LLM-Explorer: Towards Efficient and Affordable LLM-based Exploration for Mobile Apps

Shanhui Zhao[1,†], Hao Wen[1,†], Wenjie Du[1,2], Cheng Liang[1,3], Yunxin Liu[1,5], Xiaozhou Ye[4], Ye Ouyang[4], Yuanchun Li[1,5,6,‡]

[1] Institute for AI Industry Research (AIR), Tsinghua University [2] Hong Kong University of Science and Technology [3] Beijing University of Posts and Telecommunications [4] AsiaInfo Technologies (China), Inc [5] Shanghai Artificial Intelligence Laboratory [6] Beijing Academy of Artificial Intelligence (BAAI)

ABSTRACT

Large language models (LLMs) have opened new opportunities for automated mobile app exploration, an important and challenging problem that used to suffer from the difficulty of generating meaningful UI interactions. However, existing LLM-based exploration approaches rely heavily on LLMs to generate actions in almost every step, leading to a huge cost of token fees and computational resources. We argue that such extensive usage of LLMs is neither necessary nor effective, since many actions during exploration do not require, or may even be biased by the abilities of LLMs. Further, based on the insight that a precise and compact knowledge plays the central role for effective exploration, we introduce LLM-Explorer, a new exploration agent designed for efficiency and affordability. LLM-Explorer uses LLMs primarily for maintaining the knowledge instead of generating actions, and knowledge is used to guide action generation in a LLM-less manner. Based on a comparison with 5 strong baselines on 20 typical apps, LLM-Explorer was able to achieve the fastest and highest coverage among all automated app explorers, with over 148x lower cost than the state-of-the-art LLM-based approach.

CCS CONCEPTS

• **Human-centered computing** → **Ubiquitous and mobile computing**; • **Computing methodologies** → *Artificial intelligence*.

† Co-primary authors.

‡ Corresponding author: Yuanchun Li (liyuanchun@air.tsinghua.edu.cn).

ACM MobiCom '25, November 3–7, 2025, Hong Kong, China

ACM ISBN 979-8-4007-1129-9/25/11…$15.00
https://doi.org/10.1145/3680207.3723494

KEYWORDS

Mobile App Exploration, Large Language Models, Software Testing, LLM-based Agent

ACM Reference Format:
Shanhui Zhao[1,†], Hao Wen[1,†], Wenjie Du[1,2], Cheng Liang[1,3],, Yunxin Liu[1,5], Xiaozhou Ye[4], Ye Ouyang[4], Yuanchun Li[1,5,6,‡]. 2025. LLM-Explorer: Towards Efficient and Affordable LLM-based Exploration for Mobile Apps. In *The 31st Annual International Conference on Mobile Computing and Networking (ACM MobiCom '25), November 3–7, 2025, Hong Kong, China.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3680207.3723494

1 INTRODUCTION

Automated mobile app exploration is a long-standing research problem, with lots of important applications including app testing [7, 9, 16–19, 22, 23], malware detection [1, 3, 4, 32], and in-app data crawling [10, 11, 14]. The performance of mobile app exploration is usually measured by coverage, *i.e*. the number of activities or lines of code reached in a limited period of time. Higher activity coverage correlates with better overall system performance. Several existing approaches, including Humanoid [17], GPTDroid [21], and DroidAgent [39], have adopted this metric to evaluate their effectiveness. Recently, emerging intelligent smartphone agents [13, 15, 34] also rely on exploration to collect necessary knowledge for task automation. Since the main interface of mobile apps is the graphical user interface (GUI or UI for short), the exploration of mobile apps is also usually grounded by GUI - The exploration agent navigates between different GUI states of an app by sending different GUI actions (touch, scroll, input text, etc.), just like how human users interact with the app.

The key question in mobile app exploration is how to generate the GUI actions that can efficiently discover new functionalities in the app. The major policies include random [7, 24, 25] (randomly selecting which UI elements to interact with and how), model-based [2, 16, 31, 38] (creating a model or representation of the UI and derive test cases to systematically cover different parts of the app functions), and learning-based [12, 17, 37] (leveraging machine learning or

reinforcement learning techniques to generate test actions). Many explorers adopt a mixture of different policies. Despite lots of existing attempts, there is still much room of improvement in app exploration. Mobile apps are dynamic, featuring a wide variety of user interfaces with diverse combinations of UI elements, and unique UI actions leading to different states. This complexity makes comprehensive exploration challenging, emphasizing the need for strategic testing methods [19, 28, 30, 40]. Therefore, we aim to efficiently cover as much app functionality as possible without needing to explore every single UI state, which might be infinite. Achieving higher and faster coverage requires a deep understanding of the apps' functionalities and historical traces, which is difficult for most existing exploration agents.

Recently, pretrained foundation models, represented by large language models (LLMs), have demonstrated remarkable performance in language understanding, reasoning, and generation. Which makes it possible for LLM-based agents to understand and execute tasks in a human-like fashion. Since the app GUI can also be represented by natural language text and images, the LLM-based agents can potentially better understand the functionalities of apps and therefore improve the exploration performance. Such an idea has already been studied in prior work [30]. For example, GPTDroid [21] introduces a method to generate testing actions by directly passing the current GUI information and historic trace to LLMs. DroidAgent [39] uses multiple autonomous agents for generating unexplored tasks, observing the interface, generating actions, and judging and reflecting on task completion. These approaches have demonstrated the ability of LLM to generate more meaningful GUI actions.

However, existing LLM-based exploration approaches have led to two important issues, limited efficiency and huge cost. The two issues originate from one cause - the excessive dependence on LLMs. Existing approaches extensively query the LLMs to generate or plan steps, which is slow and costly (*e.g.* generating a GUI action typically consumes around 500 tokens and 3 seconds, and exploring an app usually takes thousands of steps). Actually, most steps during exploration do not demand much LLM-based reasoning and planning, which is analogous to human exploration of unknown places that is aimless at most times. Moreover, asking LLMs to generate each action may bias the exploration process against unusual use cases, which are also important for exploration.

To address the above problems, we propose to combine the ability of LLMs and careful interaction knowledge modeling in mobile app exploration. Our key insight is that *the cornerstone of efficient exploration is the knowledge maintenance and utilization, instead of the action generation and planning abilities.* Specifically, similar to human or robots exploring a new environment, the key to efficient app exploration is reducing repetitive meaningless actions (*i.e.* being creative), rather than planning future actions at each step (*i.e.* being foresighted). Since LLMs are trained by learning patterns from large datasets, they are usually considered to have only weak forms of creativity [8]. On the contrary, an agent can be creative during exploration if it compares the candidate actions (and action combinations) against a high-quality knowledge. Maintaining the knowledge is mainly about information summarization, where LLMs can be more helpful.

Based on the insight, LLM-Explorer handles the automatic app exploration process with two main modules: LLM-assisted Knowledge Maintenance and Knowledge-guided Exploration. The **LLM-assisted Knowledge Maintenance** module is responsible for recording and categorizing reached UIs to prevent repetitive explorations or loops. This is crucial as variations in UI states or actions can make it challenging, even for LLMs. To address this, we introduce an app knowledge that contains abstract representations of UI states, elements and actions, as well as abstract interaction graphs to track transitions between UI states. The **Knowledge-guided Exploration** module then selects the next UI action to execute based on the maintained knowledge, with LLMs being occasionally invoked to tackle particularly complex UI actions (*e.g.* text input). This approach aims to enhance exploration efficiency by reducing unnecessary LLM queries and focusing on strategic knowledge management and interaction planning.

We evaluate the effectiveness of our LLM-Explorer approach on 20 apps, in comparison with strong baselines including DroidAgent, GPTDroid, Humanoid, Droidbot, and Monkey. We also included the human exploration performance for reference. The results have demonstrated that LLM-Explorer can achieve 4%-35% higher activity coverage than the baselines within a fixed time, matching human-level exploration efficiency on some apps. Compared with other LLM-based exploration approaches (DroidAgent and GPTDroid), LLM-Explorer can reduce the LLM cost by 9x-148x times.

Our work makes the following technical contributions:

(1) We study the LLM-based mobile app exploration with specific consideration on affordability, which can potentially benefit cost-sensitive individual mobile app developers and market-scale app analyzers.
(2) We propose a new abstraction of an app's exploration knowledge, which is maintained by LLMs and can guide efficient future exploration. We believe this abstraction is also useful for a broader scope of app analysis.
(3) Through a comprehensive evaluation, we demonstrate the effectiveness of our approach over strong baselines and the potential to advance the field of mobile app exploration.

Our code is open-sourced at https://github.com/MobileLLM/LLM-Explorer.

2 BACKGROUND AND MOTIVATION

2.1 Mobile App Exploration

App exploration is also called app traversal, crawling, or fuzzing based on different usage scenarios [14, 16, 40]. The goal of automated mobile app exploration is to traverse the functions of an app by automatically interacting with it. Since the main interface of mobile apps is GUI, the output of app exploration is usually a sequence of GUI actions, including touching, scrolling, typing text, etc. Based on different purposes of app exploration, the output of exploration may also include the comprehensive report detailing the discovered bugs or issues, the performance metrics of the app, the data crawled from the app, or the screenshots or videos recording the exploration process. More efficient and sufficient exploration usually leads to better performance in downstream applications, such as more bugs detected (for app testing), more precise malware classification (for malware detection), and more accurate task automation (for task automation).

The major performance metric of app exploration is the coverage. Unlike software testing approaches whose performance is usually measured with code coverage (*i.e*. number of reached source code lines), mobile app explorers usually deal with apps without source code available. The fundamental building block of an Android application is called activity, which represents a function component that provides a UI screen for users to interact with. For instance, an email app has different activities for different functions including the inbox, reading email, editing email, and settings. Therefore, measuring the effectiveness of an automated app explorer is often based on the activity coverage, *i.e*. how many unique activities can be reached within a fixed time.

Besides, implementing a mobile app explorer involves several key concepts about the GUI: 1) **UI state** refers to the current state of the mobile app visible through the user interface, which can interact with a user or an automatic explorer. 2) **UI element** is a visual component that users can interact with in a UI screen, including buttons, checkboxes, input boxes, etc. 3) **UI action** encompasses the interactions (such as clicks, text inputs, swipes) performed by users or automatic explorer to on UI elements to navigate or manipulate the app's functionality. The job of an explorer is to send appropriate UI actions to navigate between different UI states.

2.2 Difficulties of App Exploration

Despite the numerous efforts in automatic app exploration, it faces fundamental challenges that hinder its further improvement. Existing app explorers can hardly match the efficiency and effectiveness of human users in traversing the app functions. The main difficulties include:

1) Dynamic UI states. Many apps can dynamically change content and layouts based on user data, interactions, or even preferences. For example, a contact list's UI may constantly evolve due to additions or modifications to the contacts, creating potentially infinite states. This variability can trap automatic explorers in endless loops if they fail to recognize these changes as the same state.

2) Large action space. Some apps include a large number of interactive UI elements, a lot of which are similar or repetitive, (*e.g*. time selectors, date pickers, checkbox list). This complexity makes comprehensive app exploration difficult, necessitating strategic testing plans to manage the extensive variety of options effectively.

3) Non-deterministic behaviors. Mobile apps may exhibit unpredictable behaviors due to various factors, such as network latency, server-side processing, or interactions with other apps and services. These inconsistencies can make it challenging to navigate to specific UI states, hindering the explorer's ability to conduct thorough explorations.

2.3 AI and LLM for Mobile App Exploration

The aforementioned difficulties are mainly due to the lack of semantic understanding of GUI interactions, where AI techniques, especially LLMs can be helpful. Machine learning models have the potential to enhance UI exploration efficiency by detecting dynamic UI changes and reducing repetitive actions. They excel in understanding UI component dependencies and relationships, enabling them to generate context-based test cases [10, 29].

However, most existing AI-based exploration methods, including deep learning [17, 36] and reinforcement learning approaches [26, 27, 35], depend heavily on extensive training data and struggle with application generalization [6, 17]. Despite various innovations, these methods often fall short of the human ability to quickly identify interactive elements, sometimes causing inefficiency or endless loops. LLMs, with their extensive pretraining on large-scale data and alignment with human preferences, present remarkable semantic understanding and zero-shot generalization ability for unseen apps and pages. This makes using LLMs a promising solution for mimicking human users' capabilities in app exploration [21, 39].

Nevertheless, applying LLMs to mobile app exploration still presents several challenges. First, utilizing LLMs, whether deployed on the cloud or locally, incurs high inference costs. Accessing LLMs often involves considerable delays and expenses. Given that thoroughly exploring an app with high activity coverage typically needs numerous steps to explore an app (1,000+ steps at least), querying LLMs at each step leads to significant latency and costs. Second, LLMs could fall short in the creative thinking required for comprehensive app testing. They may tend to consistently select the main functions of the app based on its common sense while overlooking less apparent features [8]. This oversight can restrict

the exploration of all app activities and hinder the discovery of critical issues.

3 OUR APPROACH: LLM-EXPLORER

3.1 Overview

We propose LLM-Explorer, an automated mobile application exploration system powered by LLMs to address the aforementioned challenges. The main idea of LLM-Explorer is to maintain a high-quality app knowledge to guide the exploration process. As shown in Figure 1, LLM-Explorer comprises two main modules: the LLM-assisted Knowledge Maintenance module and the Knowledge-guided Exploration module.

The LLM-assisted Knowledge Maintenance module utilizes LLMs to simplify the raw data of the exploration process, forming abstract app knowledge. It merges states and actions with similar functions into abstract states and actions, creating an Abstract Interaction Graph for smooth and efficient app navigation. The Knowledge-guided Exploration module utilizes the insights stored in the app knowledge, strategically choosing UI actions and devising test plans with context awareness. If the chosen action is not executable in the current UI state, LLM-Explorer navigates to the corresponding UI states using the Abstract Interaction Graph for guidance.

3.2 LLM-assisted Knowledge Maintenance

The input of the LLM-assisted Knowledge Maintenance module is a raw UI exploration trace, including a list of UI test steps featuring UI states, elements, and actions. The module maintains a high-quality app knowledge composed of abstract UI states, elements, actions, and an Abstract Interaction Graph (AIG). Prior approaches [16, 17] also maintain a UI transition graph composed of raw states and actions, while we try to compress the knowledge components based on their semantic meanings. The idea is simple yet effective as it imitates how human users memorize concepts and reduce repetitive behaviors.

There are two problems to solve for maintaining the app knowledge: (i) Identifying UI states is crucial as seemingly identical UIs may represent different states, while dynamically changing UIs might belong to the same state. This distinction prevents excessive growth of app knowledge and aids in choosing efficient navigation paths. (ii) Reducing the UI action space is necessary because UIs can have many similar components, such as a calendar app's date elements for a month. Documenting every UI action increases redundancy and complicates the exploration module.

3.2.1 Knowledge Organization. LLM-Explorer builds up the exploration knowledge by summarizing the raw interaction trace. Each step of the raw trace includes the UI state, the UI element in the state, and the UI action type (`touch`, `scroll`, etc.). To properly compress the knowledge, we introduce abstract UI states, elements, actions, and interaction graph in LLM-Explorer.

An abstract UI state, symbolized as s_i^{abs}, represents a group of UI states $s_i^{(1)}, s_i^{(2)}, ..., s_i^{(j)}$ encountered during exploration that serve the same set of functions within an app. The actual states of an abstract state may differ visually with different content (*e.g.* two *Contacts* screens with different contact names), but they are expected to share similar use cases.

Abstract UI actions group together similar user interactions, often invoking the same function or API call but with different parameters. This method identifies repetitive components within an application (such as dates or contact names) that usually result in the same abstract UI state upon interaction. By categorizing these interactions under a single abstract action, it simplifies the action space and avoids potential infinite loops. An abstract UI action is defined by four key components: *Action Type*, *Actual Element*, *Exploration Flag*, and *Function*: 1) *Action Type* includes the four most common interactions: touch, long touch, scroll, and input. 2) *Actual Element* refers to specific UI elements involved in the action. These can either be a collection of elements within a single UI state that share similar functions or elements that are positioned identically across multiple actual states within an abstract state. 3) *Exploration Flag* indicates the exploration status of an action with three possible values: unexplored (not yet executed), explored (has been executed by the exploration module), and ineffective (found to be non-functional or leading to no UI changes). 4) *Function* denotes the functionality of the group of actual elements summarized by LLM.

The Abstract Interaction Graph (AIG) is a directed graph, whose nodes represent abstract UI states, and edges correspond to abstract UI actions. The source of the edge (abstract action) is the abstract UI state it is performed on, and the target is the resulting UI state after the action is performed. This model offers a more concise representation compared to the conventional UI Transition Graph (UTG) [10, 16]. By omitting redundant actions and consolidating dynamic UI states, the AIG enables a more efficient navigation of UI states.

3.2.2 Knowledge Update. LLM-Explorer updates the app knowledge after each exploration step. Each exploration step produces a tuple, representing the starting UI state (s'), the UI action taken (a'), and the ensuing UI state (s).

Firstly, the abstract states are updated by matching the new UI state with existing states. We attempted two methods to match the abstract UI states: the rule-based method and the LLM-based method. The rule-based method capitalizes on the insight that UIs with similar functions, such as $s_i^{(m)}$ and $s_i^{(n)}$, often share comparable element structures. The differences lie in some dynamic UI element properties such as text content,

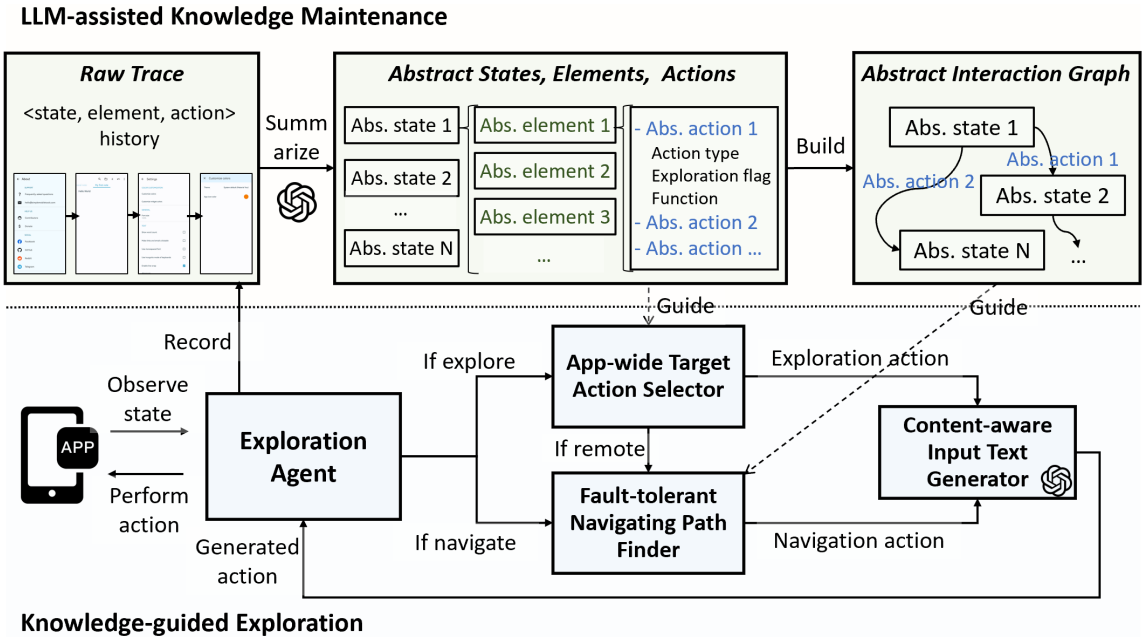

Figure 1: The workflow of LLM-Explorer.

selection status, scrolling positions, etc. Specifically, after excluding these dynamic properties, if two states $s_i^{(m)}$ and $s_i^{(n)}$ have the same set of elements, we combine them into a singular abstract UI state s_i^{abs}. On the other hand, the LLM-based method queries LLMs to determine if two UI states are functionally equivalent. It is based on the former researches that prove LLMs to be effective for summarizing UI functions and answering UI-related questions [29, 34]. We found that using the rule-based method is more effective for merging the abstract states. If the new UI state s does not belong to either of the abstract UI state in the app knowledge, LLM-Explorer recognizes it as a new abstract UI state s_i^{abs} and adds it to the app knowledge.

Secondly, LLM-Explorer updates the set of abstract UI actions. When encountering a new abstract UI state, LLM-Explorer aggregates possible UI actions that could be performed upon it into abstract UI actions $\{a_1^{abs}, a_2^{abs}...\}$, facilitated by querying LLMs. This is effective because LLMs excel at grouping repetitive UI components into generalized categories, thereby narrowing down the action space significantly. These newly incorporated abstract actions are labeled as $unexplored$. Meanwhile, for the action a that has just been taken, its status is updated based on the UI state it leads to. The abstract state resulting from this action is compared to the preceding state's abstract state. If the resulting abstract state matches the previous one, it indicates that action a didn't change the state, and the action is marked as $ineffective$. Otherwise, it is marked as $explored$.

Thirdly, LLM-Explorer updates the Abstract Interaction Graph G. If the abstract UI state s^{abs} of the current state is newly created, LLM-Explorer creates a new node for the current s^{abs} in G. Subsequently, LLM-Explorer checks whether there is a directed edge from the abstract state s'^{abs} of the last state to s^{abs} in G with the attribute of the executed action. If such an edge does not exist, it is added to the graph.

The whole process is depicted in Algorithm 1.

3.3 Knowledge-guided Exploration Policy

3.3.1 Policy Overview. Algorithm 2 illustrates the exploration process. At the beginning, LLM-Explorer analyzes the current UI state and initializes the app's knowledge with it. It then enters a loop where it continuously executes actions, and updates the knowledge. This process continues until all abstract actions in the app's knowledge have been explored, marking the end of the exploration.

At the start of each explore step, the exploration agent first chooses an abstract action from the app knowledge to be executed based on the app-wide action selection method. Next, the agent verifies if the target element of the chosen action is present in the current UI state of the app. If it is, the action is executed immediately. If not, LLM-Explorer navigates to the UI state where the target UI element is located based on the fault-tolerant navigation path finder. After executing the UI action, LLM-Explorer updates the app knowledge with the new state transition, represented as (s_i, a_i, s_{i+1}).

3.3.2 App-wide Action Selector. LLM-Explorer introduces a novel strategy that focuses on individual UI elements

Algorithm 1 Knowledge Update Process of LLM-Explorer.

Input: Existing knowledge K (including raw trace $K.T$, abstract states $K.S$, abstract actions $K.A$, abstract interaction graph $K.G$), last state s', last action a', and new state s
Output: Updated knowledge

function UPDATEKNOWLEDGE(K, s', a', s):
  Add (s', a', s) to the raw trace $K.T$
  Try classify s to existing abstract states $K.S$
  if s matches existing abstract states $K.S$ **then**
    $s^{abs} \leftarrow$ matched abstract state of s in $K.S$
  else
    $s^{abs} \leftarrow$ create new abstract state from s
    Add s^{abs} into $K.S$
  end if
  for each new action a in state s **do** ▷ LLM assisted
    if a doesn't match existing actions $K.A$ **then**
      $a^{abs} \leftarrow$ create new abstract action from a
      $a^{abs}.exploration\_flag \leftarrow$ `unexplored`
      Add a^{abs} into $K.A$
    end if
  end for
  $s'^{abs} \leftarrow$ matched abstract state of s' in $K.S$
  $a'^{abs} \leftarrow$ matched abstract action of a' in $K.A$
  $a'^{abs}.exploration\_flag \leftarrow$ `explored`
  if $s'^{abs} = s^{abs}$ **then**
    $a'^{abs}.exploration\_flag.add($`ineffective`$)$
  end if
  Update graph $K.G$ with transition $(s'^{abs}, a'^{abs}, s^{abs})$
  return Updated knowledge K
end function

Algorithm 2 Exploration Policy of LLM-Explorer.

Input: App under test, the device for testing
Output: Exploration knowledge and log

function EXPLORE-MAIN(app):
  $s_i \leftarrow$ Observe GUI state of the app ▷ Get initial state
  $K \leftarrow UpdateKnowledge(\emptyset, null, null, s_i)$ ▷ Initialize knowledge
  $nav\_steps \leftarrow \emptyset$ ▷ Initialize navigation steps
  while $K.unexplored\_abstract\_actions \neq \emptyset$ **do**
    if $nav\_steps \neq \emptyset$ **then** ▷ Try navigation
      $a_i \leftarrow$ pop next action from nav_steps
    else ▷ Try exploration
      $a_i \leftarrow SelectExploreAction(K, s_i)$
      if $a_i \notin s_i.available\_actions$ **then** ▷ Switch to navigation
        $nav\_steps \leftarrow FindNavigatePath(K, s_i, a_i)$
        $a_i \leftarrow$ pop next action from nav_steps
      end if
    end if
    if a_i is a text input action **then**
      $a_i \leftarrow GenerateInputText(s_i, a_i)$ ▷ LLM assisted
    end if
    Perform action a_i on the device
    Observe new state s_{i+1}
    $K \leftarrow UpdateKnowledge(K, s_i, a_i, s_{i+1})$
    if a_i is navigation and s_{i+1} doesn't match nav_steps **then**
      $nav\_steps \leftarrow UpdateNavigatePath(K, nav\_steps)$
    end if
  end while
end function

across the entire app. This differs significantly from the existing methods that focus on UI state granularity [16, 17, 21]. This strategy stems from the insight that exploring an app by its individual UI elements could be more effective than examining through its various UI states. This is because an application can potentially generate an infinite combination of UI states by mixing and matching different elements. However, the number of distinct UI elements, aside from some minor variations like content descriptions or colors, remains limited. As a result, the exploring algorithm could reach completion once it has examined all the UI elements. This is in contrast to UI state granularity methods, which may find themselves in continuous loops when faced with dynamic UI states. Consequently, at each exploration step, LLM-Explorer selects and executes an abstract UI action from the app knowledge, rather than selecting a UI state to investigate.

Using its high-quality app knowledge that encompasses all UI actions, LLM-Explorer is able to choose from all the UI actions encountered rather than being restricted to the actions available in the current UI state. LLM-Explorer starts by going through the abstract actions in app knowledge, pulling out actions available in the current UI state that are labeled as *unexplored*. This creates a set of UI actions, among which it randomly picks one as the next action to explore. If no unexplored actions are found in the current state, LLM-Explorer widens its search to all UI states, gathering all the unexplored elements and action types to form an action pool, from which it randomly selects the next action for exploration. Compared to the LLM agents that select actions only from the current UI state [21, 39], this strategy saves LLM queries because it does not query LLM when navigating to a specific UI state.

3.3.3 Fault-tolerant Navigating Path Finder. The exploration agent of LLM-Explorer should possess the capability to navigate to a specific UI state, especially when the required UI action is not available in the current state. This can be achieved by utilizing the abstract UI interaction graph, where the exploration agent can select the shortest path from the current to the target UI state where the UI action is available. In this graph, edges represent UI actions, allowing for sequential navigation. Should the shortest path fails, the system will attempt alternative paths for a limited number of times.

However, due to unpredictable app behavior, the target abstract UI state might sometimes be unreachable. In these cases, LLM-Explorer will restart the app and attempt to navigate from the initial state to the target UI state again. If this approach also fails, the navigation attempt will be deemed as failed, and the corresponding action will be removed from the interaction graph so that future navigations can avoid generating infeasible paths.

3.3.4 Content-aware Input Text Generator. Generating human-like input text for text boxes is complex, as it requires

understanding how to use the mobile apps. Thus, we rely on LLMs for this task. LLM-Explorer generates a structured prompt based on the name of the app under test and the current state information, and then queries LLM to obtain the input text. An example is shown in Figure 3.

3.4 Detailed Usage of LLM

In this section, we further clarify the usage of LLM in our approach and analyze the cost. LLM-Explorer involves LLMs in two parts, including knowledge organization and text input generation.

Using LLM for Knowledge Organization. The LLM is firstly utilized to categorize a group of similar UI actions, grouping them into an abstract UI action for app knowledge management, as detailed in Section 3.2.2. The structure of the prompt is illustrated in Figure 2, which is composed of four modules. It begins with an introduction including the testing app name and an explanation of the purpose of the prompt. Next, the UI is represented in the HTML syntax in line with previous studies [29]. A chain-of-thought prompting module [33] is also included to encourage LLMs' logical reasoning. The output format is required to be a json dict for straightforward parsing. LLMs are expected to generate a merging instruction for UI elements, with the aim of consolidating elements with the same functions into a single UI element. This approach simplifies the action space by combining actions on these elements, which have similar functionality, into a single abstract UI action.

Using LLM for Content-Aware Input Generation. LLM-Explorer also employs the LLM to generate text inputs for UI elements, such as input boxes, based on the current UI context. Figure 3 illustrates the prompt structure. It provides the information about the current UI state and the specific input box to be filled and requests LLMs to generate the input text. We observe that LLMs could generate human-like input data, such as usernames, phone numbers, email addresses, etc.

These two usages of LLM are uneasy for traditional AI techniques since it involves few-shot understanding of diverse UI content and free-form text generation.

According to the usages, the LLM queries of LLM-Explorer are determined by the number of unique states/actions (in knowledge organization) and the number of text input boxes (in input generation). These numbers are bounded for most of the apps, although there might be a few exceptions with very dynamic GUI. Meanwhile, these numbers are guaranteed to be smaller than the number of steps, since each unique state/action or input box maps to at least one step in the exploration. Therefore, LLM-Explorer is expected to have much slower token consumption than existing approaches that use LLM for action generation.

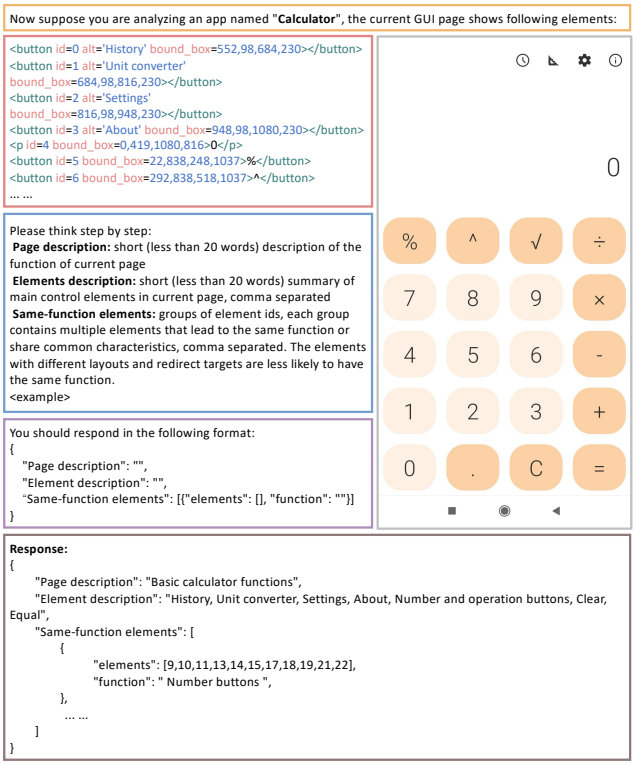

Figure 2: The prompt for knowledge organization. From top to bottom: general instructions, UI representation, chain-of-thought module, output format, and response, respectively.

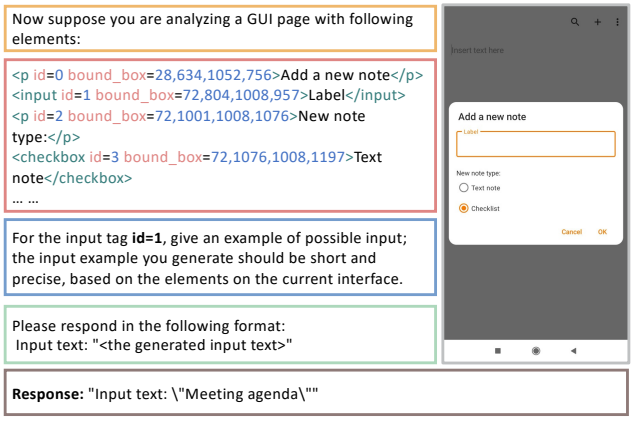

Figure 3: The prompt for content-aware input generation. From top to bottom, the prompt comprises: overall guidance, page representation, input request, response format, and response, respectively.

4 EVALUATION

We implemented LLM-Explorer with Python and Java atop DroidBot [16] and GPT-3.5 (specific version: gpt-3.5-turbo-1106), and we conducted experiments on real app exploration tasks to evaluate its performance. In total, the experiments

cost about $1,000 for GPT services, including both GPT-3.5 and GPT-4 usage, covering LLM-Explorer and all baselines.

4.1 Experimental Setup

Benchmark Apps. We mainly evaluated LLM-Explorer on 20 apps collected from F-Droid and Google Play. The apps used in our study were selected based on the following rules: 1. Apps were chosen from the most common categories in app stores, with 1-3 apps selected per category. 2. Open-source versions of apps were prioritized where available. The sources and complexity of the apps is shown in Table 1. We didn't include more apps for analysis due to the expensive cost of LLM services. However, we believe these apps are representative enough since they cover the most common app functionalities including contact management, messaging, photography, playing music, email management, and so on.

Hardware. We evaluate the end-to-end performance of LLM-Explorer on real Android devices and emulators with Android 10. The exploration agents run on a desktop with 2 NVIDIA GeForce RTX 3090 GPUs and 48GB memory.

Baselines. We chose DroidAgent [39], GPTDroid [21], Humanoid [17], DroidBot [16], and Monkey [7] as our baselines. DroidAgent and GPTDroid are LLM-based automated exploration methods. DroidAgent uses LLMs (GPT-4 and GPT-3.5) to plan tasks, observe the page, generate actions, determine task completion, etc. GPTDroid uses an LLM to choose the next action based on the current UI state. Humanoid uses a deep neural network to generate human-like UI actions during exploration. DroidBot is a generic app testing framework, in which we used the default depth-first traversal policy to explore the apps. Monkey is a popular and official command-line tool for automated app fuzz testing. Note that since the source code of GPTDroid is not fully usable at the time of our experiments, we use DroidAgent's reimplementation of GPTDroid, which uses a function-call-based action selector instead of the matching network in GPTDroid. We also use GPT-3.5 instead of GPT-3 in GPTDroid. We also compared other variations of LLM-Explorer based on different LLMs (Vicuna-13B and GPT-4 (specific version: gpt-4-1106-preview) in Section 4.4.

Reference Human Performance. Since certain app activities lack predefined navigation logic set by developers, the upper limit of activity coverage during exploration may not reach 100%. Therefore, to better understand the performance of the automated explorers, we included human performance as a reference for the upper bound of explorable activities. The human performance is collected with a user study approved by our Institutional Review Board (IRB) with the research question "How many activities of apps can human users reach?". We invited 6 experienced smartphone users to explore the 20 apps in Table 2 using our lab device. Participants were selected based on their proficiency with computers and smartphones, with priority given to those with a background in computer science. They were all from our campus, had at least five years of smartphone experience, and possessed foundational knowledge in computer science. The number of participants is determined by how many times each benchmark app can be explored. While we acknowledge the limited scale of the user study, we emphasize that six participants sufficed to explore each app twice. This provides adequate exploration coverage per app. Participants were asked to explore all the pages and elements with different functions within the apps. The exploration time for each app was required to be no less than 10 minutes and the exploration of an app ended when the participant felt there was no more to explore. During the process, our backend system automatically recorded the activities explored by the participants. In the end, each app was explored for more than twice, and we took the union of explored states as the final result. To the best of our knowledge, this is also the first time in the community to compare against reference human performance in app exploration.

Metrics. Similar to most existing work on mobile app exploration, we test our method and the baselines by comparing the progressive coverage, *i.e*. the improvement of coverage over time and the achieved final coverage. Since measuring the source code coverage is difficult for compiled APKs, we opt for monitoring the activity coverage (*i.e*. the ratio of Activities reached by the agent among all Activities defined in the app), which is also a common practice.

4.2 Exploration Effectiveness and Efficiency

To evaluate the effectiveness and efficiency of LLM-Explorer, we test 20 popular apps with LLM-Explorer and the five baselines. To ensure fair comparisons, each method was given a fixed exploration time of 2 hours.

Table 2 presents the final activity coverage achieved by each method. The results show that LLM-Explorer is able to achieve a higher activity coverage than all baseline methods. While there is still some gap to human performance, it can attain a similar or even higher level of coverage than human exploration on some simple apps.

Figure 4 and Figure 5 illustrate the progressive activity coverage of LLM-Explorer and baselines over time and over the number of steps, respectively.

As shown in Figure 4, at the beginning of exploration, LLM-Explorer, Humanoid, and DroidBot achieved similar coverage growth rates, which were significantly higher than DroidAgent, GPTDroid, and Monkey. This is because they could send actions at a faster speed and the actions could easily reach new activities in the early stage. Gradually, the growth rates slowed down as discovering new activities became harder, while LLM-Explorer and DroidAgent began

Table 1: Information about the benchmark apps

App Name	From	Activity Count	Complexity	App Name	From	Activity Count	Complexity
Activity Diary	F-Droid	11	Medium	App Launcher	F-Droid	8	Low
Calculator	F-Droid	12	Medium	Calendar	F-Droid	18	Medium
Camera	F-Droid	8	Low	Clock	F-Droid	14	Medium
Contacts	F-Droid	12	Medium	Draw	F-Droid	8	Low
File Manager	F-Droid	15	Medium	Gallery	F-Droid	23	High
Google Mail	Google Play	64	High	Keyboard	F-Droid	10	Low
Markor	F-Droid	11	Medium	Music Player	F-Droid	16	Medium
My Expenses	F-Droid	53	High	Notes	F-Droid	10	Low
Open Tracks	F-Droid	24	High	SMS Messenger	F-Droid	16	Medium
Voice Recorder	F-Droid	12	Medium	Wikipedia	F-Droid	57	High

Table 2: The activity coverage achieved by LLM-Explorer and baselines on 20 typical apps.

App	LLM-Explorer	DroidAgent	GPTDroid	Humanoid	DroidBot	Monkey	Human
Activity Diary	90.91%	**100.00%**	54.55%	90.91%	81.82%	45.45%	100.00%
App Launcher	**87.50%**	**87.50%**	75.00%	75.00%	12.50%	37.50%	87.50%
Calculator	**83.33%**	**83.33%**	50.00%	58.33%	**83.33%**	25.00%	83.33%
Calendar	**61.11%**	**61.11%**	22.22%	55.56%	33.33%	16.67%	61.11%
Camera	**87.50%**	**87.50%**	75.00%	75.00%	**87.50%**	50.00%	87.50%
Clock	**57.14%**	42.86%	28.57%	**57.14%**	28.57%	35.71%	57.14%
Contacts	**75.00%**	**75.00%**	41.67%	58.33%	50.00%	58.33%	83.33%
Draw	62.50%	**75.00%**	12.5%	62.50%	**75.00%**	50.00%	75.00%
File Manager	**73.33%**	26.67%	26.67%	40.00%	53.33%	46.67%	53.33%
Gallery	**52.17%**	47.83%	21.74%	44.00%	21.74%	13.04%	59.09%
Google Mail	**25.00%**	21.88%	4.69%	21.88%	7.81%	3.13%	28.13%
Keyboard	**70.00%**	**70.00%**	10.00%	**70.00%**	**70.00%**	**70.00%**	70.00%
Markor	**54.55%**	36.36%	36.36%	18.18%	36.36%	18.18%	54.55%
Music Player	62.50%	62.50%	12.50%	56.25%	**68.75%**	18.75%	75.00%
My Expenses	**26.42%**	**26.42%**	7.55%	15.09%	13.21%	18.87%	52.38%
Notes	**80.00%**	70.00%	10.00%	70.00%	70.00%	10.00%	80.00%
Open Tracks	**66.67%**	**66.67%**	29.17%	33.33%	37.50%	50.00%	50.00%
SMS Messenger	81.25%	**87.5%**	18.75%	43.75%	50.00%	25.00%	75.00%
Voice Recorder	**66.67%**	50.00%	25.00%	58.33%	**66.67%**	25.00%	66.67%
Wikipedia	28.07%	24.56%	10.53%	**42.11%**	26.32%	7.02%	35.71%
Average	**64.58%**	60.13%	28.62%	52.28%	48.68%	31.22%	66.74%

to have higher growth rates than the others with the help of LLMs. Eventually, LLM-Explorer was able to achieve the highest activity coverage.

In Figure 5, the coverage of LLM-Explorer gradually converged at about 1500 steps, faster than most of the baselines. During exploration, although DroidAgent's per-step actions were more effective in contributing to activity coverage improvement, DroidAgent's each step was more time-consuming due to the fact that it requested LLM extensively for reasoning and planning, with the lowest number of actions generated in the 2-hour exploration. We also observed that the activity coverage growth rate of DroidAgent decreased with the number of steps at the end of exploration, which we thought was because LLM-based action selection became less effective for discovering more detailed and unusual activities. In Table 3, we show the average single-step time of LLM-Explorer and baselines.

The advantages of LLM-Explorer over the baselines could be attributed to the following reasons: (i) LLM-Explorer merges states with identical functionality into a single abstract state, thereby eliminating redundant exploration of states with the same functionality. This strategy allows the exploration process to focus on discovering more unknown features of the app under test. Our further analysis found that

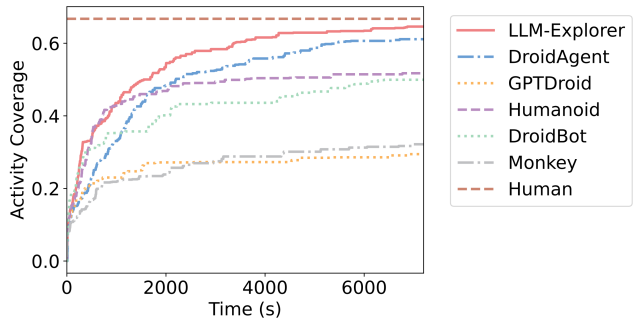

Figure 4: Progressive activity coverage of LLM-Explorer and baselines over time. The brown dotted line is the reference human performance.

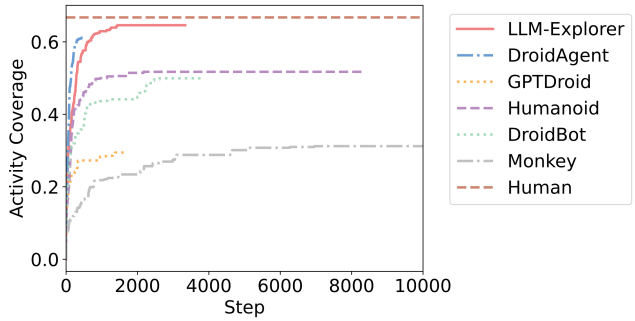

Figure 5: Progressive activity coverage of LLM-Explorer and baselines over steps within 2 hours. The brown dotted line is the reference human performance. The maximum step differs for each method due to the different per-step time. Note that Monkey produced over 20,000 steps in 2 hours of exploration, but the activity coverage nearly converged after 10,000 steps, so this figure only shows up to 10,000 steps.

Table 3: Average per-step time of LLM-Explorer and baselines. LE: LLM-Explorer, DA: DroidAgent, GD: GPTDroid, HU: Humanoid, DB: DroidBot, MO: Monkey.

Method	LE	DA	GD	HU	DB	MO
Time (seconds)	5.19	19.10	5.59	3.19	2.94	0.79

each app has 128 abstract states on average. LLM-Explorer reduced the numbers of redundant state and action visits by 4.08 and 14.71 per abstract state, which effectively reduces the exploration space. (ii) LLM-Explorer performs exploration based on abstract actions, which allows a more comprehensive coverage of all functionalities within the current app, including those that are not primary features, such as viewing the list of contributors on the "About" page of *Notes* app. (iii) LLM-Explorer's LLM-assisted knowledge maintenance strategy of merging functionally identical elements avoids

Table 4: The average number of tokens and average cost consumed by LLM-Explorer, DroidAgent and GPTDroid when exploring an app. GPT-3.5, GPT-3.5-16K, and GPT-4 represent the tokens consumed when exploring an app using the corresponding model, respectively. Cost denotes the total cost of LLM when exploring an app.

Method	GPT-3.5	GPT-3.5-16K	GPT-4	Cost($)
LLM-Explorer	96,884.2	0	0	0.11
DroidAgent	1,384,079.0	1,141,095.29	335,913.29	16.31
GPTDroid	993,939.35	0	0	1.07

LLM-Explorer repeatedly exploring duplicate elements with the same functionality. For example, within the "file import" page of the *File Manager* app, LLM-Explorer can combine the file name buttons on the page into the same abstract element. This makes it possible to explore the page by importing a file only once, thereby accomplishing the exploration of the file import functionality.

We further analyzed why LLM-Explorer failed in some cases: (i) LLM-Explorer merged different states into an abstract state based on rules, but if there was a unique element in the state, LLM-Explorer could not efficiently merge the state with the previous state. We also tried using LLM to merge states, but the result was not satisfactory, so we didn't adopt this solution in the end. (ii) In LLM-assisted knowledge maintenance, LLM may incorrectly treat elements with different functions as having the same function, resulting in the merging of elements that could reach different activities into one abstract element, causing LLM-Explorer to miss some activities when exploring. Examples are given in section 4.5.

4.3 LLM Cost of Exploration

LLM-Explorer uses the LLM in two scenarios including knowledge maintenance and input text generation. GPTDroid and DroidAgent use LLMs mainly for action generation, and also for planning, observing, etc. We compare the number of tokens and API fees consumed by different methods during the exploration process. Table 4 shows the average cost of using LLM to explore an app. As compared to DroidAgent (which achieved comparative coverage as ours), LLM-Explorer significantly reduced the average cost of exploring an app from $16.31 to $0.11, which was over 148 times less.

We also compared the number of query and the number of query tokens during exploration for different methods, as shown in Table 5. LLM-Explorer reduces the number of query by 18.48 and 9.40 times compared to DroidAgent and GPTDroid respectively. At the same time, LLM-Explorer used fewer input tokens while obtaining approximately twice the number of output tokens compared to other methods, and also had the lowest average token count per query.

Table 5: Number of queries and average number of query tokens by LLM-Explorer, DroidAgent and GPTDroid.

Method	LLM-Explorer	DroidAgent	GPTDroid
Number of queries	157.12	2903.15	1476.51
Input tokens per query	507.15	933.71	623.00
Output tokens per query	109.47	51.80	50.17

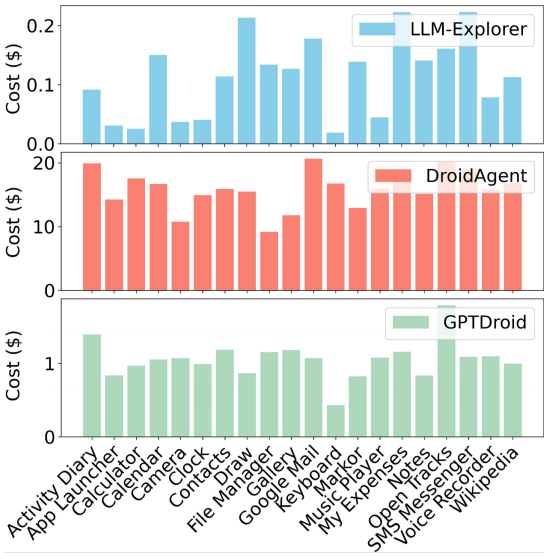

Figure 6: The cost of exploration in all apps under test by LLM-Explorer, DroidAgent, GPTDroid, respectively.

To understand the relation between the cost of the exploration process and the complexity of the apps, we analyzed the exploration cost of LLM-Explorer, DroidAgent and GPTDroid on different apps. As shown in Figure 6, the cost of LLM-Explorer is more adaptive for different apps, with lower cost on the apps with smaller number of activities (*App Launcher*, *Keybord*, etc.) while higher cost on more complex apps (*My Expenses*, *SMS Messenger*, etc.). However, DroidAgent and GPTDroid do not clearly show this adaptivity, and won't significantly reduce costs when exploring simpler apps. Meanwhile, the costs increased linearly with the step count in DroidAgent and GPTDroid, while the increase of LLM-Explorer was sublinear, as shown in Figure 7. Such adaptivity and sublinearity come from the way LLM-Explorer uses LLM, which has been discussed in Section 3.4.

4.4 Model Variation Test

Given the pivotal role of the LLM (GPT-3.5) in the exploration policy of LLM-Explorer, we evaluate its influence by replacing it with a weaker and a stronger variant. The results are shown in Figure 8.

Using a Smaller Local LLM. We first tested the performance of LLM-Explorer with Vicuna-13B [5]. We quantized

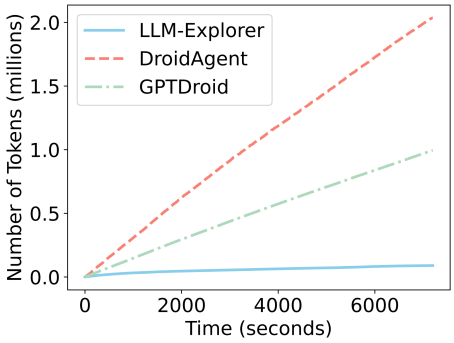

Figure 7: The progressive LLM token consumption of LLM-Explorer, DroidAgent, and GPTDroid.

it with AWQ [20] to 4bit and deployed it on a desktop computer equipped with an NVIDIA GeForce RTX 3090 GPU. As shown in Figure 8, there were slight decreases in coverage for most of the apps when using LLM-Explorer with Vicuna-13B instead of the default GPT-3.5. The reason was mainly due to the decreased quality of LLM responses. Specifically, the smaller Vicuna-13B model may return responses with incorrect formats, invalid element ids, or wrong classifications, leading to inaccurate summarization of abstract elements. Although we had queried the LLM for multiple times to increase the chances of good responses, the problem was not easily avoidable. However, the Vicuna-based LLM-Explorer still exhibited acceptable overall performance, suggesting that LLM-Explorer does not heavily rely on a larger LLM and may be further improved with local model training.

Using GPT-4. We also evaluated the performance of LLM-Explorer with GPT-4. As depicted in Figure 8, there were slight increases in coverage for most apps, attributed to the higher quality responses from GPT-4. However, despite the superior reasoning capability of GPT-4 compared to GPT-3.5, the enhancement in average coverage was not significant. Using GPT-3.5 in LLM-Explorer is good enough for most apps, except for some complicated apps (*e.g.* Wikipedia) that demand more advanced knowledge management. These findings suggest that using the most powerful LLMs may not be necessary and we choose GPT-3.5 as the default choice.

4.5 Case Studies

We further dive deeper into the succeeded and failed cases of LLM-Explorer to understand its advantages and limitations.

Cases outperforming human. LLM-Explorer achieved better coverage than human on three apps including *File Manager*, *Open Tracks* and *SMS Messenger*. We compared the activities reached by human and LLM-Explorer to understand the reasons. In Open Tracks, human participants missed the *Marker Edit* and *Marker Detail* activities. This was because these two activities require the creation of a marker first, but

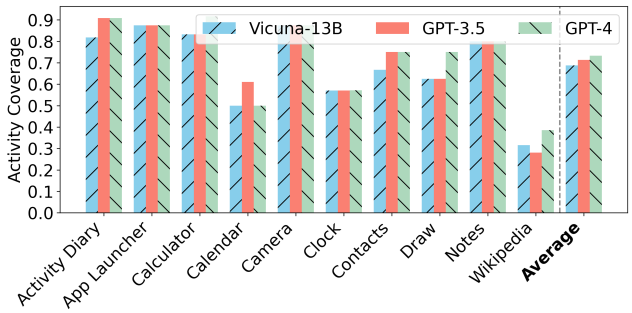

Figure 8: The activity coverage achieved by LLM-Explorer on 10 apps with different LLMs.

the creation of a marker fails when the GPS signal is weak. For other activities that LLM-Explorer explored but human users did not, it was mainly due to human participants overlooking detailed functions, such as the *Recycle Bin Conversations* activity in SMS Messenger, and the *Save As* activity in File Manager. The feasible path to reach the *Save As* activity is shown in the top half of Figure 9. To reach the activity, one needs to select a file, click the "More Options" button, click the "Share" button, and finally click the "Save as" button on the screen. The "Save as" was invisible when the human user selected a folder (instead of a file) to share, and the *Save As* activity was unreachable. Such a detailed function was uneasy to be noticed by human users, while automated explorers can solve this problem through extensive traversal.

Effects of LLM-assisted Knowledge Management. The knowledge of LLM-Explorer could help to reduce the number of redundant actions. Figure 10 shows the same-function element groups determined by the LLM in three different GUI pages. The elements in each box were considered to have the same function and their actions were stored as one abstract action. In the *Calculator* app, LLM-Explorer grouped the number entry buttons as a cluster, and all arithmetic symbol buttons as another cluster. In the *About* page, LLM-Explorer was able to group several communication platforms in the *SOCIAL* section as a cluster having the same function. In the *Import Folder* page, LLM-Explorer can group several folders together. Such groupings reduced a lot of meaningless actions and action combinations.

Effects of Content-aware Input Text Generation. LLM-Explorer also uses LLM to generate text input. Figure 11 shows LLM-Explorer's input when creating a new contact in the *Contacts* app. LLM-Explorer was able to fill in the contact's name, phone number, and address information based on the element description. After entering the user name "John Doe", it could generate email address information associated with the contact name. These meaningful inputs made LLM-Explorer easier to pass the input checks in the apps.

We also analyzed the situations where LLM-Explorer performed worse. The main causes are as follows.

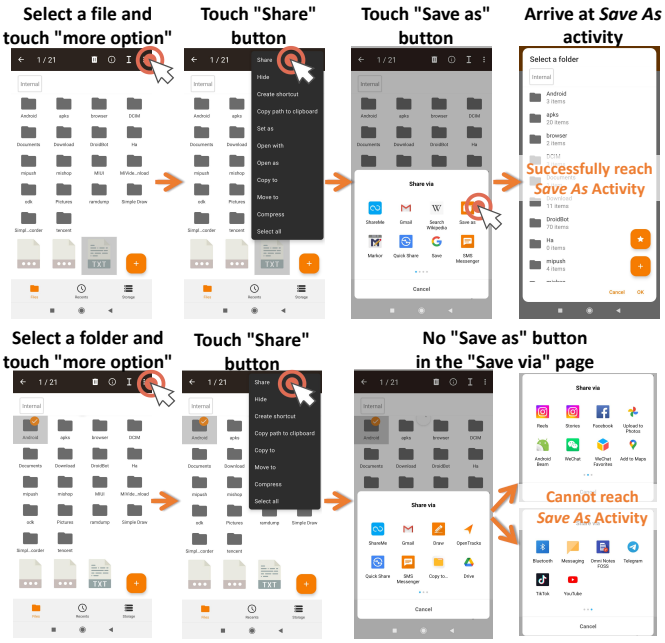

Figure 9: A case where LLM-Explorer outperformed the human performance. Top: LLM-Explorer successfully reached the *Save As* activity by selecting a file to save. Bottom: Human users only tried to select a folder, which couldn't reach the *Save As* activity.

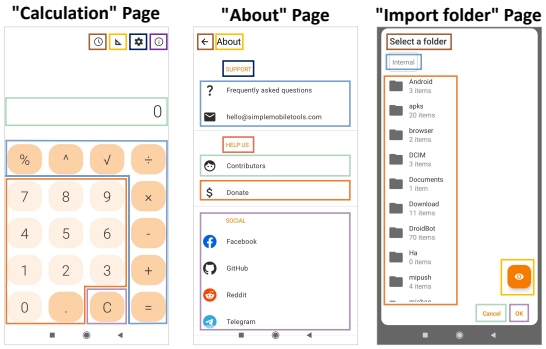

Figure 10: Illustrations of same-function elements determined by LLM-assisted knowledge maintenance module.

(i) **Failed to reach activities requiring cross-app jumps.** LLM-Explorer performed poorly for activities that need to be navigated across apps. For example, as shown in Figure 12, in the *Contacts* app, the path to reach the *Insert Or Edit Contact Activity* was to click on the *Dialpad* button and then click the "Add" button on the *Dialpad* page. However, since the *Dialpad* page used by Contacts belongs to the *Dialer* app (not the app under test), LLM-Explorer automatically returned to the *Contacts* app without further exploration. This kind of failure could potentially be fixed by allowing the agent to explore outside the target app for more steps.

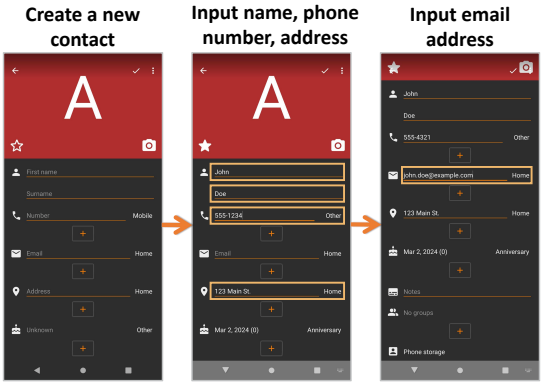

Figure 11: Examples of meaningful text inputs generated by LLM-Explorer in the *Contacts* app.

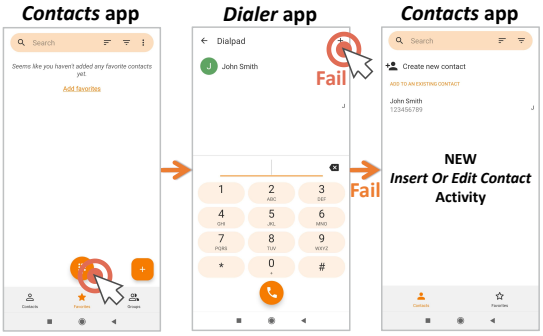

Figure 12: LLM-Explorer failed to reach the activity that required cross-app navigation.

(ii) **Errors in maintaining the knowledge.** In the LLM-assisted knowledge maintenance module, the LLM may falsely classify elements with different functions into the same group, resulting in some useful actions being grouped into the same abstract action in the knowledge and ignored in future explorations. For example, when LLM-Explorer explored the page shown in Figure 13, the knowledge maintenance module incorrectly judged the elements in the purple box as having the same function. As a result, in the app knowledge, actions of these elements were combined as one abstract action. However, in fact, they could navigate to different GUIs. Similarly, in the *About* page of the contacts app, the `touch` actions on the "Privacy policy" button and the "Third-party licenses" button were combined into the same abstract action. LLM-Explorer only touched the "Privacy policy" button and completed the exploration of this part early, missing the chance to visit the *License* activity by touching the "Third-party licenses" button. These errors could be mitigated by using better LLMs to generate more precise abstractions.

5 DISCUSSION

Cross-app navigation. LLM-Explorer automatically navigates back when detecting a transition to another app. This design ensures that the exploration remains focused on the specified app but hinders the discovery of activities that require cross-app transitions. A potential solution is to merge the abstract interaction graphs of different apps into a unified graph, allowing LLM-Explorer to search for possible cross-app return paths when transitioning to another app and enabling more effective cross-app navigation and exploration.

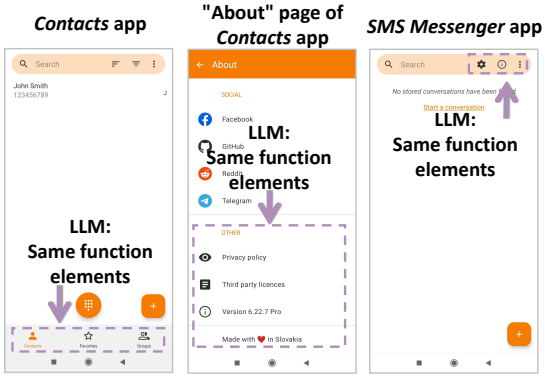

Figure 13: The LLM-assisted knowledge maintenance module of LLM-Explorer may mistakenly group actions with different functions into one abstract action.

Utilization of VLMs in app exploration. Incorporating visual modalities, such as screenshots, into prompts using state-of-the-art multimodal models like GPT-4o is an effective approach to maintaining higher-quality app knowledge and has the potential to enhance system performance. Screenshots provide crucial information that may not be accurately conveyed through text alone, such as descriptions of images and icons within the interface. However, adding visual modalities typically leads to increased token consumption, which consequently raises system overhead.

6 CONCLUSION

We present an LLM-based exploration agent for mobile apps. Experiment results show that our method can achieve effective and efficient exploration, outperforming strong baselines. By wisely choosing when and how to use LLM in the exploration process, we can significantly reduce the cost of LLMs while maintaining high exploration performance. We believe that the synergy between LLM and well-organized domain knowledge is the key to making intelligent agents and services more affordable.

ACKNOWLEDGEMENT

This work is supported by National Natural Science Foundation of China (Grant No.62272261), Tsinghua University (AIR)–AsiaInfo Technologies (China) Inc. Joint Research Center, Wuxi Research Institute of Applied Technologies, Tsinghua University (Grant No.20242001120) and Beijing Academy of Artificial Intelligence (BAAI). Wenjie Du and Cheng Liang contributed as interns at Tsinghua University.

REFERENCES

[1] Anshul Arora, Sateesh K. Peddoju, Vikas Chouhan, and Ajay Chaudhary. 2018. Hybrid Android Malware Detection by Combining Supervised and Unsupervised Learning. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (MobiCom '18)*. Association for Computing Machinery, New York, NY, USA, 798–800. https://doi.org/10.1145/3241539.3267768

[2] Nataniel P. Borges, Maria Gómez, and Andreas Zeller. 2018. Guiding App Testing with Mined Interaction Models. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft '18)*. Association for Computing Machinery, New York, NY, USA, 133–143. https://doi.org/10.1145/3197231.3197243

[3] Abhijit Bose, Xin Hu, Kang G. Shin, and Taejoon Park. 2008. Behavioral detection of malware on mobile handsets. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services (MobiSys '08)*. Association for Computing Machinery, New York, NY, USA, 225–238. https://doi.org/10.1145/1378600.1378626

[4] Haipeng Cai. 2020. Assessing and improving malware detection sustainability through app evolution studies. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 2 (2020), 1–28.

[5] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. 2023. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality. https://lmsys.org/blog/2023-03-30-vicuna/

[6] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. Association for Computing Machinery, New York, NY, USA, 845–854. https://doi.org/10.1145/3126594.3126651

[7] Android Developers. [n. d.]. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/other-testing-tools/monkey.

[8] Giorgio Franceschelli and Mirco Musolesi. 2023. On the creativity of large language models. *arXiv preprint arXiv:2304.00008* (2023).

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

[10] Forrest Huang, Gang Li, Tao Li, and Yang Li. 2023. Automatic Macro Mining from Interaction Traces at Scale. *arXiv preprint arXiv:2310.07023* (2023).

[11] Haojian Jin, Minyi Liu, Kevan Dodhia, Yuanchun Li, Gaurav Srivastava, Matthew Fredrikson, Yuvraj Agarwal, and Jason I. Hong. 2018. Why Are They Collecting My Data? Inferring the Purposes of Network Traffic in Mobile Apps. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 2, 4, Article 173 (dec 2018), 27 pages. https://doi.org/10.1145/3287051

[12] Yuanhong Lan, Yifei Lu, Zhong Li, Minxue Pan, Wenhua Yang, Tian Zhang, and Xuandong Li. 2024. Deeply Reinforcing Android GUI Testing with Deep Reinforcement Learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 71, 13 pages. https://doi.org/10.1145/3597503.3623344

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

[14] Tong Li, Yong Li, Mohammad Ashraful Hoque, Tong Xia, Sasu Tarkoma, and Pan Hui. 2022. To What Extent We Repeat Ourselves? Discovering Daily Activity Patterns Across Mobile App Usage. *IEEE Transactions on Mobile Computing* 21, 4 (2022), 1492–1507. https://doi.org/10.1109/TMC.2020.3021987

[15] Yuanchun Li, Hao Wen, Weijun Wang, et al. 2024. Personal LLM Agents: Insights and Survey about the Capability, Efficiency and Security. *arXiv preprint arXiv:2401.05459* (2024).

[16] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: A Lightweight UI-Guided Test Input Generator for Android. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17)*. IEEE Press, 23–26. https://doi.org/10.1109/ICSE-C.2017.8

[17] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A deep learning-based approach to automated black-box android app testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1070–1073.

[18] Chieh-Jan Mike Liang, Nicholas D. Lane, Niels Brouwers, et al. 2014. Caiipa: Automated Large-Scale Mobile App Testing through Contextual Fuzzing. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking (MobiCom '14)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2639108.2639131

[19] Hao Lin, Jiaxing Qiu, Hongyi Wang, Zhenhua Li, Liangyi Gong, Di Gao, Yunhao Liu, Feng Qian, Zhao Zhang, Ping Yang, and Tianyin Xu. 2023. 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 '23)*. Association for Computing Machinery, New York, NY, USA, Article 45, 17 pages. https://doi.org/10.1145/3570361.3613259

[20] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. 2023. AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration. *arXiv* (2023).

[21] Zhe Liu, Chunyang Chen, Junjie Wang, et al. 2023. Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*. IEEE/ACM.

[22] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. 2023. Fill in the Blank: Context-aware Automated Text Input Generation for Mobile GUI Testing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1355–1367. https://doi.org/10.1109/ICSE48619.2023.00119

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

[24] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 224–234. https://doi.org/10.1145/2491411.2491450

[25] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-Objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/2931037.2931054

[26] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 153–164. https://doi.org/10.1145/3395363.3397354

[27] Andrea Romdhana, Alessio Merlo, Mariano Ceccato, and Paolo Tonella. 2022. Deep reinforcement learning for black-box testing of android

apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 1–29.

[28] Konstantin Rubinov and Luciano Baresi. 2018. What Are We Missing When Testing Our Android Apps? *Computer* 51, 4 (2018), 60–68. https://doi.org/10.1109/MC.2018.2141024

[29] Bryan Wang, Gang Li, and Yang Li. 2023. Enabling Conversational Interaction with Mobile UI Using Large Language Models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 432, 17 pages. https://doi.org/10.1145/3544548.3580895

[30] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing with Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering* (2024), 1–27. https://doi.org/10.1109/TSE.2024.3368208

[31] 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 *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. Association for Computing Machinery, New York, NY, USA, 738–748. https://doi.org/10.1145/3238147.3240465

[32] Yin Wang, Ming Fan, Xicheng Zhang, Jifei Shi, Zhaoyu Qiu, Haijun Wang, and Ting Liu. 2024. LIReDroid: LLM-Enhanced Test Case Generation for Static Sensitive Behavior Replication. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware (Internetware '24)*. Association for Computing Machinery, New York, NY, USA, 81–84. https://doi.org/10.1145/3671016.3671404

[33] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[34] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2024. Autodroid: Llm-powered task automation in android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*. 543–557.

[35] Husam N Yasin, Siti Hafizah Ab Hamid, and Raja Jamilah Raja Yusof. 2021. Droidbotx: Test case generation tool for android applications using Q-learning. *Symmetry* 13, 2 (2021), 310.

[36] Faraz YazdaniBanafsheDaragh and Sam Malek. 2021. Deep GUI: Black-box GUI Input Generation with Deep Learning. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 905–916. https://doi.org/10.1109/ASE51524.2021.9678778

[37] Faraz YazdaniBanafsheDaragh and Sam Malek. 2022. Deep GUI: black-box GUI input generation with deep learning. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE '21)*. IEEE Press, 905–916. https://doi.org/10.1109/ASE51524.2021.9678778

[38] Chao-Chun Yeh, Shih-Kun Huang, and Sung-Yen Chang. 2013. A black-box based android GUI testing system. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '13)*. Association for Computing Machinery, New York, NY, USA, 529–530. https://doi.org/10.1145/2462456.2465717

[39] Juyeon Yoon, Robert Feldt, and Shin Yoo. 2023. Autonomous Large Language Model Agents Enabling Intent-Driven Mobile GUI Testing. *arXiv preprint arXiv:2311.08649 (ICST 2024)* (2023).

[40] Li Lyna Zhang, Chieh-Jan Mike Liang, Wei Zhang, and Enhong Chen. 2017. Towards A Contextual and Scalable Automated-testing Service for Mobile Apps. In *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications (HotMobile '17)*. Association for Computing Machinery, New York, NY, USA, 97–102. https://doi.org/10.1145/3032970.3032972