LLMDroid: Enhancing Automated Mobile App GUI Testing Coverage with Large Language Model Guidance

CHENXU WANG*, Huazhong University of Science and Technology, China
TIANMING LIU*, Monash University, Australia
YANJIE ZHAO, Huazhong University of Science and Technology, China
MINGHUI YANG, OPPO, China
HAOYU WANG†, Huazhong University of Science and Technology, China

With the rapid development of Large Language Models (LLMs), their integration into automated mobile GUI testing has emerged as a promising research direction. However, existing LLM-based testing approaches face significant challenges, including time inefficiency and high costs due to constant LLM querying. To address these issues, this paper introduces LLMDroid, a novel testing framework designed to enhance existing automated mobile GUI testing tools by leveraging LLMs more efficiently. The workflow of LLMDroid comprises two main stages: Autonomous Exploration and LLM Guidance. During Autonomous Exploration, LLMDroid utilizes existing testing tools while leveraging LLMs to summarize explored pages. When code coverage growth slows, it transitions to LLM Guidance to strategically direct testing towards unexplored functionalities. This approach minimizes LLM interactions while maximizing their impact on test coverage. We applied LLMDroid to three popular open-source Android testing tools and evaluated it on 14 top-listed apps from Google Play. Results demonstrate an average increase of 26.16% in code coverage and 29.31% in activity coverage. Furthermore, our evaluation under different LLMs reveals that LLMDroid outperform existing step-wise approaches with significant cost efficiency, achieving optimal performance at $0.49 per hour using GPT-4o among tested models, with a cost-effective alternative achieving 94% of this performance at just $0.03 per hour. These findings highlight LLMDroid's effectiveness in enhancing automated mobile app testing and its potential for widespread adoption.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Mobile App Testing, GUI Testing, Large Language Models, Code Coverage

ACM Reference Format:
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. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE046 (July 2025), 22 pages. https://doi.org/10.1145/3715763

*Chenxu Wang and Tianming Liu are the co-first authors.
†Corresponding author: Haoyu Wang (haoyuwang@hust.edu.cn).

Authors' Contact Information: Chenxu Wang, Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China, csecxwang@hust.edu.cn; Tianming Liu, Monash University, Melbourne, Australia, Tianming.Liu@monash.edu; Yanjie Zhao, Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China, yanjie_zhao@hust.edu.cn; Minghui Yang, OPPO, Dongguan, China, yangminghui@oppo.com; Haoyu Wang, Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China, haoyuwang@hust.edu.cn.

ACM 2994-970X/2025/7-ARTFSE046
https://doi.org/10.1145/3715763

1 Introduction

The proliferation of the mobile app ecosystem has intensified the need for more effective automated GUI testing of mobile apps, which plays a crucial role in ensuring app quality for developers and facilitating secure vetting processes for app stores. Consequently, it has remained a persistent and vibrant research topic over the past decade [16, 27], spawning a diverse array of tools and approaches.

Except for random-based tools like the official Android Monkey [9] that commonly serve for stress testing, most automated app GUI testing tools operate by iteratively acquiring GUI information from app pages on the testing device and generating executable actions accordingly. Within this framework, different tools employ varied strategies to enhance testing performance. Initially, these tools often relied on model-based or probability-based strategies [15, 32, 40]. However, these approaches often struggled to achieve comprehensive testing coverage. The development of AI technologies has caused drastic changes in testing strategies, with deep learning-based [33, 50] and reinforcement learning-based approaches [28, 37, 38] continuously emerging, aiming to elevate testing performance. Nevertheless, there remains substantial room for improvement in testing coverage, as research has revealed that these tools often become trapped in loops or disproportionately focus on a limited subset of app pages [18, 23, 48].

The recent advancements in Large Language Models (LLMs), with their enhanced capability at semantic understanding, have led to their widespread application across various domains of software engineering [22]. Consequently, researchers have also begun exploring the potential of LLMs to enhance automated GUI testing. However, these efforts [24, 31, 45–47, 49] mostly focus on leveraging LLMs to build agents capable of executing specific tasks described in natural language (e.g., "Follow User" for Twitter). For works aimed at improving the overall performance of app GUI testing through LLMs, GPTDroid [35] engages in functionality-aware dialogues with the LLM to determine the next action to be executed on the current page, while DroidAgent [51] prompts the LLM to generate task goals and subsequently attempts to achieve these goals under LLM guidance.

However, all of the aforementioned works rely on testing apps under step-by-step LLM guidance, where every action executed is based on inputting the description of current app page to the LLM and depending on its output for the next action. This testing mode presents two significant challenges: time efficiency and cost. The continuous querying of LLMs introduces substantial inefficiencies, both in terms of response time and end-to-end latency. Moreover, it incurs considerable economic costs, particularly for extended testing periods. The combination of these temporal and financial constraints creates barriers to the widespread application of these approaches.

To address these challenges, this paper proposes LLMDroid, a novel testing framework designed to enhance the performance of existing automated mobile GUI testing tools by leveraging LLMs. We use code coverage as our primary metric to guide enhancements and demonstrate effectiveness, as it is the most widely adopted measure [16, 27] upon which other metrics depend. The workflow of LLMDroid comprises two main stages: Autonomous Exploration and LLM Guidance. Initially, LLMDroid enters the Autonomous Exploration stage, utilizing the existing tool to be enhanced for app testing. During this phase, LLMDroid employs LLMs to summarize explored pages. When code coverage growth slows down, LLMDroid transitions to the LLM Guidance stage, where it leverages the LLM to strategically direct the testing process, identifying which app pages and functionalities should be explored next to overcome coverage bottlenecks. LLMDroid alternates between these two stages to maximize testing coverage.

On the technical front, when leveraging LLMs to summarize explored app pages, we have designed a GUI summary prompt that is both functionality-oriented and importance-aware (§3.1.2). This approach facilitates more effective LLM guidance in the subsequent stage. Additionally, we have developed a novel algorithm to detect low-growth states in code coverage during testing

(§3.1.3). In the LLM Guidance stage, we have implemented diverse strategies to navigate the app to the target page offline (which was proven effective in §4.2) while utilizing LLM capabilities to execute the target functionality. This hybrid approach represents our attempt to optimize resource utilization and cost-effectiveness.

We apply LLMDroid to three popular open-source Android automated testing tools, each based on different techniques and strategies: Droidbot [32], Humanoid [33], and Fastbot [37], to demonstrate our effectiveness and generalizability. Different from previous studies requiring code coverage analysis, which often use open-source apps, we construct our experimental dataset primarily from Google Play's top listings to better reflects the scale and complexity of contemporary commercial apps, aligning more closely with real-world app testing scenarios. Our experimental results reveal that, on average across these three tools, LLMDroid achieved a **26.16%** increase in code coverage and a **29.31%** increase in activity coverage, with the most significant improvements reached 30.30% for code coverage and 41.29% for activity coverage, underscoring LLMDroid's effectiveness. We also measured the effectiveness rate of LLM Guidance instances and provide an in-depth analysis for failure cases. Moreover, our evaluation of LLMDroid under LLMs with varying capabilities and pricing reveals that LLMDroid outperform existing step-wise approaches with significant cost efficiency. For optimal performance of LLMDroid among tested models using GPT-4o, LLMDroid incurs a cost of **only $0.49 for one-hour testing**. Notably, LLMDroid achieves **94%** of this performance using GPT-4o mini [7] at merely **$0.03 per hour**, 6.7% of the cost of GPT-4o. These findings highlight the cost-effectiveness of LLMDroid and its huge potential for large-scale applications.

Ultimately, LLMDroid aims to provide and inspire insights into **new paradigms for leveraging LLMs to enhance automated app testing**.

In conclusion, the major contributions of this paper are as follows:

- We propose LLMDroid, a novel testing framework designed to enhance the performance of existing automated mobile GUI testing tools with LLMs, providing a new paradigm for integrating LLMs into automated app testing.
- We demonstrate LLMDroid's effectiveness by applying it to three popular testing tools. To facilitate future effort in this domain, we have made the source code for all these implementations, along with our dataset, results, and a guideline for integrating LLMDroid into existing testing tools, publicly available on our website[1].
- We evaluate LLMDroid on 14 popular closed-source apps from Google Play, and demonstrate an average increase of 26.16% in code coverage and 29.31% in activity coverage.
- Our evaluation of LLMDroid under different LLMs highlights its cost-effectiveness and its future application prospects, with optimal performance among tested models achieved using the pricey GPT-4o at a cost of only $0.49 per hour. Notably, 94% of the optimal performance can be attained at merely $0.03 per hour using GPT-4o mini.

2 Related Work

2.1 Automated App GUI Testing Based on LLM

To date, GPTDroid [35] and DroidAgent [51] are, to our knowledge, the only two works that focus on comprehensively testing mobile apps by leveraging LLMs, both of which rely on step-by-step LLM guidance. GPTDroid engages in a dialogue with the LLM to determine the next actions to be executed. During this process, it prompts the LLM with historical tested functionalities and requests the LLM to consider the current functionality under test and whether it represents a new functionality. DroidAgent, conversely, prompts the LLM to plan testing tasks prior to execution. It

[1] https://github.com/security-pride/LLMDroid

then leverages the LLM to execute these tasks, followed by a reflection phase to gather knowledge that can inform future task planning.

Meanwhile, other research efforts focus on using LLMs to address specific challenges in app testing. For instance, Cui et al. [17] and QTypist [34] employ LLMs to generate context-aware text inputs for app testing scenarios (e.g., search bars). AdbGPT [19], ReBL [44], and CrashTranslator [25] utilize LLMs to reproduce bug reports and crashes based on stack traces. Ju et al. [26] and VisionDroid [36] leverage LLMs' multimodal capabilities to detect non-crash bugs in apps. These diverse approaches contribute to the broader integration of LLMs in app testing, demonstrating the potential for LLM-driven enhancements across various aspects of the app testing process.

2.2 Other Enhancements to Automated App GUI Testing

Several previous works have identified shortcomings in existing testing tools, particularly their tendency to become trapped in loops or disproportionately focus on a limited subset of app pages. In response, various techniques have been proposed to enhance testing efficiency and coverage. Notably, TimeMachine [18] introduced an innovative approach of preserving snapshots of simulated Android devices for interesting app pages. This method supports continued testing of previously explored but insufficiently tested pages, allowing for more comprehensive coverage. Fax [48] and Delm [23] aim to boost code coverage by directly jump-starting Android activities that are not the main entry of the app. They achieve this by constructing activity contexts and intents, enabling exploration of otherwise hard-to-reach app pages. AdaT [20] addressed the inefficiency of fixed time intervals between testing actions. By implementing a screenshot-based deep learning method to determine page load completion, AdaT significantly reduces wasted time during the testing process. These innovative techniques can also apply to LLMDroid, presenting opportunities for further enhancement of automated app testing.

3 Approach

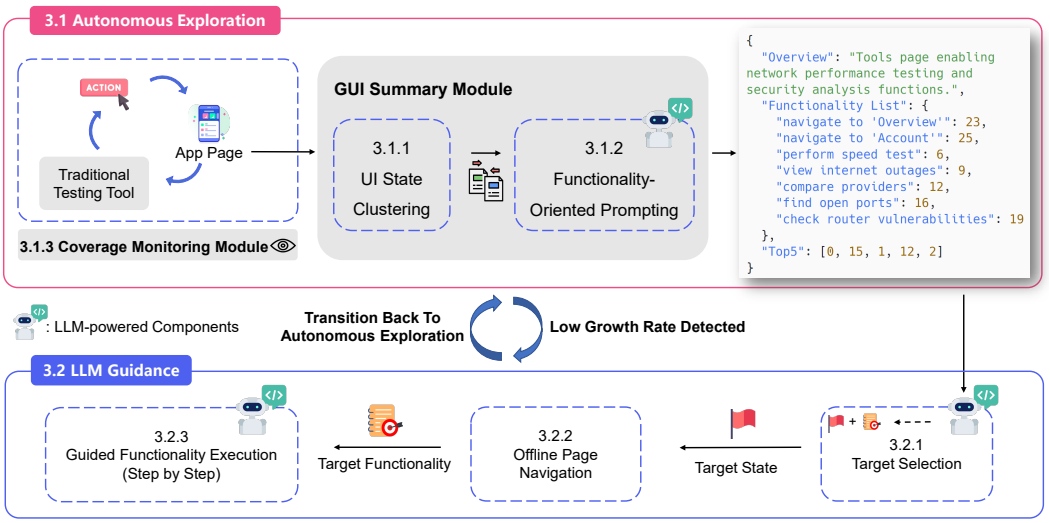

Fig. 1. Workflow of LLMDroid

LLMDroid is designed as a framework to enhance existing automated mobile GUI testing tools by leveraging Large Language Models (LLMs). In this paper, we use code coverage as the primary

metric to guide our enhancements and showcase the effectiveness of LLMDroid. This decision is driven by the fact that code coverage, as a measuring metric, is objective, calculable, and serves as a fundamental metric upon which other metrics depend, such as activity coverage, UI coverage, and number of faults detected. Code coverage is also the most prevalent used metric in the domain of automated mobile GUI testing [16, 27].

The overall workflow of LLMDroid, as shown in Figure 1, is divided into two main stages: **Autonomous Exploration** (§3.1) and **LLM Guidance** (§3.2). When testing an app, LLMDroid initially enters the Autonomous Exploration stage, where it leverages an existing tool to be enhanced to test the app. This underlying tool generally operates by continuously acquiring GUI information of app pages from the device, extracting executable actions, and selecting an action to execute. During this stage, LLMDroid incorporates two additional modules:

- The ***Coverage Monitoring*** module continuously monitors the code coverage of the app in real-time and directs LLMDroid to switch from the Autonomous Exploration stage to the LLM Guidance stage when the growth of code coverage slows down.
- The ***GUI Summary*** module is responsible for communicating with the LLM to understand and summarize the pages explored during the Autonomous Exploration stage. The output of this module serves as crucial information for the LLM to provide guidance in the next stage.

The LLM Guidance stage leverages the LLM to guide the testing process on which app page and functionality the testing should explore next when the code coverage growth slows down. LLMDroid then navigates to the specified app page, engages the identified functionality, and switches back to the Autonomous Exploration stage to continue testing, until the Coverage Monitoring module determines that it is necessary to enter the LLM Guidance stage again.

In summary, LLMDroid alternates between the two stages until the allocated testing time is exhausted. **Shying away from step-by-step LLM guidance which current LLM-based testing works heavily rely on**, the two stages of LLMDroid work in synergy to maximize testing coverage while optimizing resource usage, saving time and computational costs of LLM interactions. LLMDroid aims to provide and inspire insights into **new paradigms for leveraging LLMs to enhance automated app testing**.

3.1 Autonomous Exploration

In this stage, LLMDroid leverages an existing automated mobile GUI testing tool to autonomously explore the app, while simultaneously using the LLM to understand and summarize the explored pages. The ***GUI Summary*** module clusters similar UI pages (§3.1.1) and summarizes them by querying LLM on a PageCluster basis. We design the prompt for GUI summary in a functionality-oriented and importance-aware manner (§3.1.2). The LLM takes the page information as input, and outputs a summary of the page and a list of functionalities that could be achieved within the page. These outputs serves as crucial information for the LLM Guidance stage. The ***Coverage Monitoring*** module monitors the app's code coverage in real-time. We set a threshold for the growth rate of code coverage and dynamically adjust this threshold with our algorithm (§3.1.3). When the growth rate continuously falls below this threshold, the testing transitions to the LLM Guidance stage.

3.1.1 UI Page Clustering. We first introduce our UI page clustering approach as it is essential for understanding the workflow of LLMDroid and for reducing the interaction frequency with LLM for GUI summary.

The term "UI page" here refers to the user interface state of an app at a given moment, therefore in some literature, this concept is also referred to as "UI state". UI page differentiation is a common problem for automated mobile app GUI testing approaches. Some methods consider pages as different even with minimal changes (e.g. [32]), which would introduce excessive overhead for LLM

interaction and is clearly unsuitable for LLMDroid. Conversely, an oversimplified approach might treat all UI pages within the same Activity as a single UI state. While this is undoubtedly the most economical method, it fails to account for the significant variations that can exist between pages in the same Activity, particularly when considering Android Fragments [4]. Therefore, we adopt a more balanced strategy and differentiates UI pages based on their similarity.

The similarity is determined by the number of the same widgets two UI pages share. We define two widgets being identical when their class, resource-id, supported actions, width, and height attributes are all identical. The similarity is calculated by the Dice Coefficient [3], which is commonly used to measure similarity between two sets: $\frac{2|A \cap B|}{|A|+|B|}$, where $|A \cap B|$ is the number of identical views shared between two UI pages, and $|A|$ and $|B|$ are the total number of views in each UI page respectively. If the similarity exceeds a certain threshold T_S, the two pages are considered similar and merged into a **PageCluster**. This method is especially useful to apps featuring dynamic content feeds, which are ubiquitous in modern apps.

We establish PageClusters in real-time during the app testing process. The first reached UI page within the PageCluster is called the Root Page, of which each PageCluster only has one. Whenever a new page is reached, we first check the similarity with the Root Pages of all PageClusters and add this page to the PageCluster where its Root Page has the highest similarity with this page, given that the similarity is above the threshold. If no similarity is above the threshold, a new PageCluster is created with this new page being the Root Page.

Prompt Timing for GUI Summary. For each PageCluster, LLMDroid only interacts with the LLM once for GUI Summary (to be introduced in §3.1.2), basing the prompt on the Root Page, to reduce the expenses on LLM interaction. Therefore, the timing of such interactions in the Autonomous Exploration stage is when a new PageCluster is established, which often occurs more frequently during the initial phases of testing when the app's unexplored territory is vast, leading to frequent discoveries of new pages and consequently making it uncommon for apps to reach code coverage ceiling at the beginning. Conversely, as testing progresses, it becomes increasingly common for the Autonomous Exploration to encounter coverage barrier, necessitating LLM Guidance for the next moves that costs additional LLM interaction. Therefore, our design balances the LLM interaction time and resources needed between the two stages.

We also notice a flaw in this design: the LLM summary of a PageCluster is limited to the Root Page only, potentially overlooking information in other UI pages within the cluster despite their similarity. However, querying the LLM at PageCluster's establishment is the most resource-friendly approach, rather than at the end when no more similar pages are added to the PageCluster (which is an indeterminate timing) or at any other timing in between. Our mediation to this problem is that when LLMDroid is vacant for LLM interaction, i.e., no query for any stage is pending, we extract unique widgets from non-Root Pages within the PageCluster and prompt the LLM again for a more comprehensive summary of the PageCluster.

3.1.2 Prompt Design for GUI Summary. During Autonomous Exploration, we provide the LLM with information about the explored UI pages, requesting the LLM to summarize the UI page in a functionality-oriented and importance-aware manner. The output of GUI Summary serves as crucial input for LLM to provide guidance, which will be introduced in §3.2. The GUI summary process operates **concurrently** with the app testing to fully utilize the original testing tool's efficiency.

Table 1 presents a concise summary of all prompt designs utilized in LLMDroid. Each row in the table offers a distilled version of the original prompts, which we elaborate on individually in the subsequent sections. For the sake of brevity, we have omitted the output requirements from the prompts and included simplified output examples instead, to illustrate the expected format and content of the responses. This table is divided into four sections: GUI Context Pattern, Query

Table 1. Prompt Designs in LLMDroid

ID	Prompt Type	Instantiation
		GUI Context Pattern
1	App Information	I'm now testing an app called Fing on Android. This is an network tool app, which can...
2	HTML Description	**Activity:** xxx.ui.main.MainActivity **Current Page Id: 8** <scroller id=0 class="FrameLayout" direction="vertical, horizontal" > ..... <scroller id=3 class="ScrollView" resource-id="scroll_view" direction="vertical" > This is a WIFI <button id=5 class="FrameLayout" resource-id="select_network" ></button> ..... </scroller>
3	Page Abstraction	"Page8": { **"Overview"**: "The main activity interface for network monitoring and navigation.", **"Top Q Functionality List"**: ["navigate to 'Tool' tab", "select network", ...] },
		Query Pattern
4	Query Summary	Based on the HTML description of this page, your tasks include: **a. Page Overview**: Summarize the current page within 30 words, ... **b. Functionality Analysis**: Analyze the functionalities contained in this page. List the functionalities with the corresponding element ID in the HTML tag. The functionality list should be sorted by importance, defined as: **c. Functionality Importance**: A functionality's importance increases if it can potentially trigger a new page or result in more code being executed. Specifically: - Navigation-related functionalities or any other functionalities could trigger new pages ... - Functionalities central to the page's main purpose, ... **d. Page Importance Ranking**: Assess this page's significance relative to the entire app, ...
5	Query Target	Which page should we go next, and what functionality would be most appropriate to test in the target page? You can follow these strategies: a. Prioritize choosing functionalities related to navigation. b. Choose other functionalities which can trigger transition or lead to undiscovered pages. .....
6	Query Step	I have already executed [...], did I finish testing the target functionality {...}? If not, what action should I perform next on the current page?
		Prompt Generation Rule
7	GUI Summary	App Information + HTML Description + Top P_1 Pages + Query Summary
8	Target Selection	App Information + Top P_2 Pages + Query Target
9	Functionality Execution	App Information + HTML Description + Query Step
		Output Example
10	GUI Summary	{ "Overview": "Main page of the app ...", "Functionality List": {"navigate to Tool'": 22,"view devices": 18,"select network": 5, ... }, "ID of Top 5 Pages": [0, 1, 2, 8, 3]}
11	Target Page	{"Target Page": "Page15", "Target Functionality": "perform speed test"}
12	Functionality Execution	{"Finished": false, "Element Id": 6, "Action Type": 0}

Pattern, Prompt Generation Rule, and Output Example. As demonstrated in **Row 1**, the prompt first includes the app name and an overall description of the app, giving the LLM an overview of the app.

Describing UI Pages with HTML. Subsequently, the prompt contains a description of the UI page being summarized (**Row 2**), which includes the attributes of each UI widget within the page and the hierarchical relationships between them. We choose to describe the UI page in HTML format in LLMDroid following Wang et al. [43]. Their work suggests that HTML representation might better align with LLMs' training data distribution, as LLMs are extensively trained on web data including HTML, and HTML is naturally suited for representing UI hierarchies.

Each HTML tag represents a UI widget, with which an ID is also assigned for reference. Similar to [43], LLMDroid employs HTML tag names to represent the actionable attributes of each UI widget. Specifically, we use the following mapping: <button> for clickable elements, <checkbox> for checkable items, <scroller> for scrollable components, <input> for editable fields, and <p> for unactionable widgets. Each HTML tag encapsulates several attributes that provide detailed information about the corresponding widget. These attributes include the widget's class, resource-id, text content, and content-desc (an additional textual description of the widget for accessibility purposes). This comprehensive widget information is extracted from the GUI view tree, which is accessible through the Android system's AccessibilityService [1].

Functionality-oriented and Importance-aware GUI Summary. During this phase, we first ask the LLM to generate a concise overview of the UI page in less than 30 words (**Row 4a**). Next, we instruct the LLM to analyze the page and generate a list of functionalities, each associated with a specific UI widget (**Row 4b**). During this process, we also request the LLM to rank these functionalities based on their importance (**Row 4c**). Since our goal is to perform comprehensive testing of the app and maximize coverage, we ask LLM to consider a functionality as important if it is navigation-related (those that can potentially direct the app to another different page) or related to the core functionality of the app. This process closely resembles how humans test an app: we first get an overview of the page to establish a general understanding, then select widgets that can trigger new pages or are crucial to the app for testing.

An example output is demonstrated in the first two lines of **Row 10**, within which, for each functionality, we also include the ID of its corresponding widget (as assigned in HTML tags) in the page.

Within the same prompt, we also task the LLM with assessing the importance of the current page within the context of the entire app (**Row 4d**). Despite our efforts to cluster similar pages and query the LLM by PageClusters, the testing process can still generate a substantial number of PageClusters. This volume makes it impractical to feed all clusters to the LLM during the Guidance stage while expecting LLM to have a good understanding. By evaluating the importance of pages, we can prioritize crucial ones for subsequent LLM guidance.

Given the diversity and complexity of mobile apps, it is challenging to establish a universal standard for LLM to accurately measure the importance of one page alone. Therefore, instead of assigning absolute importance scores, we opt for a relative ranking approach. We maintain a dynamic list of the top P_1 important pages, based on the LLM's previous assessments. When summarizing each new page, we present the LLM with this list, as shown in **Row 7**. Each page is abstracted as in **Row 3**, with a concise overall description of the page and its top Q_1 most important untested functionalities. A functionality is deemed untested if its corresponding UI widget in this page is unexplored. The LLM then compares the current page against these top pages, potentially changing the ranking if the new page is deemed more important. An example output of this page importance assessment is shown in the last line of **Row 10**. The length of this list is always up to P_1 to prevent overwhelming the LLM with too many comparisons among excessive information.

The value of P_1 and Q_1 should be primarily determined by the LLM's capacity to process and understand lengthy prompts. In our implementation using GPT-4o [6], we empirically set P_1 to

5, as we observed that larger values occasionally led to degraded LLM performance; and Q_1 to 5, because we found that the top 5 important untested functionalities, combined with the overall page description, provides sufficient context for the LLM to understand the page.

3.1.3 Code Coverage Monitoring. This module monitors the code coverage in real-time. We employs a dynamic threshold mechanism for the code coverage growth rate, which, when continuously breached, triggers a transition from the Autonomous Exploration stage to the LLM Guidance stage.

Growth Rate Calculation and Low Growth Detection. Let $c_1, c_2, ..., c_n$ represent n code coverage data points, each being the total code coverage of the app after a testing action is executed by LLMDroid. The growth rate between adjacent points is calculated as $g_{i-1} = (c_i - c_{i-1})/c_{i-1}$. We define a window of size W, and a low growth rate threshold T which is dynamically adjusted as described in the next paragraph. If all growth rates within this window fall below the dynamic threshold, we consider the code coverage to be in a low growth state. This approach accounts for the common scenario where, as testing progresses, only a few actions trigger new function executions, leading to code coverage increases being scattered and occasional.

Dynamic Threshold Adjustment. Determining an appropriate low growth rate threshold is challenging due to the diversity of mobile apps in their total number of methods and achievable code coverage percentages. To address this, we introduce a dynamic baseline growth rate (G), calculated as the average of all previous growth rates: $G = (g_1 + g_2 + ... + g_{n-1})/(n-1)$. For each new code coverage data point after a new action is executed, we update G and calculate the difference between the current growth rate and G: $\Delta g = g_n - G$. We then adjust the threshold T using an exponential function: $T_n = T_{n-1} \cdot e^{\Delta g}$. This dynamic adjustment mechanism allows the threshold to adapt to the changing trends in code coverage growth. When Δg is positive, indicating above-average growth, the threshold increases. Conversely, when Δg is negative, the threshold decreases, making it easier to detect growth even when code coverage naturally slows as testing progresses.

We empirically determined the initial growth rate threshold T_0 to be 0.05, based on our observations that the code coverage growth rate typically exceeds this value when a new page or functionality is triggered. To prevent excessive threshold reduction as testing progresses, we implemented a lower bound of 0.01 for this dynamic threshold. We set the window size W to a relatively high value of 80, meaning that a sequence of up to 80 low-growth actions will prompt the interruption of autonomous exploration. Such configuration minimizes disruption to the original tool's testing strategies, while simultaneously optimizing resource utilization by reducing the frequency of LLM interactions and fully leveraging the rapid action execution inherent to autonomous exploration. We also performed a set of controlled experiments trying to determine an optimal window size, testing values of 60, 70, 80, 90, and 100, and revealed minimal variations in code coverage across these different window sizes, with 80 slightly higher than 60 and 100 by less than 1%. Notably, we observed no clear trend in performance across this range of window sizes.

Transition to LLM Guidance. As autonomous exploration progresses, it may eventually encounter a scenario where the original testing tool struggles to trigger new pages, looping among a limited set of pages. When the code coverage growth rate consistently falls below the dynamically adjusted threshold, we consider the testing to have reached a bottleneck. At this point, LLMDroid interrupts its current autonomous exploration and transitions to the LLM Guidance stage, leveraging the LLM to overcome the bottleneck and explore new areas of the app.

3.2 LLM Guidance

The LLM Guidance stage aims to navigate LLMDroid to an appropriate page and functionality for subsequent testing to overcome bottlenecks encountered during autonomous exploration. This stage is divided into three modules:

- The ***Target Selection*** module utilizes output of the GUI Summary module, and selects a target page and a target functionality within this page to be tested next.
- The ***Offline Page Navigation*** module leverages historical testing traces from autonomous exploration to efficiently guide the app to the target page without relying on LLM assistance.
- Finally, the ***Guided Functionality Execution*** completes the execution of the target functionality under LLM's step-by-step guidance.

3.2.1 Target Selection. In this module, we prompt the LLM with the top important pages obtained in the GUI Summary module, and instruct it to select **a target page** and **a target functionality within this page** for subsequent testing.

As mentioned in §3.1.2, feeding all explored pages to the LLM is impractical. Therefore, mirroring our prompt design in GUI Summary, we provide the top P_2 important pages to the LLM in this module (**Row 8**), with each page including its overall description and its top Q_2 most important untested functionalities. This design significantly reduces the prompt length while directing the LLM's attention to more crucial pages.

Our selection strategy prioritizes navigation-related functionalities that could potentially lead to the discovery of previously unexplored pages during the Autonomous Exploration phase. We instruct the LLM to favor these options in its decision-making process (**Row 5**).

Following our previous setup, we maintain Q_2 at 5. However, we increase P_2 to 10 in this module. This adjustment is feasible because, unlike in the GUI Summary module, the LLM is not also tasked with analyzing and summarizing UI pages for their functionalities here. The higher P_2 value also provides the LLM with a broader range of choices and a more comprehensive context of the already explored pages, allowing the LLM to make more informed selections for further testing. We obtain the list of top P_2 important pages by extending beyond the P_1 (which we set to 5) pages. This extension is derived from locally maintaining pages eliminated from the P_1 list.

3.2.2 Offline Page Navigation. In this module, the tool attempts to automatically navigate the app to the target page. To enhance testing efficiency and conserve resources, this navigation does not rely on the LLM but is instead performed locally leveraging testing traces from autonomous exploration, considering such navigation may necessitate multiple steps, where substantial testing time could be wasted under LLM's step-by-step guidance.

Navigating from the current page to the target page presents significant challenges. App transitions are often unidirectional, meaning that certain actions cannot be simply reversed with a back event. Additional complexities, such as pop-ups, further complicate the backtracking process.

Therefore, we restart the app to consistently begin from the app's initial page for each navigation attempt. We determine a shortest path to the target page using Dijkstra's algorithm on the unidirected UI page transition graph (built from execution traces during Autonomous Exploration), and replays this path by performing the necessary UI actions successively and verifying after each step whether the actual page reached aligns with the expected node in the path.

To maximize navigation success rates, we have implemented the following replay strategies:

- Fuzzy Matching: To mitigate potential mismatches caused by dynamic page changes, our algorithm employs page similarity calculations for fuzzy matching of pages along the navigation path. If the current page is not an exact match with the expected page but exceeds a similarity threshold T_S (consistent with §3.1.1), this navigation step is still considered successful.
- Step Skipping: In cases where the current page does not match the expected one, the algorithm checks for a match with the subsequent page in the navigation path. If a match is found, the algorithm skips the current node and proceeds with navigation actions for the next node.

- Failure Handling: When the shortest path navigation fails, we infer potential app changes and sequentially attempt navigation using the most recently established paths from the app's initial page to the target page. Each failure also triggers a reduction in the similarity threshold by 0.05. The tool attempts navigation to the target page up to three times. If these attempts fail, we revert to the LLM to select a new target on a different page, which we also set a maximum of three attempts, and revert to autonomous exploration if all failed.

3.2.3 Guided Functionality Execution. After navigating to the target page, this module engages in a step-by-step dialogue with the LLM to execute the target functionality.

As shown in **Row 9** and **Row 6**, for each step, the tool provides the LLM with a description of the app and the current page, again in HTML format, reports previous actions taken to execute the functionality, and queries whether the functionality has been successfully executed. If the functionality execution is incomplete, we instruct the LLM to specify the next actions required and generate any necessary text input, such as for search boxes. We limit this guided execution to a maximum of five steps due to the time-intensive nature of step-by-step LLM guidance, and considering most target functionalities are typically straightforward, often requiring minimal interactions.

After this module, the LLM Guidance phase concludes. Ideally, this process should lead to the discovery of a new page. However, even if no new page is uncovered, we assume that the LLM has directed us to an important page with higher exploratory potential. Consequently, the tool transitions back to the Autonomous Exploration phase to swiftly examine the app from the current page, and re-enters the LLM Guidance phase when deemed necessary. This alternation continues until LLMDroid's pre-allocated testing time is exhausted.

4 Experimental Results

To evaluate LLMDroid's performance, we applied it to three popular open-source Android automated testing tools. §4.1 details LLMDroid's implementation and our experimental setup, including the rationale behind tool selection and the composition of our experimental dataset. The remainder of this section is structured around the following three research questions (RQs):

- RQ1 (§4.2): To what extent can LLMDroid enhance the coverage of the original tools? How effective are LLM Guidances in improving code coverage?
- RQ2 (§4.3): How does LLMDroid perform when utilizing LLMs of varying capabilities? How much does LLMDroid cost? How does LLMDroid compare with other LLM-based app testing tools relying on step-by-step guidance?
- RQ3 (§4.4): What is the optimal UI page similarity threshold T_S for use in UI page clustering, as described in §3.1.1?

4.1 LLMDroid Implementation and Experimental Setup

Throughout our experiments, we employ OpenAI's GPT-4o [6], an advanced LLM model to date, given the sophisticated nature of our task. We also evaluate LLMDroid's performance on different LLMs in RQ2 (§4.3).

Code Coverage Monitoring. For code coverage monitoring, we collect real-time code coverage data using AndroLog [39] in our evaluation, a state-of-the-art code coverage analysis tool based on static APK instrumentation using Soot. This tool enables code coverage monitoring in a **black-box** manner, i.e., without requiring the source code of app under test. It also demonstrates superior performance compared to its competitors. AndroLog offers various levels of code coverage granularity, including class, method, and statement levels. Among these options, we opt for method-level granularity to strike a balance between precision and overhead. To better support real-world testing

scenarios where developers have source code access, we also provide a JaCoCo-based version of LLMDroid on our website that supports white-box coverage monitoring.

Tool Selection. We applied LLMDroid to three popular open-source Android automated testing tools: Droidbot [32], Humanoid [33] (based on Droidbot), and Fastbot [37]. These tools respectively represent model-based, deep learning-based, and reinforcement learning-based approaches, which are the three major techniques that previous automated testing tools have relied upon. Our selection criteria were as follows: firstly, tools needed to be open-source, as some alternatives only released binaries; secondly, we required tools that are actively maintained and support recent Android versions, as earlier versions are incompatible with current trending apps. Although Droidbot and Humanoid were initially released in 2017 and 2019 respectively, they continue to be actively maintained and are widely adopted in recent works. Also, Humanoid is based on Droidbot. This significantly reduced our effort in applying LLMDroid, which can be non-trivial for non-developers of the original tool. Fastbot was included due to its recent development by industry (Bytedance), its efficiency, and stability. Inherited from the original Android Monkey, it boasts an impressive input speed of up to 12 actions per second [2]. While we did not utilize such high-frequency inputs in our current study, we believe tools like Fastbot hold greater potential for LLM integration. We designate these enhanced tools as LLMDroid-Droidbot, LLMDroid-Humanoid, and LLMDroid-Fastbot. To facilitate future efforts in this domain, we have made the source code for all these implementations available on our website, along with a step-by-step instruction with code examples of how to integrate LLMDroid into an existing testing tool.

Experimental Dataset. Our dataset is primarily derived from Google Play's top listings and their recommended similar apps. We deliberately chose these closed-source commercial apps over open-source alternatives, which are commonly used in studies requiring code coverage monitoring. This decision was driven by the observation that open-source apps often fail to represent the scale and complexity of contemporary commercial apps. By selecting popular commercial apps, our study aligns more closely with real-world app testing scenarios.

However, opting for closed-source apps introduced additional challenges to our dataset construction process. Despite utilizing AndroLog, a state-of-the-art black-box code coverage analyzer based on the actively maintained and well-adopted Soot framework [42], we encountered limitations. While AndroLog successfully instrumented most of our initially collected apps, we observed that some may crash upon launch after Soot instrumentation. We further discuss this limitation in §5.

After careful testing, we finalized a dataset comprising 14 apps that are both relatively popular and technical compatible. These apps cover various popular app categories, as detailed in Table 2.

Table 2. App Metadata within our Experimental Dataset

ID	App Name	Category	Downloads	Apk Size
1	Wish: Shop and Save	Shopping	500M+	43MB
2	Twitch: Live Streaming	Entertainment	100M+	157MB
3	Wikipedia	Education	50M+	33.2MB
4	Quora: the knowledge platform	Education	50M+	10.3MB
5	Fing - Network Tools	Tools	50M+	41.5MB
6	Lefun Health	Health	10M+	142MB
7	Red Bull TV: Videos & Sports	Sports	10M+	19.3MB
8	Time Planner: Schedule & Tasks	Tools	5M+	12.5MB
9	NewPipe	Entertainment	5M+	13.7MB
10	NHK WORLD-JAPAN	News	1M+	7.39MB
11	Renpho Health	Health	1M+	100MB
12	Mood Tracker: Self-Care Habits	Life	500K+	42.9MB
13	CafeNovel - Fiction & Webnovel	Book	10K+	19.0MB
14	OceanEx	Commercial	10K+	14.9MB

Other Experimental Setup. We conducted our experiments on a Google Pixel 4 running Android 11. For apps requiring authentication, we pre-registered accounts and manually logged in prior to testing to ensure functionality accessibility throughout the testing process. The UI page similarity threshold T_S mentioned in §3.1.1 is set to 0.6 through an experiment detailed in §4.4.

To maintain fairness, we allocated a consistent 60-minute testing duration for each tool on every app, a testing time widely adopted in previous GUI testing research such as [21, 32, 35, 41]. We also set a time interval of 3 seconds between actions for all tools and their enhanced versions. In addition to code coverage, we also measured activity coverage for performance evaluation. To mitigate potential biases, following the practice of Liu et al. [35], we executed each testing tool three times per app and used the highest coverage metrics for our analysis.

During our evaluation, we chose not to include fault detection metrics due to two significant challenges. First, Androlog's Soot instrumentation introduced occasional crashes, making it difficult to distinguish between genuine app failures and instrumentation-induced crashes. Second, recent research by Lan et al. [29] reveals inherent limitations in using fault detection as an evaluation metric, empirically demonstrating that fault detection results exhibit high instability.

4.2 Tool Performance (RQ1)

Table 3. Coverage and their Improvements over the Three Tools

Metrics	Monkey	Droidbot			Humanoid			Fastbot		
		Original	LLM	Impr.	Original	LLM	Impr.	Original	LLM	Impr.
ACC	8.86%	9.21%	11.69%	**26.90%**	11.05%	13.40%	**21.29%**	11.36%	14.80%	**30.30%**
AAC	11.34%	15.90%	22.46%	**41.29%**	19.16%	23.66%	**23.51%**	24.40%	30.05%	**23.13%**

LLM: LLMDroid-enhanced versions, Impr.: Improvement Rate
ACC: Average Code Coverage, AAC: Average Activity Coverage

Overall Improvement. Table 3 illustrates the average code coverage and activity coverage of each tool among all apps in our dataset, and their improvement rate under LLMDroid. We also included results of the widely used Monkey tool [9] for comparision.

Among the three tools, LLMDroid-Fastbot demonstrated the most significant improvement in code coverage by **30.30%**, followed by the enhanced versions of Droidbot and Humanoid, with code coverage improvements of 26.90% and 21.29% respectively. On average, the three LLMDroid-enhanced tools achieved a **26.16%** increase in code coverage and a **29.31%** increase in activity coverage. These substantial improvements underscore the effectiveness of LLMDroid across tools relying on different underlying techniques. Notably, LLMDroid-Droidbot exhibits an exceptionally high increase in activity coverage of **41.29%**, yet this is not correspondingly reflected in its code coverage increase. This discrepancy can be attributed to Droidbot's inherently limited capabilities, as evidenced by its markedly lower coverage in both metrics when compared to other tools. This result in more improvement room in activity coverage. However, even when LLMDroid-Droidbot successfully enters new activities, it often fails to fully exploit these activities to maximize code coverage.

Our dataset comprises 42 code coverage improvement rate data points, derived from applying the three tools across 14 apps. Figure 2 provides a visual representation of their distribution in both absolute numbers and percentages. Notably, only 16.67% of the improvement rates fall below 10%, while nearly half (47.62%) exceed 20%. This distribution underscores LLMDroid's relatively consistent and robust performance across diverse apps and testing tools.

Code Coverage Increase over Time. Figure 3 illustrates the code coverage progression over time for Fastbot and LLMDroid-Fastbot on a single app (Wish), providing a clearer demonstration of how LLMDroid enhances code coverage. The marked points on the graph indicate instances when

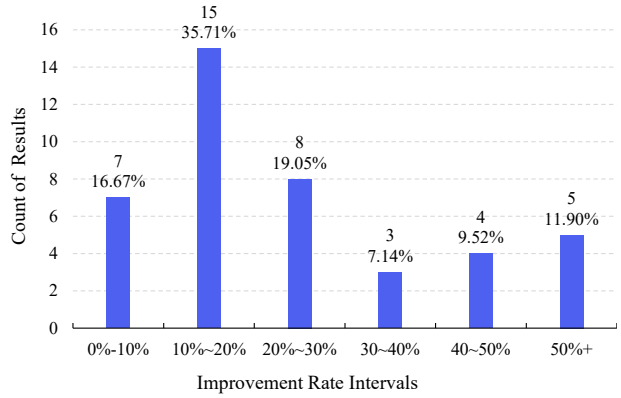

Fig. 2. The Distribution of 42 Improvement Rates

LLMDroid-Fastbot requested LLM guidance, which occurs after periods of stagnant code coverage. While Fastbot shows only occasional and irregular increases in code coverage, LLMDroid-Fastbot exhibits a more consistent upward trend, with code coverage increases occurring after instances of LLM guidance, highlighting the effectiveness of LLMDroid.

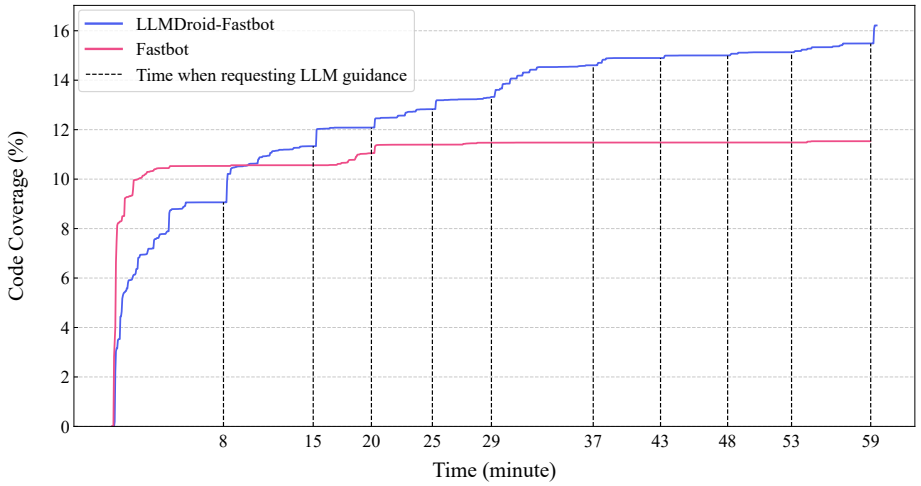

Fig. 3. The Increase of the App *Wish*'s Code Coverage Over Time

Effectiveness of LLM Guidance Instances and Analysis of Failure. LLMDroid's effectiveness depends on whether instances of LLM guidance can result in code coverage increase. Naturally, not all LLM guidance instances result in such increase. Therefore, we aim to evaluate how many LLM guidance instances are effective and analyze the reasons behind unsuccessful cases.

We first examined the frequency of LLM guidance instances for each app across our enhanced tools. On average, LLMDroid-Fastbot triggered **9.17** instances per app over one-hour testing, while LLMDroid-Droidbot and LLMDroid-Humanoid showed lower frequencies of **6.42** and **6.17**

instances per app, respectively. This is due to that, as described in §3.1.3, we set a fixed window size of 80 for determining low coverage growth states, meaning that 80 consecutive low-growth actions from the original tool would trigger a transition to LLM guidance. While we standardized the time interval setting between actions at 3 seconds for all tools (as mentioned in §4.1), Fastbot demonstrates faster action inputs due to its superior architecture. Fastbot's design incorporates an on-device server, whereas Droidbot and Humanoid operate from a PC, resulting in communication delays between the PC and the test device.

We then sought to establish an appropriate method for evaluating the effectiveness of each LLM guidance instance. Recognizing that code coverage typically demonstrates rapid growth in the early stages of testing, as illustrated in Figure 3, we devised a method to account for this pattern. Our approach involved calculating the difference between the code coverage at the first LLM guidance instance and the final code coverage achieved for the app. An LLM guidance instance was deemed effective if it resulted in a coverage increase exceeding 5% of this calculated difference. To quantify this increase, we measured the growth in code coverage from the point of LLM guidance until the app entered its next low-growth state. Initially, we considered setting this threshold at 10%. However, this proved overly stringent, as some instances generated substantial increases, consequently diminishing the relative percentage for others. For instance, at a 10% threshold, only the instances at 8, 29, and 59 minutes in Figure 3 would be classified as effective, which we determined to be an unreasonable assessment; while adjusting the threshold to 5% resulted in only the instances at 37, 43, 48, and 53 minutes being categorized as ineffective, which more accurately reflects the true impact of the LLM guidance instances.

Utilizing the aforementioned criteria, we calculated the percentage of effective guidance instances relative to the total number of guidance attempts. Overall, 47.84% of LLM guidance instances were deemed effective, with LLMDroid-Fastbot, LLMDroid-Humanoid, and LLMDroid-Droidbot achieving effectiveness rates of 42.48%, 55.00%, and 48.24%, respectively. The comparatively lower performance of LLMDroid-Fastbot can be attributed to its higher frequency of LLM guidance instances. This increased frequency potentially makes it more susceptible to encountering B2 and B4 scenarios, as detailed in the subsequent analysis. Conversely, LLMDroid-Humanoid demonstrates higher effectiveness rates than LLMDroid-Droidbot. This superior performance likely stems from Humanoid's enhanced capabilities in exploring unexplored territories as directed by LLM guidance. This assumption is evident by the higher coverage achieved by the original Humanoid tool compared to Droidbot, as shown in Table 3.

We proceeded to analyze the reasons behind ineffective guidance instances. First, we examined whether failures in our offline page navigation (§3.2.2) prevented LLMDroid from reaching the LLM-recommended target pages. **Notably, only 12.11% of ineffective guidance instances were due to navigation failures.** This underscores the efficacy of our optimization strategies in offline page navigation and validates our decision to perform this navigation without LLM interaction.

Further analysis of other ineffective guidance instances revealed two primary categories: A. LLM guidance led to new pages that ultimately proved ineffective (accounting for 18.14% of cases). B. LLM's recommendation failed to lead to new pages altogether (69.75%).

Within Category A, we identified the following specific reasons:

- **A1**: Insufficient exploration of newly discovered pages, resulting in a failure to trigger subsequent increases in code coverage. During the Autonomous Exploration phase following the discovery of a new page, the testing tool might prematurely exit the page. This could occur through actions such as clicking the back button or by activating buttons that lead to previously explored pages.

- **A2**: The newly discovered page itself lacks significant potential to explore. This could be due to the page having relatively simple functionalities, thus failing to trigger more new pages, or because the new pages reuse code that has already been explored.

For Category B, we identified the following reasons:

- **B1**: LLM misinterpretation of UI control functionality. The LLM occasionally selected functionalities it believed would trigger new pages, which in reality did not. For instance, a "back to top" button on certain pages merely scrolls the list to the top rather than navigating to a new page, but the LLM might misinterpret it as navigation-related.
- **B2**: Navigation to previously explored pages. This occurs because the LLM suggests functionalities based solely on their navigational potential without considering their destination pages, particularly when these destinations can be reached through multiple paths. For example, in an app with a navigation bar, the LLM might suggest clicking an unexecuted tab button based on its navigational nature, unaware that its destination tab has already been explored via other routes.
- **B3**: Selection of unexplored functionalities that fail to trigger new pages. While these functionalities may lead to small coverage increases, they do not meet our effectiveness threshold, as exemplified by the instances at 43 and 48 minutes in Figure 3.
- **B4**: Selection of functionalities already explored on other pages. Although we prevent the LLM from selecting previously explored functionalities within the current page, identical functionalities may exist and have been explored on other pages.

These identified failure reasons provide valuable insights into potential avenues for improving LLMDroid, which we elaborate on in §5.

4.3 Performance and Cost under Different LLMs (RQ2)

This evaluation serves a dual purpose: assessing LLMDroid's performance and costs under LLMs with varying capabilities, and comparing it with existing LLM-based testing approaches - DroidAgent [10, 51] and GPTDroid [14, 35] (introduced in §1 and §2.1), which pursue high app testing coverage through step-by-step LLM guidance. For this evaluation, we select LLMDroid-Fastbot for comparison, and maintain the same experimental setup and dataset of 14 apps used in RQ1 for consistency. The evaluation metrics include code/activity coverage, API cost, and LLM interaction statistics (frequency and total time taken) per app-hour. The last two metrics further demonstrate the advantages of LLMDroid's design - particularly its concurrent GUI summary during Autonomous Exploration (§3.1.2), as opposed to the step-by-step LLM guidance approach where each LLM interaction blocks the testing process.

We apply three LLM models for this evaluation: GPT-3.5-turbo [5], GPT-4o-mini [7], and GPT-4o [6], where GPT-4o mini is positioned as a cost-efficient alternative to supersede GPT-3.5 with lower pricing but enhanced capabilities. Given the rapid development of LLMs and their changing prices, we note the model versions and their pricing during this experiment [13][2]. We did not evaluate GPTDroid and DroidAgent with GPT-4o as their step-by-step guidance nature would incur high API costs, and that they primarily used GPT-3.5 in their original settings.[3]

The experimental results are presented in Table 4, where the "Impr. over better" indicates LLMDroid's metrics improvement compared to the better-performing tool between DroidAgent and GPTDroid, regardless of the model applied.

[2]Prices per 1M tokens at experiment time in January 2025 [13]: gpt-3.5-turbo-0125: $0.50/$1.50 (input/output); gpt-4o-mini-2024-07-18: $0.15/$0.60; gpt-4o-2024-08-06: $2.50/$10.00.

[3]Except for DroidAgent which uses GPT-4 for task planning and GPT-3.5 for the more frequent task execution operations. In this evaluation, we configure DroidAgent to consistently use GPT-4o for task planning to preserve its performance, while applying GPT-3.5-turbo and GPT-4o-mini respectively for task execution.

Table 4. Performance Comparison of LLMDroid with Different LLMs against Existing Step-wise Tools

Metrics Per App-hour	Code Coverage	Activity Coverage	Cost	Interaction Frequency	Total Interaction Time
DroidAgent-3.5	10.77%	18.71%	$1.721	1761.00	2508.90s
DroidAgent-4o mini	10.89%	18.05%	$1.086	1691.86	2762.24s
GPTDroid-3.5	7.43%	7.33%	$0.207	724.29	2185.15s
GPTDroid-4o mini	7.45%	8.84%	$0.059	679.43	2131.72s
LLMDroid-3.5	13.01%	28.20%	$0.087	94.71	437.64s
Impr. over better	**19.47%**	**50.72%**	**-31.82%**	**617.38%**	**387.09%**
LLMDroid-4o-mini	13.47%	27.92%	$0.033	101.07	433.98s
Impr. over better	**23.69%**	**49.23%**	**77.78%**	**572.24%**	**391.20%**
LLMDroid-4o[4]	14.33%	28.28%	$0.494	102.00	399.40s
Impr. over better	**31.59%**	**51.15%**	**-88.01%**	**566.11%**	**433.73%**

Code Coverage and Activity Coverage. In terms of code coverage, as shown in Table 4, LLMDroid-Fastbot achieved optimal performance among tested models with GPT-4o, showing a 31.59% improvement over the better-performing baseline tool between DroidAgent and GPTDroid. The versions with GPT-4o-mini and GPT-3.5-turbo demonstrated improvements of 23.69% and 19.47%, respectively. **Comparing with Fastbot**'s original performance (11.36% code coverage, ref. Table 3), LLMDroid-Fastbot with GPT-4o, GPT-4o-mini, and GPT-3.5-turbo improved by **26.14%**, **18.57%**, and **14.52%**, respectively. These results indicate a positive correlation between LLMDroid's code coverage and the underlying LLM's capabilities. Interestingly, this pattern did not manifest in activity coverage, where LLMDroid-Fastbot performed consistently across all three models, maintaining approximately 50% improvement over DroidAgent. This consistency might be attributed to the coarse granularity of activity coverage as a metric, as empirically shown by Lan et al. [29] that activity coverage may sometimes fail to align with code coverage metrics.

The relatively lower performance of LLMDroid-Fastbot with GPT-3.5-turbo in code coverage can be attributed to its limitations in handling complex tasks. It often overlooked functionalities associated with UI widgets and therefore provided less accurate page summary. It also frequently suggested non-existent pages during page ranking. We posit that these complex tasks exceeded GPT-3.5's capabilities. These issues were substantially mitigated under GPT-4o-mini and GPT-4o.

LLM Interaction Frequency and Cost. By strategically combining autonomous exploration with selective LLM guidance, LLMDroid significantly reduces LLM interactions while achieving superior testing effectiveness. LLMDroid-Fastbot requires only around 100 interactions per hour, a substantial reduction compared to step-wise approaches like DroidAgent and GPTDroid, as shown in Table 4. This reduction in interaction frequency translates to decreased LLM interaction time, with LLMDroid-Fastbot requiring only around 400 seconds per hour. Moreover, through concurrent execution of LLM interactions during Autonomous Exploration (§3.1.2), LLMDroid-Fastbot experiences test-blocking LLM interaction times of merely 98s, 122s, and 107s (2.7%, 3.4%, and 3,0% of testing duration) for the three models. In contrast, DroidAgent and GPTDroid's step-by-step approaches result in LLM interactions blocking at each test action, leading to significant blocking

[4]LLMDroid-Fastbot's results under GPT-4o in RQ2's experiment are slightly lower than in RQ1 (ref. Table 3). Despite using identical APKs and experimental setups, we observed consistently lower performance in three apps during RQ2 (January 2025) compared to RQ1 (August 2024), potentially due to backend changes in these apps' servers.

times of 69.7% and 59.2%. **LLMDroid substantially mitigates a challenge in integrating LLMs into app testing - the overhead from LLM generation delays and network latency**, enabling the execution of more test actions (94.4% of original Fastbot) within the same timeframe.

The reduction in interaction frequency also results in significant cost efficiency: **LLMDroid-Fastbot incurs costs of only $0.49, $0.03, and $0.09 per hour** when using GPT-4o, GPT-4o-mini, and GPT-3.5-turbo respectively. Compared with DroidAgent, even using GPT-4o, LLMDroid-Fastbot costs less than half while achieving over 30% better code coverage. This gap widens when both use GPT-4o-mini, where LLMDroid-Fastbot achieves over 23.69% better code coverage at just 3% of the cost. While GPTDroid achieves comparable costs to LLMDroid-Fastbot, the code coverage difference remains significant.

When comparing the three models within LLMDroid-Fastbot, **GPT-4o-mini demonstrates particularly impressive cost efficiency, achieving 94% of GPT-4o's code coverage performance and 71% of its Fastbot improvement at only $0.03 per hour (6.7% of GPT-4o's cost). This cost-effectiveness highlights its huge potential for large-scale applications** in real-world scenarios. Even using GPT-4o, LLMDroid-Fastbot remains a reasonable cost of $0.49 per hour.

These results demonstrate that LLMDroid achieves better performance at lower costs compared to step-wise approaches, while providing valuable reference for those who consider applying other commercial, proprietary, or open-source models (e.g., Llama [12], DeepSeek [11]) to LLMDroid.

4.4 Determining UI Page Similarity Threshold T_S (RQ3)

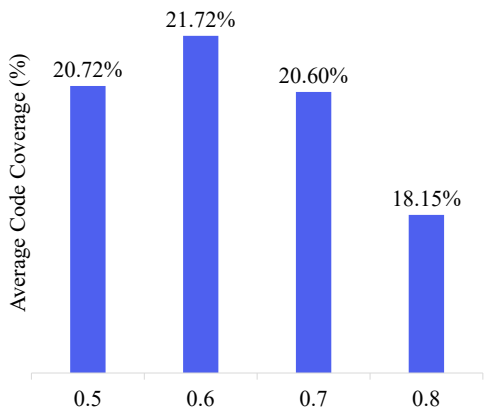

Fig. 4. Average Code Coverage under Different Thresholds

As discussed in §3.1.1, we need to set a similarity threshold T_S for determining whether two UI pages should be considered similar and merged into a PageCluster. This threshold value plays a vital role in LLMDroid's performance: a higher threshold could lead to frequent LLM interactions for summarizing UI pages explored during Autonomous Exploration, while a lower threshold might impair LLMDroid's ability to differentiate between distinct UI pages, resulting in inadequate UI page summary. This, in turn, hinders LLMDroid's capacity to recommend these pages and their associated functionalities during the LLM Guidance phase.

To determine the optimal similarity threshold value, we conducted a controlled experiment using LLMDroid-Fastbot with the GPT-4o model on a randomly selected subset of five apps from our dataset. We tested similarity thresholds of 0.5, 0.6, 0.7, and 0.8 across different experimental groups.

Initially, we intended to include a 0.9 threshold, but this was abandoned due to the prohibitively high frequency of LLM interactions and associated OpenAI API costs. The results, presented in Figure 4, show the average code coverage achieved for each threshold group. Our findings indicate that 0.6 is an appropriate threshold value, where higher thresholds not only increased LLM costs but also led to performance degradation, due to an excessive number of unfinished LLM queries for UI page summarization prior to the LLM Guidance phase.

5 Limitations and Future Work

Issues Regarding Code Coverage Monitoring. While Androlog [39] represents the state-of-the-art research tool in black-box code coverage monitoring and significantly outperforms its competitors, we observed apps exhibited dysfunctional behavior post-instrumentation, primarily due to Androlog's reliance on Soot for static APK instrumentation. There remains a need for improved code coverage analysis approaches for black-box apps. Notably, for developers testing their own apps—which represents the majority of LLMDroid's intended application scenarios—source code access enables the use of more established white-box coverage monitoring tools like Jacoco [8]. We provide such a Jacoco-based version of LLMDroid on our website.

Limited Page Comprehension of LLMs. During our testing process, we observed instances where LLMs failed to identify critical components in their functionality analysis based on HTML descriptions of pages. Notably, some widgets lack semantic information, either having no text or possessing resource-IDs that are difficult to interpret, which hinders LLM's page understanding.

To mitigate these issues, several approaches could be considered. One potential solution is the incorporation of traditional image caption models like pix2struct [30], as utilized in MemoDroid [31]. Another approach is the adoption of multimodal large models. Industry leaders such as Tencent with AppAgent [49] and MM-Navigator [47] with Microsoft's participation have already made pioneering attempts in applying multimodal large models for app testing.

However, we opted not to implement these solutions in the current version of LLMDroid due to their high computational costs, which contradict our intention for LLMDroid to be cost and time-efficient. As these technologies become more accessible and affordable in the future, their integration into LLMDroid may become a viable option to enhance its page comprehension capabilities.

More Detailed Experiments for Parameter Determination. LLMDroid incorporates numerous parameters that warrant more comprehensive experiments for optimal determination. While we have conducted experiments to determine the similarity threshold T_S for UI page clustering (§3.1.1) and the window size W for detecting low code coverage growth states (mentioned in §3.1.3), several other parameters were set empirically. These include the initial growth rate threshold T_0, as well as P and Q, which dictate the number of pages and functionalities within each page to be sent to the LLM in different stages of LLMDroid.

Moreover, it is crucial to recognize that these parameters may have interdependencies that affect the overall performance of LLMDroid. Despite our efforts in conducting experiments to determine some of these parameters, our current approach does not account for such dynamics and complexities. A more comprehensive study of these interrelationships and their impact on LLMDroid's performance is left for future work. Nevertheless, even with the current parameter setup, which may not be fully optimized, LLMDroid still demonstrates significant effectiveness.

Other Potential Improvements for LLMDroid. Our in-depth analysis of failed LLM guidance instances in §4.2 has inspired several avenues for potential improvement of LLMDroid. For instance, to address the B1 issue of LLM misinterpretation of UI control functionality, we could implement a more precise definition of navigation-related functionality. This, coupled with chain-of-thought reasoning, could significantly mitigate such misinterpretations. For B2 and B4 scenarios, where the LLM recommends previously explored pages or functionalities, we propose developing a

correlation mechanism that identifies pages accessible through different functionalities and identical functionalities existing across different pages. Implementing this feature could prevent redundant recommendations by the LLM, thereby improving the efficiency of the exploration process.

6 Conclusion

This paper introduces LLMDroid, a novel framework that enhances existing automated mobile GUI testing tools by efficiently leveraging Large Language Models (LLMs). LLMDroid addresses the challenges of time inefficiency and high costs associated with constant LLM querying in current approaches. By alternating between Autonomous Exploration and LLM Guidance stages, LLMDroid minimizes LLM interactions while maximizing their impact on test coverage. Our evaluation on 14 popular apps from Google Play, using three different testing tools, demonstrates significant improvements in both code coverage and activity coverage. Evaluation under different LLMs reveals that LLMDroid outperform existing step-wise approaches with significant cost efficiency. These results highlight the potential of LLMDroid to revolutionize automated mobile app testing by providing a more efficient and effective integration of LLMs into the testing process.

Data Availability

Our artifact, including the implementation of LLMDroid on Droidbot, Humanoid, and Fastbot, a guideline for integrating LLMDroid into existing testing tools, and our experimental datasets and results, is available at https://github.com/security-pride/LLMDroid.

Acknowledgment

We sincerely thank all the reviewers for their valuable suggestions and comments. This work was partially supported by the National Natural Science Foundation of China (grant No.62072046) and the Key R&D Program of Hubei Province (2023BAB017, 2023BAB079).

References

[1] 2024. AccessibilityService | Android Developers. https://developer.android.com/reference/android/accessibilityservice/AccessibilityService.
[2] 2024. bytedance/Fastbot_Android: Fastbot(2.0) is a model-based testing tool for modeling GUI transitions to discover app stability problems. https://github.com/bytedance/Fastbot_Android.
[3] 2024. Dice-Sørensen coefficient. https://en.wikipedia.org/wiki/Dice-S%C3%B8rensen_coefficient.
[4] 2024. Fragments | Android Developers. https://developer.android.com/guide/fragments.
[5] 2024. GPT-3.5 | OpenAI. https://platform.openai.com/docs/models/gpt-3-5-turbo.
[6] 2024. GPT-4o | OpenAI. https://openai.com/index/hello-gpt-4o/.
[7] 2024. GPT-4o mini | OpenAI. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/.
[8] 2024. Java Code Coverage Library. https://github.com/jacoco/jacoco.
[9] 2024. UI/Application Exerciser Monkey | Android Developers. https://developer.android.com/studio/test/other-testing-tools/monkey.
[10] 2025. coinse/droidagent: DroidAgent: Intent-Driven Mobile GUI Testing with Autonomous LLM Agents. https://github.com/coinse/droidagent.
[11] 2025. DeepSeek. https://www.deepseek.com/.
[12] 2025. Llama. https://www.llama.com/.
[13] 2025. Pricing - OpenAI API. https://platform.openai.com/docs/pricing.
[14] 2025. testinging6/GPTDroid. https://github.com/testinging6/GPTDroid.
[15] 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. doi:10.1109/ms.2014.55
[16] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for android: Are we there yet?(e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 429–440. doi:10.1109/ase.2015.89

[17] Chenhui Cui, Tao Li, Junjie Wang, Chunyang Chen, Dave Towey, and Rubing Huang. 2024. Large Language Models for Mobile GUI Text Input Generation: An Empirical Study. *arXiv preprint arXiv:2404.08948* (2024). doi:10.48550/arXiv.2404.08948

[18] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of android apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 481–492. doi:10.1145/3377811.3380402

[19] Sidong Feng and Chunyang Chen. 2024. Prompting Is All You Need: Automated Android Bug Replay with Large Language Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3608137

[20] Sidong Feng, Mulong Xie, and Chunyang Chen. 2023. Efficiency matters: Speeding up automated testing with gui rendering inference. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 906–918. doi:10.1109/icse48619.2023.00084

[21] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 269–280. doi:10.1109/icse.2019.00042

[22] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79. doi:10.1145/3695988

[23] Han Hu, Han Wang, Ruiqi Dong, Xiao Chen, and Chunyang Chen. 2024. Enhancing GUI Exploration Coverage of Android Apps with Deep Link-Integrated Monkey. *ACM Transactions on Software Engineering and Methodology* (2024). doi:10.1145/3664810

[24] Yongxiang Hu, Xuan Wang, Yingchuan Wang, Yu Zhang, Shiyu Guo, Chaoyi Chen, Xin Wang, and Yangfan Zhou. 2024. AUITestAgent: Automatic Requirements Oriented GUI Function Testing. *arXiv preprint arXiv:2407.09018* (2024). doi:10.48550/arXiv.2407.09018

[25] Yuchao Huang, Junjie Wang, Zhe Liu, Yawen Wang, Song Wang, Chunyang Chen, Yuanzhe Hu, and Qing Wang. 2024. Crashtranslator: Automatically reproducing mobile application crashes directly from stack trace. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3623298

[26] Bangyan Ju, Jin Yang, Tingting Yu, Tamerlan Abdullayev, Yuanyuan Wu, Dingbang Wang, and Yu Zhao. 2024. A Study of Using Multimodal LLMs for Non-Crash Functional Bug Detection in Android Apps. *arXiv preprint arXiv:2407.19053* (2024). doi:10.48550/arXiv.2407.19053

[27] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. 2018. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability* 68, 1 (2018), 45–66. doi:10.1109/tr.2018.2865733

[28] Yuanhong Lan, Yifei Lu, Zhong Li, Minxue Pan, Wenhua Yang, Tian Zhang, and Xuandong Li. 2024. Deeply Reinforcing Android GUI Testing with Deep Reinforcement Learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3623344

[29] Yuanhong Lan, Yifei Lu, Minxue Pan, and Xuandong Li. 2024. Navigating Mobile Testing Evaluation: A Comprehensive Statistical Analysis of Android GUI Testing Metrics. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 944–956. doi:10.1145/3691620.3695476

[30] Kenton Lee, Mandar Joshi, Iulia Raluca Turc, Hexiang Hu, Fangyu Liu, Julian Martin Eisenschlos, Urvashi Khandelwal, Peter Shaw, Ming-Wei Chang, and Kristina Toutanova. 2023. Pix2struct: Screenshot parsing as pretraining for visual language understanding. In *International Conference on Machine Learning*. PMLR, 18893–18912.

[31] Sunjae Lee, Junyoung Choi, Jungjae Lee, Hojun Choi, Steven Y Ko, Sangeun Oh, and Insik Shin. 2023. Explore, select, derive, and recall: Augmenting llm with human-like memory for mobile task automation. *arXiv preprint arXiv:2312.03003* 3, 7 (2023), 8. doi:10.48550/arXiv.2312.03003

[32] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 23–26. doi:10.1109/icse-c.2017.8

[33] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. 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, 1070–1073. doi:10.1109/ase.2019.00104

[34] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. 2023. Fill in the blank: Context-aware automated text input generation for mobile gui testing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1355–1367. doi:10.1109/icse48619.2023.00119

[35] 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. doi:10.1145/3597503.3639180

[36] Zhe Liu, Cheng Li, Chunyang Chen, Junjie Wang, Boyu Wu, Yawen Wang, Jun Hu, and Qing Wang. 2024. Vision-driven Automated Mobile GUI Testing via Multimodal Large Language Model. *arXiv preprint arXiv:2407.03037* (2024). doi:10.48550/arXiv.2407.03037

[37] Zhengwei Lv, Chao Peng, Zhao Zhang, Ting Su, Kai Liu, and Ping Yang. 2022. Fastbot2: Reusable automated model-based gui testing for android enhanced by reinforcement learning. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5. doi:10.1145/3551349.3559505

[38] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 153–164. doi:10.1145/3395363.3397354

[39] Jordan Samhi and Andreas Zeller. 2024. AndroLog: Android Instrumentation and Code Coverage Analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 597–601. doi:10.1145/3663529.3663806

[40] 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. doi:10.1145/3106237.3106298

[41] Ting Su, Jue Wang, and Zhendong Su. 2021. Benchmarking automated GUI testing for Android against real-world bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 119–130. doi:10.1145/3468264.3468620

[42] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224. doi:10.1145/1925805.1925818

[43] Bryan Wang, Gang Li, and Yang Li. 2023. Enabling conversational interaction with mobile ui using large language models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–17. doi:10.1145/3544548.3580895

[44] Dingbang Wang, Yu Zhao, Sidong Feng, Zhaoxu Zhang, William GJ Halfond, Chunyang Chen, Xiaoxia Sun, Jiangfan Shi, and Tingting Yu. 2024. Feedback-Driven Automated Whole Bug Report Reproduction for Android Apps. *arXiv preprint arXiv:2407.05165* (2024). doi:10.1145/3650212.3680341

[45] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2024. AutoDroid: LLM-powered Task Automation in Android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*. 543–557. doi:10.1145/3636534.3649379

[46] Hao Wen, Hongming Wang, Jiaxuan Liu, and Yuanchun Li. 2023. Droidbot-gpt: Gpt-powered ui automation for android. *arXiv preprint arXiv:2304.07061* (2023). doi:10.48550/arXiv.2304.07061

[47] An Yan, Zhengyuan Yang, Wanrong Zhu, Kevin Lin, Linjie Li, Jianfeng Wang, Jianwei Yang, Yiwu Zhong, Julian McAuley, Jianfeng Gao, et al. 2023. Gpt-4v in wonderland: Large multimodal models for zero-shot smartphone gui navigation. *arXiv preprint arXiv:2311.07562* (2023). doi:10.48550/arXiv.2311.07562

[48] Jiwei Yan, Hao Liu, Linjie Pan, Jun Yan, Jian Zhang, and Bin Liang. 2020. Multiple-entry testing of android applications by constructing activity launching contexts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 457–468. doi:10.1145/3377811.3380347

[49] Zhao Yang, Jiaxuan Liu, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2023. Appagent: Multimodal agents as smartphone users. *arXiv preprint arXiv:2312.13771* (2023). doi:10.48550/arXiv.2312.13771

[50] Faraz YazdaniBanafsheDaragh and Sam Malek. 2021. Deep GUI: Black-box GUI input generation with deep learning. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 905–916. doi:10.1109/ase51524.2021.9678778

[51] Juyeon Yoon, Robert Feldt, and Shin Yoo. 2024. Intent-Driven Mobile GUI Testing with Autonomous Large Language Model Agents. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 129–139. doi:10.1109/ICST60714.2024.00020

Received 2024-09-13; accepted 2025-01-14