

# AUITestAgent: Automatic Requirements Oriented GUI Function Testing

Yongxiang Hu\*

School of Computer Science  
Fudan University  
Shanghai, China

Yu Zhang

Meituan  
Shanghai, China

Xuan Wang\*

School of Computer Science  
Fudan University  
Shanghai, China

Shiyu Guo

Meituan  
Shanghai, China

Yingchuan Wang\*

School of Computer Science  
Fudan University  
Shanghai, China

Chaoyi Chen

Meituan  
Beijing, China

Xin Wang

School of Computer Science  
Fudan University  
Shanghai Key Laboratory of  
Intelligent Information Processing  
Shanghai, China

Yangfan Zhou

School of Computer Science  
Fudan University  
Shanghai Key Laboratory of  
Intelligent Information Processing  
Shanghai, China

## ABSTRACT

The Graphical User Interface (GUI) is how users interact with mobile apps. To ensure it functions properly, testing engineers have to make sure it functions as intended, based on test requirements that are typically written in natural language. While widely adopted manual testing and script-based methods are effective, they demand substantial effort due to the vast number of GUI pages and rapid iterations in modern mobile apps. This paper introduces AUITestAgent, the first automatic, natural language-driven GUI testing tool for mobile apps, capable of fully automating the entire process of GUI interaction and function verification. Since test requirements typically contain interaction commands and verification oracles, AUITestAgent can extract GUI interactions from test requirements via dynamically organized agents. Then, AUITestAgent employs a multi-dimensional data extraction strategy to retrieve data relevant to the test requirements from the interaction trace and perform verification. Experiments on customized benchmarks<sup>1</sup> demonstrate that AUITestAgent outperforms existing tools in the quality of generated GUI interactions and achieved the accuracy of verifications of 94%. Moreover, field deployment in Meituan has shown AUITestAgent's practical usability, with it detecting 4 new functional bugs during 10 regression tests in two months.

\*This work was performed during the authors' internship in Meituan.

<sup>1</sup><https://github.com/bz-lab/AUITestAgent>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

Conference'17, July 2017, Washington, DC, USA

© 2024 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## KEYWORDS

Automatic Testing, Mobile Apps, Functional Bug, In-context Learning

### ACM Reference Format:

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. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## 1 INTRODUCTION

Since GUI (Graphical User Interface) is the medium through which users interact with mobile apps, developers need to verify that it functions as expected. The most direct approach to this verification is to check the UI functionality according to requirements, specifically by executing GUI interactions and checking GUI responses. Despite being effective, the vast number of GUI pages and the rapid pace of app iterations result in a large volume of testing tasks. Therefore, to avoid expending a large amount of human effort, automation is essential.

Since the test requirements are usually expressed in natural language, we call this form of testing *natural language-driven GUI testing* for mobile apps, which remains an underexplored field. The most relevant work is natural language-driven GUI interaction[10, 28, 33]. They typically utilize large language models (LLMs) to analyze natural language commands and GUI pages, and ultimately generate interaction commands. However, unlike app interaction tasks, which focus on the interaction result and are goal-oriented, GUI testing requires checking the correctness of GUI responses during the interaction process and is, therefore, step-oriented. Consequently, the commonly used trial-and-error strategies in these interaction methods often result in inefficient interaction traces and insufficient success rate for GUI testing.

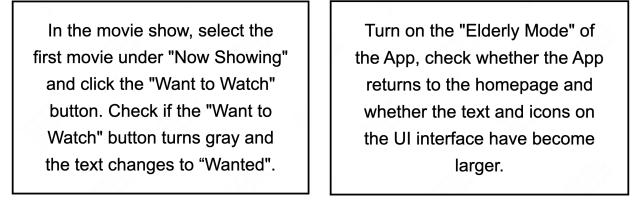
In industry practices, performing *natural language-driven GUI testing* still largely relies on intensive manual efforts. For example, such GUI testing in Meituan is mainly performed manually or through GUI testing scripts. Specifically, testing engineers should read and understand test requirements and manually select UI elements to interact and check by XPath, a language for pointing to different parts from a UI hierarchy file [8]. Then, multiple rules are designed to conduct GUI interactions and verifications. However, mobile apps today typically have multiple business lines (e.g., hotel booking, taxi booking, and food delivery), each involving a tremendous number of GUI functions and rapid development iterations. Consequently, conducting GUI function testing demands substantial manual effort for test script development and maintenance. This paper aims to reduce such manual efforts in *natural language-driven GUI testing*.

Despite GUI testing having more steps and therefore being more complex than interactions for testing engineers, we found that the step-oriented characteristic of GUI testing tasks is advantageous. This structure allows us to utilize step descriptions to design prompts, enabling us to achieve higher UI interaction success rates and verification accuracy, thereby attaining industry-level usability.

Although encouraging, there are still challenges posed by diverse testing requirements and the complex, huge amount of GUI pages. The key to reducing such human efforts lies in simplifying GUI testing tasks to align with LLMs' capabilities. In this regard, we designed AUITestAgent, the first automatic natural language-driven GUI testing tool for mobile apps. It takes GUI testing requirements as input, performs tests on the specified app, and outputs the test results.

As for AUITestAgent's implementation, three strategies are employed to simplify GUI testing tasks. Firstly, although GUI testing contains GUI interaction and function verification, the verification results do not actually impact the interaction. Therefore, AUITestAgent decouples interaction and verification into two separate modules, performing verification after interaction. As for diverse interaction commands, we designed two agent organization patterns. For simple instructions, AUITestAgent allows the *Executor* to perform directly, while complex instructions are broken down by the *Planner* first. As for GUI function verification, it requires processing numerous GUI screenshots, each containing tens of GUI elements. Since LLMs struggle to handle questions with many images attached accurately [13, 20], AUITestAgent extracts requirement-related information from three dimensions: GUI elements, GUI pages, and interaction traces. Then, verification will be conducted based on this integrated information.

We analyze the design effectiveness of AUITestAgent with a set of experiments. We prove that AUITestAgent perform the best in GUI interaction compared with AppAgent [35] and MobileAgent [28], which are both well-known natural language-driven interaction tools. Our verification experiment shows that AUITestAgent can recall 90% GUI functional bugs and provide reasonable explanations while maintaining a false positive rate of less than 5%. We also report our field experiences in applying AUITestAgent in Meituan. To date, AUITestAgent has been deployed across xx business lines and has detected xx new functional bugs. We summarize the contributions of this paper as follows.



(a) Requirement with specific steps      (b) Requirement in concise expression

**Figure 1: Practical Testing Requirements**

- We demonstrate that the step-oriented characteristic of GUI functional testing is well-suited for LLMs-based automatic testing.
- We propose AUITestAgent, the first automatic natural language-driven GUI testing tool for mobile apps.
- We show that AUITestAgent is an effective tool via real-world cases on commercial apps that serve tremendous end-users. We summarize the lessons learned, which can shed light on the automation of GUI testing.

## 2 PRACTICES OF GUI FUNCTION TESTING

The design of AUITestAgent is motivated by the practical testing status of Meituan's mobile app. Meituan is one of the largest online shopping platforms over the world, with nearly 700 million transacting users and about 10 million active merchants. Bugs on GUI pages in such apps would inevitably deteriorate user experience.

The Meituan app features a vast number of GUI pages and functionalities. Despite numerous automated testing tools proposed in academia [6, 9, 11, 12, 24, 27], they typically focus on detecting non-functional bugs (e.g., app crash) or require considerable manual intervention. Consequently, these tools fail to reduce the human effort required for GUI functional testing substantially.

During our practice in Meituan, GUI function verification is mainly performed manually or through hard-coded scripts. Specifically, testing engineers need to collaborate with both requirements and development teams to define test requirements. As shown in Figure 1, these requirements are typically in natural language. After the GUI development is completed, testing engineers verify these test requirements manually on real devices. Additionally, regression tests are set for core functionalities for cost considerations. Implementing GUI function regression testing involves developing hard-coded GUI testing scripts. This requires testing engineers to select UI elements that need interaction or verification from the UI hierarchy [3] and perform GUI interaction and function verification via rule development.

For commonly used commercial apps, there are usually many business lines, each with numerous functions, and each function involves multiple GUI interfaces. As a result, the workload for GUI functional testing is substantial. Although GUI test scripts can reduce some of the manual effort, they are prone to becoming outdated due to GUI changes brought by app iterations. Therefore, extensive use of these scripts can lead to considerable maintenance costs.

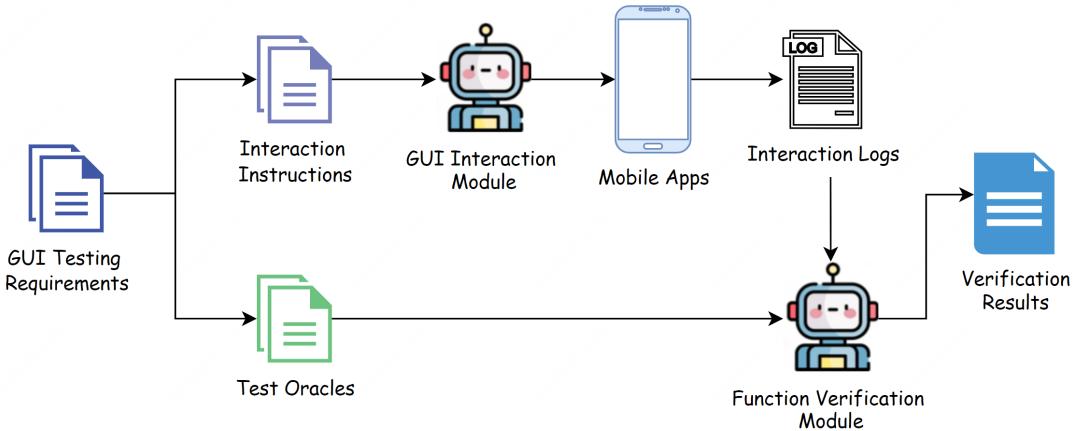


Figure 2: Overview of AUITestAgent

### 3 TOOL DESIGN AND TESTING WORKFLOW

#### 3.1 Overview

Figure 2 presents an overview of AUITestAgent, which comprises two main modules: GUI interaction and function verification. Specifically, AUITestAgent first analyzes the GUI test requirements to extract interaction commands and oracles for function verification. The GUI interaction module then dynamically selects the appropriate agent organization pattern based on the complexity of the interaction commands. If the commands consist of specific steps, the interactions will be performed directly by the *Executor*. Otherwise, if the commands are expressed concisely, a more complex pattern involving a *Planner* and *Monitor* will be used. Finally, the function verification module analyzes the interaction logs based on the test oracles. These logs contain information about the GUI pages encountered during the interactions, including screenshots, LLM outputs, and the UI actions performed. AUITestAgent extracts oracle-relevant data from the interaction logs and outputs the verification results and reason analysis.

#### 3.2 Natural Language Driven GUI Interaction

As discussed in Section 2, the interaction commands in different GUI test requirements vary significantly. As shown in Figure 1(a), target elements and corresponding UI actions are specified in explicit commands. In contrast, concise commands such as Figure 1(b) may abstract multiple UI actions into a single interaction command, which is more challenging to perform. Such differences require AUITestAgent to dynamically organize suitable agents to collaborate, ensuring the interaction commands are executed accurately.

Specifically, concise interaction commands require the agent to accurately interpret the user's intent behind the commands, assess the current state from GUI screenshots and interaction records, and determine the appropriate UI actions to take. Therefore, performing concise commands demands interaction memory and reasoning capabilities, which are difficult to achieve through a single agent due to the context length limitations. In contrast, while detailed interaction commands are easier to execute, using multiple agents increases unnecessary costs. It may also amplify the hallucination

problem [16] of LLMs due to cumulative effects, thereby reducing the quality of generated UI actions.

To address these challenges, the GUI interaction module of AUITestAgent employs two agent organization patterns, *Specific steps pattern* and *Concise expression pattern*, as shown in Figure 3. The GUI interaction module first uses LLMs to recognize the type of input interaction commands. If the interaction commands consist of a series of specific steps, AUITestAgent will convert them into a list of single-step UI actions via the specific step pattern. Otherwise, if the test requirement contains concise interaction commands, the GUI interaction module will use the concise expression pattern to handle them.

**3.2.1 Specific steps pattern.** Specific steps pattern executes each operation in the operation list sequentially through the collaboration of three LLM agents: the *Observer*, the *Selector*, and the *Executor*.

The **Observer** agent is to identify all interactive UI elements in the current GUI page and infer their functions. Specifically, the *Observer* processes the screenshot of the current GUI page using Set-of-Mark[34], assigning numeric IDs and rectangular boxes to all interactive elements. This approach highlights the interactive elements in the GUI screenshots, making it easier for LLMs to recognize. Finally, a multi-modal large language model (MLLM) is used to infer the function of each numbered UI element.

Although MLLMs possess image analysis capabilities, they do not perform well in observing GUI images, especially in recognizing UI elements and analyzing their functions due to the complexity of GUIs[14]. Besides, MLLMs are prone to Optical Character Recognition (OCR) hallucinations when recognizing text in images[15], while text is crucial for understanding the functionality of many elements within the GUI. To tackle these challenges, we enhance the GUI analysis capability of the *Observer* agent through multi-source input and knowledge base augmented generation.

Specifically, we select GUI screenshots and UI hierarchy file [3] as the *Observer*'s input. For the UI hierarchy file, *Observer* first parses the UI hierarchy file to identify all the interactive nodes(*i.e.*, whose clickable, enabled, scrollable, or long-clickable attribute is set to true). Related attributes of these nodes are also collected, including coordinate information (from 'bounds' attribute), textual

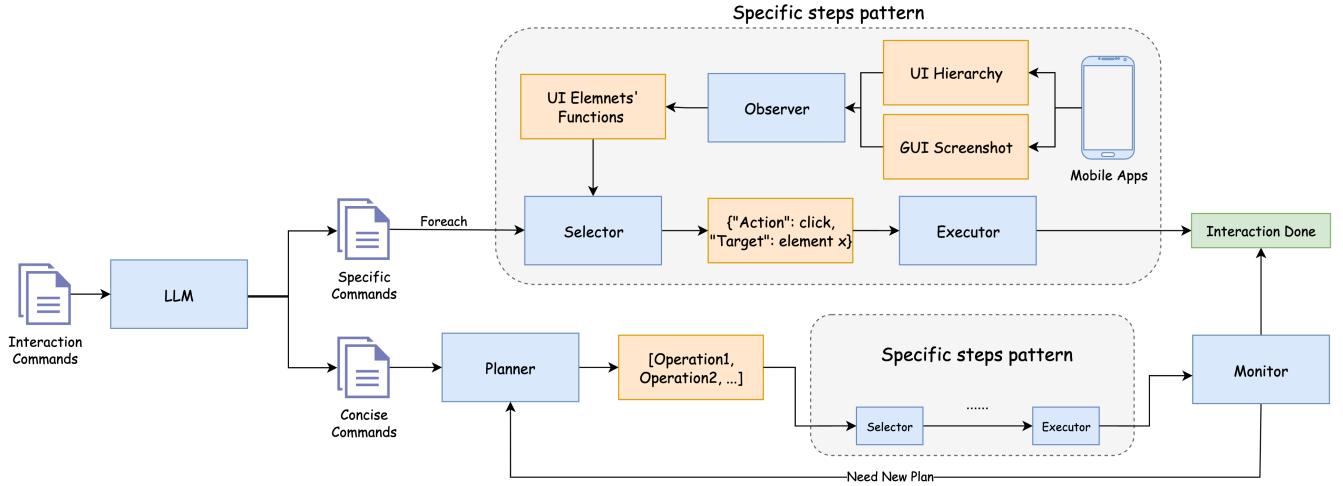


Figure 3: Workflow of GUI Interaction Module

information (from ‘text’ and ‘content-desc’ attribute), and functional information (*e.g.*, nodes with a “EditText” class are likely to be text input fields). For GUI screenshots, *Observer* uses the vision-ui model from Meituan [4] to identify all elements in the GUI and perform OCR to recognize the text visible to users. The recognition results serve as a supplement to the UI hierarchy, especially in cases where obtaining the UI hierarchy file fails or when the page contains WebView [7] components.

In order to make it easier for LLMs to describe the functions of elements, the *Observer* then marks the numerical ID and rectangular boxes of each element on the GUI image based on their coordinates. Finally, a system prompt that contains the processed screenshot and corresponding elements’ data, including element IDs, text information, and other property information, is assembled. This prompt is then used as input to the MLLM to infer the function of UI elements.

Given that the LLM is not well-acquainted with the design of mobile apps, particularly those in Chinese, we have observed that the *Observer* agent occasionally mispredicts the functions of UI elements, especially those UI icons that lack textual information. To tackle this problem, a UI element function knowledge base is formed to enhance element recognition. For instance, we manually collect and label some elements that are difficult to recognize, primarily icons that do not contain text. Each element in the knowledge base contains its screenshot, appearance descriptions, and function. When processing the GUI, the *Observer* uses CLIP [26] to match elements from the knowledge base with elements on the current page and then includes the matched elements’ appearance descriptions and functions in the system prompt for MLLM. This helps it infer the functions of these error-prone elements.

The **Selector** agent is set to select the operation target from the elements listed by *Observer* based on the natural language description of a single-step command. It generates the necessary parameters for the GUI action and then calls the corresponding function to perform. Specifically, we have defined the following UI actions for *Selector*:

- `click(target: int)`: This function is used to click an element listed by *Observer*. The parameter `target` is the element’s ID.
- `longPress(target: int)`: This function is similar to `click`, while its function is to long press a particular element.
- `type(target: int, text: str)`: this function is used to type the string ‘`text`’ into an input field. It is worth noting that the `target` parameter is the ID of the input field. The function will click on this element before typing to ensure the input field is activated.
- `scroll(target: int, dir: str, dist: str)`: This function is called to scroll on the element `target` in a direction ‘`dir`’(up, down, left, right) and distance ‘`dist`’(short, medium, long).

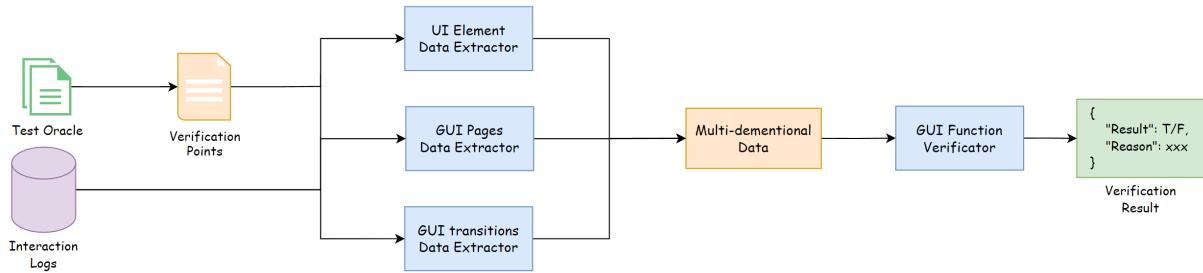
Finally, the **Executor** agent performs the UI actions generated by *Selector* on the app under testing (AUT). For actions targeting specific elements, the *Executor* first consults the coordinates provided in the *Observer*’s results, converting the ID of the target element into specific page coordinates. Then *Executor* performs the actual interaction action through the Android Debug Bridge (adb) tool[1].

**3.2.2 Concise expression pattern.** Concise expression pattern is also achieved through the collaboration of multiple agents. As shown in Figure 3, it adds two agents, *Planner* and *Monitor*, on top of the specific step pattern to handle concise commands.

The **Planner** agent formulates a plan based on the concise interaction commands, where each step in the plan is a specific interaction command aimed at progressing the completion of the interaction commands. Then, for each step in the plan, the specific step pattern is used for GUI action execution.

Another agent, the **Monitor**, is involved because, unlike the specific step pattern that requires only the sequential execution of each operation in the list, the concise expression pattern necessitates a role to determine whether the user’s commands have been completed (*i.e.*, whether the user’s intent has been achieved).

Specifically, the *Monitor* determines whether the interaction commands have been completed based on the current GUI screenshot and the record of executed operations. If the commands are completed, the GUI interaction module outputs the interaction log

**Figure 4: Workflow of Function Verification Module**

and stops. Otherwise, the *Monitor* describes the current GUI state and the operations it believes are necessary to complete the rest interactions. Its output serves as feedback for the *Planner*, and in the next iteration, the *Planner* will make a new plan.

### 3.3 Function Verification

To our knowledge, AUITestAgent is the first work to investigate GUI function verification. Since AUITestAgent decouples GUI interaction from verification, its verification is conducted based on interaction log analysis. Consequently, we formulate function verification as a semantic analysis problem of GUI pages.

Although MLLMs possess image analysis capabilities, we still need to simplify GUI function verification tasks to align with the capabilities of LLMs, to achieve stable performance. Specifically, there are three challenges in GUI function verification. First, the diversity of test oracles makes them difficult to process. Second, the sheer volume of data in interaction logs presents a challenge, as LLMs struggle to process excessively lengthy inputs [13], which may even exceed their maximum input capacity. Lastly, according to our experience, even the most capable MLLMs are not adept at semantic analysis of multiple images, especially complex images like GUI screenshots.

**3.3.1 Simplifying Test Oracles.** To tackle challenges posed by test oracles' complexity, AUITestAgent will convert them to a list of simple questions, termed *verification points* in this paper. According to our experiences in Meituan, test requirements often encompass multiple verification points, with each relating to different pages and UI components within the interaction log. As shown in Figure 4, AUITestAgent initially breaks down the test oracles extracted from the test requirements into individual verification points. Each verification point is tailored to address a single question, thereby streamlining the handling of test oracles.

**3.3.2 Multi-dimensional Data Extraction.** As for the challenges posed by the scale of log data and semantic analysis of images, AUITestAgent approaches information extraction from three perspectives, transforming it into textual information. Specifically, data contained in interaction logs can be categorized into three dimensions, as shown in Table 1. The GUI element dimension contains element functions and properties such as location and shape. Another dimension, the GUI page, includes page types (e.g., form page and list page) and categories (e.g., payment page, advertising page). And the page state transitions dimension, integrating

**Table 1: Categorization of interaction log data.**

Type	Description	Example
UI element	Element functions and properties	A submit button at the bottom-right corner.
GUI page	Page structures and categories	A list-structured advertisement page.
Page state transitions	integrate the first two aspects.	A payment page can be reached from an order-placement page by clicking the submit button.

the first two aspects (e.g., a payment page can be reached from an order-placement page by clicking *Confirm* button).

AUITestAgent leverages multiple agents to extract information relevant to the current verification point from these dimensions and records it as text. Considering the analysis of GUI element functions and page types has been performed in Section 3.2, AUITestAgent will reuse such content to enhance processing speed.

**3.3.3 Function Verification.** After requirement-related data are collected, AUITestAgent will integrate the verification points with corresponding data extracted from the logs. Finally, the judgment will be executed using MLLM. Furthermore, we employ the JSON mode [5] to standardize the output, ensuring a structured delivery of judgments and explanations.

## 4 EVALUATION

In this section, we evaluate the performance of AUITestAgent by answering the following research questions.

- **RQ1:** How do the quality and efficiency of AUITestAgent in generating GUI interactions compare to the existing approaches?
- **RQ2:** How do the effectiveness of AUITestAgent in perform GUI function verification?
- **RQ3:** To what extent can AUITestAgent perform natural language-driven testing in practical commercial mobile apps?

RQ1 assesses the effectiveness of AUITestAgent's GUI interaction model, examining its accuracy and efficiency in converting natural language commands into GUI actions. In RQ2, we construct a dataset containing 20 bugs, simulating real-world scenarios to evaluate the effectiveness of the GUI function verification module. In RQ3, we validate the practical applicability of AUITestAgent by examining its performance across GUI pages in Meituan.

### 4.1 Evaluation Setup

**4.1.1 Dataset.** To the best of our knowledge, AUITestAgent is the first automatic natural language-driven GUI testing tool for mobile

**Table 2: Difficulty levels of interaction tasks.**

Difficulty Levels	Definition	Examples	Step <sub>ideal</sub>	Score <sub>vag</sub>
L1	Step <sub>ideal</sub> + Score <sub>vag</sub> ≤ 3	Click the Like button on the first post.	1	1
		Click the Top Charts tab, then click to view the details of "Honor of Kings".	2	1
		Click the "Explore" tab, then click the "New releases".	2	1
L2	3 < Step <sub>ideal</sub> + Score <sub>vag</sub> ≤ 7	Click the "Movie Showings" button, click on the first movie under "Now Playing," and then click the "Want to Watch" button.	3	1
		Click the "Takeout" button, click on the "Food" section, and then click to enter the first store in the store list.	3	1
		View the detail page of the first notification in the notifications list.	3	3
L3	Step <sub>ideal</sub> + Score <sub>vag</sub> > 7	Search for "Hello World" and then play any song from the search list.	4	4
		Click the "Jobs" tab, search for QA Engineer in the search bar, click to view the details of any job from the search results, and then click Save.	6	1.5
		Send an email to {address1} and {address2} with the subject "Test" and the body "Hello".	8	2.66

**Table 3: The categories of test oracles.**

Category	Definition	Examples
No Response	App does not react when it should have.	No change after clicking order.
		Stays on message edit page after sending.
Unexpected Result	App's response violates the original functional logic.	Likes decreased by 1 after liking.
		Review page opened after clicking install.
Data Inconsistency	Self-conflicts of data presented in the GUI.	Ratings for the same seller are inconsistent across pages.
		Price does not equal original minus discount.

apps. Since there is no publicly available dataset, we contract testing scenarios and corresponding requirements for RQ1-RQ2. As shown in Table 2, given the diversity of practical test requirements depicted in Figure 1, we construct interaction requirements of various complexities along with corresponding verification oracles of various types.

For RQ1, we assessed the difficulty of interactions from two perspectives: the ideal number of interaction steps (*i.e.*, the number of steps required for manual interaction, recorded as  $Step_{ideal}$ ) and the vagueness of the requirements (*i.e.*, metric  $Score_{vag}$  is defined as the  $Step_{ideal}$  divided by the number of interaction commands in the requirements). As shown in Table 2, we use  $Step_{ideal}$  and  $Score_{con}$  to describe the complexity level of interaction tasks. Based on these two perspectives, we have categorized the interaction tasks into three difficulty levels. We then created ten interaction tasks for each difficulty level from eight popular apps *i.e.*, Meituan, Little Red Book, Douban, Facebook, Gmail, LinkedIn, Google Play and YouTube Music.

For RQ2, as the apps we selected are not open-source and bugs are not common in the online versions of these mature commercial apps, we needed a robust method to effectively measure AUITestAgent's performance in executing function verification. Two authors of this paper analyzed 50 recent historical anomalies at Meituan,

categorizing them into three types (as shown in Table 3). Based on these categories, we manually constructed 20 test oracles. Specifically, we selected 20 interaction tasks from the 30 crafted for RQ1 and manually created corresponding GUI anomaly manifestations. Consequently, as shown in Table 3, the data for RQ2 includes 20 test oracles, with each oracle associated with two interaction traces, one correct and the other having conflict with a verification point. Therefore, there are 40 verification tasks for RQ2, half of which contain anomalies.

For RQ3, we applied AUITestAgent to 18 test requirements from the real business scenarios at Meituan. By presenting the bugs identified by AUITestAgent in real scenarios, we demonstrate its practical effectiveness.

**4.1.2 Baselines.** To demonstrate the advantages of AUITestAgent in converting natural language into GUI interaction, we compared it with two state-of-the-art natural language-driven GUI interaction tools. AppAgent [35] is the first to propose a natural language-driven approach for mobile app interaction, featuring an innovative design that incorporates an exploratory learning phase followed by interaction. This addresses the challenge of LLMs' lack of familiarity with apps by offering two learning modes: unsupervised exploration and human demonstration. For fairness, in RQ1, we utilized the unsupervised exploration mode and set the maximum

**Table 4: Quality comparison of converted GUI interactions.**

Difficulty level	L1				L2				L3				Avg.			
	TC	CS	CT	SE												
AUITestAgent	<b>1.00</b>	<b>0.97</b>	<b>1.00</b>	<b>0.97</b>	<b>0.80</b>	<b>0.91</b>	<b>0.93</b>	<b>1.00</b>	<b>0.50</b>	<b>0.96</b>	<b>0.86</b>	<b>1.00</b>	<b>0.77</b>	<b>0.94</b>	<b>0.93</b>	<b>0.99</b>
MobileAgent	0.70	0.68	0.80	0.90	0.60	0.58	0.78	0.85	0.20	0.54	0.64	1.00	0.50	0.60	0.74	0.90
AppAgent	0.60	0.61	0.95	0.77	0.30	0.43	0.62	<b>1.00</b>	0.30	0.45	0.55	0.81	0.40	0.49	0.70	0.84

**Prompt:****Task**

The screenshots represent pages visited during the user's navigation. And the user's actions to navigate between these pages are **{actions}**.

Given the test task: **{test\_task}**, your reply should be in JSON format as shown below:

```
{"checkpoints": [{"checkpoint": {"judgement": "reason"}, ...}]}
```

**Examples**

For example, if the test task is .....

For another example, if the test task is .....

**Figure 5: Prompt of RQ2 Baseline**

number of exploration steps to  $Step_{ideal} + 5$ . Different from AppAgent, MobileAgent [28] employs a visual perception module for operation localization based solely on screenshots without needing underlying files. It utilizes LLM agents for holistic task planning and incorporates a self-reflection method to correct errors and halt once tasks are completed.

Since AUITestAgent is the first to focus on natural language-driven GUI function verification and there are no existing studies in this field, we constructed a verification method based on multi-turn dialogue [32] as a baseline. Specifically, We constructed a prompt of a two-turn dialogue with MLLM, as shown in figure 5.

**4.1.3 Implementation and Configuration.** For RQ1 and RQ2, to explore the upper limits of AUITestAgent's capabilities, we selected one of the state-of-the-art MLLM, GPT-4o [2], as the underlying model for AUITestAgent and three baselines. As for AUITestAgent's practical implementation within Meituan, we select another MLLM. Optimizing for efficiency, and given that all of these methods are device-independent, we conducted our experiments on multiple Android devices via the adb tool. For fairness, in RQ1, we set the maximum number of trial steps for all candidates to  $Step_{ideal} + 5$ .

## 4.2 RQ1: Quality of GUI Interactions

Initially, three authors manually executed each of the 30 interaction tasks. For each task, we recorded the manual interaction trace as

$Trace_i^{ma}$ , which serves as the ground truth for interaction task  $i$ .

$$Trace_i^{ma} = \{action_1, \dots, action_k\} \quad (1)$$

The ideal number of interaction steps,  $Step_{ideal}$ , is defined as the length of  $Trace_i^{ma}$ . For the action sequence generated by tool  $x$ , we denote it as  $Trace_i^x$ .

$$Trace_i^x = \{action_1, \dots, action_m\} \quad (2)$$

Due to the trial-and-error interaction strategy of the baseline interaction tool, we incorporated a parser in RQ1 to filter out redundant actions (*i.e.*, revert and repeat the last action) and meaningless interactions (*e.g.*, do not click on a UI element) in  $Trace_i^x$ . The filtered result is denoted as  $Trace\_P_i^x$ .

Four metrics are used to evaluate the interaction quality of AUITestAgent and baselines.

- *Task Completion*: Abbreviated as TC, indicates whether interaction tasks have been successfully executed.
- *Correct Step* (CS): Represents the correct actions in  $Trace_i^x$ .
- *Correct Trace* (CT): Represents the longest common prefix between  $Trace_i^x$  and the ground truth *i.e.*,  $Trace_i^{ma}$ . It can indicate the degree of completion of the interaction task  $i$ .
- *Step Efficiency* (SE) is used to measure the interaction efficiency of the action traces generated by AUITestAgent and baselines when task completion occurs. It is represented by the ratio of  $Trace\_P_i^x$  to  $Trace_i^x$ .

The formal representations of these metrics are as follows:

$$TC_i = \begin{cases} 1 & \text{if } Trace\_P_i^x == Trace_i^{ma} \\ 0 & \text{Otherwise} \end{cases} \quad (3)$$

$$CS_i = \frac{|Trace_i^x \cap Trace_i^{ma}|}{|Trace_i^x|} \quad (4)$$

$$CT_i = \frac{|LCP(Trace_i^x, Trace_i^{ma})|}{|Trace_i^{ma}|} \quad (5)$$

$$SE_i = \begin{cases} |Trace\_P_i^x| / |Trace_i^x| & \text{if } TC_i == 1 \\ 0 & \text{Otherwise} \end{cases} \quad (6)$$

As shown in Table 4, AUITestAgent achieved the highest scores across four types of metrics, with an overall task completion rate of 77%, significantly outperforming the two baseline methods. Although AppAgent incorporates an exploratory process, its direct use of the unprocessed UI hierarchy as input for LLMs results in excessively long and noisy data due to the complex GUI interfaces of the popular apps selected for RQ1. Consequently, this approach

**Table 5: Effectiveness comparison of function verification.**

Task Type	Correct Function Verification					Anomaly Detection				
	Oracle Acc.	Point Acc.	Reason Acc.	Completion Tokens	Prompt Tokens	Oracle Acc.	Point Acc.	Reason Acc.	Completion Tokens	Prompt Tokens
AUITestAgent	<b>0.95</b>	<b>0.97</b>	0.91	1345	34987	<b>0.85</b>	<b>0.91</b>	<b>0.84</b>	1288	33715
GPT-4o	0.90	0.95	<b>0.95</b>	<b>123</b>	<b>5695</b>	0.30	0.53	0.55	<b>129</b>	<b>5695</b>

leads to the lowest performance metrics for AppAgent across all evaluated criteria. This also demonstrates the effectiveness of the multi-source GUI understanding implemented in AUITestAgent.

Furthermore, as the difficulty of interaction tasks increases, four metrics for AUITestAgent and the baselines decline. Although Mobile-Agent performs well at levels L1 and L2, it struggles with tasks that involve numerous interaction commands or require concise execution. This reflects the significance of the design in AUITestAgent's GUI interaction module.

Therefore, the conclusion of RQ1 is that AUITestAgent can effectively convert GUI interaction tasks of different difficulty levels into corresponding GUI actions. Since the following function verification requires that interaction tasks be completed first, AUITestAgent possesses the capability to carry out subsequent function verification.

### 4.3 RQ2: Effectiveness of Function Verification

Since some test oracles in Meituan contain multiple verification points, 8 of the 20 test oracles we designed for RQ2 include multiple verification points. Before conducting the experiments, we manually divided these test oracles into 32 verification points to facilitate subsequent analysis.

We evaluated the verification performance of AUITestAgent and the multi-turn dialogue-based baseline from three perspectives: test oracle, verification point, and reasoning in explanation. To ensure fairness, we also tracked the number of tokens consumed by each method in the experiment. Specifically, we designed five metrics for RQ2.

*Oracle Acc.* represents the proportion of correctly judged tasks. A task is considered correctly judged by an oracle only if all included verification points are accurately assessed and the corresponding explanations are reasonable. Similarly, *Point Acc.* and *Reasoning Acc.* measure the accuracy of verification point judgments and the correctness of their explanations. Additionally, we present the *Completion Tokens* and *Prompt Tokens* for each task, which can indicate the economic costs associated with each method.

As shown in Table 5, AUITestAgent achieved an accuracy of 0.90 in making correct judgments on test oracles across two types of tasks. Due to the greater difficulty of anomaly detection compared to correct function verification, AUITestAgent achieved an *Oracle Acc.* of 0.85 in anomaly detection, slightly lower than in correct function verification. Considering that the 40 cases in RQ2 contain 64 verification points, of which 20 have anomalies, AUITestAgent successfully identified 18 anomalous points, resulting in 2 false positives. This corresponds to a recall rate of 90% and a false positive rate of 4.5%.

Conversely, the baseline method reached an accuracy of 1.0 in the correct function verification task but only achieved an *Oracle Acc.* of 0.35 in anomaly detection, indicating its difficulty in understanding questions in RQ3 and its tendency to output a 'no anomaly' judgment.

This data not only highlights the challenges LLMs face in handling GUI functional testing but also validates the effectiveness of multi-dimensional data extraction in AUITestAgent.

### 4.4 RQ3: Performance in Practical Implementations

In RQ3, we discuss the practical implementation of AUITestAgent in real business scenario *i.e.*, *Video* of Meituan's app, to show its practical effectiveness. Specifically, 10 test requirements with L1/L2 level interaction commands are selected to cover various checkpoints in short video playback and live show scenarios. These requirements have been integrated into the weekly regression automation process for this business line in Meituan.

In the video scenario within Meituan, in addition to supporting normal video playback functions (pause and play, dragging the progress bar, and swiping up and down to switch videos), it also supports clicking on merchant and product cards to view merchant information and purchase products. The GUI page in this scenario is complex, with numerous checkpoints that need testing. In the past, each iteration relied on manually writing test scripts or performing manual interactions for verification. These 10 test requirements contain 18 verification points in the video scenario, saving 1.5 Person-Days (PD) in each round of regression testing. Since AUITestAgent's launch, it has detected 4 new bugs from the video scenario of Meituan's app during 10 rounds of regression testing. Details of these bugs are shown in Table 6.

## 5 THREATS TO VALIDITY

One potential threat arises from the fact that AUITestAgent employs a structure that decouples interaction from verification. Specifically, in industry testing scenarios, a small number of requirements necessitate performing interactions during function verification (*e.g.*, repeatedly clicking the next video on a video playback page until a hotel-related live stream appears, then checking the details of the business to ensure the business name matches the live stream). Currently, AUITestAgent lacks the capability to verify such requirements. Given their rarity, this limitation does not have much impact on AUITestAgent's applicability in industry settings.

Another threat stems from AUITestAgent's reliance on general-purpose LLMs. General-purpose large language models such as GPT-4 are trained on extensive datasets [25]. However, these datasets lack domain knowledge crucial for GUI function testing, such as app

Table 6: RQ3-bug Information.

Test requirement	Bug Image	Test Result
Swipe up once, tap on the shop information card at the bottom of the screen (the card contains information such as the shop name and rating), then swipe down once. Check if the shop details half-screen modal appears after tapping the shop information card in step two. Verify that the shop details half-screen modal disappears after swiping down in step three.		<p>“reason”: “Based on the user’s navigation and interaction records, the test task was not successfully fulfilled. The records do not provide clear evidence that the store details half-popup layer was displayed or dismissed. Specifically, after the user clicked on the store information card (Action 1), there is no indication on Page 2 that the store details half-popup layer was shown. Additionally, after the user swiped down (Action 2), Page 3 does not indicate that the store details half-popup layer was dismissed. Both Page 0 and Page 3 explicitly state that the store details half-popup layer is ‘invisible’. Therefore, the required checks for the popup layer’s visibility and dismissal were not confirmed.”</p>

function designs and GUI components. Consequently, we acknowledge the current limitations of AUITestAgent’s capabilities. In the future, we plan to construct a dataset specific to GUI functions and enhance AUITestAgent’s proficiency in UI testing by employing a retrieval-augmented generation (RAG) approach [18].

## 6 RELATED WORK

Since the quality of the GUI is closely related to user experience, various testing methods targeting bugs in GUI have been proposed [6, 17, 31]. Traditionally, developing GUI testing scripts has been the main method for automating GUI function tests in industry practice. Since these scripts are labor-intensive to develop and are easily obsolete, several studies have been conducted on automatically generating or repairing GUI testing scripts. For instance, AppFlow[12] utilizes machine learning techniques to automatically identify screen components, enabling testers to write modular testing libraries for essential functions of applications. CraftDroid[19] uses information retrieval to extract human knowledge from existing test components of applications to test other programs. CoSer[9] constructs UI state transition graphs by analyzing app source code and test scripts to repair obsolete scripts. Although AUITestAgent is also a method for testing GUI functions, it differs from these studies as it neither relies on nor generates hard-coded testing scripts, thereby offering better generalizability and maintainability.

Due to their extensive scale of training data and robust logical reasoning abilities, LLMs are increasingly utilized in mobile app testing. Several methods are proposed using LLMs to generate GUI interactions or translating natural language commands into GUI actions. QTypist [21] focuses on generating semantic textual inputs for form pages to enhance exploration testing coverage. GPT-Droid [22] extracts GUI page information and widget functionality from the UI hierarchy file, using this data to generate human-like interactions. AppAgent [35], Mobile-Agent [28], DroidBot-GPT [30]

and AutoDroid [29] leverage LLMs to process natural language descriptions and GUI pages, translating natural language commands into GUI actions. Different from these tools, AUITestAgent can directly perform GUI testing on mobile apps. Moreover, our experiments demonstrate that the techniques employed by AUITestAgent, including dynamically organizing agents, lead to the highest quality of GUI interaction generation.

VisionDroid [23] focuses on non-crash bug detection of GUI pages, highlighting the absence of testing oracles and utilizing large language models (LLMs) to detect unexpected behaviors. In contrast, AUITestAgent concentrates on the industry’s practical needs. By emphasizing the challenges of implementing practical testing requirements, we have developed an industry-applicable automatic natural language-driven GUI testing method.

## 7 CONCLUSION

In this paper, we propose AUITestAgent, an automatic approach to perform natural language-driven GUI testing for mobile apps. In order to extract GUI interactions from test requirements, AUITestAgent utilizes dynamically organized agents and constructs multi-source input for them. Following this, a multi-dimensional data extraction strategy is employed to retrieve data relevant to the test requirements from the interaction trace to perform verifications. Our experiments on customized benchmarks show that AUITestAgent significantly outperforms existing methods in GUI interaction generation and could recall 90% injected bugs with a 4.5% FPr. Furthermore, unseen bugs detected from Meituan show the practical benefits of using AUITestAgent to conduct GUI testing for complex commercial apps. These findings highlight the potential of AUITestAgent to automate the GUI testing procedure in practical mobile apps.

## REFERENCES

- [1] Accessed: 2024. *Android Debug Bridge (adb)*. <https://developer.android.com/tools/adb>
- [2] Accessed: 2024. *Hello GPT-4o*. <https://openai.com/index/hello-gpt-4o/>
- [3] Accessed: 2024. *Layouts in Views*. <https://developer.android.com/develop/ui/views/layout/declaring-layout>
- [4] Accessed: 2024. *Meituan-Dianping/vision-ui: Visual UI analysis tools*. <https://github.com/Meituan-Dianping/vision-ui>
- [5] Accessed: 2024. *OpenAI Platform: JSON mode*. <https://platform.openai.com/docs/guides/text-generation/json-mode>
- [6] Accessed: 2024. *UI/Application Exerciser Monkey*. <https://developer.android.com/studio/test/other-testing-tools/monkey>
- [7] Accessed: 2024. *WebView | Android Developers*. <https://developer.android.com/reference/android/webkit/WebView>
- [8] Accessed: 2024. *XML Path Language (XPath)*. <https://www.w3.org/TR/1999/REC-xpath-19991116/>
- [9] Shaoheng Cao, Minxue Pan, Yu Pei, Wenhua Yang, Tian Zhang, Linzhang Wang, and Xuandong Li. 2024. Comprehensive Semantic Repair of Obsolete GUI Test Scripts for Mobile Applications. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 90:1–90:13. <https://doi.org/10.1145/3597503.3639108>
- [10] 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, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 67:1–67:13. <https://doi.org/10.1145/3597503.3639137>
- [11] 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 *Proc. of the 41st International Conference on Software Engineering, ICSE. IEEE / ACM*, 269–280.
- [12] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *Proc. of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*. ACM, 269–282.
- [13] Wenbo Hu, Yifan Xu, Yi Li, Weiyue Li, Zeyuan Chen, and Zhiwen Tu. 2024. BLIVA: A Simple Multimodal LLM for Better Handling of Text-Rich Visual Questions. In *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2024, February 20-27, 2024, Vancouver, Canada*, Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan (Eds.). AAAI Press, 2256–2264. <https://doi.org/10.1609/AAAI.V38I3.27999>
- [14] Yongxiang Hu, Jiazheng Gu, Shuqing Hu, Yu Zhang, Wenjie Tian, Shiyu Guo, Chaoyi Chen, and Yangfan Zhou. 2023. Appaction: Automatic GUI Interaction for Mobile Apps via Holistic Widget Perception. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 1786–1797. <https://doi.org/10.1145/3611643.3613885>
- [15] Wen Huang, Hongbin Liu, Minxin Guo, and Neil Zhenqiang Gong. 2024. Visual Hallucinations of Multi-modal Large Language Models. *CoRR* abs/2402.14683 (2024). <https://doi.org/10.48550/ARXIV.2402.14683> arXiv:2402.14683
- [16] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Yebin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.* 55, 12 (2023), 248:1–248:38. <https://doi.org/10.1145/3571730>
- [17] Wing Lam, Zhengkai Wu, Dengfeng Li, Wenyu Wang, Haibing Zheng, Hui Luo, Peng Yan, Yuetang Deng, and Tao Xie. 2017. Record and replay for Android: are we there yet in industrial cases? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 854–859. <https://doi.org/10.1145/3106237.3117769>
- [18] Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.).
- [19] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test Transfer Across Mobile Apps Through Semantic Mapping. In *Proc. of the 34th IEEE/ACM International Conference on Automated Software Engineering, ASE. IEEE*, 42–53.
- [20] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Trans. Assoc. Comput. Linguistics* 12 (2024), 157–173. [https://doi.org/10.1162/TACL\\_A\\_00638](https://doi.org/10.1162/TACL_A_00638)
- [21] 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 *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1355–1367.
- [22] 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 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 100:1–100:13. <https://doi.org/10.1145/3597503.3639180>
- [23] 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. *CoRR* abs/2407.03037 (2024).
- [24] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 94–105.
- [25] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). <https://doi.org/10.48550/ARXIV.2303.08774> arXiv:2303.08774
- [26] Alex Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models From Natural Language Supervision. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event, Marina Meila and Tong Zhang (Eds.)*. PMLR, 8748–8763.
- [27] 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, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 245–256.
- [28] Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024. Mobile-Agent: Autonomous Multi-Modal Mobile Device Agent with Visual Perception. *CoRR* abs/2401.16158 (2024). <https://doi.org/10.48550/ARXIV.2401.16158> arXiv:2401.16158
- [29] Hao Wen, Yuanchun Li, Guohong Liu, Shanhuai Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2023. Empowering LLM to use Smartphone for Intelligent Task Automation. *CoRR* abs/2308.15272 (2023).
- [30] Hao Wen, Hongming Wang, Jiaxuan Liu, and Yuanchun Li. 2023. DroidBot-GPT: GPT-powered UI Automation for Android. *CoRR* abs/2304.07061 (2023).
- [31] Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. 2023. An Empirical Study of Functional Bugs in Android Apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 1319–1331. <https://doi.org/10.1145/3597926.3598138>
- [32] Yi Xu, Hai Zhao, and Zhuosheng Zhang. 2021. Topic-Aware Multi-turn Dialogue Modeling. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 14176–14184.
- [33] An Yan, Zhengyuan Yang, Wanrong Zhu, Kevin Lin, Linjie Li, Jianfeng Wang, Jianwei Yang, Yiwu Zhong, Julian J. McAuley, Jianfeng Gao, Zicheng Liu, and Lijuan Wang. 2023. GPT-4V in Wonderland: Large Multimodal Models for Zero-Shot Smartphone GUI Navigation. *CoRR* abs/2311.07562 (2023). <https://doi.org/10.48550/ARXIV.2311.07562> arXiv:2311.07562
- [34] Jianwei Yang, Hao Zhang, Feng Li, Xueyan Zou, Chunyuan Li, and Jianfeng Gao. 2023. Set-of-Mark Prompting Unleashes Extraordinary Visual Grounding in GPT-4V. *CoRR* abs/2310.11441 (2023). <https://doi.org/10.48550/ARXIV.2310.11441> arXiv:2310.11441
- [35] Chi Zhang, Zhao Yang, Jiaxuan Liu, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2023. AppAgent: Multimodal Agents as Smartphone Users. *CoRR* abs/2312.13771 (2023). <https://doi.org/10.48550/ARXIV.2312.13771> arXiv:2312.13771