KuiTest: Leveraging Knowledge in the Wild as GUI Testing Oracle for Mobile Apps

1st Yongxiang Hu
School of Computer Science
Fudan University
Shanghai, China
yongxianghu23@m.fudan.edu.cn

2nd Yu Zhang
Meituan
Shanghai, China
zhangyu233@meituan.com

3rd Xuan Wang
School of Computer Science
Fudan University
Shanghai, China
xuanwang23@m.fudan.edu.cn

4th Yingjie Liu
School of Computer Science
Fudan University
Shanghai, China
yjliu24@m.fudan.edu.cn

5th Shiyu Guo
Meituan
Shanghai, China
guoshiyu03@meituan.com

6th Chaoyi Chen
Meituan
Beijing, China
chenchaoyi@meituan.com

7th Xin Wang
School of Computer Science
Fudan University
Shanghai Key Laboratory of
Intelligent Information Processing
Shanghai, China
xinw@fudan.edu.cn

8th Yangfan Zhou
School of Computer Science
Fudan University
Shanghai Key Laboratory of
Intelligent Information Processing
Shanghai, China
zyf@fudan.edu.cn

***Abstract*—In industrial practice, UI (User Interface) functional bugs typically manifest as inconsistent UI input and corresponding response. Such bugs can deteriorate user experiences and are, therefore, a major target of industrial testing practice. For a long time, testing for UI functional bugs has relied on rule-based methods, which are labor-intensive for rule development and maintenance. Given that the UI functional bugs typically manifest where an app's response deviates from the user's expectations, we proposed the key point of reducing human efforts lies in simulating human expectations. Due to the vast in-the-wild knowledge of large language models (LLMs), they are well-suited for this simulation. By leveraging LLMs as UI testing oracle, we proposed `KuiTest`, the first rule-free UI functional testing tool we designed for Meituan, one of the largest E-commerce app providers serving over 600 million users. `KuiTest` can automatically predict the effect of UI inputs and verify the post-interaction UI response. We evaluate the design of `KuiTest` via a set of ablation experiments. Moreover, real-world deployments demonstrate that `KuiTest` can effectively detect previously unknown UI functional bugs and significantly improve the efficiency of GUI testing.**

***Index Terms*—Automatic GUI Testing, Functional Bug, Mobile Apps**

I. Introduction

As mobile apps become increasingly complex nowadays, bugs are inevitable [1]. Tremendous research efforts have been put into combating non-functional bugs, *e.g.*, those cause laggy UI (User Interface) [2], [3], freezing UI [4], [5], or app to crash [6], [7]. Functional bugs, those cause incorrect UI interactions (*i.e.*, inconsistent UI input and its corresponding response), are still a challenge to quality assurance of apps in industry practice [8]–[10]. In industrial practice, functional bug testing largely relies on manual efforts, with rule-based testing scripts. Specifically, such scripts check the correctness of UI interactions with pre-defined rules according to the business logic of the target app [11], [12], which requires substantial manual effort to develop and maintain.

Although there is a rich literature on automatic functional bug testing tools [11], [13], [14], they typically attempt to reduce manual effort by implementing more generalized pre-defined rules, such as replacing the hard-coded scripts with natural language descriptions. However, these methods still require manual rule definition and adaptation, which restricts their practical usability in industrial environments.

Hence, to reduce such manual effort, this paper aims to develop a rule-free tool for automatic UI functional testing of mobile apps. Given that incorrect UI interactions typically manifest where an app's response deviates from common user expectations, simulating these expectations can allow us to eliminate traditional rule-based detection methods. Since large language models (LLMs), especially those general-purpose models, are trained on vast datasets, their vast in-the-wild knowledge enables them to simulate such human expectations at a low cost. In this paper, we share our experiences in leveraging large language models as UI test oracles, facilitating their use in the design of `KuiTest`, a rule-free UI function testing tool for mobile apps.

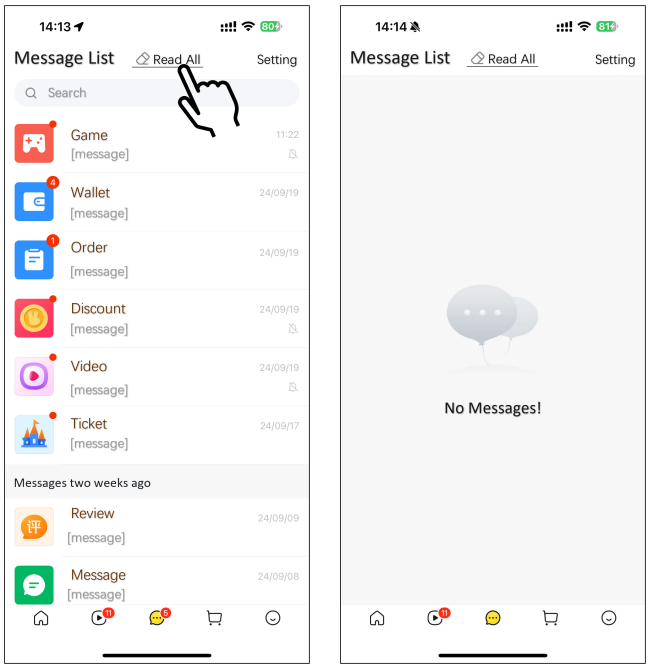
However, since LLMs are not trained for UI testing, two challenges arise in KuiTest's implementation. First, GUI interaction testing involves domain-specific knowledge, making it difficult for general LLMs, *i.e.*, LLMs trained for general purposes like ChatGPT [15], to handle. To address this, KuiTest mimics manual testing by splitting functional testing into two sub-tasks: predicting the effect of a UI input (*e.g.*, the outcome after clicking a heart-shape button) and verifying the post-interaction UI response accordance. This decomposition allows the UI function testing task to better align with the capabilities of general LLMs. Second, although users can easily understand UI inputs and their responses, both our field experiences and existing research [16] show that LLMs struggle to handle these tasks via screenshots. To solve this, KuiTest constructs multimodal inputs by providing both textual descriptions alongside with screenshots to assist LLMs in better understanding the GUI, as LLMs perform significantly better when processing text compared to images [17]–[19].

Comprehensive experiments are conducted to evaluate the design effectiveness of KuiTest. Specifically, our ablation experiment shows the benefits of testing task decomposition. Furthermore, results show that KuiTest outperforms the baselines in predicting UI input influence and verifying post-interaction UI response. In addition, KuiTest can recall 86% UI functional bugs and provide reasonable explanations while maintaining a false positive rate of less than 1.2%. We have also field-deployed KuiTest to test a real-life shopping app from Meituan, one of the largest E-commerce app providers serving over 600 million users. It has successfully detected 41 previously unknown bugs and significantly reduced manual effort in GUI testing.

We summarize the contributions of this paper as follows.

- We share our field experience on UI functional bug detection in industry practice. We show that the key obstacle to its automation is the poor generalization ability of the widely-used rule-based oracles.
- By studying the manifestations of UI functional bugs, we propose that these rule-based oracles can be replaced by users' expectations. Given vast in-the-wild knowledge of LLMs, they are well-suited to simulate such human expectations. We propose the idea of an LLM-based oracle for UI functional testing.
- We design and implement KuiTest, the first rule-free UI functional testing tool for mobile apps. Comprehensive experiments and real-world case studies on widely-used mobile apps demonstrate the effectiveness of KuiTest. We summarize the lessons learned, offering insights into the automation of GUI testing.

The rest of the paper is organized as follows. Section II introduces practices of mobile app testing and highlights their limitations in two representative scenarios. In Section III, we demonstrate the design of KuiTest by proposing each of its two modules. Then, three questions are answered in Section IV to evaluate KuiTest's performance and usability. Section V elaborates our practical cases for the deployment of KuiTest.



(a) Pre-interaction screenshot (b) Confusing GUI response

Fig. 1: Example of UI functional bug

Threats to validity and future work are discussed in Section VI. Finally, the paper is concluded in Section VIII.

II. Motivation Study

In this section, we demonstrate the motivation of KuiTest by presenting our practical experience in Meituan. The implementation of KuiTest will fill two gaps in industrial practices: automatic UI functional testing for newly launched features and functional verification during regression testing.

A. App Launch on New Platforms

As demonstrated earlier, UI functional testing in the industry is primarily conducted through hard-coded scripts. Specifically, testers bind UI interactions and verification to UI elements in the hierarchy files based on attributes or relative path [20]. Therefore, in popular commercial apps, a large number of UI testing scripts are developed to detect logic anomalies from GUI in time.

However, when an app is launched on a new platform (*e.g.*, HarmonyOS Next [21]), the changes in the hierarchy structure often cause these scripts to become obsolete. Such obsolescence will lead to large volumes of manual script repairing, making it challenging to complete the migration of GUI logic testing within a short time. Additionally, the bug count tends to be high when an app is newly launched on a platform, significantly increasing the workload for engineers performing manual testing to detect UI functional bugs.

To reduce the workload of manual testing, there is a great demand for a functional testing method that can quickly adapt to new platforms. Specifically, unlike manual or script-based

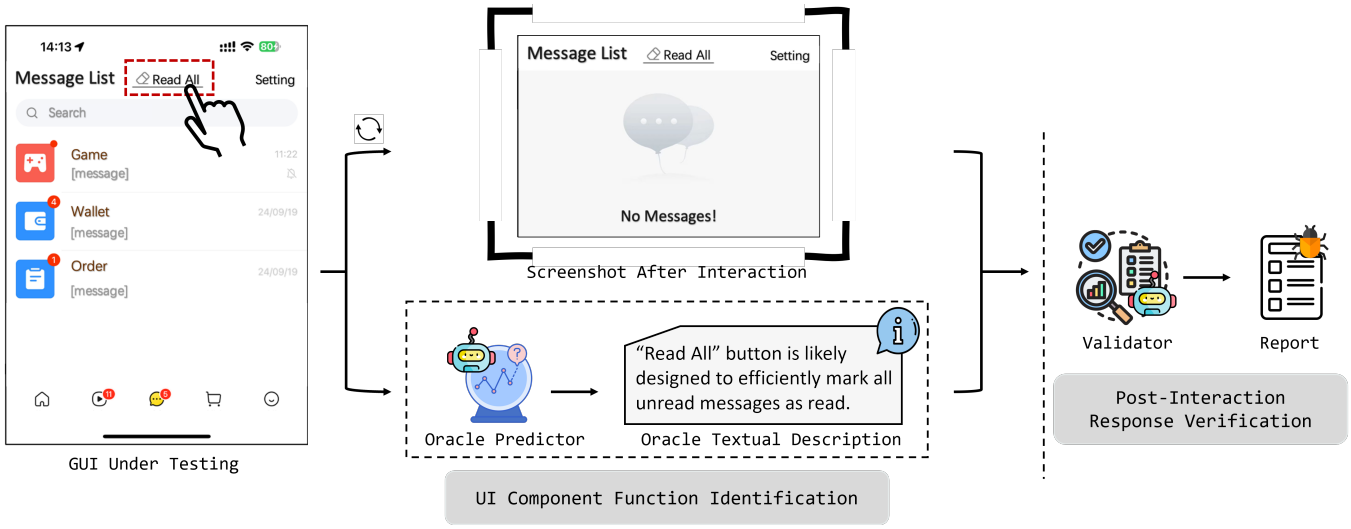


Fig. 2: Overview of KuiTest

testing, this method does not require the ability to detect functional bugs triggered by complex traces of UI. Being capable of automatically traversing UI pages of the app under testing (AUT) and alerting human testers to high-risk areas would significantly reduce the workload of manual testing and improve the efficiency of UI functional testing.

B. Semantic Verification in Regression

In addition to platform changes, continuously iterated UI designs also render widely-used UI testing scripts obsolete, resulting in high maintenance costs [12]. Consequently, the huge manual efforts in scripts' development and maintenance make it impractical for script-based testing to cover all UI functions in a complex app. Given the increasing complexity of mobile apps today, it is now prevalent for apps to go live without comprehensive testing.

Although the automation of functional testing remains underdeveloped, the automatic detection of non-functional bugs is well-studied [6], [22]–[24]. For example, random testing tools, like Google's Monkey [6], have retained popularity in the industry due to their low execution costs. These tools operate by randomly interacting with the AUT, simulating various user interactions across the app until a crash is triggered. They can achieve this without any predefined test scripts, making them a cost-effective solution for detecting UI crash bugs.

However, their reliance on simple oracles makes them incapable of detecting UI functional bugs. In this regard, we aim to design a new testing tool by attaching new oracles to their exploration strategy, and therefore achieve automatic UI functional testing. Specifically, by deploying it in UI regression testing, this tool can significantly broaden the coverage of UI testing, providing early warnings for UI functional bugs before going live.

III. Tool Design

A. Overview

Figure 2 shows the overview of KuiTest, the UI functional testing tool we designed for mobile apps. Given that UI functional testing is a complicated task for general LLMs, we divide this process into two modules for KuiTest to enhance the reliability of LLMs' outputs. Specifically, KuiTest first predicts the effect of UI inputs by analyzing the function of interactive UI components (*e.g.*, a confirm button or heart-shaped icon). Second, KuiTest combines the predicted component function with the post-interaction screenshot and prompts the LLMs to determine whether any functional bugs are present.

B. Decomposition of Testing Task

Since UI functional testing requires understanding the semantics behind the GUI, it is challenging for LLMs to handle. Therefore, directly using LLMs for UI functional testing would introduce reliability issues, such as hallucinations [25], [26] and misjudgment.

Given the challenges of directly leveraging LLMs to handle a complex task, there is an alternative to breaking down the whole task into more manageable parts, a widely-used strategy [27], [28]. This decomposition can enhance the reliability of LLM outputs when handling complex tasks. Therefore, in KuiTest, we followed the idea of task decomposition to improve LLMs' usability in UI functional testing.

Indeed, task decomposition for LLMs, especially in industrial settings, is a trade-off between reliability and efficiency. While breaking down tasks can improve the reliability of LLM responses by reducing the complexity of the problem, dividing a task into too many small parts can result in higher computational costs and lead to low efficiency. Therefore, we aim to find a balance that ensures testing quality while maintaining efficiency and keeping a low cost.

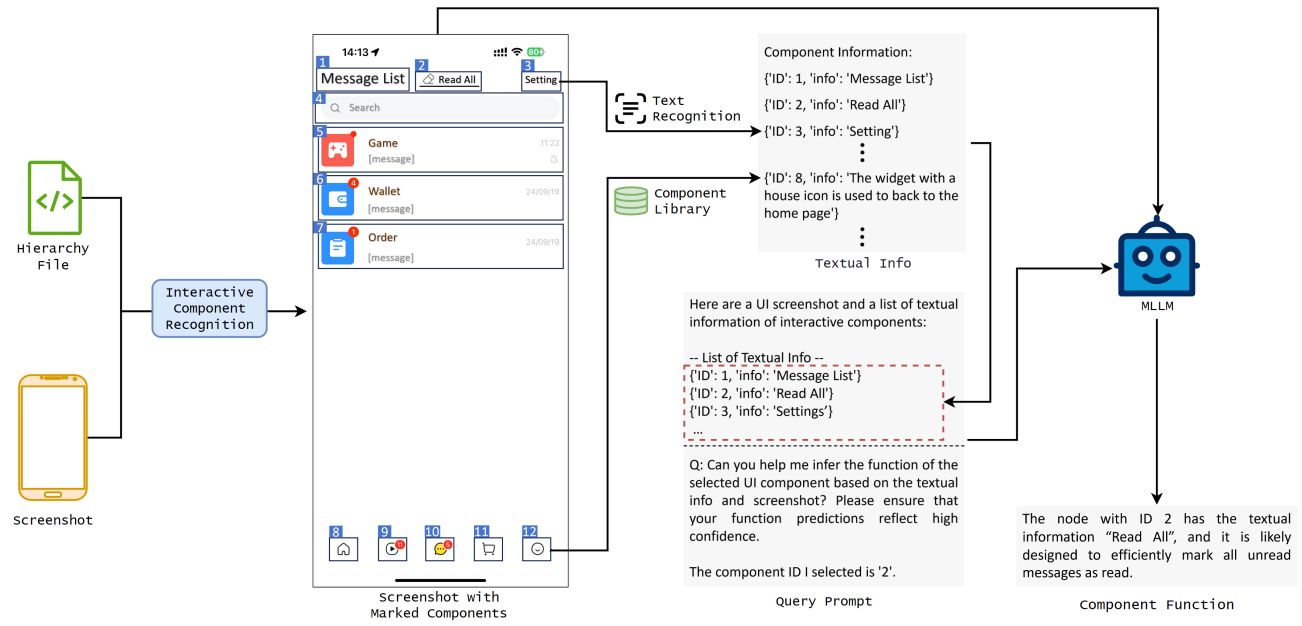


Fig. 3: Workflow of UI component function identification module in KuiTest

To achieve this balance, we borrowed the procedure from manual testing. Specifically, the manual testing process begins by analyzing the function of an interactive component. Then, a UI input is performed on this component, followed by checking whether the UI response aligns with the prediction.

Therefore, UI functional testing is divided into two parts for KuiTest: *UI Component Function Identification* and *Post-Interaction Response Verification*. Specifically, the workflow of KuiTest starts by predicting the function of interactive UI components in the *UI Component Function Identification* part. Then, followed by the *Post-Interaction Response Verification*, in which we let LLMs evaluate whether the post-interaction response aligns with the prediction, using in-the-wild knowledge to detect potential UI functional bugs.

C. UI Component Function Identification

Before identifying the functions of UI components, KuiTest first detects all interactive elements within the UI pages under testing. As shown in Figure 3, KuiTest achieves this by utilizing both the screenshot and UI hierarchy (*e.g.*, XML [29] on the Android platform). Specifically, the UI hierarchy is retrieved using the Android Debug Bridge (adb) tool [30], which is then parsed to identify all interactive components (*i.e.*, those with a *true* value for 'clickable' or 'long-clickable' attributes). Additionally, KuiTest employs a deep-learning model, vision-ui [31], to directly identify interactive elements from the UI screenshot as a supplementary method. This method is particularly useful for recognizing components that may be missing from the UI hierarchy files, such as WebView [32] components in Android apps. Once these interactive components are identified, we adopt the widely-used Set-of-Mark (SoM) [33], outlining these components with bounding boxes and assigning each a unique ID on the screenshot.

After the detection of interactive UI components, KuiTest then conducts functional identification with the help of MLLMs. Although UI components are typically regarded as intuitive and easy for humans to understand, both our experiments and related works [14], [16] show that the visual comprehension ability of MLLMs is inadequate for UI function prediction. Therefore, to assist MLLMs in identifying UI components' functions, KuiTest employs a multimodal input strategy that integrates both the image (*i.e.*, UI screenshot) and textual information. Specifically, after recognizing and annotating interactive components, KuiTest extracts the associated text for each component through optical character recognition (OCR). This extracted text is then linked to its corresponding component in a query prompt, and both the prompt and annotated screenshot are provided as inputs to the MLLM. By leveraging this multimodal approach, the MLLM can predict component functions from both visual and textual perspectives.

However, not all UI components contain textual information. Many frequently used icons lack accompanying text (*e.g.*, components 8 to 12 in Figure 3), making our multimodal input strategy less effective when dealing with these icons. To overcome this limitation, we implemented an icon function library to enhance the MLLMs' ability to understand these text-less components. Specifically, we collected commonly used text-less components from apps in Meituan to form the library. Each component is manually labeled with its name and function, and this information is stored alongside the component's image embeddings using CLIP (Contrastive Language-Image Pre-training) [34].

When encountering a text-less UI component, KuiTest will search for the most similar component in the library

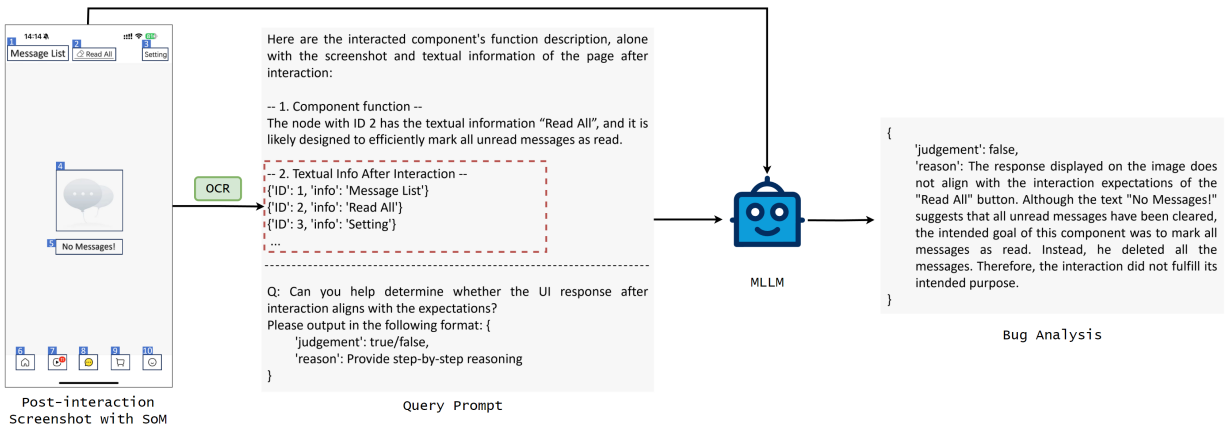


Fig. 4: Workflow of Post-Interaction Response Verification Module in KuiTest

before function recognition. Cosine similarity is used for calculating the similarity between the target component's image embeddings, denoted as V_{UI}, and the embeddings of the components in the library, denoted as V_{Lib_i} for $i = 1, 2, \ldots, n$.

$$\text{cos\_similarity}(V_{UI}, V_{Lib_i}) = \frac{V_{UI} \cdot V_{Lib_i}}{\|V_{UI}\| \|V_{Lib_i}\|} \quad (1)$$

$$\text{max\_similarity} = \max_{i \in [1,n]} \{\text{cos\_similarity}(V_{UI}, V_{Lib_i})\} \quad (2)$$

If the maximum similarity score exceeds a predefined threshold δ, KuiTest considers the component to be matched and associates its function with the text-less component in the query prompt (*e.g.*, component 8 in Figure 3).

As shown in Figure 3, after constructing the screenshot with outlined interactive UI components and icon library aided textual information, KuiTest prompts the MLLM and queries the function of an interactive UI component based on its ID.

D. Post-Interaction Response Verification

Once the function of the interactive components has been identified, the next step is to perform the interaction. Specifically, KuiTest chooses different tools to simulate clicks depending on the platform. For instance, it utilizes the adb tool [30] on Android and HarmonyOS Device Connector (hdc) [35] for HarmonyOS Next. After the interaction is completed, KuiTest takes a screenshot of the post-interaction GUI, which serves as the input for the subsequent response verification process.

After the interaction, KuiTest first identifies whether there are any GUI responses, *i.e.*,, visual differences between the pre and post-interaction states. For instance, KuiTest compares the pixel-based image similarity between the pre and post-interaction screenshots. If no visible changes are detected, KuiTest marks the interaction as *no response* and flags it as an anomaly. For screenshots that show a response, KuiTest hands over the task of functional anomalies detection to MLLMs.

Specifically, as shown in Figure 4, KuiTest utilizes similar image-text information demonstrated in Section III-C to assist the MLLM in understanding the post-interaction UI semantics. Furthermore, the previously predicted component function is also incorporated as input. Then KuiTest prompts MLLMs to detect conflicts between the component's intended function and its actual response. In designing the prompt, we deliberately avoided including examples (well-known as few-shot) to ensure that the judgments of MLLMs remained uninfluenced. This approach helps preserve their generalizability in addressing the diversity of UI functional bugs. The final prompt for anomaly detection can be referred to Figure 4.

E. Configuration for Deployment

When deployed, KuiTest provides two testing strategies for engineers' choices: *random exploration* and *UI component traversal*. Specifically, in the *random exploration* strategy, engineers need to provide an AUT, a start page (default is the app's homepage), and a maximum number of steps (default is 500). KuiTest will then conduct UI functional testing from the start page by repeatedly selecting an interactive UI component randomly and verifying its response. After each round of interaction, KuiTest does not reset the page to the start state but continues with another interaction on the current page until the specified number of steps is reached.

As for the *UI component traversal* strategy, engineers need to provide an AUT and a set of GUI pages as KuiTest's input. For each UI page, KuiTest will first recognize all of the interactive UI components and then sequentially perform interactions and verify the responses. Different from the *random exploration*, after each interaction in *UI component traversal*, KuiTest will shut down and reopen the AUT, then navigate back to the target GUI page before the next interaction.

IV. Evaluation

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

- **RQ1:** What is the benefit of the task decomposition structure in KuiTest?

- **RQ2:** How does the effectiveness of KuiTest's design in identifying functions of GUI component?
- **RQ3:** To what extent can KuiTest detect UI functional bugs in real-world, practical commercial mobile apps?

RQ1 evaluates the effectiveness and efficiency of KuiTest's task decomposition by comparing its effectiveness and efficiency in detecting UI functional bugs against alternative decomposition variants. RQ2 assesses KuiTest's capability in GUI understanding by comparing its performance in GUI component function identification against three variant prompts via an ablation experiment. Finally, the practical usability of KuiTest is validated in RQ3 by recognizing manually injected real-life historical UI functional bugs across different scenarios. Moreover, in Section V, we will present the practical effectiveness of KuiTest by applying it to real testing circumstances involving thousands of UI pages in Meituan.

A. Evaluation Setup

1) Benchmark Construction: Although KuiTest provides two testing strategies (as discussed in Section III-E) for engineers' choices, the random exploration is not conducive to statistical analysis. Therefore, we opted for the GUI component traversal strategies in the quantitative experiments.

To the best of our knowledge, there is currently no publicly available benchmark specifically for UI functional bug testing. Therefore, we construct two benchmarks ourselves from Meituan's *M* app, which serves nearly 700 million customers and 10 million merchants.

Specifically, we collected historical UI functional bugs (as exampled in Figure 1) identified through manual testing at Meituan over the past six months. For **RQ1**, we randomly selected 25 UI inputs that historically triggered UI functional bugs from five different business lines in Meituan's app, *i.e.*, Travel, Hotel, Video, Comment, and Coupon. Then, 25 correctly functioning UI inputs are selected from each business line. Therefore, the benchmark for RQ1 contains 150 UI inputs with a bug ratio of 16.7% (25/150).

In **RQ2**, 250 interactive UI components are randomly collected from the five business lines to evaluate the performance of UI component function identification. To ensure diversity, we included both text components and icon components from each business line.

To thoroughly test the capabilities of KuiTest, we construct the benchmark for **RQ3** from ten different business lines (*i.e.*, Message, Movie, Takeout, Dining, Map, Travel, Leisure, Riding, Grocery, and Medicine) in Meituan. For each business line, 15 UI inputs that historically triggered UI functional bugs are randomly selected and injected into 10 correctly functioning GUI pages. In summary, the benchmark for RQ3 includes a list of 100 GUI pages under testing from Meituan, with a total of 4,664 interactive UI components, of which 150 will trigger a buggy response.

2) Baselines: Since **RQ1** evaluates the performance of KuiTest's task decomposition, we construct two different variants, *No Decomposition* and *Three-Step* strategies, to compare their performance against the *Two-step* strategy utilized by KuiTest. Specifically, as shown in Figure 5, one variant involves feeding the pre and post-interaction GUI screenshots and images of interactive components to the MLLM in a single query, and therefore referred to as *No Decomposition*. It prompts the MLLM to answer whether the UI response exhibits any anomalies directly. In contrast, the other variant divides the UI functional testing into three sub-tasks: first, predicting the functionality of the interacted component based on the pre-interaction screenshot; second, analyzing the semantic content of the page using the post-interaction screenshot; and finally, determining whether there are discrepancies between the component functionality and the page semantics.

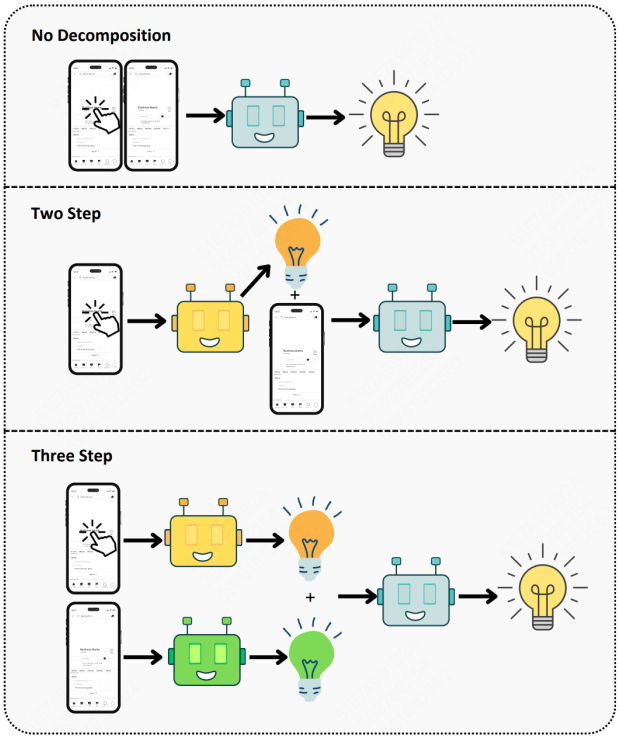


Fig. 5: Different variants of task decomposition.

In **RQ2**, we evaluate the design of GUI component function identification in KuiTest by comparing it with three variants. Specifically, we iteratively remove components from KuiTest to validate the effectiveness of each part described in Section III-C. First, we removed the icon library to create the first variant. Next, we eliminated the SoM strategy for the second variant. Then, by changing the JSON format of the text obtained through OCR to a single string, we shifted the component-level image-text connection in KuiTest to a page-level connection. Finally, we removed text information for the last variant, where the prompt for identifying GUI component functions only includes the screenshot and the position of the target component, which is referred to as *Pure MLLM*.

In **RQ3**, we assess KuiTest's ability to identify functional bugs in real-life scenarios by demonstrating its performance across different business lines within Meituan.

TABLE I: Ablation results for different decomposition strategies.

Decomposition Strategy	Travel		Hotel		Video		Comment		Coupon		Average				
	Pre.	Rec.	Pre.	Rec.	Pre.	Rec.	Pre.	Rec.	Pre.	Rec.	Acc.	Pre.	Rec.	Tok_p	Tok_c
No Decomposition	0.50	0.40	0.60	0.60	1.0	0.60	0.67	0.40	0.50	0.40	0.83	0.65	0.48	**3420**	**223**
Two-Step	**0.75**	**0.60**	**0.83**	**1.0**	**1.0**	0.80	0.75	**0.60**	**1.0**	**0.60**	**0.93**	**0.86**	**0.72**	5326	316
Three-Step	0.67	**0.60**	**0.83**	**1.0**	**1.0**	**1.0**	**1.0**	**0.60**	0.67	0.40	0.90	0.83	0.60	7012	773

[1] Acc. Pre. and Rec. represent the accuracy, precision, and recall.
[2] Tok_c and Tok_p represent the completion tokens and prompt tokens consumption in processing an interaction task.

TABLE II: Accuracy and efficiency of UI component function identification

Identify Strategy	Travel / Acc.		Hotel / Acc.		Video / Acc.		Comment / Acc.		Coupon / Acc.		Average / Acc.		Tokens	
	text	icon	text	icon	text	icon	text	icon	text	icon	text	icon	Tok_p	Tok_c
Pure MLLM	0.72	0.00	0.89	0.25	**1.00**	0.17	0.92	0.25	0.98	0.00	0.90	0.13	**1655**	120
Img+Text	0.81	0.14	0.91	0.25	**1.00**	0.17	0.92	0.25	0.98	0.00	0.92	0.16	2315	134
SoM+Text	0.89	0.14	0.95	0.25	**1.00**	0.33	**0.97**	0.25	**1.00**	0.00	0.96	0.19	2943	**97**
KuiTest	**0.91**	**1.00**	**0.96**	**1.00**	**1.00**	**1.00**	0.94	**0.75**	**1.00**	**1.00**	**0.96**	**0.95**	3136	**97**

[1] Tok_c and Tok_p represent the completion tokens and prompt tokens consumption in processing an interaction task.
[2] text and icon represent interactive UI components with and without corresponding text, respectively.

3) Metrics: As for **RQ1**, since the UI functional testing can be formulated as a binary classification problem (*i.e.*, whether anomalies are present), we choose the widely used *accuracy*, *precision*, and *recall* to compare the effectiveness of KuiTest with the two variants. Additionally, we track *token consumption* to assess their efficiency.

RQ2 evaluates whether the function identification performed by the LLMs aligns with the component's intended design. To facilitate analysis, we simplify this textual quality evaluation task into a binary classification problem and use *accuracy* as the metric to compare the performance of KuiTest with the baselines. Specifically, two authors of this paper manually reviewed the results, and in cases when their assessments differed, another author was consulted to arbitrate. Furthermore, we also recorded the prompt and completion *token consumption* in RQ2's experiments to highlight the differences in efficiency and cost between KuiTest and the baselines.

For **RQ3**, although the methods and testing data differ from RQ1, the task under investigation remains the same (*i.e.*, identifying functional bugs from GUI interactions). Therefore, we reused the *precision* and *recall* from RQ1 and added *FPr* to evaluate KuiTest's capabilities of reducing human effort in UI functional testing.

4) Implementation and Configuration: Since it is impossible to construct an icon dataset containing all possible icons of a practical commercial app due to its vast number of pages, we randomly selected 10 pages from our benchmark to create the icon library for RQ2 and RQ3, simulating KuiTest's deployment scenario. Specifically, we extracted and labeled 18 types of UI components and stored their image samples and function descriptions in the library.

All experiments were run on a MacBook Pro with a 2.6 GHz 6-core Intel Core i7 processor, while screenshots and hierarchy data were captured from a Xiaomi 14 device. Additionally, for cost efficiency, we chose to use the more affordable Claude-3-haiku [36] as our base model for KuiTest's evaluation, which is different from KuiTest's deployments in Meituan in Section V.

B. RQ1: Benefits of Testing Task Decomposition

In RQ1, we compare the effectiveness and efficiency of different task decomposition variants (shown in Figure 5) by evaluating their performance in detecting injected UI functional bugs.

The result for RQ1 is shown in Table I. The bug detection performance (*i.e.*, accuracy, precision, and recall) of two-step and three-step decomposition variants is significantly higher than that of a single query. This demonstrates the effectiveness of the widely used task decomposition strategy in enhancing the reliability of LLM outputs when handling complex problems.

As for the differences between the two-step and three-step decomposition variants, although the tasks for MLLM in the three-step variant are less complex, its overall performance is poorer than that of the two-step variant. The *post-interaction semantic extraction* procedure makes the three-step variant advance in comprehending page type and, therefore, good at handling intra-page navigation interaction, particularly in the *Video* and *Comment* scenarios. However, in the meantime, it is prone to omit subtle UI features (*e.g.*, icon color), leading to missing inter-page responses in the UI semantics. As a result, it performs poorly in identifying UI functional bugs in scenarios with frequent inter-page interactions, such as *Travel* and *Coupon*.

Therefore, KuiTest's choice of leveraging the two-step decomposition strategy to handle UI functional testing by the *UI component function identification* and *post-interaction response verification* module is effective and efficient.

TABLE III: KuiTest's performance in detecting UI functional bugs in different business lines.

Business	Message	Movie	Takeout	Dining	Map	Travel	Leisure	Riding	Grocery	Medicine	**Aggregate**
#Inputs	479	399	525	589	417	501	532	323	469	430	4664
#Bugs	15	15	15	15	15	15	15	15	15	15	150
#TP	13	12	13	11	15	12	11	14	15	13	129
#FP	8	0	3	9	3	8	7	3	3	7	52
Precision	0.62	**1.00**	0.81	0.55	0.83	0.60	0.61	0.82	0.83	0.65	0.71
Recall	0.87	0.80	0.87	0.73	**1.00**	0.80	0.73	0.93	1.00	0.87	0.86
FPr	0.02	**0.00**	0.01	0.02	0.01	0.02	0.01	0.01	0.01	0.02	0.01

[1] #Inputs, #Bugs, #TP, and #FP represent the number of UI inputs, injected bugs, recalled bugs, and false positives, respectively.

C. RQ2: Performance of UI Component Function Identification

In RQ2, we prepared the corresponding input for KuiTest and the three variants and manually counted the accuracy of their function predictions on UI components. Besides, icon components (*i.e.*, those without corresponding text) are split with the other components (referred to as text components) to evaluate the effects of the icon library.

As shown in Table II, the accuracy of the variant with the input of a screenshot and the position of the target component (named Pure MLLM) rank lowest among the four methods. This supports the statement made in Section I, which highlights the limitations of LLMs' ability to understand images. Additionally, with the addition of text and SoM as input, the accuracy of function recognition for text components improved, reaching 96%, which demonstrates its suitability for industrial use.

Moreover, none of the three baselines is capable of accurately identifying the function of UI icon components (*i.e.*, with an accuracy below 20%), whereas KuiTest achieves 95% accuracy, demonstrating the effectiveness of its icon library.

D. RQ3: Effectiveness of Response Verification

In RQ3, we evaluate KuiTest's performance in detecting these 50 UI functional bugs from the 150 UI pages in our benchmark. To simulate a real-life practical scenario, we did not restrict the range of UI inputs for KuiTest. Therefore, KuiTest has to independently traverse all interactive UI components across the 150 pages to detect UI functional bugs.

The Results for RQ3 are shown in Table III. KuiTest can identify 86% injected UI functional bugs from the ten different business lines. Although KuiTest formulates the UI response verification as a binary classification problem, the positive cases (*i.e.*, UI inputs leading to functional bugs) account for only 3.2% in our benchmark, resulting in a precision of 71% for KuiTest.

Given that the percentage of positive cases in real-life industrial testing scenarios is even lower than 3.2%, keeping a low rate of false positives (*i.e.*, incorrectly identifying a correct UI response as a UI functional bug) is crucial for reducing manual effort. Therefore, the FPr of 1.2% for KuiTest makes it possible for industrial deployment, which we will further discuss in the next section.

V. Case Studies

Since the design of KuiTest was motivated by practical industrial labor-intensive status in Meituan, it has now been integrated into an automated testing platform in Meituan for engineers' easy access. When invoking KuiTest, engineers are required to provide either a starting GUI page or a list of GUI pages from the application under test (AUT). With this information, KuiTest can autonomously conduct GUI semantic testing through random exploration or component traversal of the specified GUI list.

In this section, we show the real-life benefits by demonstrating the procedure of applying KuiTest to mobile apps in Meituan. We summarize the lessons learned, which can shed light on the automation of GUI semantic testing.

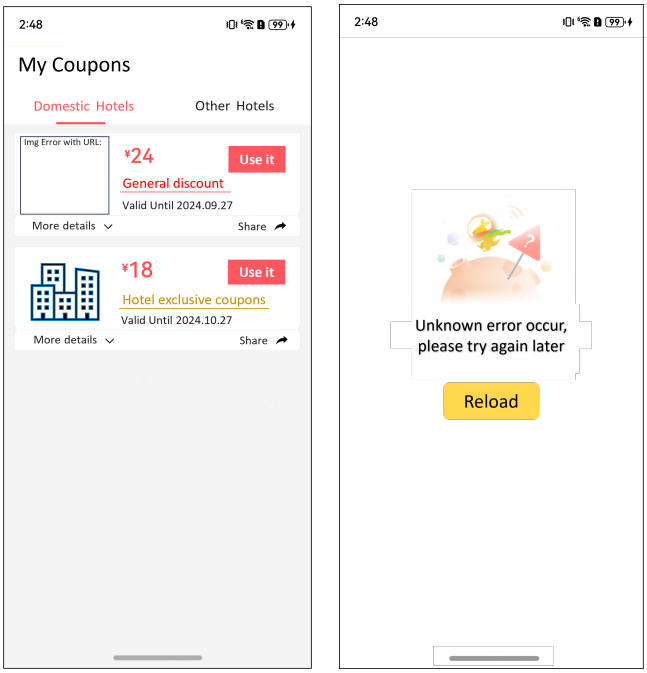
A. GUI Traverse on HarmonyOS NEXT

Nowadays, popular commercial apps typically support multiple platforms (*e.g.*, Android, iOS). Due to the longstanding challenges in GUI image analysis, GUI testing methods in industrial practice heavily rely on analyzing the hierarchy files. However, different platforms have varying support for hierarchy structures, making cross-platform testing particularly challenging.

This is also the case for the *M* app from Meituan, whose automated GUI testing primarily relies on scripts tied to platform-specific hierarchy files. Therefore, engineers in Meituan need to maintain numerous GUI testing scripts for each platform to identify app anomalies before users encounter them.

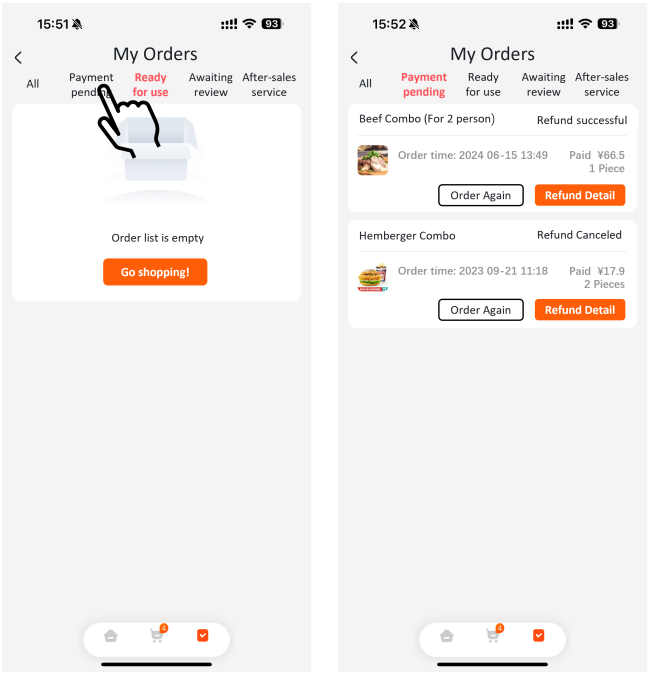
Such platform-specific GUI testing scripts will face immense pressure when a new platform (*e.g.*, *HarmonyOS NEXT*) is set to launch. Interactions and verification previously tied to apps on Android or iOS through GUI elements' IDs or attributes cannot be automatically migrated to the new hierarchy structure. Despite significant manual effort invested in developing scripts and performing manual tests, many GUI pages remain uncovered.

Therefore, as soon as KuiTest was released at Meituan, it was first used to test the *M* app's functionality on the preview version of *HarmonyOS NEXT*. Specifically, engineers selected the random exploration strategy and only needed to set a starting GUI page for the beginning. From there, KuiTest autonomously conducted GUI interactions and performed



(a) Image load failure (b) GUI page load failure

Fig. 6: Bugs of M app on Harmony OS NEXT



(a) Click for "Payment Pending" (b) Display "Refund Orders"

Fig. 7: Interaction bug found via regressions

semantic testing. Finally, the results containing GUI interaction traces and `KuiTest`'s judgments are shown to engineers.

In our testing of 31,176 UI interactions, `KuiTest` identified 32 previously unknown bugs in the *M* app, of which 31 have been confirmed. Since the *M* app for *HarmonyOS NEXT* includes many newly developed GUI pages, 28 of the 31 bugs were related to the UI loading process, such as images failing to load and GUI page load failures sampled in Figure 6.

B. Weekly Semantic Regression

As demonstrated in Section II-B, The development and maintenance of GUI testing scripts represent a significant labor burden, meaning that even on well-established platforms, page coverage of scripts remains limited, typically focusing on core business areas (*e.g.*, transaction-related pages) due to the cost-benefit trade-off.

While bugs in other areas can still negatively impact the user experience, `KuiTest` now offers a low-cost solution to enhance the coverage of GUI semantic testing. Specifically, engineers select the GUI component traversal strategy in `KuiTest` and provide a list of GUI pages to be tested. During each round of GUI regression, `KuiTest` automatically interacts with every interactive component on each page in the list, followed by semantic verification of the response.

During the six months since `KuiTest`'s release, it has been involved in 24 rounds of GUI regression testing, uncovering 12 GUI bugs from 100k cases, of which 10 have been confirmed. These include 4 display bugs (*e.g.*, blank screen issue) and 6 semantic bugs. We sampled representative semantic bugs in Figure 7.

Due to the current limitations of LLMs in understanding GUI semantics, we acknowledge that `KuiTest` cannot detect all semantic bugs resulting from GUI interactions. Testing engineers at Meituan have assessed `KuiTest`'s contribution to automated testing by evaluating the proportion of GUI scripts and manual testing efforts it could replace. Currently, this ratio is estimated to be around 15%. We will discuss this further in the next section.

VI. Threats to Validity

One potential threat arises from the fact that `KuiTest` employs the LLM-simulated human expectations as an oracle for UI functional testing. Therefore, we acknowledge that `KuiTest` 's ability to detect UI functional bugs is related to the performance of the base model. Given our focus on industrial usability when designing `KuiTest`, we currently set its testing target to one-step interactions rather than interaction sequences to keep the reliability of `KuiTest`'s judgment. In the future, we will work on designing a rule-free UI functional testing approach for sequences of interactions.

Another threat lies in `KuiTest`'s generalizability to other apps. Although `KuiTest` is designed and implemented in Meituan, it is not specifically tailored for Meituan's app. Since neither the *UI Component Function Identification* nor the *Post-Interaction Response Verification* module in `KuiTest` requires proprietary technology from Meituan, we believe

that KuiTest's workflow and performance can be readily replicated on other apps.

VII. Related Work

As mobile apps become increasingly complex, automatic GUI testing has long been studied by both academia [4], [23], [37]–[43] and industry [6], [44]–[49]. Depending on the target type of UI bugs being tested, these methods can be categorized into two categories: non-functional and functional bug testing tools.

Tools targeting UI crashes are widely-studied non-functional bug UI testing tools [50]. These tools typically employ a model-based [4], [7], heuristic [40], [51], [52] or even LLM-based [53] strategy to generate streams of UI inputs and detect crashes in the AUT. Since they do not rely on the semantics or functions of the GUI when performing UI testing, they can execute tests on almost any app with minimal adaptation. Although these tools can automatically explore an AUT and, therefore, offer a high degree of generalization, their testing scope is limited as they rely on a simple testing oracle focused solely on detecting crashes.

As for UI functional bug testing, engineers traditionally relied on crafting hard-coded UI test scripts [12]. Although this approach demonstrates strong testing capabilities in specific areas, it is quite labor-intensive to develop and maintain a large set of UI testing scripts. As a result, script-based UI testing often has low coverage in industrial practices. Therefore, some researchers have utilized automated methods to aid in generating test scripts, such as AppFlow [11], CraftDroid [54], and CoSer [12]. Others have adopted human-like testing strategies through learning-based automation methods such as Humanoid [55] and QTypist [53]. However, these methods often exhibit poor generalizability, making them challenging to be applied directly in novel scenarios.

Moreover, the advancement of LLMs empowers machines with extensive knowledge bases and strong comprehension abilities, which leads to the widespread use of LLMs in UI functional testing. Unlike previous methods, LLMs can effectively understand the semantic information of GUI pages and natural language instructions, enabling more automated UI functional testing tools. Specifically, VisionDroid [56] employs MLLMs to integrate GUI images and textual information, determining whether the AUT exhibits any of 9 categories of non-crash UI functional bugs. AUITestAgent [57] leverages LLMs' natural language understanding to replace traditional hard-coded scripts with natural language-based test requirements to improve generalizability. Although these methods can reduce the human effort required for UI functional testing, they still rely on manual rule definition and adaptation, which limits their usability in industrial practice.

Our work, KuiTest, although also an LLM-based functional testing tool, eliminates the need for predefined rule-based oracles. Instead, it simulates common user expectations, enabling adaptation-free UI functional testing in novel industrial scenarios.

VIII. Conclusion

This paper highlights the manual-intensive status of UI functional testing caused by the poor generalization of the widely-used rule-based oracles. Then, we propose KuiTest, the first rule-free UI functional testing tool we designed for Meituan to reduce such manual effort. KuiTest adopts the idea of leveraging general LLMs to simulate common user expectations and, therefore, replaces the widely-used rule oracles for UI functional testing with knowledge in the wild. Our experiments on customized benchmarks show that KuiTest can effectively identify the function of interactive UI components and could recall 86% injected UI functional bugs while maintaining a 1.2% FPr. Furthermore, practical industrial case studies show KuiTest's real-life usability in successfully detecting previously unknown UI functional bugs and capabilities of reducing human effort in UI testing.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (Project No. 62472101), the Natural Science Foundation of Shanghai (Project No. 22ZR1407900), and Meituan. In addition, we extend our heartfelt thanks to our colleagues in Meituan, specifcally, Ying, Hui, Haibo, Chen, Xianxuan, Youwei, Jiangshan, and Junlin, for their kind help and support in this work. Y. Zhou is the corresponding author.

References

[1] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, "An empirical study of android test generation tools in industrial cases," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, M. Huchard, C. Kästner, and G. Fraser, Eds. ACM, 2018, pp. 738–748.

[2] M. Jovic, A. Adamoli, D. Zaparanuks, and M. Hauswirth, "Automating performance testing of interactive java applications," in *The 5th Workshop on Automation of Software Test, AST 2010, May 3-4, 2010, Cape Town, South Africa*, H. Zhu, W. K. Chan, C. J. Budnik, and G. M. Kapfhammer, Eds. ACM, 2010, pp. 8–15.

[3] H. He, Z. Jia, S. Li, E. Xu, T. Yu, Y. Yu, J. Wang, and X. Liao, "Cp-detector: Using configuration-related performance properties to expose performance bugs," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 623–634.

[4] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "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*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 245–256.

[5] Z. Qin, Y. Tang, E. Novak, and Q. Li, "Mobiplay: a remote execution based record-and-replay tool for mobile applications," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 571–582.

[6] (Accessed: 2024) Ui/application exerciser monkey. [Online]. Available: https://developer.android.com/studio/test/other-testing-tools/monkey

[7] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical GUI testing of android applications via model abstraction and refinement," in *Proc. of the 41st International Conference on Software Engineering, ICSE*. IEEE / ACM, 2019, pp. 269–280.

[8] Y. Xiong, M. Xu, T. Su, J. Sun, J. Wang, H. Wen, G. Pu, J. He, and Z. Su, "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*, R. Just and G. Fraser, Eds. ACM, 2023, pp. 1319–1331.

[9] Y. Huang, J. Wang, Z. Liu, S. Wang, C. Chen, M. Li, and Q. Wang, "Context-aware bug reproduction for mobile apps," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2336–2348.

[10] S. Kang, J. Yoon, and S. Yoo, "Large language models are few-shot testers: Exploring llm-based general bug reproduction," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2312–2323.

[11] G. Hu, L. Zhu, and J. Yang, "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, 2018, pp. 269–282.

[12] S. Cao, M. Pan, Y. Pei, W. Yang, T. Zhang, L. Wang, and X. Li, "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, 2024, pp. 90:1–90:13.

[13] Z. Liu, C. Li, C. Chen, J. Wang, B. Wu, Y. Wang, J. Hu, and Q. Wang, "Vision-driven automated mobile gui testing via multimodal large language model," *CoRR*, vol. abs/2407.03037, 2024.

[14] Y. Hu, X. Wang, Y. Wang, Y. Zhang, S. Guo, C. Chen, X. Wang, and Y. Zhou, "Auitestagent: Automatic requirements oriented gui function testing," 2024. [Online]. Available: https://arxiv.org/abs/2407.09018

[15] (Accessed: 2024) Introducing chatgpt. [Online]. Available: https://openai.com/blog/chatgpt

[16] K. You, H. Zhang, E. Schoop, F. Weers, A. Swearngin, J. Nichols, Y. Yang, and Z. Gan, "Ferret-ui: Grounded mobile UI understanding with multimodal llms," *CoRR*, vol. abs/2404.05719, 2024.

[17] Q. Jiao, D. Chen, Y. Huang, Y. Li, and Y. Shen, "Enhancing multimodal large language models with vision detection models: An empirical study," *CoRR*, vol. abs/2401.17981, 2024.

[18] J. Zhang, J. Hu, M. Khayatkhoei, F. Ilievski, and M. Sun, "Exploring perceptual limitation of multimodal large language models," *CoRR*, vol. abs/2402.07384, 2024.

[19] Z. Li, D. Liu, C. Zhang, H. Wang, T. Xue, and W. Cai, "Enhancing advanced visual reasoning ability of large language models," *CoRR*, vol. abs/2409.13980, 2024.

[20] (Accessed: 2024) Xml path language (xpath). [Online]. Available: https://www.w3.org/TR/1999/REC-xpath-19991116/

[21] (Accessed: 2024) Harmonyos next beta1. [Online]. Available: https://developer.huawei.com/consumer/cn/doc/harmonyos-releases-V5/releasenotes-beta1-V5

[22] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, X. Che, D. Wang, and Q. Wang, "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, 2024, pp. 100:1–100:13.

[23] T. Su, Y. Yan, J. Wang, J. Sun, Y. Xiong, G. Pu, K. Wang, and Z. Su, "Fully automated functional fuzzing of android apps for detecting non-crashing logic bugs," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–31, 2021.

[24] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, Z. Tian, Y. Huang, J. Hu, and Q. Wang, "Testing the limits: Unusual text inputs generation for mobile app crash detection with large language model," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 137:1–137:12.

[25] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," *ACM Comput. Surv.*, vol. 55, no. 12, pp. 248:1–248:38, 2023.

[26] W. Huang, H. Liu, M. Guo, and N. Z. Gong, "Visual hallucinations of multi-modal large language models," *CoRR*, vol. abs/2402.14683, 2024.

[27] Q. Wu, G. Bansal, J. Zhang, Y. Wu, S. Zhang, E. Zhu, B. Li, L. Jiang, X. Zhang, and C. Wang, "Autogen: Enabling next-gen LLM applications via multi-agent conversation framework," *CoRR*, vol. abs/2308.08155, 2023.

[28] C. Chan, W. Chen, Y. Su, J. Yu, W. Xue, S. Zhang, J. Fu, and Z. Liu, "Chateval: Towards better llm-based evaluators through multi-agent debate," in *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[29] (Accessed: 2024) Layouts in views. [Online]. Available: https://developer.android.com/develop/ui/views/layout/declaring-layout

[30] (Accessed: 2024) Android debug bridge (adb). [Online]. Available: https://developer.android.com/tools/adb

[31] (Accessed: 2024) Meituan-dianping/vision-ui: Visual ui analysis tools. [Online]. Available: https://github.com/Meituan-Dianping/vision-ui

[32] (Accessed: 2024) Webview | andriod developers. [Online]. Available: https://developer.android.com/reference/android/webkit/WebView

[33] J. Yang, H. Zhang, F. Li, X. Zou, C. Li, and J. Gao, "Set-of-mark prompting unleashes extraordinary visual grounding in GPT-4V," *CoRR*, vol. abs/2310.11441, 2023.

[34] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever, "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*, M. Meila and T. Zhang, Eds. PMLR, 2021, pp. 8748–8763.

[35] (Accessed: 2024) hdc usage guidelines. [Online]. Available: https://device.harmonyos.com/en/docs/apiref/doc-guides/ide-command-line-hdc-0000001078799258

[36] (Accessed: 2024) Claude 3 haiku: our fastest model yet. [Online]. Available: https://www.anthropic.com/news/claude-3-haiku

[37] W. Wang, W. Lam, and T. Xie, "An infrastructure approach to improving effectiveness of android UI testing tools," in *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, C. Cadar and X. Zhang, Eds. ACM, 2021, pp. 165–176.

[38] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE Computer Society, 2017, pp. 23–26. [Online]. Available: https://doi.org/10.1109/ICSE-C.2017.8

[39] F. Y. B. Daragh and S. Malek, "Deep GUI: black-box GUI input generation with deep learning," in *Proc. of the 36th IEEE/ACM International Conference on Automated Software Engineering, ASE*. IEEE, 2021, pp. 905–916.

[40] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *Proc. of the 42nd International Conference on Software Engineering, ICSE*. ACM, 2020, pp. 481–492.

[41] W. Guo, L. Shen, T. Su, X. Peng, and W. Xie, "Improving automated GUI exploration of android apps via static dependency analysis," in *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 2020, pp. 557–568.

[42] C. Li, Y. Jiang, and C. Xu, "Cross-device record and replay for android apps," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, A. Roychoudhury, C. Cadar, and M. Kim, Eds. ACM, 2022, pp. 395–407.

[43] T. Su, J. Wang, and Z. Su, "Benchmarking automated GUI testing for android against real-world bugs," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 119–130.

[44] D. Ran, Z. Li, C. Liu, W. Wang, W. Meng, X. Wu, H. Jin, J. Cui, X. Tang, and T. Xie, "Automated visual testing for mobile apps in an industrial setting," in *Proc. of the 44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP)*. IEEE, 2022, pp. 55–64.

[45] Y. Hu, J. Gu, S. Hu, Y. Zhang, W. Tian, S. Guo, C. Chen, and Y. Zhou, "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*, S. Chandra, K. Blincoe, and P. Tonella, Eds. ACM, 2023, pp. 1786–1797.

[46] Y. Hu, H. Jin, X. Wang, J. Gu, S. Guo, C. Chen, X. Wang, and Y. Zhou, "Autoconsis: Automatic gui-driven data inconsistency detection of mobile apps," in *Proceedings of the 46th International Conference on Software*

Engineering: Software Engineering in Practice, ICSE-SEIP 2024, Lisbon, Portugal, April 14-20, 2024. ACM, 2024, pp. 137–146.

[47] W. Lam, Z. Wu, D. Li, W. Wang, H. Zheng, H. Luo, P. Yan, Y. Deng, and T. Xie, "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*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 854–859.

[48] H. Zheng, D. Li, B. Liang, X. Zeng, W. Zheng, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated test input generation for android: Towards getting there in an industrial case," in *Proc. of the 39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP*. IEEE Computer Society, 2017, pp. 253–262.

[49] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated test input generation for android: are we really there yet in an industrial case?" in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, T. Zimmermann, J. Cleland-Huang, and Z. Su, Eds. ACM, 2016, pp. 987–992.

[50] T. Su, L. Fan, S. Chen, Y. Liu, L. Xu, G. Pu, and Z. Su, "Why my app crashes? understanding and benchmarking framework-specific exceptions of android apps," *IEEE Trans. Software Eng.*, vol. 48, no. 4, pp. 1115–1137, 2022.

[51] K. Mao, M. Harman, and Y. Jia, "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*, A. Zeller and A. Roychoudhury, Eds. ACM, 2016, pp. 94–105.

[52] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: segmented evolutionary testing of android apps," in *Proc. of the 22nd ACM SIGSOFT Symposium on Foundations of Software Engineering, FSE*. ACM, 2014, pp. 599–609.

[53] Z. Liu, C. Chen, J. Wang, X. Che, Y. Huang, J. Hu, and Q. Wang, "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, 2023, pp. 1355–1367.

[54] J. Lin, R. Jabbarvand, and S. Malek, "Test transfer across mobile apps through semantic mapping," in *Proc. of the 34th IEEE/ACM International Conference on Automated Software Engineering, ASE*. IEEE, 2019, pp. 42–53.

[55] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A deep learning-based approach to automated black-box android app testing," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 1070–1073. [Online]. Available: https://doi.org/10.1109/ASE.2019.00104

[56] Z. Liu, C. Li, C. Chen, J. Wang, B. Wu, Y. Wang, J. Hu, and Q. Wang, "Vision-driven automated mobile GUI testing via multimodal large language model," *CoRR*, vol. abs/2407.03037, 2024.

[57] Y. Hu, X. Wang, Y. Wang, Y. Zhang, S. Guo, C. Chen, X. Wang, and Y. Zhou, "Auitestagent: Automatic requirements oriented GUI function testing," *CoRR*, vol. abs/2407.09018, 2024.