GUARDIAN: A Runtime Framework for LLM-Based UI Exploration

Dezhi Ran
Key Lab of HCST (PKU), MOE; SCS,
Peking University
Beijing, China
dezhiran@pku.edu.cn

Hao Wang
Peking University
Beijing, China
tony.wanghao@stu.pku.edu.cn

Zihe Song
University of Texas at Dallas
Richardson, USA
zihe.song@utdallas.edu

Mengzhou Wu
Peking University
Beijing, China
wmz@stu.pku.edu.cn

Yuan Cao
Peking University
Beijing, China
cao_yuan21@stu.pku.edu.cn

Ying Zhang
Key Lab of HCST (PKU); NERC of SE,
Peking University
Beijing, China
zhang.ying@pku.edu.cn

Wei Yang
University of Texas at Dallas
Richardson, USA
wei.yang@utdallas.edu

Tao Xie
Key Lab of HCST (PKU), MOE; SCS,
Peking University
Beijing, China
taoxie@pku.edu.cn

Abstract

Tests for feature-based UI testing have been indispensable for ensuring the quality of mobile applications (*apps* for short). The high manual labor costs to create such tests have led to a strong interest in *automated feature-based UI testing*, where an approach automatically explores the App under Test (AUT) to find correct sequences of UI events achieving the target test objective, given only a high-level *test objective description*. Given that the task of automated feature-based UI testing resembles conventional AI planning problems, large language models (LLMs), known for their effectiveness in AI planning, could be ideal for this task. However, our study reveals that LLMs struggle with following specific instructions for UI testing and replanning based on new information. This limitation results in reduced effectiveness of LLM-driven solutions for automated feature-based UI testing, despite the use of advanced prompting techniques.

Toward addressing the preceding limitation, we propose GUARDIAN, a runtime system framework to improve the effectiveness of automated feature-based UI testing by offloading computational tasks from LLMs with two major strategies. First, GUARDIAN refines UI action space that the LLM can plan over, enforcing the instruction following of the LLM by construction. Second, GUARDIAN deliberately checks whether the gradually enriched information invalidates previous planning by the LLM. GUARDIAN removes the invalidated UI actions from the UI action space that the LLM can plan over, restores the state of the AUT to the state before the execution of the invalidated UI actions, and prompts the LLM to re-plan with the new UI action space. We instantiate GUARDIAN with ChatGPT and construct a benchmark named *FESTIVAL* with 58 tasks from 23 highly popular apps. Evaluation results on FESTIVAL show that GUARDIAN achieves 48.3% success rate and 64.0% average completion proportion, outperforming state-of-the-art approaches with 154% and 132% relative improvement with respect to the two metrics, respectively.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; **Runtime environments**; • **Computing methodologies** → **Natural language processing**.

Keywords

UI Testing, Mobile Testing, Android Testing, Large Language Models, Runtime System, Sequential Planning

ACM Reference Format:
Dezhi Ran, Hao Wang, Zihe Song, Mengzhou Wu, Yuan Cao, Ying Zhang, Wei Yang, and Tao Xie. 2024. GUARDIAN: A Runtime Framework for LLM-Based UI Exploration. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680334

1 Introduction

To ensure the high quality of mobile applications (*apps* for short), feature-based UI testing [61] focuses on validating the core functionalities of the App Under Test (*AUT*), and is indispensable [27, 30, 31, 46] yet often incurs significant manual costs [31, 50, 61, 62]. To reduce manual costs, it is a long-sought goal to automatically generate feature-based UI tests directly from test objectives [27], denoted as *automated feature-based UI testing* in this paper. Given an AUT and a test objective as inputs, automated feature-based UI testing explores the AUT to find a sequence of UI actions (i.e.,

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680334

UI events) to achieve the test objective. In particular, automated feature-based UI testing sequentially selects one UI action on a UI screen, triggers the UI action on the AUT, and selects the next UI action on the new UI screen until the test objective is achieved. Consequently, automated feature-based UI testing is inherently a sequential planning problem [7, 57, 63], which has been effectively tackled by large language models (LLMs) [14, 18, 52, 57, 70, 75].

Despite the promising effectiveness of LLMs for sequential planning problems, state-of-the-art approaches of LLM-based feature-based UI testing [52, 70, 75] are shown to achieve low effectiveness according to our empirical investigation (Section 4.2) and one may hypothesize two likely contributing factors for the low effectiveness. First, LLMs may struggle to comprehend the content or environment (i.e., the UI elements) within their operational scope. Second, LLMs may not be skilled at formulating planning strategies specific to such exploration tasks (i.e., feature-based UI testing).

To empirically investigate the preceding hypothesized contributing factors, in this paper, we conduct a preliminary study (Section 2) to produce two findings. First, interestingly, our study results (Section 2.1) contradict the first hypothesized factor, revealing that LLMs are, in fact, quite capable of comprehending UI content. LLMs correctly select UI actions with over 95% accuracy, even amidst distracting elements. Second, our study results (Section 2.2) show that the primary cause of failure in existing approaches is the second factor: inadequate planning strategies. This issue encompasses two major aspects of challenges discussed below, as revealed in our analysis (whose details are in Section 2.2).

Challenges. *Failing to follow domain-specific task instructions.* While LLMs can be experts in understanding UI content, LLMs lack domain knowledge in UI exploration [63, 70] and existing approaches incorporate UI-testing-specific instructions into prompts to help. However, LLMs fail to follow these instructions, leading to low effectiveness. As detailed in our preliminary study, despite Droidbot-GPT [70] using explicit instruction prompts to avoid selecting already selected actions, 36% of the planned actions are simply repeating historical actions.

Failing to replan based on new information. The task of automated UI testing involves highly dynamic, real-time exploration of the AUT, given the fact that the same functionality can be implemented in various ways in different apps. As exploration progresses, newly uncovered information can invalidate previously planned UI actions. Figure 5 presents an example on the Quizlet app, where an LLM is instructed to activate the night mode. On the front page, the LLM chooses to click the search button to pursue a shortcut setting of the night mode. While finding that the search tab cannot achieve its previous planning purpose, the LLM fails to replan with the new information, stuck in repeating clicking the search tab.

Addressing the preceding challenges with advanced prompt engineering is not sufficient, based on both theoretical and empirical evidence. First, empirical findings [11, 44] suggest that transformer LLMs often reduce multi-step compositional reasoning into linearized subgraph matching, rather than developing systematic problem-solving skills. This implies that, in tasks like UI testing, where multi-step reasoning is crucial, the performance of LLMs is likely to decline as task complexity increases [11]. Second, studies [23] have shown that pretrained language models inherently

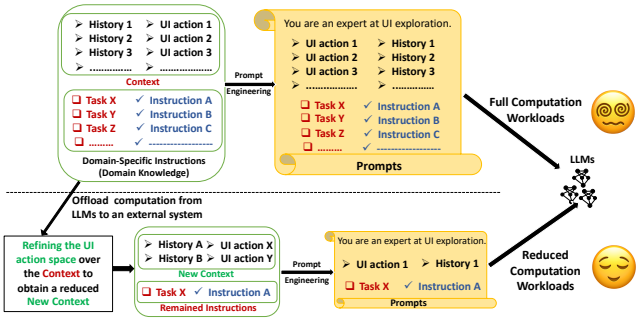

Figure 1: Computation Offloading from LLMs to external systems.

possess a statistical lower bound on hallucination rates, independent of their architecture or the quality of their training data. This makes them prone to generating inaccurate information, which can undermine their reliability in critical tasks. Last, LLMs have been empirically demonstrated to be less effective in following multiple or fine-grained instructions [20, 22, 56]. This limitation becomes even more significant in scenarios involving long contexts, where the models' ability to accurately follow instructions tends to decline [26, 32].

To fundamentally tackle the preceding theoretical and empirical limitations of prompt engineering, we introduce *computation offloading* [9, 51] to LLM-based UI exploration, inspired by the insight that some tasks and instructions in automated feature-based UI testing can be formulated as tasks computable in an external system. These tasks represent refinement strategies of the action space based on historical exploration. For example, the instruction of avoiding repetitive action selection used by Droidbot-GPT [70] can be written as a function that removes UI actions if they appear in historical actions. As shown in Figure 1, by offloading such computation tasks from LLMs to external systems, we can reduce the task complexity as well as the length of contexts, which circumvent the theoretical limitations caused by computational complexity and empirical limitations caused by multiple instructions following and long context handling.

To effectively offload the computation workloads from LLMs, in this paper, we propose GUARDIAN, the first and general runtime framework for LLM-based UI exploration. Figure 2 presents the system architecture, consisting of a domain knowledge loader, memory, and execution engine. The domain knowledge loader consists of optimizers, validators, and error handlers. The optimizers refine the action space, the validators assess the need for replanning, and the error handlers reset the AUT for replanning. This approach is closely tied to the nature of UI exploration, where instructions can be offloaded to the runtime framework and viewed as constraints on the action space. By externally executing these constraints, disallowed actions are removed from the context, ensuring that the new context adheres to the instructions, thus making the action space more concise and the set of instructions more manageable. Implementing this step requires specific programs, corresponding to the drivers in our execution engine. The memory component is crucial for both refining the action space and facilitating replanning. The memory component keeps a record of blocked actions, aiding in the

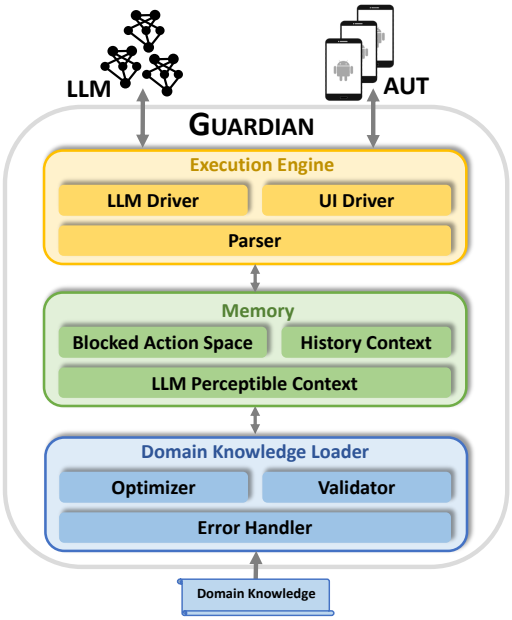

Figure 2: System Architecture of GUARDIAN.

adjustment of the action space. Additionally, the memory component maintains a historical context that includes all UI states visited and LLM perceptible context—the refined context that LLM can access based on the information available at the time of planning. This historical data is essential for effectively replanning allowing the framework to restore previous states and adjust the course of action as needed. The execution engine includes a parser, an LLM driver, and a UI driver. The LLM driver manages the interaction with the LLM, receiving its responses, while the UI driver is responsible for executing actions on the AUT and collecting the UI action space data. The parser plays a key role in processing both the LLM responses and the UI action space information, coordinating with the memory component to manage the flow of instructions and ensure they are applied correctly within the refined action space.

To evaluate the effectiveness of GUARDIAN, we instantiate GUARDIAN with ChatGPT [40] and construct a benchmark named FESTIVAL including 58 feature-based UI tests from 23 highly popular industrial Android apps [66] that are widely used in previous work [5, 10, 65–67]. We compare GUARDIAN with three state-of-the-art LLM-based approaches including Droidbot-GPT [70], ReAct [75], and Reflexion [52]. Evaluation results on FESTIVAL show that GUARDIAN successfully generates 48.3% fully correct tests and completes 64.0% of a test on average, substantially outperforming the state-of-the-art baseline approaches with respect to the two metrics with 154% and 132% relative improvement, respectively. We also conduct detailed experiments to investigate the effectiveness of the individual algorithm designs and their contributions to the effectiveness of GUARDIAN.

In summary, this paper makes the following main contributions:

- An empirical study revealing the poor planning strategies of LLM-based approaches in automated feature-based UI testing.
- A publicly available runtime framework GUARDIAN [45] for effectively offloading computations from LLMs with domain knowledge.
- Extensive evaluations demonstrating the effectiveness of GUARDIAN.

2 Preliminary Study

While LLM-based approaches have been successfully applied to sequential planning problems [52, 70, 74, 75], our evaluation in Section 4.2 shows that ReAct [75], Reflexion [52], and Droidbot-GPT [70], three state-of-the-art LLM-based approaches, all exhibit low effectiveness when applied to automated feature-based UI testing. This observation leads us to investigate the underlying causes of their low effectiveness. We propose two primary hypotheses to explain the low effectiveness observed: First, LLMs may lack the capability to understand UI content, which is essential for successful automated feature-based UI testing. Second, LLMs might comprehend UI content adequately, but the prompting strategies employed in existing LLM-based approaches may not be functioning as effectively as intended.

Consequently, we further conduct a detailed analysis to investigate the following two research questions:

- **RQ1**: How effectivecan LLMs comprehend UI content?
- **RQ2**: How effective are the prompting strategies in existing approaches in terms of achieving their intended outcomes?

2.1 RQ1: UI Comprehension Capability of LLMs

This section investigates the root cause of the limited effectiveness of LLM-based approaches in automated feature-based UI testing, specifically focusing on whether this problem stems from LLMs' inadequate comprehension of UI content. For instance, Droidbot-GPT's initial evaluation on simpler open-source apps may not adequately represent its performance with more complex, industrial apps. To address this problem, our experiments are designed to assess both the comprehension capability of LLMs for UI content and their robustness against the increasing complexity of UI elements.

Experiment Setup. Our experimental framework centers around the *UI action selection* task on the MoTIF dataset [5]. In this task, LLMs are prompted to select one and only one UI element from a list that would fulfill a given low-level instruction. The instructions, derived from natural language descriptions accompanying the MoTIF dataset, are carefully selected to exclude vague annotations like "click a button". From the 28 tasks in the MoTIF dataset, we extract 60 specific instructions for our experimental dataset. The length of candidate UI elements to select from ranges from 3 to 109, with the average length being 28.5 and the standard deviation being 20.2.

To further test the LLMs' robustness, we augment the UI element list with irrelevant elements sourced from different mobile applications, thereby creating a more challenging and noisy environment for the LLMs to navigate. Suppose the length of candidate UI elements to select from is n. We add 50%, 100%, and 200% noisy UI elements. In other words, after adding the noisy UI elements under the three conditions, the length of candidate UI elements to select from becomes $1.5n$, $2n$, $3n$, respectively.

Study results of UI comprehension capability. Table 1 shows the UI action selection accuracy of different models. In our 60-task

Table 1: Effectiveness of UI Content Comprehension.

Model	Original	+50% Noise	+100%	+200%
Seq2Act	80.0%	-	-	-
GPT-3.5	96.7%	96.7%	96.7%	93.3%
GPT-4	98.3%	96.7%	96.7%	96.7%

assessment, GPT-3.5 achieved an accuracy of 96.7%, while GPT-4 achieved an impressive accuracy of 98.3%, 1.23 times the performance of the leading non-LLM-based approach Seq2Act, which is fine-tuned on the MoTIF dataset. Our inspection of the only failed case of GPT-4 further confirms the high effectiveness of LLMs for UI comprehension. The ground-truth UI element in the failed task is a searchField given by the MoTIF dataset. GPT-4 selected a "wrong" UI element, the search icon, different from the ground-truth UI element. However, clicking on either of the two UI elements on the app can achieve the instruction "open the search field icon". Consequently, we suspect that the failed case is not due to the limited effectiveness of LLMs, but the limitation of the MoTIF dataset.

In terms of robustness against noisy UI elements and scale of UI element candidates, LLMs showed a relatively small decrease in performance, dropping from 96.7% to 93.3% accuracy even with the addition of 200% noisy elements. This slight decline, considering the potential semantic relevance of the added noisy elements, underscores the LLMs' resilience to increased UI content complexity.

Answer to RQ1: LLMs demonstrate expert-level comprehension of UI content and maintain robust performance, indicating that UI comprehension ability is not the limiting factor in the effectiveness of current LLM-based approaches.

2.2 RQ2: Unintended Outcomes against Prompting Strategies

Acknowledging the effectiveness of LLMs in understanding UI content, our next focus shifts to examining the effectiveness of complex prompting strategies. Specifically, we aim to assess whether these strategies are accurately followed by LLMs and if they indeed yield the intended outcomes. This investigation is conducted through a combination of quantitative and qualitative analyses to provide a comprehensive understanding of the efficacy of prompting strategies in the context of automated feature-based UI testing.

Experiment Setup. The experiment setup for this research question is the same as the setup in Section 4.1.

Quantitative analysis of instruction following. Our analysis begins with Droidbot-GPT, designed to follow a specific instruction: the LLM should not select any UI actions that have already been chosen in previous trials. Droidbot-GPT presents the LLM with a list of previously selected UI actions and instructs it to avoid them. Compliance with this instruction is assessed by tracking the number of times each UI element is selected; any repeated selection of a UI element is considered a violation.

Unfortunately, as depicted in Figure 3, violations of this instruction are frequent. Out of 449 different UI elements, 288 UI action selections (approximately 64% of all selections) adhere to the instruction. There are numerous instances of violations: 88 UI action selections (20%) involve a UI element being selected more than twice.

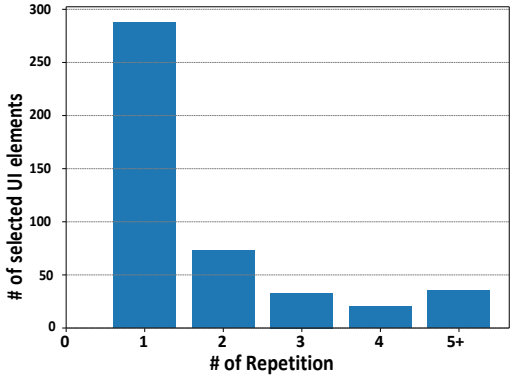

Figure 3: Violations against No Repetitive UI Action.

```
start the app AccuWeather
click view "Open navg. drawer"
click view with text "Settings"
click view with text "F, mph, in"
click view with text "F, mph, in"
click view with text "F, mph, in"
click view with text "Manage Notif"
click view with text "Locations"
go back
click view with text "Locations"
click view with text "Edit Location"
click view with text "Locations"
click view with text "Manage Notif"
click view with text "Locations"
```

Figure 4: Action Selection of Droidbot-GPT on AccuWeather.

Figure 4 presents an example of such violations. Not only does ChatGPT violate the instruction of not selecting an already selected UI element, but the violation happens continuously two times when clicking view with text "F, mph, in", and happens continually three times when clicking view with text "Locations".

Consequently, the prompting strategy of Droidbot-GPT does not actually work as intended, explaining the poor effectiveness of Droidbot-GPT on FestiVal.

Qualitative analysis of plan adaptation. Among the leading approaches, Reflexion uniquely incorporates a mechanism to analyze and adapt from previous failed trials. This ability to modify strategies based on past failures contributes significantly to its high success rate. Given that a single functionality can be implemented in various ways across different applications [31], the capacity to adjust initial plans in response to new insights gained from interaction with the Application Under Test (AUT) is vital for enhancing the success of automated feature-based UI testing.

However, the observed unintended behaviors in Droidbot-GPT raise questions about the efficacy of Reflexion's reflection mechanism. Unlike the straightforward quantification possible with Droidbot-GPT's instruction adherence, evaluating Reflexion's adaptive strategies requires a more nuanced approach. Therefore, we conducted a manual inspection of a sample of Reflexion's logs. In one notable instance with the Quizlet app, following a failed trial, Reflexion generated a specific reflection:

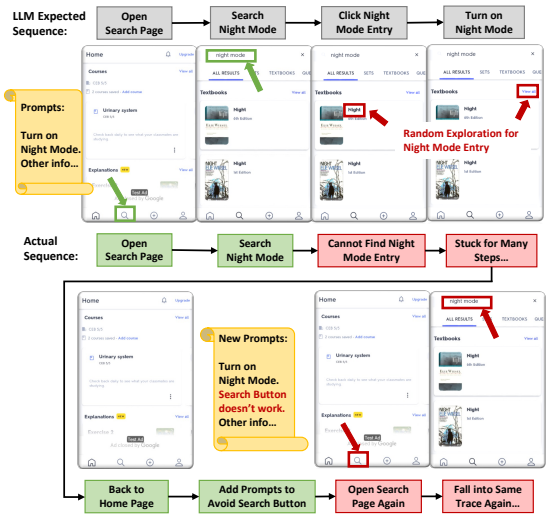

Figure 5: A motivating example of enforcing instruction following and plan adjustment with UI action space refinement.

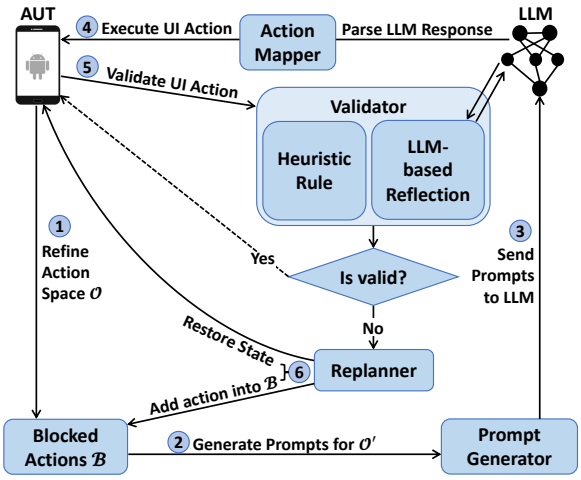

Figure 6: Workflow of GUARDIAN in one round. Only one UI action is executed in Step 4 and validated in Step 5. GUARDIAN repeats the workflow till the test objective reached.

> Reflection: In my previous attempts, I failed to realize that the necessary UI element may not be included in the given list. This led to a repetitive and ineffective approach. Next Plan: I should seek additional information or a different set of UI elements to identify the night mode toggle or switch. This could involve checking the app's settings menu, searching for user guides or tutorials, or reaching out to customer support for guidance...

Despite Reflexion's mechanism to generate alternative strategies following a failed attempt, in subsequent trials the LLM repeatedly selects the same UI action that led to the previous failure. This behavior indicates a violation of the intended reflection process and results in repeated unsuccessful attempts to complete the task. This pattern highlights a critical gap in the LLM's ability to effectively adapt and learn from past interactions, undermining the potential benefits of Reflexion's adaptive planning capability.

Answer to RQ2: Prompting-based strategies, including both explicit instructions and adaptive mechanisms, fail to consistently guide LLMs to follow the outlined instructions or adaptations. This inconsistency is the key factor contributing to the observed low effectiveness.

2.3 Motivating Example

In this section, we use a motivating example to illustrate why existing approaches fail, and how the failure can be overcome by transforming the instructions into action space refinement and replanning with restoration.

Consider the task of enabling night mode in the Quizlet app, as depicted in Figure 5. Initially, the LLM conceives a plan, hypothesizing that the app's search function can be used to enable night mode quickly. Acting on this plan, the LLM selects the search tab (the upper sequence of UI screens in Figure 5). However, it soon discovers that the app doesn't support a quick setting for night mode through the search tab. Without a replanning strategy, the LLM blindly and randomly explores the search page, stuck on the search page for many steps leading to low effectiveness.

In the subsequent trial, without advanced prompting strategy [52], the LLM is instructed not to click on search button again. Unfortunately, due to the instruction violation against these negation instructions, the LLM ignores the "search button doesn't work" instruction, goes into the search page and gets stuck on the search page again, leading to the low effectiveness.

In the first failure case, GUARDIAN can identify the planning when the search tab the LLM chosen does not match the UI state after clicking the search tab, and then restore the AUT to the home page. In the second failure case, GUARDIAN refines the action space by removing the search tab from the action space, ensuring the LLM not to enter the search page again.

3 GUARDIAN Approach

3.1 System Architecture of GUARDIAN

Figure 2 presents the system architecture of GUARDIAN. GUARDIAN takes a test objective, an AUT, and an LLM as inputs. To improve the effectiveness of automated feature-based UI testing, GUARDIAN consists of three modules: *Domain Knowledge Loader*, *Memory Module*, and *Execution Engine*.

3.1.1 Domain Knowledge Loader. As shown in our preliminary study (detailed in Section 2.2), encoding domain knowledge in prompts may not work as intended. Consequently, instead of tuning prompts for domain knowledge incorporation, GUARDIAN uses the domain knowledge loader to incorporate the domain knowledge. Specifically, the domain knowledge loader transforms the domain knowledge into algorithms of the action space optimizer, the validator, and the error handler. These algorithms refine the action space of the LLM (detailed in Section 3.2.2) or change the state of planning (detailed in Section 3.2.4) to enforce the LLM's exploration to conform to the domain knowledge, improving the effectiveness of LLM-based UI exploration. Table 2 presents the concrete domain

Algorithm 1 Main Algorithm of GUARDIAN

Require: an LLM LLM, app under test AUT

$\mathcal{O} \leftarrow Null$ ▷ action space
$\mathcal{B} \leftarrow \emptyset$ ▷ blocked action set
$\mathcal{H} \leftarrow []$ ▷ exploration history of UI states
while not achieve test objective **do**
  $\mathcal{O} \leftarrow AUT.dump\_hierarchy()$
  $S \leftarrow AUT.get\_state()$
  $\mathcal{H}.append(\mathcal{O})$
  $\mathcal{O}' \leftarrow []$ ▷ step 1
  for $a \in \mathcal{O}$ **do**
    if $(a, S) \notin \mathcal{B}$ **then**
      $\mathcal{O}'.append(a)$ ▷ step 2
  $prompt \leftarrow PromptConstruct(\mathcal{O}', H)$
  $response \leftarrow LLM.call(prompt)$ ▷ step 3
  $action \leftarrow Mapping(response, \mathcal{O}')$
  $AUT.execute(action)$ ▷ step 4
  $\mathcal{O} \leftarrow AUT.dump\_hieararchy$ ▷ step 5
  $ReflectPrompt \leftarrow (prompt, action, \mathcal{O})$
  $LMCheck \leftarrow LLM.call(ReflectPrompt)$ ▷ LLM-based reflection, return True if considered invalidated
  if $\mathcal{O} \in \mathcal{H}$ **then**
    $HeuristicCheck \leftarrow True$ ▷ action leads to a loop or unresponsive UI state.
  else
    $HeuristicCheck \leftarrow False$
  if $HeuristicCheck \vee LMCheck$ **then**
    $\mathcal{B}.add((action, S))$ ▷ step 6
    $AUT.restore$ ▷ step 6
  else
    $\mathcal{B}.add(action)$ ▷ block repeating actions

knowledge borrowed from existing work [10, 52, 67, 70] and used for the current implementation of GUARDIAN. First, Time Machine [10] and Vet [67] reported that an effective strategy of automated UI testing should avoid exploration loops and tarpits, where automated UI testing tools repetitively explore a small fraction (of the app functionality) that is already visited before. Based on these insights, we refine the action space by removing UI actions (from the action space) that lead to exploration tarpits (detailed in Section 3.2.2) and replan when a loop or an exploration tarpit is encountered (detailed in Section 3.2.4). Second, in the design of LLM agents, Droidbot-GPT [70] proposed to avoid selecting an already selected UI action, and Reflexion [52] proposed to use the self-reflection mechanism of LLMs to re-generate planning. Based on these insights, we refine the action space to avoid selecting repeated UI actions (detailed in Section 3.2.2) and replan when the LLM determines that the current plan is invalidated by subsequent exploration feedback (detailed in Section 3.2.4).

3.1.2 Memory Module. The memory module stores the history context, blocked actions determined by the domain knowledge, and the LLM perceptible context which is the working memory of the LLM. The history context stores all the UI contexts encountered during the exploration on the AUT. Each UI context contains an Activity name, a UI hierarchy, the planned UI action by the LLM on the UI hierarchy, and the corresponding UI transition after executing the LLM's planned UI action on the AUT. The blocked action space, i.e., $\mathcal{B}$, stores the actions that are blocked according to the domain knowledge (detailed in Sections 3.2.2 and 3.2.4). Each item in $\mathcal{B}$ is a tuple (UI action x, UI state s). Each UI action is uniquely identified using its textual attributes obtained from the XML file of the UI hierarchy, and each UI state is identified using the UI layout ignoring the text field (used by previous work [55, 67]). On UI state s, Guardian blocks UI action x if $(x, s) \in \mathcal{B}$.

Table 2: Domain Knowledge Borrowed from Existing Work.

Knowledge Source	Domain Knowledge
Automated UI Testing	
Time Machine [10]	Avoiding Exploration Loops
Vet [67]	Avoiding Exploration Tarpits
LLM Agents	
Reflexion [52]	Alternative Planning after Failure
Droidbot-GPT [70]	Avoiding Action Repetition

Figure 7: Running Example of GUARDIAN.

3.1.3 Execution Engine. The execution engine proxies the interaction between the LLM and the AUT. On the LLM side, the execution engine provides the LLM with prompts describing instructions and the refined action space, receives the LLM's output, and parses the LLM's output into an executable UI action. On the AUT side, the execution driver executes UI actions on the AUT, dumps the UI hierarchy file representing the screen of the AUT, and parses the UI hierarchy file to obtain the current action space on the AUT.

3.2 Workflow of GUARDIAN

3.2.1 Workflow Overview. Figure 6 presents GUARDIAN's one-round workflow of assisting the LLM in generating one UI action and Algorithm 1 illustrates the details of each step in the workflow. The workflow is repeated until the test objective is reached.

Refining action space. In each round during the automated feature-based UI testing, GUARDIAN first enumerates a list of available UI actions present in the current AUT interface to formulate the action space O (step 1). GUARDIAN then refines the action space by removing the blocked actions on the current UI state according to $\mathcal{B}$, yielding a refined action space O' (step 2, elaborated in Section 3.2.2).

Prompting the LLM with refined action space. With the refined action space O', GUARDIAN generates prompts and sends the prompt to the LLM to obtain the LLM's response (step 3). GUARDIAN parses the LLM's response into an executable UI action, and executes the UI action on the AUT (step 4).

Replanning via state restoration. After executing the UI action, GUARDIAN obtains the UI state on the AUT to validate whether the UI action yields expected outcomes (step 5). If the UI action is validated, GUARDIAN enters the next iteration. Otherwise, the UI action is invalidated and GUARDIAN deliberately replans by restoring the AUT state and adding the UI action along with the corresponding UI state to the blocked action set $\mathcal{B}$ (step 6), detailed in Section 3.2.4). GUARDIAN iterates the workflow shown in Figure 6 until the test objective is reached.

3.2.2 Refining the Action Space. In each iteration within automated feature-based UI testing, a UI action is chosen from a set denoted as *UI action space*. This space includes all possible UI actions for the current iteration. As shown in our preliminary study (Section 2.1), LLMs are experts at selecting a UI action from the UI action space. Consequently, GUARDIAN converts UI testing specific instructions to the tasks that the LLM is an expert at, i.e., selecting a UI action from an action space. In particular, GUARDIAN maintains a blocked UI action set $\mathcal{B}$, and refines the action space O according to $\mathcal{B}$ (Lines 9-11 in Algorithm 1). GUARDIAN uses the domain knowledge presented in Table 2 to maintain $\mathcal{B}$. First, inspired by Droidbot-GPT [70], we block a UI action if it has already been chosen in the same UI state in previous iterations (Line 27 in Algorithm 1). Second, inspired by Time Machine [10] and Vet [67], we block a UI action if it leads to an exploration loop or an unresponsive UI state (i.e., a tarpit) (Lines 19-22 in Algorithm 1, detailed in Section 3.2.4). Third, inspired by Reflexion [52], any UI action invalidated by the LLM's reflection is also blocked (Lines 16-18 in Algorithm 1, detailed in Section 3.2.4). The lower part of Figure 7 shows a running example of blocking the search tab with GUARDIAN, effectively addressing the limitations of existing approaches depicted in Figure 5.

3.2.3 Prompt and Parser Design. After obtaining the safe UI action space O', GUARDIAN proceeds to articulate the refined UI action space, the test objective, the current plan, and the task instructions within a prompt, as illustrated in Figure 8.

GUARDIAN also allocates a unique identifier to each UI action within O' by sorting the UI action list and enumerating each action based on its index in the list, subsequently assigning the identifier "index-i" to the i-th UI action. These unique identifiers are crucial for extracting the selected UI action from the LLM's response with regular expressions. The description of each UI action includes its event type , resource-id, textual representation, and accessibility information (as retrieved by UIAutomator [16]), provided these attributes are available.

```
You are a UI testing expert helping me <test
    objective> on <app name>.
You have selected <validated UI action history>.
Currently we have <Action space size> UI actions:
index-0: a Button (a11y information: scroll to see
     more options) to swipe
index-1: a View (resource_id widget_form_edittext,
     text password) to text
...
Your task is to select one UI action that helps
    achieve the <test objective>. First, think
    about which UI action satisfies our need, and
    then select only one UI action by its
    identifier.
```

Figure 8: Prompt Design of UI Representation.

GUARDIAN supports four primary types of UI actions:

- **Click**: click on the center point of a given UI element.
- **Long Click**: press on the center point of the UI element for one second.
- **Swipe**: query the LLM for the direction and distance of the swipe, and then perform the swipe operation.
- **Text**: generate an additional query to the LLM for generating an appropriate string input, which is then entered into the given UI element.

3.2.4 Replanning via State Restoration. After executing the planned UI action on the AUT, GUARDIAN validates whether the previous UI action yields expected outcomes. If the UI action does not yield expected outcomes, GUARDIAN invalidates the UI action for the given test objective and replans a UI action. Inspired by the domain knowledge from Reflexion [52] and automated UI testing tools [10, 67] (listed in Table 2), GUARDIAN employs a combination of LLM feedback and heuristic rules to validate whether the outcome of a UI action is expected based on its prior planning.

LLM-based reflection (Lines 16-18 in Algorithm 1). Inspired by Reflexion [52], after executing a UI action, GUARDIAN consults the LLM to determine whether the result matches the expected outcome, based on the initial plan (including both the prompt and the LLM's response) and the post-execution UI screen. A response of "No" from the LLM indicates that the action is invalidated.

Heuristic rule check (Lines 19-22 in Algorithm 1). Inspired by Time Machine [10] and Vet [67], GUARDIAN evaluates the action's results against known exploration pitfalls including unresponsiveness [67], repetitive exploration [67], and exploration loops [10]. An action is invalidated if it either fails to alter the UI screen or leads to a previously visited screen within the same trial.

If a UI action is identified as invalidated by either LLM-based reflection or the heuristic rule check (Line 23 in Algorithm 1), GUARDIAN adds it to the block action set (Line 24 in Algorithm 1). When a UI action is invalidated, GUARDIAN automatically restores the UI state to the state before the invalidated UI action is executed (Line 25 in Algorithm 1). Actions leading to unresponsiveness require no further steps. For actions resulting in a return to a previously visited UI screen, GUARDIAN replays preceding actions up to the point just before the invalidated action. If invalidated by

LLM feedback, particularly for "click" actions, Guardian triggers a "back" action on the AUT to revert to the previous state. The upper part of Figure 7 shows a running example of invalidating the search tab and restores the AUT to the home page, effectively addressing the limitations of existing approaches depicted in Figure 5. Once the UI state is restored, Guardian initiates the next iteration, where the LLM is presented with a refined UI action space (detailed in Section 3.2.2), excluding the previously invalidated UI action.

4 Evaluation

To assess the effectiveness of Guardian and its individual algorithms, we conduct comprehensive evaluations to answer the following research questions:

- **RQ3:** How effective is Guardian compared to state-of-the-art approaches?
- **RQ4:** How effective is action space refinement in improving the effectiveness of automated feature-based UI testing?
- **RQ5:** How effective is replanning via restoration in improving the effectiveness of automated feature-based UI testing?
- **RQ6:** How effective is Guardian on "unseen data"?

4.1 Evaluation Setup

Test platform. All experiments are conducted on the official Android x64 emulators running Android 6.0 on a server with four AMD EPYC 7H12 64-Core Processors. Each emulator is allocated with 4 dedicated CPU cores, 2 GB RAM, and 2 GB internal storage. We manually write auto-login scripts for apps requiring a login to access the features used in the evaluation. Each of these scripts is executed only once before the corresponding app starts to be tested in each test run.

Benchmark. We reuse an existing benchmark MoTIF [5] and collect additional tasks on popular industrial apps [66] to construct the FestiVal (**FE**ature-ba**S**ed UI tes**TI**ng e**VAL**uation) benchmark for the study. The MoTIF dataset [5] consists of 344 vision-language navigation tasks on 125 mobile apps. The vision-language navigation tasks are similar to feature-based UI testing. We install the provided APKs on emulators and execute the ground-truth tasks on the AUT. If the APK can be installed and the execution trace is the same as the provided UI hierarchy traces, we add it to FestiVal. We obtain 28 tasks runnable on our test platform. Since the task complexity collected from the MoTIF dataset is relatively low (consisting of 2-4 UI actions per task), we collect 30 additional tasks (consisting of 4-13 UI actions per task) on popular industrial apps [48, 65–67]. Finally, we obtain 58 tasks from 23 popular mobile apps for evaluation.

Prompting-based LLM approaches. We use the following three state-of-the-art LLM approaches that are designed for or can be adapted for automated feature-based UI testing.

- **ReAct** [75] directs LLMs to produce both verbal reasoning traces and actions related to a given task in an interleaved manner, enabling the model to engage in dynamic reasoning.
- **Reflexion** [52] utilizes verbal reinforcement learning to enable agents to learn from prior failures, mirroring the iterative learning process observed in humans tackling complex tasks.
- **Droidbot-GPT** [70] is an LLM-based UI navigation approach. Taking a test objective and the current UI screen of the AUT as inputs, Droidbot-GPT iteratively selects the most proper UI action to achieve the test objective, with explicit domain knowledge outlined in the prompt.

Note that all of the state-of-the-art approaches use prompt engineering [33] to instruct the LLM to follow their designed strategies. For ReAct and Reflexion, we implement them for the automated feature-based UI testing task based on their code written for WebShop [73], a web simulation environment resembling mobile apps. We use the same prompt design for describing the UI content (detailed in Section 3) as Guardian. A major difference in our implementation of ReAct and Reflexion is that we manually block the "back" action on the front page. Otherwise the two approaches will deterministically fail to finish any task by repeatedly clicking the "back" button on the front page. For Droidbot-GPT, we use its publicly available implementation [69].

We also compare the effectiveness of non-LLM-based approaches used by the MoTIF dataset, including Seq2Seq [53], MOCA [54], and Seq2Act [27]. We simply use the released parameters of these models along with MoTIF.

Evaluation Metrics. We evaluate the effectiveness by comparing the ground-truth UI action sequence with the UI action sequence generated by an approach. We use *success rate* (SR) and *average completion proportion* (ACP), which are commonly used for evaluating the effectiveness of automated feature-based UI testing [5, 70]. We use subsequence [71] when computing the average completion proportion and success rate for all approaches in the same way. Suppose that the ground-truth UI action sequence is $GT = [a_1, ..., a_n]$, and the generated UI action sequence is $Gen = [a_1, ..., a_m]$. We check whether GT is a subsequence [71] of Gen. If GT is a subsequence of Gen, then the task is treated as a success. Note that the design choice of using subsequence does not require the UI action to be adjacent in Gen but requires only the orders to be the same, ignoring the impact of loops and irrelevant UI actions. The $SR = \frac{\text{\# of successful tasks}}{\text{\# of all tasks}}$ measures the percentage of successful tasks. Let us denote $GT_i = [a_1, .., a_i]$ to be the prefix of GT with the first i UI actions. The average completion proportion $ACP = \max \frac{i}{n} \forall i \in [1, .., n] \wedge GT_i$ is a subsequence of Gen.

4.2 RQ3. Effectiveness of Guardian

In this section, we evaluate the overall effectiveness of Guardian for automated feature-based UI testing and analyze the root causes leading to failures in using LLMs for automated feature-based UI testing.

4.2.1 Main Results. Table 3 presents the overall effectiveness of Guardian and prompting-based state-of-the-art approaches. It demonstrates that LLM-based approaches significantly outperform their non-LLM-based counterparts in automated feature-based UI testing. It is important to highlight that the non-LLM-based methods, which are trained on the MoTIF dataset, only manage to attain success rates in the single digits. This result underlines the inherent complexity and challenges posed by automated feature-based UI testing, suggesting that traditional approaches may struggle to effectively navigate this domain.

Table 3: Overall Effectiveness of GUARDIAN.

Approaches	SR (%)	ACP (%)
Non-LLM		
Seq2Seq [53]	8.8	19.8
MOCA [54]	6.9	21.2
Seq2Act [27]	1.9	5.4
Prompting-based		
ReAct [75]	13.8	25.5
Reflexion [52]	19.0	26.5
Droidbot-GPT [70]	6.9	27.6
Runtime-system-guarded		
GUARDIAN (Ours)	**48.3 (+154%)**	**64.0 (+132%)**

Table 4: Cause Analysis of GUARDIAN's Failures.

Cause	# of Failed Cases
Vision Modality Input	17 (56.7%)
Screen-level Understanding	8 (26.7%)
Element-level Understanding	5 (16.7%)

Notably, Reflexion stands out with the highest success rate among all the previous approaches, achieving 19.0%, while Droidbot-GPT leads in terms of average completion proportion, reaching 27.6%. Reflexion achieves the highest success rate, particularly noteworthy as it significantly surpasses ReAct, despite both employing almost identical prompting strategies. The key differentiator is Reflexion's additional reflection mechanism that is expected to adjust the planning based on previous failed trials. This phenomenon demonstrates the necessity of adapting planning strategies based on gradually enriched information obtained during the exploration of the AUT.

Given its deliberate and reliable replanning achieved by the runtime system instead of expecting the prompts to work for the LLM, GUARDIAN achieves 48.3% success rate, outperforming the best state-of-the-art approach Reflexion [52] with 154% relative improvement, As for the average completion proportion, GUARDIAN achieves 64.0%, outperforming the best state-of-the-art approach Droidbot-GPT [70] with 132% relative improvement. Given the safe UI action space construction algorithm, GUARDIAN reliably avoids selecting repeated UI actions, and substantially reduces the 44% repeated UI actions by Droidbot-GPT (as shown in the preliminary study). The substantial improvement demonstrates the effectiveness of GUARDIAN, the runtime-system approach to reliably improving LLM-based automated feature-based UI testing instead of extensively optimizing the prompt design.

4.2.2 Investigation of Failure Cases of GUARDIAN. To investigate the root causes of failures of GUARDIAN, we manually inspect the 30 failed tasks. Table 4 presents the analysis of root causes of failed tasks, categorized into three types. First, among the investigated cases, the most common reason that GUARDIAN cannot generate the desired test is that the UI hierarchy information is not enough to perform the task. When generating specific test steps such as clicking on an icon or image button without accessibility information, GUARDIAN cannot understand the intent of these UI elements and consequently fails to select them to achieve the task. If vision modality information can be integrated, GUARDIAN can be improved. Second, GUARDIAN sometimes ignores the scrollable elements on the current UI screen and decides to go back to the previous screen when all the elements on the current UI are unrelated to the feature. GUARDIAN can benefit from a better description of the semantics of scrollable UI elements. Third, GUARDIAN cannot decide which of the UI elements is the most related to the goal when several similar UI element descriptions appear. For example, in T_8, when trying to switch the translation language, two elements on the screen are related to the feature textually. However, one element is for switching the two languages for translation, which is only weakly related to the targeted feature. Although the element has a "swap" in its name, GUARDIAN still tends to mistake it for the correct element. These cases can be inherently difficult to solve since handling them requires more advanced LLMs for test step grounding. However, these potential improvement directions involve using advanced multi-modal models (for taking visual inputs and better UI content comprehension) and designing better prompting strategies, which are orthogonal to the design and purpose of GUARDIAN.

Table 5: Efficacy of Action Space Refinement.

Approaches	SR (%)	ACP (%)
Droidbot-GPT	6.9	27.6
REFINE-DROIDBOT	13.8 (+100.0%)	32.1 (+16.3%)
UNREFINE-GUARDIAN	34.5 (-28.5%)	51.9 (-18.9%)
GUARDIAN	**48.3**	**64.0**

Answer to RQ3: GUARDIAN substantially outperforms prompting-based SOTA approaches with 154% success rate and 132% average completion proportion relative improvement.

4.3 RQ4. Effectiveness of Action Space Refinement

In this section, we evaluate the effectiveness as well as the generalizability of refining action space to ensure the instruction following.

We compare GUARDIAN and Droidbot-GPT, examining their performance both with and without action space refinement. The comparison involves the following baselines:

- **REFINE-DROIDBOT** augments Droidbot-GPT [70] by incorporating the safe UI action space to prevent the selection of UI actions already chosen in previous trials.
- **UNREFINE-GUARDIAN** operates without the safe UI action space, maintaining all other configurations identical to GUARDIAN.

Table 5 presents the experimental results on the ablation study on the effectiveness of action space refinement to ensure instruction following. both Droidbot-GPT and GUARDIAN substantially benefits from action space refinement. Specifically, REFINE-DROIDBOT demonstrates a remarkable improvement, outperforming Droidbot-GPT with a 100.0% increase in success rate and a 16.3% increase in average completion proportion. Notably, REFINE-DROIDBOT doubles the success rate of Droidbot-GPT, underscoring the promising generalizability of this approach. Conversely, the removal of action space refinement in UNREFINE-GUARDIAN leads to a marked

Table 6: Efficacy of Adjusting Planning.

Approaches	SR (%)	ACP (%)
Static Re-Plan		
CoT [68]	1.7	11.1
ToT [74]	17.2 (+911.8%)	35.0 (+216.2%)
Dynamic Re-Plan		
Droidbot-GPT	6.9	27.6
Replan-Droidbot	19.0 (+175.4%)	34.5 (+25.0%)
Guardian-NoReplan	32.8 (-32.1%)	48.5 (-24.2%)
Guardian	**48.3**	**64.0**

decrease in performance for Guardian, with 28.5% reduction in success rate and 18.9% reduction in average completion proportion.

Answer to RQ4: Refining action space significantly enhances instruction following, markedly boosting the effectiveness of Guardian. This strategy also proves beneficial in augmenting the existing LLM-based approaches.

4.4 RQ5: Effectiveness of Replanning

In this section, we study the effectiveness of replanning via restoration, especially with enriched information obtained from exploring the AUT. To comprehensively investigate this research question, we compare Guardian with the following baselines:

- **Guardian-NoReplan** omits the planning adjustment algorithm, maintaining all other configurations identical to Guardian.
- **Replan-Droidbot** enhances Droidbot-GPT by incorporating the replanning via restoration.
- **Chain of Thought (CoT)** [68] statically builds one UI action sequence to achieve the test objective before exploring the AUT. CoT explores the AUT to find UI actions that is most similar with UI actions in the dreamed sequence.
- **Tree of Thought (ToT)** [74] improves over CoT by statically conceiving five UI action sequences, increasing the diversity of paths to the test objective.

Note that both CoT and ToT do not create new planning with the gradually enriched information on the AUT. Instead, both of them imagine possible UI action sequences of the given test objective.

Table 6 presents the experimental results from which we have three findings. First, Replanning-equipped approaches generally outperform no-replanning approaches. ToT significantly surpasses CoT, validating the hypothesis that even diversifying the initial planning can increase effectiveness. Similarly, without the planning adjustment algorithm, Guardian's effectiveness substantially drops. Second, dynamic replanning works better than static replanning. While ToT substantially outperforms CoT, it is far from attaining similar effectiveness as approaches with dynamic adjustment. Third, Replanning generally improves effectiveness across different approaches. The improved performance of Replan-Droidbot over Droidbot-GPT, and its superiority to Reflexion [52], implying that even existing LLM-based approaches benefit from dynamic replanning.

Answer to RQ5: Replanning with restoration substantially improves the effectiveness of automated feature-based UI testing.

Table 7: Effectiveness on Unseen Data.

Approaches	SR (%)	ACP (%)
ReAct	0	9.6
Reflexion	16.7	28.3
Droidbot-GPT	8.3	21.8
Guardian	**58.3 (+249.1%)**	**66.9 (+136.4%)**

4.5 RQ6: Effectiveness on Unseen Data

Guardian's utility is demonstrated through its ability to consistently boost the LLM's performance beyond its initial configuration regardless of data contamination, which is evident in our results where Guardian + LLM consistently outperforms the original LLM.

Unseen-data collection. For further validation, we conduct additional experiments to investigate the effectiveness of Guardian on unseen data. We recruit two participants, undergraduate students majoring in computer science from a university class in software engineering. We first train the two participants for two hours about automated feature-based UI testing and the task format of FestiVal. We have two constraints on these tasks. First, performing the tasks requires login with non-trivial authentication such as filling emailed verification code. Second, the tasks should come from popular industrial apps, improving the representativeness of the evaluation. We give the participants the freedom to pick any popular industrial apps available on Google Play, and write any tasks for these apps. In the end, we obtain 12 tasks written by these two participants. The 12 tasks come from 5 highly popular apps, and the average length of these 12 tasks is 4.2.

Result analysis. Table 7 presents the effectiveness of Guardian and baseline approaches on unseen data. On the 12 tasks, Guardian achieves 58.3% success rate, outperforming the best state-of-the-art approach Reflexion with 249.1% relative improvement. As for average completion proportion, Guardian achieves 66.9% average completion proportion, outperforming the best state-of-the-art approach Reflexion with 136.4% relative improvement. Guardian achieves a better improvement of the LLM's effectiveness in the additional experiments than the evaluation results on FestiVal. These results further validate the effectiveness of Guardian in improving the LLM's performance. We manually inspect to figure out likely root causes of the failed cases and the suspected root causes are listed with Table 4. Specifically, failing to incorporate vision information remains the primary likely cause for the failed cases of Guardian, and these failed cases can be addressed with multi-modal models [41] and are left as future work.

5 Discussions

In this section, we discuss the limitations of the current Guardian's implementation and the future work to improve Guardian.

Limited evaluation of LLMs. Currently, we implement and evaluate Guardian with only one LLM namely GPT-3.5. While instruction violations of LLMs are found to be a prevalent issue [56], especially in long-context scenarios [32, 59], whether instruction violations are prevalent in automated feature-based UI testing remains as open problem. Nevertheless, since Guardian does not rely on LLM-specific designs for the domain knowledge, we expect that Guardian can be used for improving a broader scope of LLMs, left as our future work.

Costs of incorporating domain knowledge. Currently, we manually implement the domain knowledge within the GUARDIAN framework. To incorporate more domain knowledge and improve the usability of GUARDIAN, automating the process of turning domain-knowledge description into executable programs within the GUARDIAN framework is needed and is a promising direction left for future work.

Assumptions on input modality. Currently, GUARDIAN is instantiated with ChatGPT, which only accepts textual inputs. Mobile apps contain rich visual information useful for automated feature-based UI testing. How to incorporate the visual information of mobile apps can be an interesting yet challenging direction, specifically how to define visual-specific instructions, which we plan to explore in our future work.

Assumptions on Test Objectives. In the evaluation, we assume each task has a unique ground truth UI action sequence. However, in practice, the same test objective on the same app can have more than one paths to access (denoted as alternative paths [31]). We make our best effort by adding constraints in test objectives to make the paths to be unique. However, it can be interesting and challenging to generate all possible paths to achieve the same test objective, left as our future work.

6 Threats to Validity

The main external threat to the validity concerns the representativeness of the subject apps and approaches selected for our evaluation. To mitigate the impact of the bias introduced by app selection, we use highly popular industrial apps widely used by related work to make the constructed benchmark more practical compared to only using the tasks from MoTIF. To mitigate the impact of the bias introduced by baseline selection, we choose state-of-the-art approaches.

The threats to internal validity are instrumentation effects that can bias our results, including faults in our implementation of GUARDIAN and parameter selection of GUARDIAN. Faults in our implementation of GUARDIAN may affect the effectiveness of GUARDIAN. To reduce these threats, all authors carefully test and validate GUARDIAN on the Quizlet app to assure the behavior of GUARDIAN is as expected, and we set the temperature of LLM backbone to a low level to reduce the randomness of LLMs.

7 Related Work

Automated UI testing. Automated UI testing has been a hot research topic [1, 3, 6, 10, 15, 17, 19, 28, 36–38, 43, 49, 55, 72, 76] as well as an industry practice [38, 47, 77], falling into four main categories: (1) Some tools generate test inputs randomly and/or apply evolutionary algorithms upon these test inputs [10, 15, 36, 38, 76]. (2) Some other tools conduct systematic exploration [1, 3, 37]. (3) Model-based tools and their variants use a *UI transition model* to determine the current test progress and target not-yet-explored functionalities [6, 17, 19, 28, 55]. (4) Machine-learning-based tools [25, 29, 43, 48, 49] adopt deep learning or reinforcement learning techniques to guide the exploration of the AUT. We gain domain knowledge from existing automated UI testing tools [10, 67] for implementing GUARDIAN.

Sequential planning with LLMs. LLM-based sequential planning [7, 18, 42, 57, 63, 75] is an emerging field, having wide applications such as robotics [2, 39], video games [35], and virtual reality [58]. It has been demonstrated that LLMs are powerful for planning problems [18, 57, 63, 75]. ReAct [75] interleaves reasoning and acting to enhance the LLM planning. Reflexion [52] deliberately reflects on its previous failed trials. MemGPT [42] uses virtual memory to address the long context issue.

Large language models for software testing. Large language models [4] have gained significant attention and popularity in various areas [24, 60] including software engineering [12, 21] and testing [64]. Most existing work in software testing using LLMs focuses on embedding domain knowledge in prompts to adapt LLMs for specific tasks. Liu et al.[34] employ LLMs for textual input generation by transforming the task into a blank-filling problem. TitanFuzz [8] prompts LLMs to generate valid API sequences and parameters to fuzz deep-learning libraries. ADBGPT [13] uses prompt engineering to adapt LLMs for reproducing Android bug reports, different from the exploratory nature of automated feature-based UI testing. Our work in this paper focuses on improving the effectiveness of LLMs with a runtime-system approach. Complementing the existing work, GUARDIAN is a runtime framework external to an LLM and flexibly incorporates domain knowledge, providing a starting point to use external systems to assist an LLM with software engineering tasks.

8 Conclusion

Feature-based UI testing has been extensively adopted in industrial practices, but automated feature-based UI testing remains an open challenge. Despite the success and trend of using large language models (LLMs) for resembling planning problems, we have found two major challenges of existing prompting-based approaches: LLMs' low ability to follow task-specific instructions, and to replan based on enriched information.

To address the preceding challenges, in this paper, we have proposed GUARDIAN, a runtime framework with two key designs. First, GUARDIAN refines the action space with domain-specific knowledge to ensure instruction following. Second, GUARDIAN replans via restoration to promptly adjust the exploration on the AUT. We have constructed a benchmark named FESTIVAL containing 58 unique tasks. Evaluation results on FESTIVAL have shown that GUARDIAN can achieve 48.3% success rate and 64.0% completion rate, outperforming state-of-the-art approaches with 154% and 132% relative improvement with respect to the two metrics, respectively. Further experiments have confirmed the effectiveness of individual algorithm designs within GUARDIAN.

Acknowledgments

Tao Xie is the corresponding author. This work was partially supported by NSFC under Grant No. 62161146003, Grant No. 623B2006, Grant No.2021YFF1201103, NSF grant CCF-2146443, and the Tencent Foundation/XPLORER PRIZE.

References

[1] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone Apps. In *FSE*.

[2] Peter Anderson, Qi Wu, Damien Teney, Jake Bruce, Mark Johnson, Niko Sünderhauf, Ian Reid, Stephen Gould, and Anton van den Hengel. 2018. Vision-and-language navigation: Interpreting visually-grounded navigation instructions in real environments. In *CVPR*.

[3] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of Android Apps. In *OOPSLA*.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *NIPS* 33 (2020), 1877–1901.

[5] Andrea Burns, Deniz Arsan, Sanjna Agrawal, Ranjitha Kumar, Kate Saenko, and Bryan A Plummer. 2022. A dataset for interactive vision-language navigation with unknown command feasibility. In *ECCV*. Springer, 312–328.

[6] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI testing of Android apps with minimal restart and approximate learning. In *OOPSLA*.

[7] Abhishek Das, Samyak Datta, Georgia Gkioxari, Stefan Lee, Devi Parikh, and Dhruv Batra. 2018. Embodied question answering. In *CVPR*. 1–10.

[8] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. arXiv:2212.14834 [cs.SE]

[9] Hoang T Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. 2013. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless communications and mobile computing* 13, 18 (2013), 1587–1611.

[10] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of Android Apps. In *ICSE*.

[11] Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Sean Welleck, Peter West, Chandra Bhagavatula, Ronan Le Bras, et al. 2024. Faith and fate: Limits of transformers on compositionality. *NIPS* (2024).

[12] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533* (2023).

[13] Sidong Feng and Chunyang Chen. 2024. Prompting is all you need: Automated Android bug replay with large language models. In *ICSE*. 1–13.

[14] DiFei Gao, Lei Ji, Luowei Zhou, Kevin Qinghong Lin, Joya Chen, Zihan Fan, and Mike Zheng Shou. 2023. AssistGPT: A general multi-modal assistant that can plan, execute, inspect, and learn. (2023).

[15] Google. 2021. Android Monkey. https://developer.android.com/studio/test/monkey

[16] Google. 2023. UI Automator. https://developer.android.com/training/testing/uiautomator

[17] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *ICSE*.

[18] Tanmay Gupta and Aniruddha Kembhavi. 2023. Visual programming: Compositional visual reasoning without training. In *CVPR*.

[19] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile Apps. In *MobiSys*.

[20] Qianyu He, Jie Zeng, Wenhao Huang, Lina Chen, Jin Xiao, Qianxi He, Xunzhe Zhou, Jiaqing Liang, and Yanghua Xiao. 2024. Can Large Language Models Understand Real-World Complex Instructions?. In *AAAI*.

[21] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. arXiv:2308.10620 [cs.SE]

[22] Yuxin Jiang, Yufei Wang, Xingshan Zeng, Wanjun Zhong, Liangyou Li, Fei Mi, Lifeng Shang, Xin Jiang, Qun Liu, and Wei Wang. 2023. Followbench: A multi-level fine-grained constraints following benchmark for large language models. *arXiv preprint arXiv:2310.20410* (2023).

[23] Adam Tauman Kalai and Santosh S Vempala. 2024. Calibrated language models must hallucinate. In *STOC*.

[24] Enkelejda Kasneci, Kathrin Sessler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser, Georg Groh, Stephan Günnemann, Eyke Hüllermeier, Stephan Krusche, Gitta Kutyniok, Tilman Michaeli, Claudia Nerdel, Jürgen Pfeffer, Oleksandra Poquet, Michael Sailer, Albrecht Schmidt, Tina Seidel, Matthias Stadler, Jochen Weller, Jochen Kuhn, and Gjergji Kasneci. 2023. ChatGPT for good? On opportunities and challenges of large language models for education. *Learning and Individual Differences* (2023).

[25] Yavuz Koroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. 2018. QBE: QLearning-based exploration of Android applications. In *ICST*.

[26] Tianle Li, Ge Zhang, Quy Duc Do, Xiang Yue, and Wenhu Chen. 2024. Long-context llms struggle with long in-context learning. *arXiv preprint arXiv:2404.02060* (2024).

[27] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. 2020. Mapping natural language instructions to mobile UI action sequences. In *ACL*. Association for Computational Linguistics, Online.

[28] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: A lightweight UI-guided test input generator for Android. In *ICSE-C*.

[29] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A deep learning-based approach to automated black-box Android app testing. In *ASE*.

[30] Jun-Wei Lin, Navid Salehnamadi, and Sam Malek. 2020. Test automation in open-source android apps: A large-scale empirical study. In *ASE*. 1078–1089.

[31] Jun-Wei Lin, Navid Salehnamadi, and Sam Malek. 2022. Route: Roads not taken in ui testing. *TOSEM* (2022).

[32] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *TACL* 12 (2024), 157–173.

[33] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2021. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *arXiv preprint arXiv:2107.13586* (2021).

[34] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. 2022. Fill in the Blank: Context-aware automated text input generation for mobile GUI testing. *arXiv preprint arXiv:2212.04732* (2022).

[35] Corey Lynch and Pierre Sermanet. 2021. Language conditioned imitation learning over unstructured data. *Robotics: Science and Systems* (2021). https://arxiv.org/abs/2005.07648

[36] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for Android Apps. In *ESEC/FSE*.

[37] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: Segmented evolutionary testing of Android Apps. In *FSE*.

[38] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *ISSTA*.

[39] Dipendra Misra, Andrew Bennett, Valts Blukis, Eyvind Niklasson, Max Shatkhin, and Yoav Artzi. 2018. Mapping instructions to actions in 3D environments with visual goal prediction. In *EMNLP*. Association for Computational Linguistics, Brussels, Belgium, 2667–2678. https://doi.org/10.18653/v1/D18-1287

[40] OpenAI. 2024. Introducing to ChatGPT. https://openai.com/blog/chatgpt

[41] OpenAI. 2024. Learn how to use GPT-4 to understand images. https://platform.openai.com/docs/guides/vision

[42] Charles Packer, Vivian Fang, Shishir G Patil, Kevin Lin, Sarah Wooders, and Joseph E Gonzalez. 2023. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560* (2023).

[43] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In *ISSTA*.

[44] Binghui Peng, Srini Narayanan, and Christos Papadimitriou. 2024. On limitations of the transformer architecture. *arXiv preprint arXiv:2402.08164* (2024).

[45] Dezhi Ran. 2024. Publicly available implementation of Guardian. https://github.com/PKU-ASE-RISE/Guardian

[46] D. Ran, Y. Fu, Y. He, T. Chen, X. Tang, and T. Xie. 2024. Path Toward Elderly Friendly Mobile Apps. *Computer* 57, 06 (jun 2024), 29–39. https://doi.org/10.1109/MC.2023.3322855

[47] Dezhi Ran, Zongyang Li, Chenxu Liu, Wenyu Wang, Weizhi Meng, Xionglin Wu, Hui Jin, Jing Cui, Xing Tang, and Tao Xie. 2022. Automated visual testing for mobile apps in an industrial setting. In *ICSE-SEIP*.

[48] Dezhi Ran, Hao Wang, Wenyu Wang, and Tao Xie. 2023. Badge: Prioritizing UI events with hierarchical multi-armed bandits for automated UI Testing. In *ICSE*.

[49] Andrea Romdhana, Alessio Merlo, Mariano Ceccato, and Paolo Tonella. 2022. Deep reinforcement learning for black-box testing of Android apps. *TOSEM* (2022).

[50] Gregg Rothermel, Mary Jean Harrold, Jeffery Von Ronne, and Christie Hong. 2002. Empirical studies of test-suite reduction. *STVR* (2002).

[51] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The case for VM-based cloudlets in mobile computing. *IEEE pervasive Computing* (2009).

[52] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. arXiv:2303.11366 [cs.AI]

[53] Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. 2020. Alfred: A benchmark for interpreting grounded instructions for everyday tasks. In *CVPR*. 10740–10749.

[54] Kunal Pratap Singh, Suvaansh Bhambri, Byeonghwi Kim, Roozbeh Mottaghi, and Jonghyun Choi. 2021. Factorizing perception and policy for interactive instruction following. In *ICCV*. 1888–1897.

[55] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android Apps. In *ESEC/FSE*.

[56] Jiao Sun, Yufei Tian, Wangchunshu Zhou, Nan Xu, Qian Hu, Rahul Gupta, John Frederick Wieting, Nanyun Peng, and Xuezhe Ma. 2023. Evaluating large language models on controlled generation tasks. *arXiv preprint arXiv:2310.14542* (2023).

[57] Dídac Surís, Sachit Menon, and Carl Vondrick. 2023. Vipergpt: Visual inference via python execution for reasoning. *arXiv preprint arXiv:2303.08128* (2023).

[58] Fuwen Tan, Song Feng, and Vicente Ordonez. 2019. Text2Scene: Generating compositional scenes from textual descriptions. In *CVPR*.
[59] Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. 2020. Long range arena: A benchmark for efficient transformers. *arXiv preprint arXiv:2011.04006* (2020).
[60] Arun James Thirunavukarasu, Darren Shu Jeng Ting, Kabilan Elangovan, Laura Gutierrez, Ting Fang Tan, and Daniel Shu Wei Ting. 2023. Large language models in medicine. *Nature Medicine* (2023).
[61] Suresh Thummalapenta, Saurabh Sinha, Nimit Singhania, and Satish Chandra. 2012. Automating test automation. In *ICSE*. 881–891.
[62] Markos Viggiato, Dale Paas, Chris Buzon, and Cor-Paul Bezemer. 2022. Using natural language processing techniques to improve manual test case descriptions. In *ICSE-SEIP*. 311–320.
[63] Bryan Wang, Gang Li, and Yang Li. 2023. Enabling conversational interaction with mobile ui using large language models. In *CHI*.
[64] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2023. Software testing with large language model: Survey, landscape, and vision. arXiv:2307.07221 [cs.SE]
[65] Wenyu Wang, Wing Lam, and Tao Xie. 2021. An infrastructure approach to improving effectiveness of Android UI testing tools. In *ISSTA*.
[66] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An empirical study of Android test generation tools in industrial cases. In *ASE*.
[67] Wenyu Wang, Wei Yang, Tianyin Xu, and Tao Xie. 2021. Vet: identifying and avoiding UI exploration tarpits. In *ESEC/FSE*.
[68] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903* (2022).
[69] Hao Wen, Hongming Wang, Jiaxuan Liu, and Yuanchun Li. 2023. Droidbot-GPT. https://github.com/GAIR-team/DroidBot-GPT
[70] Hao Wen, Hongming Wang, Jiaxuan Liu, and Yuanchun Li. 2023. DroidBot-GPT: GPT-powered UI automation for Android. *arXiv preprint arXiv:2304.07061* (2023).
[71] Wikipedia. 2024. Subsequence. https://en.wikipedia.org/wiki/Subsequence
[72] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A grey-box approach for automated GUI-model generation of mobile applications. In *FASE*.
[73] Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2022. Webshop: Towards scalable real-world web interaction with grounded language agents. *NIPS* 35 (2022), 20744–20757.
[74] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601* (2023).
[75] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629* (2022).
[76] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. DroidFuzzer: Fuzzing the Android Apps with intent-filter tag. In *MoMM*.
[77] Haibing Zheng, Dengfeng Li, Beihai Liang, Xia Zeng, Wujie Zheng, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2017. Automated test input generation for Android: Towards getting there in an industrial case. In *ICSE-SEIP*.

Received 2024-04-12; accepted 2024-07-03