buttons, text fields, checkboxes, and images. Buttons trigger actions, text fields accept user input, checkboxes allow selections, and images enhance visual feedback [27].

Layout managers control how these UI components are arranged on the screen. In Android, layouts such as `ConstraintLayout`, manage alignment and positioning of UI elements, while iOS uses the Auto Layout system. These mechanisms ensure that UI elements render consistently across devices and screen sizes [19]. In Android, layouts are typically defined in XML.

Table 1 summarizes common types of user interactions, also known as UI events, which define how users engage with interface elements. The touch interactions consist of the tap, which is used to select a specific UI element, along with the double tap, which is used to tap twice to trigger an action, and the long press, that can be described as holding a touch for a while to trigger options.

In addition to basic touch interactions, gesture actions allow users to interact with the application through more complex gestures, such as pinches or swipes. The swipe is used to move a finger in a direction (left, right, down, or up), the pinch is used to zoom in or out by bringing fingers together or apart, and the drag is used to move UI elements to a new location. Finally, there is typing and input, which consists of typing text to enter it into input fields and selecting text, usually to highlight it for editing or copying [15].

Table 1: Types of actions in UI

Action Type	Actions
Touch Interactions	Tap, Double Tap, Long Press
Gesture Actions	Swipe, Pinch, Drag
Typing and Input	Type Text, Select Text

These actions trigger application behavior and form the foundation for interactive mobile experiences. Additional UI aspects such as styles—colors, fonts, and visual themes—also play a significant role in user perception and usability [15].

3.2 Graphic User Interface

The Graphical User Interface (GUI) represents the visual layer of the application through which users interact with software elements. GUIs typically include windows, buttons, icons, menus, and input components [9]. These components are often event-driven, meaning that user interactions (e.g., clicking a button) trigger specific application behavior [18].

An emerging task in this domain is UI referring, which involves generating descriptive references for UI components based on a screenshot [15]. This task plays an important role in accessibility, automation, and documentation of GUI behavior.

In Android development, Jetpack Compose[3] has become a modern alternative to XML-based UI design. It enables declarative and reactive UI construction using Kotlin,[4] a statically typed programming language supported by Google for Android development.

3.3 GUI Testing

Several approaches are used to test GUIs. Two common ones are random testing and functional testing. In manual random testing, testers rely on intuition and experience to explore application behavior [28]. Automated random testing tools—such as Monkey and MonkeyRunner—generate sequences of random UI events without requiring knowledge of the application's internal structure [3]. This black-box strategy is simple and broadly applicable.

In contrast, functional testing requires a clearer understanding of the application's intended behavior. It aims to verify that all application functions work as expected according to their specifications. This approach often requires greater human involvement and more precise test design [28].

4 Approach

This section presents our automated approach for generating GUI tests using Large Language Models (LLMs). The goal is to reduce manual effort by leveraging multimodal inputs—specifically GUI screenshots and UI metadata—to produce test intentions and executable scripts.

Our approach is guided by two research questions:

(RQ1) Can LLMs generate GUI tests from multimodal inputs (screenshots + UI metadata)?

(RQ2) How do different LLM models compare in generating the tests?

To address **RQ1**, we proposed a methodology that leverages GUI screenshots and extracted UI metadata as multimodal inputs. We designed prompt formats combining these inputs and used them to infer test intentions generated by the LLMs. We then analysed the generated tests.

To address **RQ2**, we performed a quantitative analysis of the test generation performance of each LLM model. For this, we computed the mean and standard deviation of test intentions generated, total

[3] https://developer.android.com/compose
[4] https://kotlinlang.org/

actions, and total UI elements covered across the evaluated applications. In addition, we evaluated the generated test scripts using the following information: number of test methods, UI interactions, user actions, and lines of code (LOC).

The methodology is organized into two main stages: (1) data preparation, including screenshot and UI element extraction, and (2) test generation using structured prompts. We describe each of these components in detail below.

Figure 1 illustrates the first stage of the overall methodology. We selected a diverse set of Android application packages (APKs) available in their official GitHub repositories. The selection was guided by the aim of including applications with varying interface functionalities, ensuring that our approach could be evaluated across different scenarios.

Once selected, each APK was downloaded and imported into the Android Studio Integrated Development Environment (IDE), which provided the necessary tools for interacting with the application interfaces.

We used UIAutomator Viewer (UIViewer) to extract structured information about the UI components present on each screen. This tool enabled us to retrieve precise details about the hierarchy, layout bounds, component types (e.g., buttons, text fields, checkboxes), resource identifiers, and other relevant properties of the interface elements.

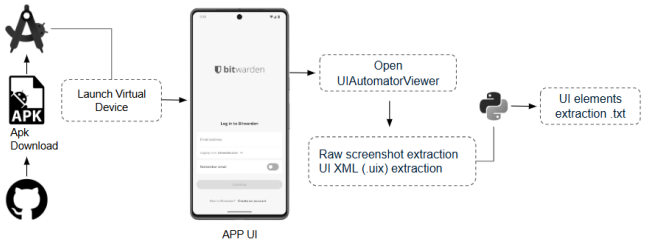

Figure 1: Screenshot and UI information extraction process

Figure 2 shows the second stage, which focuses on generating tests based on the previously collected and structured data. We constructed two prompt formats, one designed to generate GUI test intentions, capturing the user's goals and expected interactions with the application, and another prompt to produce executable test scripts.

Both prompt types were constructed to include a set of inputs, such as a screenshot, structured information about the UI elements present on the screen, and a textual task description. By combining visual, structural, and semantic information, these prompts were designed to guide the large language models to generate the tests.

To address **RQ1**, we measure the number of distinct test intentions and the number of actions per intention. For **RQ2**, we analyze the number of test methods (@Test annotations), UI interaction coverage (onView() calls), user action simulation (perform() calls), and lines of code (LOC).

5 Experiments

This section presents the experimental setup used to evaluate our approach for GUI test generation. We describe the selection of

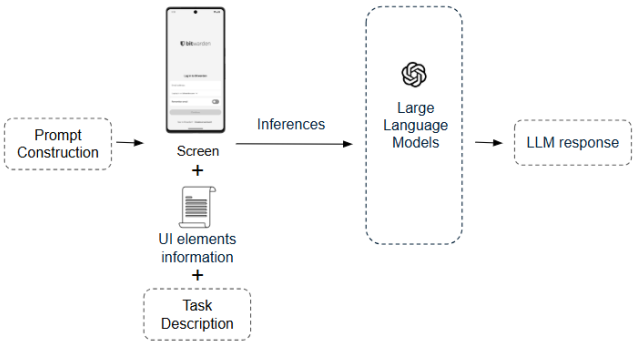

Figure 2: Prompt construction and inference process for GUI test generation

applications, data extraction procedures, prompt design, model selection, and the inference process. The goal is to assess the ability of different LLMs to generate GUI test intentions and scripts based on multimodal inputs.

We selected eight Android open-source mobile applications (Table 2) to ensure a diverse range of domains. The selection criteria included:

i. Applications from different categories, to generalize the results;
ii. Open-source licenses, allowing free access to UI assets and APKs.

Priority was given to recently updated apps to better reflect the current Android development ecosystem. After downloading the APKs from GitHub, each app was imported into Android Studio and executed to capture representative GUI screens. Screenshots were taken using the emulator's image capture tool.

The selected screens represent a variety of common and functionally diverse app scenarios to comprehensively evaluate our GUI test generation method. They include functionalities such as capturing photos (Open Camera), note-taking (Joplin Notes), fitness tracking (OpenTracks), email management (K-9 Mail), educational content creation (Ankidroid), secure password access (Bitwarden), event scheduling (Calendar), and financial transactions (Finance Management).

This selection of screens ensures coverage of diverse interaction patterns and UI elements, reflecting real-world user activities across multiple application domains, thereby enhancing the relevance and robustness of the testing evaluation. UI information was extracted using Android's UI Automator Viewer, which outputs a `.uix` XML file describing the screen's UI hierarchy. From this file, we focused on extracting relevant elements such as buttons, input fields, and navigational components. A custom Python script was developed using the `xml.etree.ElementTree` library[5] to parse and extract structured data, which was saved in `.txt` files.

The result was a dataset consisting of eight mobile applications, each represented by a single screenshot and a corresponding textual description that highlights its most significant user interface (UI) components. For each application, the screenshot provides a

[5]https://docs.python.org/3/library/xml.etree.elementtree.html

visual reference of the app's graphical layout, while the accompanying description outlines the key UI elements present in the image, such as buttons, input fields, navigation bars, and other interactive components. We cannot rely on public datasets composed solely of GUI images, because while useful for visual reference, they lack the underlying source code, dependencies, and detailed UI metadata required to successfully execute and validate generated test scripts.

Table 2: Selected open-source Android apps and their screens used in the experiments.

App	Category	Screen
Open Camera	Photo	Camera shot screen
Joplin	Notes	New note/to do screen
OpenTracks	Fitness	Home screen
K-9 Mail	Email	Inbox screen
Ankidroid	Education	New card screen
Bitwarden	Password	Login screen
Calendar	Organization	New Event
Finance M.	Finance	New transaction

5.1 Prompt Constructions

To guide the LLMs in generating GUI tests, we designed two distinct types of prompts: one for generating high-level test intentions, and another for producing executable test scripts. Both prompts combine visual input (a GUI screenshot) with structured textual input (UI element descriptions) and a task instruction.

In a typical scenario, a GUI test is designed to target a specific screen, verify the presence of UI components, simulate user actions, and validate expected behaviors. Our multimodal prompts reflect this structure by integrating visual and textual representations of the screen to enable LLM reasoning.

5.1.1 Intentional Test Prompt

The goal of this prompt format is to have the model interpret the screenshot and UI element structure, identify the screen's core functionality, and generate test intentions expressed in natural language. Each test intention corresponds to a meaningful user goal, described as a logical sequence of interface actions.

Table 3: Intentional GUI Test Prompt Structure

Prompt 1
`{Screenshot} + {UI Elements} +` **`Task Description`**
Task Description
Analyse the GUI screen and the UI elements of the application, identify the primary functional objective of the screen. Generate GUI intentional tests with actions and a logical sequence of user actions to validate the main purpose of the screen.

For example, in the case of a login screen, the model is expected to describe a sequence of steps such as entering credentials and clicking a login button, expressed in natural language. Table 3 presents the prompts used for the models.

5.1.2 Script Test Prompt

This second prompt variant instructs the LLM to generate actual GUI test code in Espresso, a framework for writing Android-based UI tests, using the same inputs. It is intended to evaluate whether the model can produce syntactically and semantically correct test scripts aligned with Android development practices.

For example, for a login interface, the script should include code to locate and interact with the username and password fields, and trigger the login button. The intent is to demonstrate if the script shows an accurate understanding of the screen's purpose and a coherent sequence of actions. Table 4 presents this specific prompt format.

Table 4: Script GUI Test Prompt Structure

Prompt 2
`{Screenshot} + {UI Elements} +` **`Task Description`**
Task Description
Analyse the GUI screen and the UI elements of the application, identify the main objective of the screen. Generate a GUI test script in Espresso to validate only the core functionality of the screen.

5.2 Model Selection

We evaluated four state-of-the-art multimodal LLMs. Three models, namely GPT-4o (OpenAI), Claude 3 Sonnet (Anthropic), and Gemini 2.5 (Google), were accessed via their applications, which are accessed through browsers and utilize the models' APIs. The fourth, Gemma 3 (12B), is an open-source model that was executed locally without fine-tuning.

This selection reflects a balance between proprietary based models and open-source alternatives, enabling a comparison of their general performance under equal conditions. All models were accessed via web browsers and used in their standard configurations without additional training or prompt tuning. Table 5 presents the models and their respective versions used in this study.

Table 5: Multimodal LLMs used

Model	Version	Parameters	Other Details
GPT-4o	2025	Proprietary	Accessed via OpenAI
Claude 3 Sonnet	2025	Proprietary	Accessed via Anthropic
Gemini 2.5	2025	Proprietary	Accessed via Gemini
Used Locally			
Gemma	3	12B	Local execution

5.3 Generation Procedures

For each app and associated screen, we submitted two prompts to each of the four models, one for test intentions and one for test scripts. This resulted in two inferences per screen, per model. In total, the experiment generated 8 apps × 2 prompts × 4 models = 64 inference outputs.

Prompts for GPT-4o, Claude 3 Sonnet, and Gemini 2.5 were submitted via a web browser that interact with the models through

their APIs. The open-source Gemma 3 model was run locally using the LM Studio.[6]

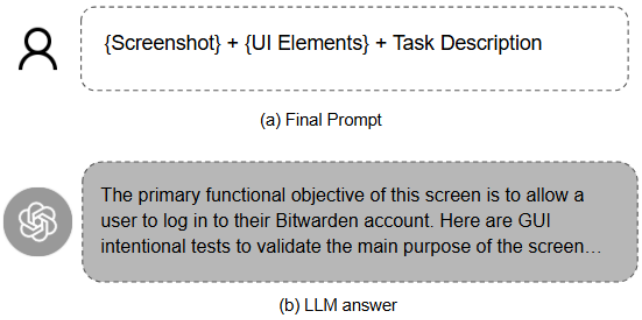

Figure 3: Inference process with LLM models

Figure 3 presents how the inferences were done when using the API approach. The outputs of each inference were saved in plain text files and used as input for the evaluation phase described in the next section.

6 Evaluation

This section presents the evaluation of the GUI tests generated by the LLMs based on the two prompt types introduced earlier: intentional tests and executable scripts. For each type, we define relevant metrics, analyze model outputs, and compare performance across models. The analysis highlights strengths and limitations of the generated content.

6.1 Intentional Test Prompt

To evaluate Prompt 1 outputs, we defined two main attributes: the number of distinct test intentions and the number of user actions described for each intention. These metrics reflect how well the model understood the functional goals of a given GUI screen and how thoroughly it mapped them to interaction sequences.

In this section, we describe the procedure used to evaluate the results generated by the models. First, the outputs obtained from the prompts related to GUI test intentions generation were evaluated based on three aspects: the test goal (test intention) and the individual interactions with the interface per intention (actions) described in Table 6.

Table 6: Evaluation attributes for GUI test intention generation

Attribute	Description
Test Intentions	Distinct functional goals described in natural language
Actions	User interactions per intention (e.g., tap, type)

Figure 4 shows a test intention generated by Gemini 2.5 for the Bitwarden app: "*Successful Login with Valid Email.*" The corresponding user actions (Figure 5) detail the expected interaction sequence, including filling out the form and submitting credentials.

[6]https://lmstudio.ai/

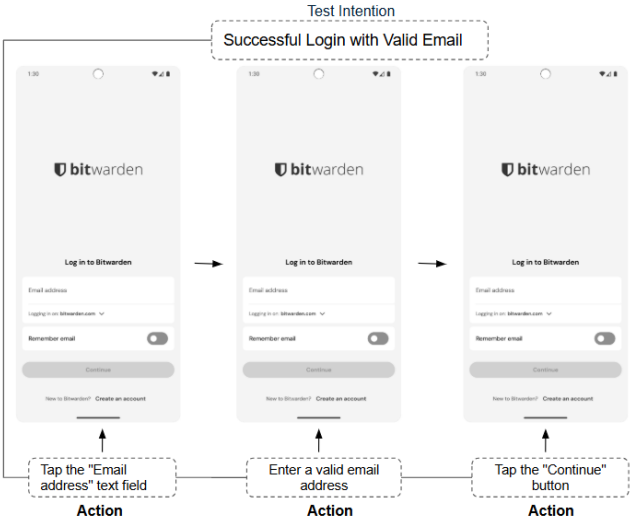

Figure 4: One test intention of the Bitwarden application

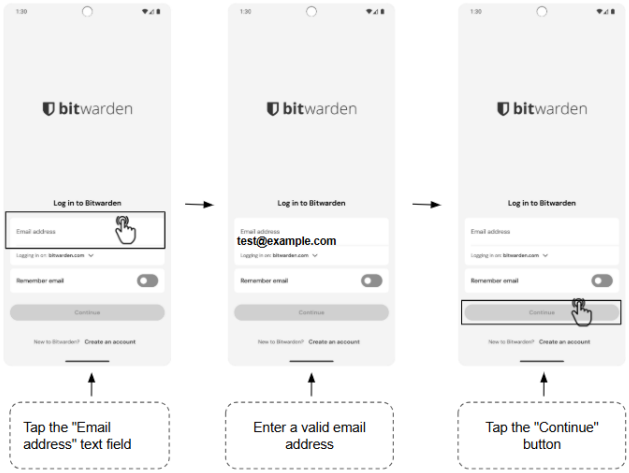

Figure 5: Actions of the test intention

Table 7 summarizes the average number of test intentions and actions generated per app, as well as totals across all 8 apps. The results in Table 7 show that the models differ in their GUI test generation capabilities, both in terms of their mean values and the standard deviation (SD), which reflects the variability of results.

Table 7: Mean, standard deviation, and total counts of the tests generated from Prompt 1 across different models.

Model	Test Intentions	Total Actions	Total Elements	Apps
	mean ± SD	mean ± SD	TI / TA	
GPT-4o	6.8 ± 0.9	14.3 ± 7.7	55 / 115	8
Claude 3 Sonnet	9.3 ± 1.8	52 ± 14.1	75 / 420	8
Gemini 2.5	5.8 ± 1.5	12.3 ± 7.1	47 / 99	8
Gemma 3	5.0 ± 0.7	18.5 ± 5	40 / 148	8

The results of the models used show differences in their test generation capabilities. Claude 3 Sonnet demonstrated the best performance, with a total of 75 GUI test intentions and 420 actions across 8 mobile applications. This corresponds to an average of 9.3 intentions per app and 52 actions per intention, indicating a tendency to produce more detailed actions per GUI test intentions, with the capacity of covering more screen functionalities.

Gemini 2.5, generated 47 GUI test intentions and 99 actions, also across 8 applications, with the average of 5.8 test intentions and 12.3 actions suggesting a less detailed generation result. GPT-4o generated 55 intentions and 115 actions in total, which results in an average of 6.8 intentions and 14.3 actions per intention, indicating that the tests are more direct.

In the case of the open-source model Gemma 3, which was executed locally, the results showed a total of 40 GUI test intentions and 148 actions across the 8 mobile applications. Although the number of test intentions generated by Gemma 3 is close to the Gemini 2.5, the average number of actions per intention is higher. This may suggest that Gemma 3 is capable of producing detailed action sequences, even if the overall test intentions appear more limited.

Model	Result
GPT-4o	Create a New Note Action: Tap the "New note" button at the bottom
Gemini 2.5	Verify "New note" button functionality Action: Tap the "New note" button
Claude 3 Sonnet	New Note Creation Actions: 1. Locate "New note" button (blue button, bottom-right) 2. Tap "New note" button 3. Verify note editor opens 4. Enter sample text 5. Save the note 6. Verify note appears in notes list
Gemma 3	Create a New Note Actions: Locate and click on the "New note" button (tag="button", bounds="[768,2833][1232,2973]"). Verify that a screen/dialog opens allowing the user to enter text for the new note. Enter some text into the note field (e.g., "Meeting notes"). Save the note. Verify that the new note appears in a list or view on the main screen.

Table 8: Comparison of generated GUI test intention by the 4 LLM models for Joplin note creation.

Table 8 presents an example a of GUI test intention generated by each model for the task creating a new note in the Joplin application. This comparison helps illustrate the results differences of each models. While all models identified the action of tapping the "New note" button, their descriptions are different. The output generated by Gemma 3 stands out for its explicit inclusion of UI element metadata, such as the element's screen coordinates (bounds="[768,2833][1232,2973]").

This suggests that Gemma 3 had a deep understanding of the UI elements information during the test generation. The model Claude 3 Sonnet also demonstrates a detailed result of the GUI test intention.

6.2 Script Test Prompt

Test scripts generated by the LLM models are focused in the Espresso framework, known for automated UI testing on Android. Since the generated scripts follow a basic Espresso structure, our analysis focuses on the main parts that are common in this testing framework. The following code block is an example of a GUI test script of the Calendar application automatically generated by GPT-4o using the Espresso testing framework without the comment and libraries lines.

```
@RunWith(AndroidJUnit4::class)
class CalendarCoreFunctionalityTest {

    @get:Rule
    val activityRule =
    ActivityScenarioRule(MainActivity::class.java)

    @Test
    fun testCreateNewTaskFromCalendar() {
        onView(withId(R.id.month_view))
            .check(matches(isDisplayed()))

        onView(withId(R.id.fab_task_icon))
            .perform(click())

        onView(withText("New Task"))
            .check(matches(isDisplayed()))
    }
}
```

In this example, the script defines one test case, as indicated by the annotation that follows.

```
@Test
fun testCreateNewTaskFromCalendar()
```

This is a standard JUnit annotation used in Espresso tests to identify test methods. The method `testCreateNewTaskFromCalendar()` specifies a sequence of interactions to validate the core functionality of creating a new task in the calendar interface.

Within the script, there are three calls to the `onView()` function.

```
onView(withId(R.id.month_view))...
onView(withId(R.id.fab_task_icon))...
onView(withText("New Task"))...
```

The `onView(...)` method is used to locate UI elements. Each call typically checks for visibility or performs an action, providing a proxy for estimating test coverage. Therefore, the number of distinct `onView(...)` calls serves as an indicator of how much of the UI is covered by the test. In this specific case, three.

The `perform(...)` method is used to simulate user actions such as clicks, typing, or swiping. In the example above, one action is performed: a click on the floating action button used to add a new task.

Executable Lines of Code (LOC) refer to all non-empty lines that are not comments (i.e., do not begin with `//`). In this case, the script contains 15 executable LOC.

These recurring components—test methods, UI interactions, user actions, and LOC—form the basis of our evaluation. Specifically, we focus on:

- **Number of `@Test` annotations**, representing the number of test cases;
- **Number of distinct `onView()` calls**, as a proxy for UI interaction coverage;
- **Number of `perform()` calls**, reflecting user action simulation;
- **Executable LOC**, indicating the overall size of the test script.

To automate this analysis, we developed a Python script that parses the generated code to extract the metrics above. Table 9 presents a comparison of the results across the four evaluated LLMs and eight mobile applications.

Table 9: Comparison of GUI test script generation across the models

Metric	Claude 3 Sonnet	Gemini 2.5	Gemma 3	GPT-4o
@Test methods	42	11	15	8
UI interactions	200	52	37	32
User actions	79	31	29	20
LOC	1,027	254	245	222

Table 9 shows that GPT-4o produced the shortest and most concise test scripts, generating 8 test methods, 32 UI interactions, 20 user actions, and 222 executable lines of code. In contrast, Claude 3 Sonnet generated the most extensive test scripts, with 42 test methods, 200 UI interactions, 79 user actions, and 1,027 lines of code—demonstrating broader functional coverage.

Gemini 2.5 falls between GPT-4o and Claude 3, with 11 test methods, 52 UI interactions, and 254 lines of code. While less comprehensive than Claude 3, it achieved more interaction coverage than GPT-4o.

The open-source model Gemma 3 generated 15 test methods, 37 UI interactions, 29 user actions, and 245 lines of code—results that are comparable to those of Gemini 2.5. Although Gemma does not reach the coverage depth of Claude 3, it demonstrates a strong capability for producing structured GUI test scripts without relying on a proprietary API. As shown in Figure 6, Gemma offers a compelling open-source alternative for automated GUI test generation.

6.3 RQ1: Multimodal GUI Test Generation

To address RQ1, we proposed a multimodal methodology that leverages GUI screenshots and extracted UI metadata as inputs to LLM models for generating comprehensive GUI tests.

Our results demonstrate that this approach successfully enables LLMs to understand mobile application interfaces and generate meaningful tests. Claude 3 Sonnet showed superior performance indicating the highest capability for identifying diverse GUI scenarios.

6.4 RQ2: Performance Analysis of LLM Models

Our quantitative analysis revealed significant differences in test generation performance among the four LLM models across both test intentions and automated script generation. For test intentions, Claude 3 Sonnet generated the most comprehensive results, while Gemini 2.5 produced more concise outputs.

In script generation, Claude 3 Sonnet again demonstrated the highest coverage, compared to GPT-4o's more accentuated approach. The open-source model Gemma 3 showed competitive results, proving to be a viable alternative to proprietary models for automated GUI test generation.

6.5 Threats to Validity

While this study provides valuable insights into the potential of LLMs for GUI test generation, threats to validity must be considered.

Generalizability. The study was conducted on a limited set of eight Android applications. As a result, the findings may not generalize to other types of mobile apps, platforms (e.g., iOS), or enterprise-scale systems with more complex UIs and workflows.

Execution Environment. All screenshots and UI metadata were captured using the Android Studio emulator. While this provides a consistent environment for testing, it may not fully reflect real-world behavior on physical devices, especially in cases involving gesture-based interactions, latency, or hardware-specific UI rendering.

Stochastic Nature of LLMs. The outputs of large language models can vary between executions due to their inherent nondeterminism. Although we used consistent prompts and configurations to minimize variability, the results may still reflect the probabilistic nature of LLM reasoning.

Static Evaluation of Scripts. We relied on static analysis of the generated test scripts. Due to the complexity of certain apps, missing dependencies, and compilation issues, it was not feasible to integrate and run all generated tests in a working Android project. As a result, we did not evaluate whether the scripts could be compiled and executed successfully, but rather whether they were syntactically correct and structurally sound.

Prompt Design Limitations. The prompts used in this study were designed to be general-purpose and consistent across models. However, fine-tuned prompts tailored to each model's capabilities might yield better results. Our choice to use unified prompts supports comparability but may underutilize specific strengths of individual models.

Despite these limitations, the results provide initial evidence of the capabilities and limitations of current LLMs for GUI test automation, particularly in multimodal contexts.

7 Related Work

This section reviews prior work at the intersection of software testing, code generation, and the use of large language models. We group related efforts into three areas: (i) LLMs for code generation, (ii) LLMs for test generation, and (iii) LLMs for GUI-related tasks.

7.1 LLMs for Code Generation

Since the introduction of transformer architectures [24], a wide range of models has been developed for code generation tasks. One prominent example is CodeGeeX [32], a multilingual code generation model with 13 billion parameters trained on diverse programming languages. CodeGeeX demonstrated strong performance on code completion and translation tasks, outperforming other models of similar scale.

Codex, developed by OpenAI [7], is another notable example. Trained on public GitHub repositories, Codex powers tools like

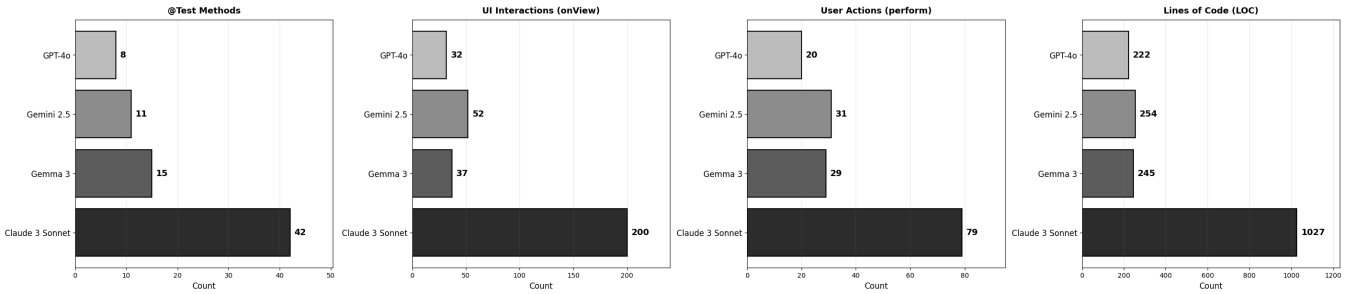

Figure 6: Comparison of GUI test script generation across the models

GitHub Copilot and supports multiple programming tasks, including function generation, code summarization, and unit test synthesis. Its performance was evaluated using benchmarks like HumanEval, showing high accuracy in generating functional Python code.

Yu et al. [30] proposed CoderEval, a benchmark with Python and Java code generation tasks selected from open source projects, focusing on evaluating the effectiveness of code generation models on real code generation tasks. The evaluation process included automatically assessing the functional correctness of generated code. The results demonstrated that the effectiveness of models in generating functions was higher than non-code functions.

Austin et al. [4] proposed the benchmark MBPP4 (Mostly Basic Python Programs) and MathQA-Python to evaluate the ability of LLMs to synthesize short Python programs from natural language descriptions. The benchmark was designed to assess the models' performance in few-shot and fine-tuning approaches.

These advances highlight the potential of LLMs to support software development workflows, including test automation.

7.2 LLMs for Test Generation

Several studies have explored the use of LLMs for test generation, particularly from natural language inputs such as bug reports or requirements.

Kang et al. [11] proposed LIBRO, a framework that uses LLMs to automatically generate test cases from bug reports. Their method focuses on reproducing software bugs using code generated by Codex and evaluated on open-source Java projects with JUnit tests.

Siddiq et al. [21] extended this work by comparing the performance of Codex, GPT-3.5-Turbo, and StarCoder on unit test generation. Their experiments used the HumanEval and EvoSuite SF110 benchmarks and found that context-aware prompt engineering significantly improves test quality.

These studies demonstrate that LLMs can effectively synthesize tests from high-level descriptions, but most previous work focuses on unit tests rather than GUI-level testing.

7.3 LLMs for GUI

LLMs have also been applied to GUI-related challenges. Liu et al. [17] introduced HintDroid, a system that uses LLMs to automatically generate hint text for input fields in Android apps. This is particularly beneficial for improving accessibility, such as for users with visual impairments.

Beyond the scope of accessibility, there has been progress in employing LLMs to automate GUI testing in mobile applications. Yoon et al. [29] propose DROIDFILLER, an LLM-based input generation technique designed to address these challenges in an industrial vendor testing context, including both in-house and third-party mobile apps.

Similarly, Wang et al. [25] developed LLMDroid, a testing framework that improves automated mobile GUI testing by combining existing tools with LLMs. The framework operates WITH autonomous exploration, where traditional tools explore the app and LLMs summarize visited screens.

Liu et al. [16] proposed a framework for mobile GUI testing that incorporates awareness of app functionalities to guide testing decisions. The method aims to mimic human interaction patterns, improving the quality and relevance of automated tests by making them more informed and targeted.

While HintDroid and focus primarily on accessibility, other approaches such as DROIDFILLER, and LLMDroid have applied LLMs to automated GUI testing. Our study explores a different dimension—automating the generation of GUI test intentions and scripts from multimodal inputs.

To the best of our knowledge, our work is among the first to evaluate the effectiveness of multiple LLMs in GUI test generation using both screenshots and structured UI data.

8 Conclusions

In this study, we investigated a multimodal approach to automate GUI test generation using large language models (LLMs). By combining screenshots and structured UI element data from eight open-source Android applications, we evaluated the performance of four LLMs—Claude 3 Sonnet, GPT-4o, Gemini 2.5, and Gemma 3—in generating both natural language test intentions and executable test scripts in Espresso.

Our findings show significant variation in performance across models. Claude 3 Sonnet consistently produced the most comprehensive and detailed outputs, both in terms of the number and quality of generated test intentions and scripts. GPT-4o, while effective, generated simpler and more concise tests. Gemini 2.5 achieved moderate results, and the open-source model Gemma 3 demonstrated competitive performance despite operating without fine-tuning.

These results indicate that LLMs are capable of supporting GUI test automation tasks, but their effectiveness depends strongly on

the specific model used. Our evaluation suggests that prompt design, input richness, and model architecture all play important roles in determining output quality.

For future work, we plan to explore:

- Support for multi-screen test flows and more complex user interactions;
- Real-device testing and runtime validation of generated scripts;
- Fine-tuning models on GUI testing tasks or incorporating feedback-driven refinement;
- Broader application domains beyond Android to assess generalizability.

ARTIFACT AVAILABILITY

All artifacts related to this study, including screenshots, UI metadata, generated test outputs, and analysis scripts, are available in the following repository: https://zenodo.org/records/15806670.

ACKNOWLEDGMENTS

Leopoldo Teixeira was partially supported by CNPq (315532/2021-1). We also acknowledge support from INES.[7]

REFERENCES

[1] David Adamo, Md Khorrom Khan, Sreedevi Koppula, and Renée Bryce. 2018. Reinforcement learning for android gui testing. In *Proceedings of the 9th ACM SIGSOFT international workshop on automating TEST case design, selection, and evaluation*. 2–8.

[2] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, et al. 2022. Flamingo: a visual language model for few-shot learning. *Advances in neural information processing systems* 35 (2022), 23716–23736.

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. 2014. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE software* 32, 5 (2014), 53–59.

[4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[5] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based android gui testing using multi-level gui comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 238–249.

[6] Ishan Banerjee, Bao Nguyen, Vahid Garousi, and Atif Memon. 2013. Graphical user interface (GUI) testing: Systematic mapping and repository. *Information and Software Technology* 55, 10 (2013), 1679–1694.

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[8] Kewei Cheng, Jingfeng Yang, Haoming Jiang, Zhengyang Wang, Binxuan Huang, Ruirui Li, Shiyang Li, Zheng Li, Yifan Gao, Xian Li, et al. 2024. Inductive or deductive? Rethinking the fundamental reasoning abilities of LLMs. *arXiv preprint arXiv:2408.00114* (2024).

[9] D Crow and BJ Jansen. 1998. The graphical user interface: An introduction. *SIGCHI Bulletin* 30, 3 (1998), 24–28.

[10] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Ryan Tsang, Najmeh Nazari, Han Wang, Houman Homayoun, et al. 2024. Large language models for code analysis: Do {LLMs} really do their job?. In *33rd USENIX Security Symposium (USENIX Security 24)*. 829–846.

[11] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2312–2323.

[12] Kristian Kolthoff, Felix Kretzer, Christian Bartelt, Alexander Maedche, and Simone Paolo Ponzetto. 2024. Interlinking user stories and GUI prototyping: A semi-automatic LLM-based approach. In *2024 IEEE 32nd International Requirements Engineering Conference (RE)*. IEEE, 380–388.

[13] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[14] Yihao Li, Pan Liu, Haiyang Wang, Jie Chu, and W Eric Wong. 2025. Evaluating large language models for software testing. *Computer Standards & Interfaces* 93 (2025), 103942.

[15] Guangyi Liu, Pengxiang Zhao, Liang Liu, Yaxuan Guo, Han Xiao, Weifeng Lin, Yuxiang Chai, Yue Han, Shuai Ren, Hao Wang, et al. 2025. Llm-powered gui agents in phone automation: Surveying progress and prospects. *arXiv preprint arXiv:2504.19838* (2025).

[16] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2024. 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*. 1–13.

[17] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Yuekai Huang, Jun Hu, and Qing Wang. 2024. Unblind text inputs: predicting hint-text of text input in mobile apps via LLM. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 1–20.

[18] Wendy L Martinez. 2011. Graphical user interfaces. *Wiley Interdisciplinary Reviews: Computational Statistics* 3, 2 (2011), 119–133.

[19] Reto Meier. 2012. *Professional Android 4 application development*. John Wiley & Sons.

[20] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic generation of programming exercises and code explanations using large language models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*. 27–43.

[21] Mohammed Latif Siddiq, Joanna Cecilia Da Silva Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. 2024. Using large language models to generate junit tests: An empirical study. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 313–322.

[22] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 245–256.

[23] Anthony Meng Huat Tiong, Junnan Li, Boyang Li, Silvio Savarese, and Steven CH Hoi. 2022. Plug-and-play vqa: Zero-shot vqa by conjoining large pretrained models with zero training. *arXiv preprint arXiv:2210.08773* (2022).

[24] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[25] Chenxu Wang, Tianming Liu, Yanjie Zhao, Minghui Yang, and Haoyu Wang. 2025. LLMDroid: Enhancing Automated Mobile App GUI Testing Coverage with Large Language Model Guidance. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 1001–1022.

[26] Zixuan Wang, Chi-Keung Tang, and Yu-Wing Tai. 2024. Audio-agent: Leveraging llms for audio generation, editing and composition. *arXiv preprint arXiv:2410.03335* (2024).

[27] Xusheng Xiao, Xiaoyin Wang, Zhihao Cao, Hanlin Wang, and Peng Gao. 2019. Iconintent: automatic identification of sensitive ui widgets based on icon classification for android apps. In *ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 257–268.

[28] Weiran Yang, Zhenyu Chen, Zebao Gao, Yunxiao Zou, and Xiaoran Xu. 2014. GUI testing assisted by human knowledge: Random vs. functional. *Journal of Systems and Software* 89 (2014), 76–86.

[29] Juyeon Yoon, Seah Kim, Somin Kim, Sukchul Jung, and Shin Yoo. 2025. Integrating LLM-Based Text Generation with Dynamic Context Retrieval for GUI Testing. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 394–405.

[30] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[31] Xiaohua Zhai, Xiao Wang, Basil Mustafa, Andreas Steiner, Daniel Keysers, Alexander Kolesnikov, and Lucas Beyer. 2022. Lit: Zero-shot transfer with locked-image text tuning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 18123–18133.

[32] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 5673–5684.

[33] Maosheng Zhong, Zhixiang Wang, Gen Liu, Youde Chen, Huizhu Liu, and Ruping Wu. 2023. Codegen-test: An automatic code generation model integrating program test information. In *2023 2nd International Conference on Cloud Computing, Big Data Application and Software Engineering (CBASE)*. IEEE, 341–344.

[7] https://www.ines.org.br