From Redundancy to Efficiency: Exploiting Shared UI Interactions towards Efficient LLM-Based Testing

Xuan Wang*, Yingchuan Wang*, Yongxiang Hu*, Yu Zhang†,
Hailiang Jin†, Shiyu Guo†, Juxing Yuan‡, Yangfan Zhou*§
*College of Computer Science and Artificial Intelligence, Fudan University, Shanghai, China
§Shanghai Key Laboratory of Intelligent Information Processing, Fudan University, Shanghai, China
†Meituan, Shanghai, China
‡Meituan, Beijing, China
xuanwang25@m.fudan.edu.cn, yingchuanwang23@m.fudan.edu.cn, yongxianghu23@m.fudan.edu.cn,
zhangyu233@meituan.com, jinhailiang@meituan.com, guoshiyu03@meituan.com, yuanjuxing@meituan.com,
zyf@fudan.edu.cn

***Abstract*—Redundant test cases, although well-studied in software engineering, are previously underexplored in UI testing of mobile apps. Our study of real-world test suites shows that, equipped with large-scale testing suites, redundancy in UI testing often manifests as redundant UI interactions. Although negligible in traditional script-based workflows, such redundancy severely impacts the efficiency of emerging Large Language Model (LLM)-based UI agents, which incur substantial decision latency and token costs from repeated LLM queries for the same interactions. To this end, based on the idea of reusing LLMs' former decisions, we present `TestWeaver`, a cost-effective LLM-based testing framework. Leveraging a semantic annotated UI Transition Graph (UTG), `TestWeaver` is capable of detecting shared interactions across test cases. It processes each interaction with a single LLM query, and reuses the result whenever the same interaction occurs. We evaluate `TestWeaver` on real-world test suites from Meituan. It achieves a 92% success rate with an average cost of $0.11 and 89.7 seconds per case, outperforming the state-of-the-art. We have also deployed `TestWeaver` in a real-world testing workflow at Meituan for over six months. `TestWeaver` has executed nearly 2,000 test cases and uncovered 10 previously undetected bugs, while reducing manual testing effort by 75%.**

***Index Terms*—Automatic UI Testing, Large Language Models, Mobile Apps, Cost Optimization.**

I. Introduction

In software engineering, large-scale testing is widely adopted for ensuring the quality and reliability of complex software systems [1]. This testing typically involves developing and maintaining a large volume of test cases. As a result, redundant test cases are inevitable [2]. Such redundancies can significantly hinder testing efficiency and increase costs, particularly in the context of modern software systems with fast development cycles and high testing demands. Consequently, detecting and removing redundant test cases to improve test efficiency has long been a prominent research topic [2]–[7].

Despite the extensive efforts to detect redundant test cases, our industrial experience in mobile app User Interface (UI) testing reveals a new form of redundancy: *redundant UI interactions*. In practice, testing whether an app functions properly requires a large number of test cases [8]–[10]. Most of these cases, as illustrated in Figure 1, are derived from specifications and expressed in natural language [11], [12], with each describing its start UI page and a sequence of UI interactions. Given the widespread functional dependencies within an app and the need for large-scale test cases for comprehensive coverage, the same UI interactions, or even identical interaction prefixes, are repeatedly appearing across different cases. We further quantify this phenomenon in Section II, showing that redundant UI interactions are prevalent across test cases, with 50% of interactions being redundant.

Despite its prevalence, interaction redundancy in UI testing is not always a major concern. For instance, in script-based testing workflows, test cases are manually converted into hard-coded UI scripts and executed using tools like Appium [13]. In such a testing scenario, redundant UI interactions have little impact as the efficiency bottleneck lies in the labour-intensive script development process. However, advanced Large Language Models (LLMs) [14] are increasingly adopted for mobile app testing [15]–[17], rendering redundant UI interactions as a new efficiency bottleneck. Specifically, to reduce the manual effort in developing scripts, many LLM-based UI agents [11], [12], [18]–[22] are proposed to directly execute test cases in natural language. Although promising for cutting manual effort, these agents typically query LLM multiple times at each step, which incurs substantial latency and token cost. Therefore, reducing redundant LLM requests in processing the same interactions has become one of the key points to improve test efficiency.

However, due to the inherent diversity of natural language, detecting redundant UI interactions before execution is non-trivial. Due to the lack of standardized guidelines, test cases authored by different testers often exhibit significant diversity. Some cases may only state the goal (*e.g.*, *"Test login functionality"*) without specifying the interactions involved, which may implicitly cover interactions that are included in other cases. Additionally, some cases may skip intermediate interactions or combine multiple interactions into one description. Hence, this diversity renders direct comparison using natural language processing techniques impractical.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00284

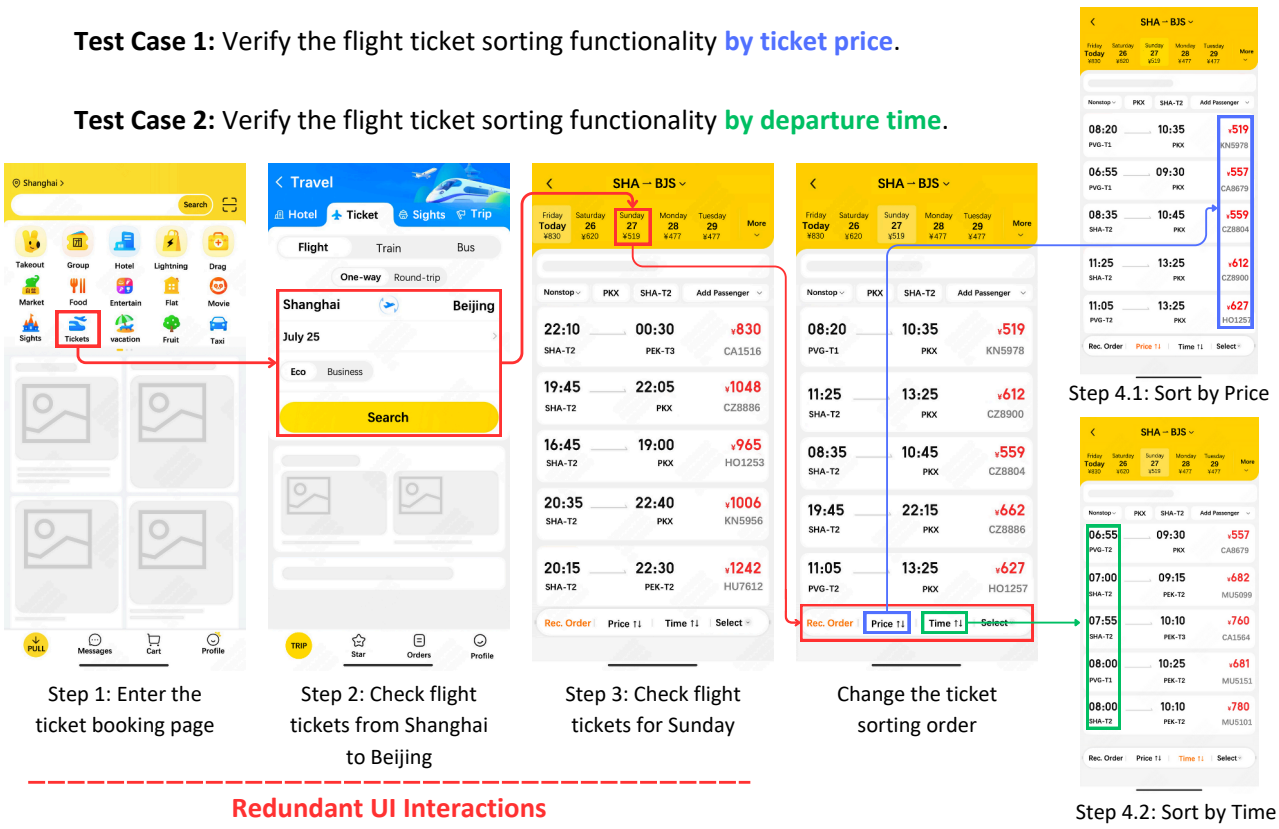

Fig. 1: Two test cases sharing redundant UI interactions.

Both test cases aim to verify the flight ticket sorting functionality in the Meituan app. Test Case 1 involves steps 1, 2, 3, and 4.1. Test Case 2 involves steps 1, 2, 3, and 4.2. They share redundant steps 1, 2, and 3.

The key to overcoming this challenge lies in standardizing diverse test cases into the same granularity. The UI Transition Graph (UTG) [23]–[27] provides an ideal solution, as it captures interactions that triggers page transitions within an app. Formally, UTG represents the structure of an app's UI as a directed graph $G = (V, E)$. Each vertex v represents a unique UI page, and each edge e represents an interaction that triggers page transitions. Based on this representation, diverse test cases can be standardized into comparable sequences via path mapping.

To this end, we implement `TestWeaver`, a tool that constructs a UTG for the app under test (AUT), maps test cases as paths in the graph, and detects redundant interactions for efficient execution. `TestWeaver` first builds the UTG through UI exploration to capture all reachable UI pages and interactions within the AUT. Then, it enhances the UTG with semantic information, ensuring more accurate mappings. Next, `TestWeaver` uses LLM to map each test case—whether high-level or detailed—onto the UTG by linking its start page to a vertex and its interactions to corresponding edges. Therefore, `TestWeaver` can exploit redundant UI interactions: each interaction is processed by LLM only once, and reused whenever it appears again in other cases.

We evaluate the effectiveness of `TestWeaver` on three real-world test suites from the Meituan app [28], a lifestyle super-app with over 1 million downloads on Google Play. `TestWeaver` can accurately standardize 96% of the test cases by leveraging semantic-enhanced UTGs, and can identify redundant interactions across them. Compared to two state-of-the-art agents, `TestWeaver` achieves the best of both worlds: 50% higher success rate, 42% lower economic cost, and 19% shorter execution time. Moreover, we deploy `TestWeaver` in Meituan, one of the largest e-commerce providers with nearly 700 million users. `TestWeaver` run 30 rounds of regression testing (around 2,000 test cases) in a real-world testing workflow, detecting 10 previously unseen bugs. Compared with prior manual testing practice, `TestWeaver` saved 75% of the involved manual effort, while also providing benefits in test case management, as confirmed by test engineers' feedback.

The contributions of this paper are as follows:

- **New problem.** We are the first to show that *redundant UI interactions* create a critical efficiency bottleneck for LLM-based UI testing. Quantified via real-world studies, this previously overlooked defect incurs up to 50% unnecessary LLM costs and hinders industrial deployment.
- **Novel idea and tool implementation.** We present

TABLE I: Overview of the Test Suites.

Module Name	#Cases	Avg. Steps	Max. Steps	Min. Steps	Representative Case
Hotel Booking	52	5.38	8	3	On the hotel listing page, change the current location to Hong Kong, claim a coupon, and then click on the first hotel in the list to access its detail page.
Travel Booking	13	8.00	14	4	On the homepage of the attraction, tap "Adult Ticket". Tap the "Book Now" button in the bottom right corner. Tap "Add", then "Add Frequent Traveler". Enter the name, ID number and contact phone number in the input box, respectively. Tap the Save button. Tap the "Done" button at the bottom. Tap "Quick Pay", then tap "Done".
Food Delivery	24	4.08	6	2	On the food delivery homepage, place an order for any discounted group-buy item.

`TestWeaver`, which can unify test cases into a standardized UTG structure and exploit redundant interactions. By mapping test cases as paths in the UTG, `TestWeaver` is able to standardize diverse test cases, cache LLM responses, and reuse the cached responses for processing natural language interactions.

- **Practical experience.** We conduct comprehensive experiments to evaluate `TestWeaver` on three real-world test suites. And we show its successful application in a practical field deployment at Meituan.

II. MOTIVATION STUDY

To highlight the motivation behind our work, we first analyze real-world test cases from Meituan, *quantifying the prevalence of redundant UI interactions*. Next, we evaluate the cost of executing these test cases using two state-of-the-art LLM-based agents to *better understand the challenges of deploying them in industrial settings*. This study aims to answer two research questions (RQs):

- **RQ1:** How prevalent are redundant interactions among real-world test cases?
- **RQ2:** How costly is it to execute real-world test cases using existing LLM-based agents?

A. Setup

1) Dataset: To answer these questions, we collect three sets of real-world test cases (*i.e.*, test suites) from the Meituan app. Each test suite focuses on a distinct feature module with unique entry points and UI navigation paths, and is developed independently by different quality assurance teams. To ensure validity, we only count redundant interactions within test suites, not across suites. Table I summarizes the three test suites we used, including the number of test cases, length distributions, and representative examples. In total, the test suites contain 89 test cases, with an average length of 5.42 steps, ranging from 2 to 14 steps.

We do not use existing public benchmarks [11], [29], [30] for RQ1, as they typically contain far fewer test cases per app than real-world industrial practice.

2) Key Definitions: Before presenting results for RQ1 and RQ2, we first define key terms used throughout this study.

a) Action Space: We categorize all involved UI interactions into the following types:

- *Tap*: tap on a specific UI widget.
- *Long press*: long pres on a specific UI widget.
- *Scroll*: scroll a UI widget or screen in a specific direction and distance.
- *Input*: input text into an activated input field when the virtual keyboard is visible.

b) Test Case: Despite their varied natural language descriptions, each test case c corresponds to a sequence of interactions starting from an initial UI page. We formally represent it as:

$$c = (v_0, \vec{e_{01}} \cdots, \vec{e_{(n-1)n}}) \tag{1}$$

where v_0 is the starting UI page, and $\vec{e_{(i-1)i}}$ represents an interaction that transitions the UI from page v_{i-1} to page v_i.

c) Redundant UI Interaction: Given this definition of UI pages, we define an interaction $\vec{e_{(i-1)i}}$ as *redundant* if there exists another interaction $\vec{e_{(j-1)j}}$ (either from the same or another test case) such that:

$$v_{i-1} = v_{j-1} \text{ and } v_i = v_j \tag{2}$$

In other words, two interactions are considered redundant if they both start from the same UI page and end on the same UI page.

3) Test Cases Execution: To obtain the ground truth interaction sequence of each test case, two authors independently executed all the test cases on real devices. The execution results were recorded using the Android Debug Bridge (adb) tool [31], capturing all the performed interactions. In cases of discrepancies, a third author would join to resolve the issue and reach a consensus. All the involved authors have over two years of experience in mobile app testing and are active users of the Meituan app.

B. RQ1: Prevalence of Redundant UI Interactions

1) Metrics: In RQ1, we use the following three metrics to quantify the prevalence of redundant interactions:

- *Redundant Interaction Diversity Rate (RIDR, %)*: Reflects how *diverse* the redundancy is at interaction level. Let I

TABLE II: RQ1: The prevalence of redundant interactions.

Test Suite	Interaction-level Metrics			Case-level Metrics			
	#Redundant Interactions	*#Total Unique Interactions*	*RIDR*	*#Cases w/ Redundancy*	*#Total Cases*	*RCCR*	*ARP*
Hotel Booking	40	48	0.83	52	52	1.00	0.97
Travel Booking	18	62	0.29	10	13	0.77	0.69
Food Delivery	14	38	0.37	22	24	0.92	0.79
Avg.	-	-	0.50	-	-	0.94	0.89

TABLE III: RQ2: The execution result of existing LLM-based UI agents.

Method	Hotel Booking				Travel Booking				Food Delivery				Avg.			
	SR ↑	*SA* ↑	*Time* ↓	*Cost* ↓	*SR* ↑	*SA* ↑	*Time* ↓	*Cost* ↓	*SR* ↑	*SA* ↑	*Time* ↓	*Cost* ↓	*SR* ↑	*SA* ↑	*Time* ↓	*Cost* ↓
AppAgent	0.19	0.69	202.84	0.56	0.23	0.45	380.40	0.91	0.25	0.67	236.28	0.55	0.21	0.60	237.79	0.61
AutoDroid	0.19	0.41	104.79	0.20	0.31	0.51	194.00	0.33	0.08	0.20	81.03	0.11	0.18	0.39	111.41	0.19

denote the set of unique interactions and $\mathrm{count}(i)$ be the occurrence count of interaction i.

$$RIDR = \frac{|\{i \in I \mid \mathrm{count}(i) > 1\}|}{|I|} \quad (3)$$

This metric reflects, among all the involved interactions, how many of them is not unique.

- *Redundant Case Coverage Rate (RCCR, %)*: Reflects how *widespread* the redundancy is at case level.

$$RCCR = \frac{\#\text{test cases that contain redundancy}}{\#\text{all test cases}} \quad (4)$$

This metric captures the proportion of test cases that exhibit any form of redundancy, regardless of how many redundant interactions they contain.

- *Average Redundancy Proportion (ARP, %)*: Reflects how *severe* the redundancy is within the affected cases. Let N be the number of test cases that contain at least one redundant interaction.

$$ARP = \frac{1}{N}\sum_{i=1}^{N} \frac{\#\text{redundant interactions in case } i}{\#\text{all interactions in case } i} \quad (5)$$

This metric captures the average share of redundant interactions in those test cases that include redundancy.

2) Results: Table II presents the results of RQ1, demonstrating that redundant interactions are prevalent in real-world UI test cases.

To assess the diversity of redundancy, we report the macro average of RIDR. On average, 50% of all distinct interaction types appear as redundant in at least one case. Such diversity suggests that simple rule-based methods to detect certain specific interactions are insufficient for detecting redundancy, highlighting the need for our UTG-based approach. Upon further analysis, we found these redundancy reasonable, typically relate to key features of each module (*e.g.*, hotel search in hotel booking).

For RCCR, the micro average is 0.94, meaning 94% of test cases contain at least one redundant interaction, confirming that redundancy is widespread. An explanation for such a high RCCR is: a test case with 0 RCCR means that all the interactions in this case are unique, and no other test cases have explored any of the interactions within it. This rarely happens in practice, as most test cases start from the same feature homepage and involve at least one or two common UI interactions.

The micro average for ARP is 0.89, showing that redundancy is not only common but also severe when it occurs. For instance, the ARP for hotel booking reaches 0.97, suggesting that test cases for this module tend to share long interaction prefixes and only diverge at the final one or two steps. According to case designers, this is intentional when they design the test cases. They aim to simulate how users would access various functions from the entry page, resulting in highly overlapping test flows. Interestingly, travel booking has a relatively low RIDR of 0.29 but still shows a high ARP of 0.69. This indicates that in this test suite, a small portion of interactions repeatedly appear. And we found them to be the essential steps for accessing most of the features in this module (*e.g.*, fill in personal information when booking tickets).

C. RQ2: Execution Cost of LLM-based Agents

1) Baselines: In RQ2, we select two state-of-the-art LLM-based agents as baselines: AppAgent [21] and AutoDroid [20]. These agents are chosen based on the following considerations: (1) they are recent and influential in the field; (2) both are open-sourced; (3) both can be successfully launched on Meituan app.

2) Metrics: To comprehensively evaluate the execution cost of LLM-based agents, we adopt the following metrics, commonly used in prior work:

- *Success Rate (SR, %)*: the percentage of test cases that are successfully completed by the agent. A case is considered successful if the entire interaction sequence generated by the agent exactly matches the ground truth sequence.
- *Step Accuracy (SA, %)*: the percentage of individual interactions that are correctly executed by the agent.
- *Average Execution Time (Time, seconds)*: the average time taken by the agent to execute each test case.

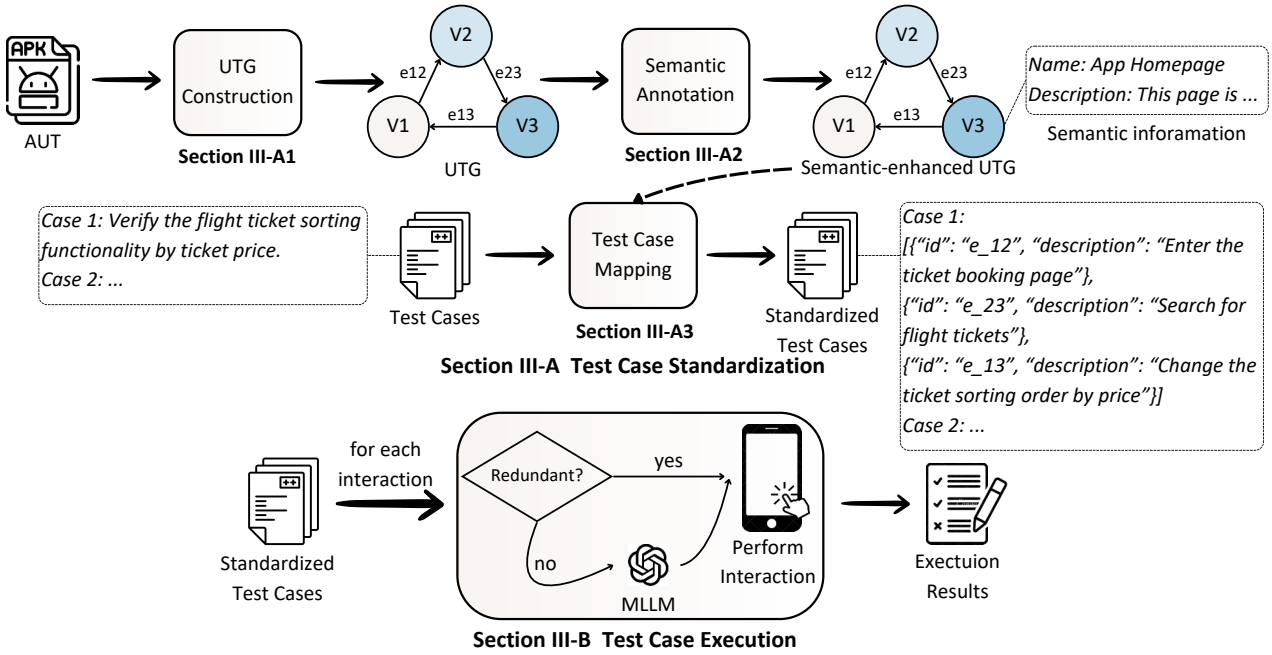

Fig. 2: Overview of `TestWeaver`.

- *Average Execution Cost (Cost, USD)*: the average monetary cost of executing each test case, estimated based on the number of prompt and completion tokens consumed by the LLM. We use GPT-4o [14] as the underlying model, and the pricing is calculated according to the official OpenAI API pricing [32].

3) Results: Table III reports the execution results of the two LLM-based agents. We make three key observations:

First, we find a clear trade-off between accuracy and execution efficiency. AppAgent, which leverages MLLMs and takes UI screenshots as input at each step, generally achieves higher step accuracy (0.60 vs. 0.39) and success rate (0.21 vs. 0.18) compared to AutoDroid, which leverages LLMs and uses structural XML [33] as input. However, this improvement comes at a significant cost: AppAgent consumes more than 2x execution time and 3x monetary cost on average.

Second, despite low success rates, the relatively high step accuracy suggests that most interactions are still correctly performed. This indicates that such agents can still alleviate manual effort in checking or executing test cases. Our analysis reveals that the low SR primarily stems from the agents' inability to determine the correct stopping point of a test case: they often continue exploring even after completing all intended interactions. This issue arises because most test cases specify only the testing objective rather than a precise sequence of steps to follow.

Third, we note that although the per-case execution cost appears low (less than $1), the total cost becomes significant in industrial settings. For instance, executing 100 test cases (a typical scale for regression testing) would require over 6.6 hours and cost more than $60 per round. Such overhead challenges the scalability and practicality of current LLM-based agents, especially for fast-paced release cycles.

III. Approach

Figure 2 provides an overview of `TestWeaver`, which comprises two main modules: *Test Case Standardization* (Section III-A) and *Test Case Execution* (Section III-B).

The first module constructs a semantic-enhanced UTG for the AUT, with test cases marked as paths onto it. For each AUT, the UTG is built only once through dynamic exploration, with each interaction uniquely identified and stored as a UTG edge. The UTG is then annotated with natural language semantics for mapping test cases as a sequence of UTG edges (*i.e.*, standardized test cases). These sequences are stored in natural language without coordinates, ensuring portability across AUT versions and device resolutions.

The second module handles the execution of the standardized test cases, processing them step-by-step. Each natural language interaction is converted by LLM only once, and the resulting LLM response is cached. This cached response is reused whenever the same interaction is encountered in other test cases, thus reducing redundant LLM queries.

A. Test Case Standardization

1) UTG Construction: `TestWeaver` constructs a comprehensive UTG through dynamic exploration, as presented in Algorithm 1. It begins by launching the AUT and capturing the app homepage as the root node of the graph. It then performs depth-first traversal using the `DFSExplore` function, recursively explore reachable UI pages.

On each UI page, `TestWeaver` first invokes *ExtractInteractiveWidgets()* to identify all the interactive widgets. This is achieved by parsing the UI hierarchy file [33] (*i.e.*, XML) and filtering relevant XML nodes. Each UI widget corresponds to one or more nodes in the XML, and we define a UI widget as interactive if any of its corresponding nodes meet any of the following criteria: (1) Any of the `clickable`, `long-clickable`, `checkable`, or `scrollable` properties is set to true; (2) the `class` property is `EditText` [34], indicating that the widget is an input field. Each interactive widget is then located by taking the union of the `bounds` properties of its corresponding XML nodes.

Then, each identified widget is interacted with *Interact()*. After each interaction, `TestWeaver` captures the resulting page via *CapturePage()*, which uses adb [31] to get the screenshot, XML, and additional metadata including the current activity and fragments.

To determine whether an interaction has led to a new page, `TestWeaver` invokes *IsNewVertex()*. If a new page is confirmed, it is added to V as a distinct vertex. And the interaction that leads to this page is added as a new edge in E. Specifically, we adopt the idea from Screen Transition Graph (STG) [35] to distinguish between UI pages, which has a finer granularity than Activity Transition Graph and aligns with how human perceive different UI pages. For each existing vertex $v \in V$, it is compared against page v' through two steps: first, `TestWeaver` verifies if v' shares the same activity name as v. Then, it calculates whether the similarity between their visible UI component (*e.g.*, fragments, menus, navigation drawers, etc.) distributions exceeds a threshold θ.

Finally, to avoid infinite loops, the exploration is bounded by a configurable depth D and step N. Once the maximum exploration depth is reached, `TestWeaver` ensures the AUT returns to the last UI page using *NavigateBackTo()*. Since directly using the system event `Back` does not always guarantee returning to the correct previous page v, in this function, `TestWeaver` restarts the AUT and replays the previously explored path to reach page v. If the maximum exploration step is reached, `TestWeaver` stops constructing.

Moreover, it is important to note that UI exploration is not equivalent to test case execution: the former aims to explore the AUT's UI and interacts randomly, while the latter focuses on validating whether specific interaction sequences cause crashes or lead to unresponsive behaviors.

2) Semantic Annotation: After constructing the UTG, `TestWeaver` uses LLM to annotate the semantic information for each UI page in V and each interaction in E. This submodule serves two purposes: First, MLLMs perform poorly when processing multiple screenshots simultaneously [36]. To ensure more accurate test case mapping, it is necessary to convert them into text form. Second, processing dozens of screenshots when mapping a test case is computationally expensive. By annotating all the elements in advance, we reduce runtime cost via one-time preprocessing with acceptable overhead.

The prompts used for annotating UI pages and interactions are shown in Table IV. We adopt a few-shot prompting strategy when designing the prompt, ensuring annotation consistency across pages and interactions.

Algorithm 1: UI Transition Graph Construction

Input: Maximum exploration depth D, Maximum exploration step N
Output: UI Transition Graph $G = (V, E)$

Function `BuildUTG(`D, N`)`**:**
  $V \leftarrow \emptyset$;
  $E \leftarrow \emptyset$;
  stepCount $\leftarrow 0$;
  LaunchApp();
  $v_0 \leftarrow$ CapturePage();
  $V \leftarrow V \cup \{v_0\}$;
  DFSExplore(v_0, V, E, D, N, *stepCount*);
  return $G = (V, E)$;
Function `DFSExplore(`v, V, E, D, N, *stepCount*`)`**:**
  if $D = 0$ ***or*** *stepCount* $\geq N$ **then**
    return
  end
  $widgets \leftarrow$ ExtractInteractiveWidgets(v);
  foreach $w \in widgets$ **do**
    if *stepCount* $\geq N$ **then**
      break;
    end
    if ***not*** $w.visited$ **then**
      Interact(w);
      stepCount $\leftarrow$ *stepCount* $+ 1$;
      $w.visited \leftarrow$ true;
      $v' \leftarrow$ CapturePage();
      if *IsNewVertex*(v', V) **then**
        $V \leftarrow V \cup \{v'\}$;
        $e \leftarrow (v, w, v')$;
        $E \leftarrow E \cup \{e\}$;
        DFSExplore(v', V, E, $D - 1$, N, *stepCount*);
      end
      NavigateBackTo(v);
    end
  end

3) Test Case Mapping: Next, `TestWeaver` standardizes natural language test cases prior to execution based on the semantic-enhanced UTG. This step addresses a key inefficiency in prior LLM-based agents: they rely on multi-turn prompting strategies like ReAct [37], requesting multiple LLM queries for each interaction (comprehend the current UI, track test progress, and locate the correct UI widget to interact with). In contrast, `TestWeaver` directly maps each test case into a sequence of concrete interactions on the UTG, removing the need for querying the LLM to track test progress at runtime.

The mapping prompt and resulting standardized test cases are illustrated in Table V. Each standardized test case is represented as a sequence of interactions, where each interaction includes:

TABLE IV: Prompts used in semantic annotation.

Prompt Type	Prompt Purpose	Prompt Content
UI Page Summary Prompt		
Role definition	Set LLM's role as a mobile app testing engineer.	You are a mobile app testing engineer.
Input context	Provide a UI screenshot of the current page.	You are presented with a UI page screenshot. <screenshot>
Task instruction	Describe the current UI page and its sections.	Summarize the type and description of the current page. Focus on the header, content, footer, and sidebar sections. Ignore the status area and keyboard.
Few-shot example	Verified example of a UI page summary.	Example: {"type": "Settings page", "header": "The title 'Settings' is displayed at the top center", "content": "A list of settings options is displayed, including ...", "footer": "The 'Save' button is at the bottom center", "sidebar": "No sidebar present", "function": "Allows the user to modify application settings." }
Output format	Require structured JSON with layout regions and functional tags.	Format the output strictly in JSON. Include the following fields: type, header, content, footer, sidebar, and function. Refer to the example.
Interaction Summary Prompt		
Role definition	Set LLM's role as a mobile app testing engineer.	You are a mobile app testing engineer.
Input context	Provide summaries of UI pages before and after the interaction, along with the interacted widget's image and XML nodes.	You are presented with two UI page descriptions and a recorded interaction. <image> <summary of UI page 1> <summary of UI page 2> <XML nodes>
Task instruction	Describe the interaction and summarize the appearance and function of the interacted widget.	Focus on the widget you interacted with: describe its appearance, shape, color, etc. Then, based on the provided summaries of the two pages, infer and describe the function of the widget.
Few-shot example	Verified example of an interaction summary.	Example: {"action": "Tap", "description": "The widget is a white circular icon", "function": "Opens the settings menu to adjust preferences." }
Output format	Require structured JSON with description and function fields.	Format the response strictly in JSON with three fields: "action" (type of the interaction), "description" (appearance of the widget) and "function" (inferred function of the widget). Refer to the example.

its edge ID in the UTG, the interaction type (*e.g.*, click, input), and natural language descriptions derived from the LLM (*i.e.*, the interacted widget's appearance and inferred function). This structured format allows identification of redundant interactions via duplicated edge IDs across cases.

B. Test Case Execution

To execute the interactions in the standardized test cases, `TestWeaver` leverages an MLLM to convert natural language descriptions into concrete screen operations, such as tapping or typing at specific coordinates. To ensure precise execution, `TestWeaver` adopts a visual grounding strategy similar to AppAgent [21]. Specifically, for each interaction, it first processes the current UI screenshot using the Set-of-Mark (SoM) technique [38], which identifies and highlights all interactive widgets within the image. This processed screenshot, together with the natural language description of the interaction, is then provided to the MLLM. The model uses this information to select the correct widget to interact with and outputs the corresponding screen operation (*e.g.*, {"action": "tap", "coordinates": [750, 2050]}). Then, `TestWeaver` performs the screen operation via the adb tool. The results of these operations are stored in a cache, where each key corresponds to the interaction's edge ID, and the value stores the generated screen operation. This cache is queried before sending any requests to the MLLM, thus preventing redundant queries for repeated interactions.

IV. Evaluation

In this section, we evaluate the performance of `TestWeaver` by answering two RQs (RQ3 and RQ4) and providing insights into deploying it. Then, we report our field experience in deploying `TestWeaver` at Meituan (RQ5).

The RQs are as follows:

- **RQ3:** What is the overhead of `TestWeaver` in standardizing test cases? How accurate are the interaction sequences within the standardized test cases?
- **RQ4:** How costly is it to execute real-world test cases using `TestWeaver`?
- **RQ5:** What is the practical usage of `TestWeaver` in real-world UI testing workflows?

A. RQ3: Overhead and Accuracy of Test Case Standardization.

Since our goal is to enable cost-effective LLM-based testing, it is both necessary and fair to quantify the overhead of the test case standardization module of `TestWeaver` —namely, constructing the UTG, performing semantic annotation, and mapping test cases. Since Meituan app is a super-app with distinct feature modules, we construct separate UTGs for each module.

TABLE V: Prompts used in test case mapping.

Prompt Type	Prompt Content
Role Definition	You are a mobile app testing engineer.
Input Context	I will provide you with a UTG and a test case written in natural language.
UTG Example	Page 1: {activity: `com.sankuai.meituan.MainActivity`, fragment: [Fragment(id:0x7f0a0059, name: `AppCategorySelectFragment`), ...], summary: {"type": "app homepage", "header": "The title 'meituan' is displayed ...", ...} } ... Edge 1: {"from": "page 1", "to": "page 3", "action": "tap", "description": ...} ...
Case Example	Verify that the flight ticket sorting functionality by ticket price works correctly.
Task Instruction	Each test case corresponds to a sequence of UI interactions. Your task is to map the given case to a list of edges in the UTG. First, identify the starting UI page. Then, trace the relevant edges step by step on the UTG.
Output Format	Respond in JSON with a key `"Edges"` listing the selected `edge_ids`, and a key `"Reason"` explaining your choices.
LLM Response	`{"Edges": [5, 7, 1, 9, 11], "Reason": "The start page is page 2. Since the test case is to verify the flight ticket sorting functionality, the expected target is page 12. A valid path from page 2 to 12 is ..."}`

The following metrics are used to measure this module:

- *Time(seconds) and Cost(USD)*: We measure the time spent by each submodule when processing the three test suites used in the motivation study. Additionally, we track the monetary cost incurred from invoking the LLM for semantic annotation and test case mapping.
- *Accuracy*: Accuracy of the generated standardized test cases is critical, as inaccurate interaction sequences would render a test case invalid. Moreover, accuracy reflects the completeness of the constructed UTG.

Table VI summarizes the results. We observe that `TestWeaver` achieves high standardization accuracy (ranging from 85% to 100%) with acceptable overhead. Processing each test suite takes approximately one hour, with monetary costs ranging from $5.02 to $8.68. This overhead is acceptable, considering that the standardized cases are reusable in future testing.

The overhead distribution across submodules and test suites is consistent. Specifically, UTG construction dominates the time consumption, accounting for 75% to 80% of the total time. This is due to the time spent waiting for UI responses during exploration. In contrast, test case mapping completes relatively quickly. The time for UTG construction and semantic annotation is consistent across modules since the UI exploration step is fixed, resulting in similar UTG sizes. Time differences mainly arise from the test case mapping submodule, which is proportional to the number of test cases involved. Semantic annotation costs approximately $3.00 per suite, as the UTG size is similar. While the test case mapping submodule costs increase with the number and length of the test cases.

As for accuracy, `TestWeaver` achieves 85% to 100% accuracy across test suites. Failures are limited to at most 2 cases per suite. We manually examined these failures and identified that they were caused by the lack of explicit start page descriptions in the test cases. When designing them, the testers default the start page of these cases to the feature module's homepage, leading to LLM errors in locating the correct start vertex in the UTG. This suggests that when deploying `TestWeaver` in real-world testing workflows, the preferences of test case designers could be included in the prompt of the test case mapping submodule for better performance.

B. RQ4: Cost of Executing Real-World Test Cases.

In RQ4, we evaluate the performance of `TestWeaver` in executing real-world test cases. We use the same dataset and evaluation metrics as in RQ2, comparing `TestWeaver` with two state-of-the-art LLM-based agents (AppAgent and AutoDroid) as baselines. It should be noted that all inaccurate standardized test cases are considered failures when calculating success rate and step accuracy.

Table VII presents the execution results. `TestWeaver` significantly outperforms both baselines, achieving an average success rate of 0.92 and step accuracy of 0.90. It achieves a 338% improvement in success rate and a 50% improvement in step accuracy, even compared with the best baseline (AppAgent). The key factor driving this huge improvement is the test case standardization module. As discussed in RQ2, the two baselines often continue interacting even after a test case is completed, resulting in a low success rate. Unlike them, the test case standardization module prevents `TestWeaver` from doing so: it executes the interaction sequences step-by-step, and stops when all the listed interactions are performed. Furthermore, the detailed description in standardized test cases reduces the chances of LLM hallucination. `TestWeaver` only needs to locate the target widget for each interaction, while the two baselines need to comprehend the current UI and determine the test progress at each step. Despite `TestWeaver`'s strong performance, it still fails in 7 cases. Four failures are due to inaccurate test case mapping, and three are caused by LLM hallucinations when locating target UI widgets.

Regarding execution efficiency, `TestWeaver` reduces the average execution time per test case to 89.71 seconds, which is 63% faster than AppAgent and 19% faster than AutoDroid. This improvement is remarkable, given that AutoDroid uses LLMs to process XML nodes at each step, while `TestWeaver` employs a more costly MLLM to process UI screenshots. Monetary costs are also drastically reduced. `TestWeaver` 's average execution cost is $0.11 per test case, 42% lower than AutoDroid and 82% lower than AppAgent.

These cost savings come from `TestWeaver` 's exploitation of redundant interactions, which avoids redundant LLM queries

TABLE VI: RQ3: Overhead and accuracy of test case standardization.

SubModule	Hotel Booking			Travel Booking			Food Delivery		
	Acc ↑	*Time* ↓	*Cost* ↓	*Acc* ↑	*Time* ↓	*Cost* ↓	*Acc* ↑	*Time* ↓	*Cost* ↓
UTG Construction	-	2492.93	-	-	2954.81	-	-	2565.99	-
Semantic Annotation	-	685.07	2.99	-	719.19	3.21	-	657.01	2.92
Test Case Mapping	0.96	144.56	5.69	0.85	35.49	2.58	1.00	67.2	2.10
Total	-	3322.56	8.68	-	3709.49	5.79	-	3290.2	5.02

TABLE VII: RQ4: Comparison of `TestWeaver` against baseline agents.

Method	Hotel Booking				Travel Booking				Food Delivery				Avg.			
	SR ↑	*SA* ↑	*Time* ↓	*Cost* ↓	*SR* ↑	*SA* ↑	*Time* ↓	*Cost* ↓	*SR* ↑	*SA* ↑	*Time* ↓	*Cost* ↓	*SR* ↑	*SA* ↑	*Time* ↓	*Cost* ↓
AppAgent	0.19	0.69	202.84	0.56	0.23	0.45	380.40	0.91	0.25	0.67	236.28	0.55	0.21	0.60	237.79	0.61
AutoDroid	0.19	0.41	104.79	0.20	0.31	0.51	194.00	0.33	0.08	0.20	81.03	0.11	0.18	0.39	111.41	0.19
`TestWeaver`	**0.92**	**0.95**	**78.08**	**0.08**	**0.77**	**0.78**	**166.44**	**0.26**	**1.00**	**1.00**	**73.36**	**0.10**	**0.92**	**0.90**	**89.71**	**0.11**

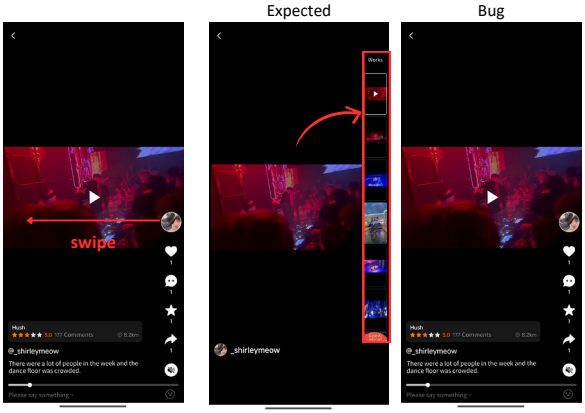

Fig. 3: A bug detected by `TestWeaver`.

by caching previously grounded interactions. We analyzed the execution logs and found that, in the hotel booking suite, 30 out of 52 cases, in the travel booking suite, 3 out of 13 cases, and in the food delivery suite, 10 out of 24 cases did not invoke MLLM at all. All interactions within these test cases are redundant and have been processed in previous test cases, confirming that `TestWeaver` eliminates unnecessary LLM queries.

This improvement is crucial for real-world scalability, particularly in large-scale test suites with frequent updates. For example, executing a suite with 100 test cases would take AppAgent 6.6 hours and $61, AutoDroid 3.1 hours and $19, while `TestWeaver` can complete it in less than 2.5 hours and $11, with a much higher success rate. Even considering the test case standardization overhead, `TestWeaver`'s total cost would be around 3.5 hours and $17—comparable to AutoDroid's cost but with significant performance advantages. Moreover, this cost advantage grows with multiple rounds of testing.

C. RQ5: Practical Usage under Industrial Settings.

With the growing popularity of short-form videos, merchants tend to upload few-second clips for product promotion [39]. To meet this demand, the Meituan app launched a short-form video feature module in September 2022, now serving for over 0.56 million merchants with 2.5 million uploaded videos. Ensuring a smooth user experience for such a large user base makes regression testing of this feature module critical for Meituan.

However, testing this feature module is both labour-intensive and time-consuming. Given the competitive app market, this feature module undergoes rapid evolution to retain user interest. Therefore, numerous test cases are created in parallel by multiple testers, accumulating over time. In the past, these cases were executed manually or through script-based testing, creating a major bottleneck. To address this, `TestWeaver` has been integrated into the testing workflow of this feature module since its released.

`TestWeaver` triggered whenever a new version of the module is released. Between June 2024 and December 2024, the module was updated 30 times, and `TestWeaver` executed 30 rounds of regression testing, covering 1,973 test cases in total. After execution, testers only need to manually review the cases that `TestWeaver` fails to execute completely, to check whether a bug presents. Compared with the past, where each regression round required on average 2 person-days for manually executing test cases and checking results, the current workflow with `TestWeaver` reduces the effort by 75%, to about 0.5 person-days (mainly for reviewing `TestWeaver`'s execution results).

This process led to the discovery of 10 previously undetected bugs. Among them, 4 were crashes triggered by specific interaction sequences, and 6 were cases where an interaction in the sequence produced no effect. Figure 3 illustrates one example: swiping left should open a list of video thumbnails, but the page did not respond.

In addition to saving manual effort during testing, `TestWeaver` also provides benefits in test case management. Testers report that the standardized test cases facilitate the

development of new cases by providing clear templates. They also simplify the maintenance of existing cases, as each case includes a detailed description of every interaction, making reviewing easier under tight schedules.

V. Threats to Validity

One potential threat arises from the fact that all test cases in our study are collected from Meituan, which may raise concerns about their representativeness. That is, whether the prevalence of redundant interactions results from some specific practices within Meituan, or this phenomenon can be observed in broader industry test cases. To alleviate this concern, the three test suites used in our experiments were developed independently by different quality assurance teams within Meituan, each targeting distinct app functionalities. Moreover, no common guideline was shared among these teams when designing test cases, reducing the likelihood of systematic bias.

Another potential threat lies in `TestWeaver`'s generalizability to other apps and test cases. Although `TestWeaver` is designed at Meituan, it relies on the reasoning ability of general LLMs, not specifically tailored for the Meituan app or test cases from Meituan. To prove this, we deploy `TestWeaver` using alternative LLMs, including Gemini 2.5 Pro [40] and Claude 3.5 Sonnet v2 [41]. And we observe consistent performance of `TestWeaver`, suggesting that `TestWeaver` 's effectiveness is not tied to a particular model. Moreover, to facilitate reproducibility, we release the core prompts of `TestWeaver` in Section III.

VI. Related Work

A. Test Case Minimization

As discussed in Section I, detecting and removing redundant test cases to improve test efficiency has been a long-standing research topic [2]–[7], commonly referred to as *Test Case(Suite) Minimization* (TSM). TSM typically aims to identify a minimal subset of test cases that satisfy given requirements (*e.g.*, coverage, execution time) while removing redundant ones.

Most existing approaches treat entire test cases, often in code form, as the unit of analysis. For example, Xue *et al.* [4] explores the problem in the context of test code, and conducts experiments on test cases for C programs. Notably, Tscope [2] shares the same form of test cases as our work, which examines redundant test cases in natural language.

Different from them, in this paper, we focus on a finer-grained and often overlooked level of redundancy: redundant UI interactions. That is, even when natural language test cases are distinct and testing different app features, they may still share overlapping UI interactions. We argue that such step-level redundancy is becoming increasingly important, especially as LLMs make it more feasible to directly consume and process natural language test cases.

B. UTG-based Testing

UTG-based testing is a type of model-based testing (MBT) approach [42]–[45], which constructs a UTG to model the AUT and systematically explore its behavior. Significant research has been conducted in this area, leading to tools such as A3E [23], DroidBot [24], WTG [25], GoalExplorer [35], StoryDroid [26], and SceneTG [27].

These methods differ primarily in their UTG representations and the level of modeling granularity. A3E builds an Activity Transition Graph, capturing transitions between Android Activities using source code and hierarchy files. WTG (Window Transition Graph) provides finer granularity by modeling transitions at the Android Window level. GoalExplorer constructs Screen Transition Graphs, also based on code and UI hierarchy, to represent screen-level navigation. StoryDroid and SceneTG go further by creating Storyboards and Scene Transition Graphs, respectively, which incorporate both code and visual information such as screenshots.

Inspired by GoalExplorer, the UTG construction model in `TestWeaver` builds UTG at the screen level, which we find well-aligned with human perception, *i.e.*, how human testers distinguish different UI pages when writing test cases. However, unlike prior works, our UTG uses natural language descriptions to represent the semantic information of UI pages and transitions. This representation is: (1) intuitive for human testers to understand and manually maintain; (2) lightweight for LLMs to process, as it avoids the high token cost associated with inputting screenshots.

C. LLM-based Testing

With their natural language understanding capabilities, and with recent advances in multimodal extensions for image analysis, LLMs are increasingly applied to mobile app UI testing [46]. Among the existing studies, one direction aims to build LLM-based autonomous agents capable of interacting with mobile UIs to fulfill specific tasks. For instance, some studies [17], [47]–[49] focus on reproducing bugs from natural language bug reports via UI interaction. Most others [11], [12], [18]–[22] focus on executing test cases in natural language, reducing the need for manual UI script development and maintenance.

This trend is gaining traction, especially in industry. For example, Feng *et al.* [12], CAT [19], and ProphetAgent [22] demonstrate the deployment of LLM-based agents under industrial settings. Feng *et al.* builds a multi-agent system to execute test cases that test multi-user app features in TikTok. CAT employs Retrieval Augmented Generation (RAG) to source examples of app usage as a few-shot learning context, assisting LLM-based agents in test case execution. ProphetAgent also builds UTG for LLM-based agents. The UTG serves as a knowledge graph for LLM in understanding the AUT. However, these methods treat each test case independently and execute them one by one. In contrast, `TestWeaver` takes a step further for cost-effective UI testing by treating the entire test suite as a whole, detecting and reusing redundant interactions across multiple test cases.

VII. Conclusion

This paper pinpoints an previously overlooked efficiency defect in mobile app UI testing: redundant UI interactions.

Such redundancy accounts for the high execution time and monetary cost of existing LLM-based UI testing methods, which hinders their adoption in industrial-scale testing. In this paper, we present `TestWeaver`, a cost-effective framework that standardizes test cases and reuses prior LLM decisions to eliminate redundant queries. Evaluations on Meituan's test suites show that `TestWeaver` achieves a 92% success rate with with an average cost of $0.11 (42% lower) and 89.7 seconds (19% shorter) per case. Furthermore, six months of industrial deployment on nearly 2,000 test cases demonstrate its practical utility, uncovering 10 previously undetected bugs while cutting manual testing effort by 75%.

In the future, we plan to deploy `TestWeaver` to additional feature modules and evaluate its scalability on larger test suites. Moreover, we also aim to enhance `TestWeaver` with RAG to mitigate errors caused by insufficient domain knowledge during test case mapping and execution.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (Project No. 62572127) and Meituan. Y. Zhou is the corresponding author.

REFERENCES

[1] H. Lin, J. Qiu, H. Wang, Z. Li, L. Gong, D. Gao, Y. Liu, F. Qian, Z. Zhang, P. Yang, and T. Xu, "Virtual device farms for mobile app testing at scale: A pursuit for fidelity, efficiency, and accessibility," in *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking, ACM MobiCom 2023, Madrid, Spain, October 2-6, 2023*, X. Costa-Pérez, J. Widmer, D. Perino, D. Giustiniano, H. Al-Hassanieh, A. Asadi, and L. P. Cox, Eds. ACM, 2023, pp. 45:1–45:17.

[2] Z. Chang, M. Li, J. Wang, Q. Wang, and S. Li, "Putting them under microscope: a fine-grained approach for detecting redundant test cases in natural language," 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. 1161–1172.

[3] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, R. E. K. Stirewalt, A. Egyed, and B. Fischer, Eds. ACM, 2007, pp. 417–420.

[4] Y. Xue and Y. Li, "Multi-objective integer programming approaches for solving the multi-criteria test-suite minimization problem: Towards sound and complete solutions of a particular search-based software-engineering problem," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 3, pp. 20:1–20:50, 2020.

[5] R. Pan, T. A. Ghaleb, and L. C. Briand, "ATM: black-box test case minimization based on test code similarity and evolutionary search," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 1700–1711.

[6] D. J. Kim, J. Yang, and T. Chen, "A first look at the inheritance-induced redundant test execution," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 114:1–114:12.

[7] A. Hora, "Monitoring the execution of 14k tests: Methods tend to have one path that is significantly more executed," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, M. d'Amorim, Ed. ACM, 2024, pp. 532–536.

[8] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the test automation culture of app developers," in *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 2015, pp. 1–10.

[9] M. L. Vásquez, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "How do developers test android applications?" in *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 2017, pp. 613–622.

[10] R. Haas, D. Elsner, E. Jürgens, A. Pretschner, and S. Apel, "How can manual testing processes be optimized? developer survey, optimization guidelines, and case studies," 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. 1281–1291.

[11] D. Ran, H. Wang, Z. Song, M. Wu, Y. Cao, Y. Zhang, W. Yang, and T. Xie, "Guardian: A runtime framework for llm-based UI exploration," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, M. Christakis and M. Pradel, Eds. ACM, 2024, pp. 958–970.

[12] S. Feng, C. Du, H. Liu, Q. Wang, Z. Lv, G. Huo, X. Yang, and C. Chen, "Agent for user: Testing multi-user interactive features in tiktok," in *47th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, SEIP@ICSE 2025, Ottawa, ON, Canada, April 27 - May 3, 2025*. IEEE, 2025, pp. 57–68.

[13] (2024) Appium: Mobile automation framework. [Online]. Available: https://github.com/appium/appium

[14] (2024) Hello gpt-4o. [Online]. Available: https://openai.com/index/hello-gpt-4o/

[15] 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.

[16] 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.

[17] S. Feng and C. Chen, "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, 2024, pp. 67:1–67:13.

[18] 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.

[19] S. Feng, H. Lu, J. Jiang, T. Xiong, L. Huang, Y. Liang, X. Li, Y. Deng, and A. Aleti, "Enabling cost-effective UI automation testing with retrieval-based llms: A case study in wechat," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, V. Filkov, B. Ray, and M. Zhou, Eds. ACM, 2024, pp. 1973–1978.

[20] H. Wen, Y. Li, G. Liu, S. Zhao, T. Yu, T. J. Li, S. Jiang, Y. Liu, Y. Zhang, and Y. Liu, "Autodroid: Llm-powered task automation in android," in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking, ACM MobiCom 2024, Washington D.C., DC, USA, November 18-22, 2024*, W. Shi, D. Ganesan, and N. D. Lane, Eds. ACM, 2024, pp. 543–557.

[21] C. Zhang, Z. Yang, J. Liu, Y. Li, Y. Han, X. Chen, Z. Huang, B. Fu, and G. Yu, "Appagent: Multimodal agents as smartphone users," in *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems, CHI 2025, YokohamaJapan, 26 April 2025- 1 May 2025*, N. Yamashita, V. Evers, K. Yatani, S. X. Ding, B. Lee, M. Chetty, and P. O. T. Dugas, Eds. ACM, 2025, pp. 70:1–70:20.

[22] Q. Kong, Z. Lv, Y. Xiong, D. Wang, J. Sun, T. Su, L. Li, X. Yang, and G. Huo, "Prophetagent: Automatically synthesizing GUI tests from test cases in natural language for mobile apps," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, FSE Companion 2025, Clarion Hotel Trondheim, Trondheim, Norway, June 23-28, 2025*, L. Montecchi, J. Li, D. Poshyvanyk, and D. Zhang, Eds. ACM, 2025, pp. 174–179.

[23] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH*

2013, Indianapolis, IN, USA, October 26-31, 2013, A. L. Hosking, P. T. Eugster, and C. V. Lopes, Eds. ACM, 2013, pp. 641–660.

[24] 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.

[25] S. Yang, H. Wu, H. Zhang, Y. Wang, C. Swaminathan, D. Yan, and A. Rountev, "Static window transition graphs for android," *Autom. Softw. Eng.*, vol. 25, no. 4, pp. 833–873, 2018.

[26] S. Chen, L. Fan, C. Chen, T. Su, W. Li, Y. Liu, and L. Xu, "Storydroid: automated generation of storyboard for android apps," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 596–607.

[27] X. Zhang, L. Fan, S. Chen, Y. Su, and B. Li, "Scene-driven exploration and GUI modeling for android apps," in *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 2023, pp. 1251–1262.

[28] (2024) Meituan. [Online]. Available: https://about.meituan.com

[29] T. Xie, D. Zhang, J. Chen, X. Li, S. Zhao, R. Cao, T. J. Hua, Z. Cheng, D. Shin, F. Lei, Y. Liu, Y. Xu, S. Zhou, S. Savarese, C. Xiong, V. Zhong, and T. Yu, "Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments," in *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, Eds., 2024.

[30] J. Chen, D. Yuen, B. Xie, Y. Yang, G. Chen, Z. Wu, L. Yixing, X. Zhou, W. Liu, S. Wang, K. Zhou, R. Shao, L. Nie, Y. Wang, J. Hao, J. Wang, and K. Shao, "Spa-bench: a comprehensive benchmark for smartphone agent evaluation," in *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025.

[31] (2024) Android debug bridge (adb). [Online]. Available: https://developer.android.com/tools/adb

[32] (2025) Pricing - OpenAI API. [Online]. Available: https://platform.openai.com/docs/pricing

[33] (2024) Layouts in views. [Online]. Available: https://developer.android.com/develop/ui/views/layout/declaring-layout

[34] (2024) Edittext. [Online]. Available: https://developer.android.com/reference/android/widget/EditText

[35] D. Lai and J. Rubin, "Goal-driven exploration for android applications," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 115–127.

[36] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Trans. Assoc. Comput. Linguistics*, vol. 12, pp. 157–173, 2024.

[37] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," in *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[38] 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.

[39] Y. K. Dwivedi, E. Ismagilova, D. L. Hughes, J. Carlson, R. Filieri, J. Jacobson, V. Jain, H. Karjaluoto, H. Kefi, A. S. Krishen, V. Kumar, M. M. Rahman, R. Raman, P. A. Rauschnabel, J. E. Rowley, J. Salo, G. A. Tran, and Y. Wang, "Setting the future of digital and social media marketing research: Perspectives and research propositions," *Int. J. Inf. Manag.*, vol. 59, p. 102168, 2021.

[40] (2025) Gemini 2.5: Our most intelligent ai model. [Online]. Available: https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/#gemini-2-5-thinking

[41] (2024) Introducing computer use, a new claude 3.5 sonnet, and claude 3.5 haiku. [Online]. Available: https://www.anthropic.com/news/3-5-models-and-computer-use

[42] Y. M. Baek and D. Bae, "Automated model-based android GUI testing using multi-level GUI comparison criteria," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, D. Lo, S. Apel, and S. Khurshid, Eds. ACM, 2016, pp. 238–249.

[43] 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.

[44] 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.

[45] S. Fischer, R. Ramler, W. K. G. Assunção, A. Egyed, C. Gradl, and S. Auberger, "Model-based testing for a family of mobile applications: Industrial experiences," in *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume A, SPLC 2023, Tokyo, Japan, 28 August 2023- 1 September 2023*, P. Arcaini, M. H. ter Beek, G. Perrouin, I. Reinhartz-Berger, M. R. Luaces, C. Schwanninger, S. Ali, M. Varshosaz, A. Gargantini, S. Gnesi, M. Lochau, L. Semini, and H. Washizaki, Eds. ACM, 2023, pp. 242–253.

[46] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Trans. Software Eng.*, vol. 50, no. 4, pp. 911–936, 2024.

[47] D. Wang, Y. Zhao, S. Feng, Z. Zhang, W. G. J. Halfond, C. Chen, X. Sun, J. Shi, and T. Yu, "Feedback-driven automated whole bug report reproduction for android apps," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, M. Christakis and M. Pradel, Eds. ACM, 2024, pp. 1048–1060.

[48] Y. Su, D. Liao, Z. Xing, Q. Huang, M. Xie, Q. Lu, and X. Xu, "Enhancing exploratory testing by large language model and knowledge graph," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 98:1–98:12.

[49] Y. Huang, J. Wang, Z. Liu, M. Li, S. Wang, C. Chen, Y. Hu, and Q. Wang, "One sentence can kill the bug: Auto-replay mobile app crashes from one-sentence overviews," *IEEE Trans. Software Eng.*, vol. 51, no. 4, pp. 975–989, 2025.