Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions

Zhe Liu[1,2],Chunyang Chen[3], Junjie Wang[1,2,*], Mengzhuo Chen[1,2], Boyu Wu[2,4],
Xing Che[1,2], Dandan Wang[1,2], Qing Wang[1,2,5,*]
[1]State Key Laboratory of Intelligent Game, Beijing, China
Institute of Software Chinese Academy of Sciences, Beijing, China;
[2]University of Chinese Academy of Sciences, Beijing, China; [*]Corresponding author;
[3]Technical University of Munich, Munich, Germany; [4]DiDi Global Inc;
[5]Science & Technology on Integrated Information System Laboratory
liuzhe181@mails.ucas.ac.cn,Chunyang.chen@monash.edu,junjie@iscas.ac.cn,wq@iscas.ac.cn

ABSTRACT

Automated Graphical User Interface (GUI) testing plays a crucial role in ensuring app quality, especially as mobile applications have become an integral part of our daily lives. Despite the growing popularity of learning-based techniques in automated GUI testing due to their ability to generate human-like interactions, they still suffer from several limitations, such as low testing coverage, inadequate generalization capabilities, and heavy reliance on training data. Inspired by the success of Large Language Models (LLMs) like ChatGPT in natural language understanding and question answering, we formulate the mobile GUI testing problem as a Q&A task. We propose GPTDroid, asking LLM to chat with the mobile apps by passing the GUI page information to LLM to elicit testing scripts, and executing them to keep passing the app feedback to LLM, iterating the whole process. Within this framework, we have also introduced a functionality-aware memory prompting mechanism that equips the LLM with the ability to retain testing knowledge of the whole process and conduct long-term, functionality-based reasoning to guide exploration. We evaluate it on 93 apps from Google Play and demonstrate that it outperforms the best baseline by 32% in activity coverage, and detects 31% more bugs at a faster rate. Moreover, GPTDroid identifies 53 new bugs on Google Play, of which 35 have been confirmed and fixed.

KEYWORDS

Automated GUI testing, Large language model

ACM Reference Format:
Zhe Liu[1,2],Chunyang Chen[3], Junjie Wang[1,2,*], Mengzhuo Chen[1,2], Boyu Wu[2,4],, Xing Che[1,2], Dandan Wang[1,2], Qing Wang[1,2,5,*]. 2024. Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639180

1 INTRODUCTION

In recent years, mobile apps have become an indispensable part of our daily life, with millions of apps available for download from app stores like the Google Play Store [4] and Apple App Store [3]. With the rise of app importance in our daily life, it has become increasingly critical for app developers to ensure that their apps are of high quality and perform as expected for users. To avoid time-consuming and labor-extensive manual testing, automated GUI (Graphical User Interface) testing is widely used for quality assurance of mobile apps [43, 44, 47, 77, 78, 83], i.e., dynamically exploring mobile apps by executing different actions such as scrolling and clicking to verify the app functionality.

Unfortunately, existing GUI testing tools such as probability-based or model-based ones [31, 56, 63] suffer from low testing coverage, meaning that they may miss important bugs and issues. This is because of the complex and dynamic nature of modern mobile apps [18, 22, 31, 52, 55, 56], which can have hundreds or even thousands of different screens, each with its own unique set of interactions and possible user actions and logic. In addition, test inputs generated by these methods are significantly different from real users' interaction traces [53], resulting in low testing coverage. To address these limitations, there has been a growing interest in using deep learning (DL) [34, 79] and reinforcement learning (RL) [50, 54] techniques for automated mobile GUI testing. By learning from human testers' behavior, these methods aim to generate human-like actions and interactions that can be used to test the app's GUI more thoroughly and effectively. These approaches are based on the idea that the more closely the actions performed by the testing algorithm mimic those of a human user, the more comprehensive and effective the testing will be.

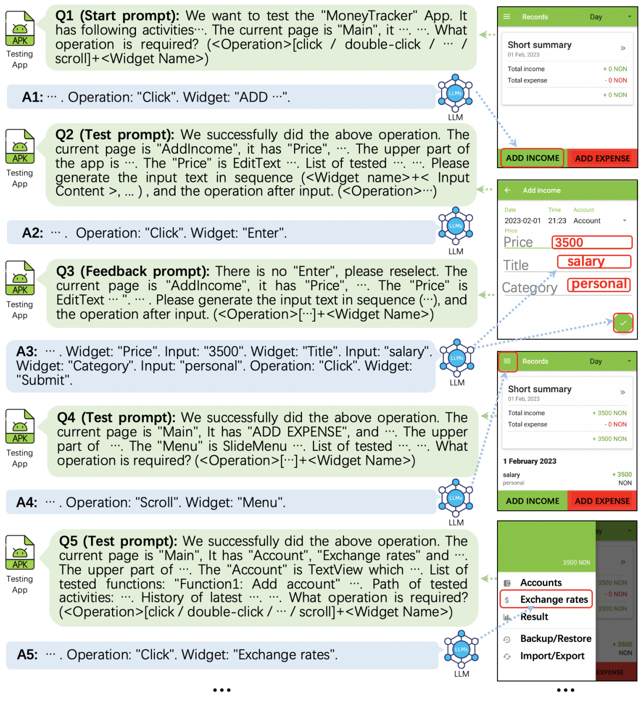
Nevertheless, there are still some limitations with these DL or RL-based GUI testing methods. First, learning algorithms require large amounts of data which is difficult to collect from real-world users' interactions. Second, learning algorithms are designed to learn and predict from training data, so they may not generalize well to new, unseen situations, as apps are constantly evolving and updating. Third, mobile apps can be non-deterministic, meaning that the outcome of an action may not be the same every time it is performed (e.g., clicking the "delete" button from a list with the last content would produce an empty list for which the delete button no longer works) which specifically makes it difficult for RL algorithms to learn and make accurate predictions. Therefore, another more effective approach to generate human-like actions is highly needed to test mobile apps thoroughly.

Large Language Models (LLMs) [17, 58, 68, 84] such as GPT-3/4 have emerged as a powerful tool for natural language understanding and question answering. Recent advances in LLM have triggered various studies examining the use of these models for software development tasks[30, 37, 73] The ChatGPT [58] (Chat Generative Pre-trained Transformer) from OpenAI, has billions of parameters and is trained on a vast dataset comprising test scripts and bug reports. Its exceptional performance across diverse domains and topics demonstrates the LLM's ability to comprehend human knowledge and interact with humans as a knowledgeable expert. Inspired by ChatGPT, we formulate the GUI testing problem as a questions & answering (Q&A) task, i.e., asking the LLM to play the role as a human tester to test the target app.

We propose GPTDroid for automated GUI testing, which asks LLM to chat with mobile apps by passing the GUI page information to LLM to elicit testing scripts and execute them to keep passing the app feedback to LLM, iterating the whole process. To convert the visual information of the app GUI into the corresponding natural language description, we first extract the semantic information of the app and GUI page by decompiling the target app and view hierarchy files, and design linguistic patterns to encode the information as the prompt of LLM. We then utilize few-shot learning by providing demonstrations with the output template to facilitate the LLM generating desired executive commands to execute the app.

Nevertheless, there are two main challenges during the interactive Q&A GUI testing process. The first is the local dilemma. Different from the LLM-based program repair or unit test generation which mainly targets a determined piece of software, GPTDroid formulates the GUI testing as a multi-turn task and the LLM faces varying GUI pages, i.e., interacting between LLM and mobile app to explore various pages of the app. During the interaction process, it is hard for the LLM to clearly and accurately remember the historical explorations, especially those that happened long before. Because of this, the LLM might only rely on the recent interactive information to make the decision, while omitting the global viewpoints, which can make the exploration fall into a local dilemma and hinder it from achieving higher coverage. The second is the low-level dilemma. Being fed with the descriptive GUI information, the LLM can easily focus more on the low-level semantics as the widgets or activities, yet less on the high-level semantics as the functionalities which is achieved with sequences of operations with the widgets/activities. However, the functional aspect of the mobile app is of high interest to testers and users, and has long been an obstacle to existing techniques.

To overcome these challenges, within GPTDroid, we develop a functionality-aware memory mechanism. It builds a testing sequence memorizer to record all the interactive testing information in terms of the explored activities and widgets. It also queries the LLM about the function-level progress of the testing during the iterative process, e.g., which function is under test, to enable the LLM to conduct the explicit reasoning by itself. And the information is then encoded into a functionality-aware memory prompt and fed into the LLM to enable the LLM in deciding the meaningful operation sequence to explore the app's functionality and conduct global exploration to cover unexplored areas.


Q1 (Start prompt): We want to test the "MoneyTracker" App. It has following activities···. The current page is "Main", it ···. ···. What operation is required? (<Operation>[click / double-click / ··· / scroll]+<Widget Name>)

A1: ··· . Operation: "Click". Widget: "ADD ···".

Q2 (Test prompt): We successfully did the above operation. The current page is "AddIncome", it has "Price", ···. The upper part of the app is ···. The "Price" is EditText ···. List of tested ···. ···. Please generate the input text in sequence (<Widget name>+< Input Content >, ...) , and the operation after input. (<Operation>···)

A2: ··· . Operation: "Click". Widget: "Enter".

Q3 (Feedback prompt): There is no "Enter", please reselect. The current page is "AddIncome", it has "Price", ···. The "Price" is EditText ··· ". ··· . Please generate the input text in sequence (···), and the operation after input. (<Operation>[···]+<Widget Name>)

A3: ··· . Widget: "Price". Input: "3500". Widget: "Title". Input: "salary". Widget: "Category". Input: "personal". Operation: "Click". Widget: "Submit".

Q4 (Test prompt): We successfully did the above operation. The current page is "Main", It has "ADD EXPENSE", and ···. The upper part of ···. The "Menu" is SlideMenu ···. List of tested ···. ···. What operation is required? (<Operation>[···]+<Widget Name>)

A4: ··· . Operation: "Scroll". Widget: "Menu".

Q5 (Test prompt): We successfully did the above operation. The current page is "Main", It has "Account", "Exchange rates" and ···. The upper part of ···. The "Account" is TextView which ···. List of tested functions: "Function1: Add account" ···. Path of tested activities: ···. History of latest ···. ···. What operation is required? (<Operation>[click / double-click / ··· / scroll]+<Widget Name>)

A5: ··· . Operation: "Click". Widget: "Exchange rates".

Figure 1: Demonstrated example of how GPTDroid works.

One example chat log can be seen in Figure 1. LLM can understand the app GUI, and provide detailed actions to navigate the app (e.g., A1-A5 at Figure 1). To compensate for its wrong prediction (A2 at Figure 1), the real-time feedback by GPTDroid guides it to regenerate the input until triggering a valid page transition. It remains clear testing logic even after a long testing trace to make complex reasoning of actions (A3, A4 at Figure 1), and it can prioritize to test important functions earlier (e.g., A5 at Figure 1). A more detailed analysis of GPTDroid's capability is in Section 5.

To evaluate the effectiveness of GPTDroid, we carry out an experiment on 93 popular Android apps in Google Play with 143 bugs. Compared with 10 common-used and state-of-the-art baselines, GPTDroid can achieve more than 32% boost in activity coverage and 20% boost in code coverage than the best baseline, resulting in 75% activity coverage and 66% code coverage. As GPTDroid can cover more activities, the method can detect 31% more bugs with a faster speed than the best baseline. Apart from the accuracy of our GPTDroid, we also evaluate the usefulness of our GPTDroid by detecting unseen crash bugs in real-world apps from Google Play. Among 223 apps, we obtain 53 crash bugs with 35 of them being confirmed and fixed by developers, while the remaining are still pending. To reveal reasons behind the promising performance of our approach, we further investigate the experiment results qualitatively and summarize 4 findings including function-aware exploration through long meaningful testing trace,function-aware prioritization, valid text input and compound action.

The contributions of this paper are as follows:

- **Vision.** The first work to formulate the automatic GUI testing problem to an interactive question & answering task to let the LLM conduct the whole app testing by understanding the GUI semantic information and automatically inferring possible operation steps.

- **Technique.** A function-aware automatic GUI testing approach `GPTDroid`[1] which designs the function-aware memory mechanism to enable the LLM to focus more on the global and functional viewpoints of the mobile app.
- **Evaluation.** Effectiveness and usefulness evaluation of the `GPTDroid` in the real-world apps with practical bugs detected (Section 3 and 4).
- **Insight.** Detailed qualitative analysis revealing the reasons why LLM can generate human-like and functionality-aware actions for app testing (Section 5).

2 APPROACH

We model the GUI testing as a Question & Answering (Q&A) problem, i.e., asking the LLM to play a role as a human tester, and enabling the interactions between the LLM and the app under testing. To realize this, we propose `GPTDroid`, as demonstrated in Figure 2, with nested loops.

In the outer loop, it extracts the GUI context information of the current GUI page, encodes them into prompt questions for LLM, decodes LLM's feedback answer into actionable operation scripts to execute the app, and iterates the whole process. Specifically, in each iteration of the testing, `GPTDroid` first obtains the view hierarchy file of the mobile app and extracts the GUI context including the app information, information of the current GUI page, and details of each widget in the page. We then design linguistic patterns for generating the GUI prompt as input of LLM. We utilize the idea of few-shot learning to enable the LLM's output to conform with our expected standards which can be directly executed in the app, by providing the demonstrations as reference.

In the inner loop, it builds a testing sequence memorizer to record all the detailed interactive testing information, e.g., the explored activities and widgets. During the process, the memorizer also stores the functionality-level progress of testing, e.g., which function is under test, which is derived by querying the LLM and is to enable LLM to conduct the explicit reasoning by itself. We also design linguistic patterns to encode the information into the functionality-aware memory prompt, to equip LLM with the capability of retaining knowledge of the whole testing and conducting the long-term reasoning. The prompt in both the outer loop and inner loop would together input into the LLM for querying the next operation.

2.1 GUI Context Extraction

Despite its excellence on various tasks, the performance of LLM can be significantly influenced by the quality of its input, i.e., whether the input can precisely describe what to ask [16, 35, 89]. In the scenario of this interactive mobile GUI testing, we need to accurately depict the GUI page currently under test, as well as its contained widgets information from a more micro perspective, and the app information from a more macro perspective. This Section describes which information will be extracted, and Section 2.2 will describe

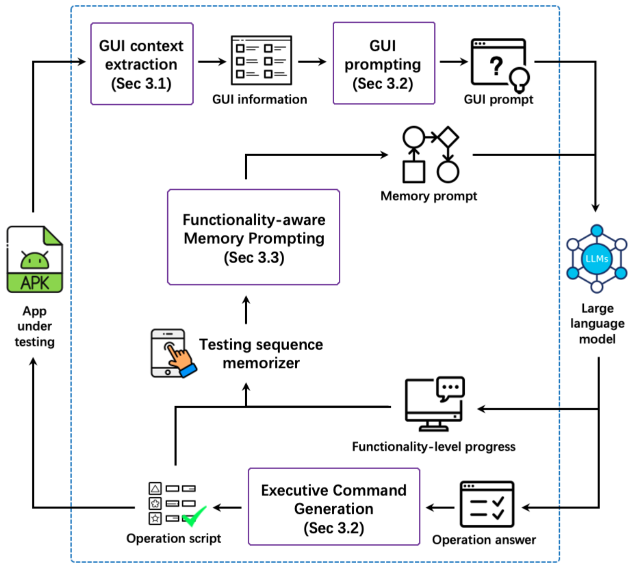

GUI context extraction (Sec 3.1)
GUI information
GUI prompting (Sec 3.2)
GUI prompt
Memory prompt
Functionality-aware Memory Prompting (Sec 3.3)
APK
App under testing
LLMs
Large language model
Testing sequence memorizer
Functionality-level progress
Executive Command Generation (Sec 3.2)
Operation script
Operation answer

Figure 2: Overview of `GPTDroid`.

how we organize the information into the style that LLM can better understand. GUI context relates to the information of the app, the GUI page currently tested, and all the widgets on the page. The app information is extracted from the *AndroidMaincast.xml* file, while the other two types of information are extracted from the view hierarchy file, which can be obtained by UIAutomator [67]. Table 1 presents the summarized view of them.

App information provides the macro-level semantics of the app under testing, which facilitates the LLM to gain a general perspective about the functions of the app. The extracted information includes the name of the app and the name of all its activities.

Page GUI information provides the semantics of the current page under testing during the interactive process, which facilitates the LLM to capture the current snapshot. We extract the activity name of the page, all the widgets represented by the "text" field or "resource-id" field (the first non-empty one in order), and the widget position of the page. For the position, inspired by the screen reader [61, 69, 86], we first obtain the coordinates of each widget in order from top to bottom and from left to right, and the widgets whose ordinate is below the middle of the page is marked as lower, and the rest is marked as upper.

Widget information denotes the micro-level semantics of the GUI page, i.e., the inherent meaning of all its widgets, which facilitates the LLM in providing actionable operational steps related to these widgets. The extracted information includes "text", "hint-text", and "resource-id" field (the first non-empty one in order), "class" field, and "clickable" field. To avoid the empty textual fields of a widget, we also extract the information from nearby widgets to provide a more thorough perspective, which includes the "text" of parent node widgets and sibling node widgets.

2.2 GUI Prompting and Executive Command Generation

With the extracted information, we design linguistic patterns to generate prompts for inputting into the LLM. We first conduct

[1]We release the source code, dataset, and detailed experimental results on our website https://github.com/franklinbill/GPTDroid. And we further provide a demo video https://youtu.be/ZeCrowmvxCg to facilitate the understanding of `GPTDroid`.

Table 1: Extracted GUI information and examples.

Id	Attribute	Description	Examples
		GUI context - App information	
1	AppName	Name of the app under testing	AppName = "Money Tracker"
2	Activities	List of names for all activities of the app, obtained from *AndroidManifest.xml* file	Activities = ["Main", "AddAccount", "Import", "Income", ...]
		GUI context - page GUI information	
3	ActivityName	Activity name of the current GUI page	ActivityName = "AddPersonalInformation"
4	Widgets	List of all widgets in current page, represented with text/id	Widgets = ["Edit Account", "btn_income", ...]
5	Position	Relative position of widgets, obtained through their coordinates	Upper = ["Welcome", ...], Lower = ["Add Income", ...]
		GUI context - widget information	
6	WidgetText	Widget text, obtained by field 'text' or 'hint-text'	WidgetText = "Welcome to the Money Tracker!"
7	WidgetID	Widget ID, obtained by field 'resource-id'.	WidgetID = "add_account"
8	WidgetCategory	Category: TextView, EditText, ImageView, etc, obtained by field 'class'	WidgetCategory = "TextView"
9	WidgetAction	Widget action, obtained by field 'clickable', such as click, input, etc.	WidgetAction = "Click"
10	NearbyWidget	Nearby widgets, obtained by the text of parent widgets and sibling widgets	NearbyWidget = "your income: [SEP] $ "

Table 2: The example of linguistic patterns of GUI prompts and generation rules.

Id	Pattern type	Sample of linguistic patterns/rules	Instantiation
		GUI context patterns: *GUIContext*	
1	App information	We want to test the <*AppName*> App. It has the following activities, including <*Activities*>.	We want to test "Money tracker" App. It has the following activities, including "Main", "AddAccount", "Import", "Setting", ...
2	Page GUI information	The current page is <*ActivityName*>, it has <*Widgets*>. The upper part of the app is <*Position*>, the lower part is <*Position*>.	The current page is "Main", it has "Income", "Add", "Delete", ... The upper part of the app is "Welcome to ..., Delete, ...", the lower part of the app is "Income, ...".
3	Widget information	The widgets which can be operated are <*WidgetText / WidgetID*>. <*WidgetText/WidgetID*> is <*WidgetCategory*> which can <*WidgetAction*> and its nearby widget is <*NearbyWidget*>.	The widgets which can be operated are "Add", "Delete", "Edit Account",... . "Add" is Button which can be clicked and its nearby widget is "Add account, ..." , "Delete" is TextView which can be clicked and its nearby widget is ... .
		Operation & feedback question patterns: *OperationQuestion*	
4	Querying general action	Action operation question + <*Output Template*>	What operation is required? (<*Operation*>[click / double-click / long press / scroll]+<*Widget Name*>)
5	Querying text input	Input operation question + <*Output Template*>	Please generate the input text in sequence, and the operation after input. (<*Widget name*>+<*Input Content*>, ...) and provided (<*Operation*>[click]+<*Widget name*>)
6	Testing feedback	Feedback question	There is no <*WidgetText / WidgetID*> on the current page, please reselect.
		Prompt generation rules	
1	**Start Prompt:** *GUIContext*[1,2,3] + *OperationQuestion*[4/5]		
2	**Test Prompt:** We successfully did the above operation. *GUIContext*[2,3] + *OperationQuestion*[4/5]		
3	**Feedback Prompt:** Sorry, <*OperationQuestion*>[6] + <*GUIContext*>[3] + <*OperationQuestion*>[4/5]		

Notes: "[1,2, ..., 6]" means the id of each pattern. If there is "EditText" on the page, GPTDroid selects "Querying text input" pattern. For the other widgets, GPTDroid selects "Querying general action" pattern.

preprocessing for the information, to facilitate the follow-up design. We tokenize attributes by the underscore and Camel Case [6] considering the naming convention in app development.

2.2.1 *Linguistic Patterns of GUI Prompt.*

To design the patterns, each of the five annotators is asked to write the prompt sentence following regular prompt template [14, 16, 26], and questions the LLM for generating the operation steps. We then check to what extent the recommended operation is reasonable considering the whole testing process. With the prompt sentences, the five annotators then conduct card sorting [60] and discussion to derive the linguistic patterns. As shown in Table 2, this process comes out with 6 linguistic patterns corresponding with the three sub-types of information in Table 1 and two operation & feedback patterns.

Pattern related to GUI context (Table 2-Id 1,2,3) We design three patterns to describe the overview of the GUI page currently under testing, respectively corresponding to the app information, page GUI information, and widget information in Table 1.

Pattern related to operation & feedback question (Table 2-Id 4,5,6) We also design patterns to describe operation and feedback questions. For the operational questions, we ask the LLM what operation is required. We also provide the output template in the prompt to enable the LLM to generate a desired executive command for testing the app, and details are in Section 2.2.3. Note that, we separate the patterns for querying general action (e.g., click) and text input (e.g., input certain text) with *pattern-Id 4 and 5* respectively, to enable the generated commands can better match the widgets in the GUI page. For the feedback question, after deciding the previous operation is not applicable, we inform the LLM that there is no such widget on the current page, and let it re-try.

2.2.2 *Prompt Generation Rules.*

Since the designed patterns describe information from different points of view, we combine the patterns from different viewpoints and generate the prompt rules as shown in Table 2. We design three kinds of prompts respectively for starting the test, routine inquiry, and getting feedback in case of error occurred. Note that, due to the robustness of the LLM, the prompt sentence doesn't need to follow the grammar completely.

Test prompt is the most commonly used prompt for informing the LLM of the current status and query for the next operation. Specifically, we tell the LLM the GUI context, i.e., the information about the current GUI page and detailed widget information; then ask the LLM which operation is required.

Feedback prompt is used for informing the LLM error occurred and re-try for querying the next operation. Specifically, we first tell LLM its generation operation cannot correspond to the widget on the page; re-provide it the detailed widget information of the page and let the LLM recommend the operation again.

Besides the above two kinds of prompts, we additionally design **start prompt** to start the testing of the app and only used it once. Different from the test prompt, it provides the LLM with the app information including all activities for a global overview.

2.2.3 ***Executive Command Generation.*** After inputting the generated prompt, LLM will output the natural language sentence of operation, e.g., *input 3500 for price, input salary in title widget and personal in category widget, then click submit which depicts the example output for the second image in Figure 1.* Considering that a testing operation can be expressed in different ways and with different words, it is challenging to map natural language testing operations to the app for execution. Therefore, we utilize the idea of in-context learning to provide LLM with the output template, including available operations and operation primitives, which can be mapped directly to the instructions for executing the app.

Available operations. We identify five commonly used standard operations for mobile applications, including click, double-click, long press, scroll and input. Although there are other customizations, such as custom gestures, they are not common, and we leave them for future work.

Output templates. We design different operation primitives for the above-mentioned operations to represent widgets and executive commands. The above five operations can be divided into two main categories, i.e., *action* (the first four operations) and *input*. For the action category, we formulate it as <Operation>[click / double-click / long press / scroll] + <Widget name>, e.g., *Operation: "Click". Widget: "ADD INCOME".*. For the input category, the GUI page usually involves text field widgets for entering specific values, and the follow-up operations (usually for submitting the text input). We formulated it as <Widget name> + <Input content>, e.g., *Widget: "Price", Input: "3500"*, followed by <Operation> + <Widget name>, e.g., Operation: "Click". Widget: "Submit" . Table 4 provides an example answer from LLM for these two categories.

2.3 Functionality-aware Memory Prompting

With the context extraction, GUI prompting and executive command generation in the previous two sections, `GPTDroid` can already conduct the automated GUI testing. This Section proposes the function-aware memory prompting, which further improves the capability of the approach in retaining the knowledge during the iterative testing process and understanding the functional aspects of the mobile app, so as to generate the function-aware operations to guide the operation from global viewpoints.

To achieve this, we build a testing sequence memorizer (Section 2.3.2) to record all detailed testing information. We also query the LLM about the function-level progress of the testing (Section 2.3.1) in each iterative testing step, and store it into the memorizer. We then design linguistic patterns to encode the information into the functionality-aware memory prompt (Section 2.3.3), and query the LLM together with the prompt in the previous section.

2.3.1 ***Functionality-level Progress.*** Due to the importance of functionalities for app users, we hope our automated GUI testing can conduct the exploration from the viewpoints of the functionalities. Hence we need to make LLM understand what function is currently being tested derived from the explored activities and widgets. Specifically, we design a prompt to ask LLM what functionality is currently being tested and whether it has been completed in Section 2.3.3. And the LLM would provide us with the functionality-level progress as <Function name> + <Status>, in which 'YES' denotes it has completed testing the specific functionality.

To facilitate the LLM to make reasonable functionality-level decisions, we also provide the LLM with the list of functionalities of the app in the prompt. This information is first extracted from the app description file and the activity names to serve as the initial seed, and will continuously let the LLM output the refined information during the iterative testing process.

2.3.2 ***Testing Sequence Memorizer.*** We design a testing sequence memorizer to keep the record of testing, including the set of tested functions (in Section 2.3.1), the testing path of activity, the set of tested activities with page visits number, the set of tested widgets of the current page with widgets visits number.

Specifically, during the iteration, when an operation is conducted, we can obtain the status of the function (<Function name>+<Status>), the operation of the widget (<Operation>+<Widget name>), and the test path of activity (<Activity1>+<Operation>+<Activity2>)). Then the operation memorizer is updated accordingly. In detail, the visit number of the widget is updated by finding the same widget in the operation memorizer with the "text" field and "resource-id" field of the widget. The visit number of activity is updated by finding the same activity in the memorizer with the "ActivityName" field.

2.3.3 ***Functionality-aware Memory Prompt.*** Based on the information in the testing sequence memorizer, we further construct a functionality-aware memory prompt to facilitate the LLM to keep an eye on the functionality during recommending the next operation. It consists of three parts, including the explored functionalities, the covered activities, and the recently tested operations, and it would refer to the testing sequence memorizer in each iteration to fetch the latest information. We follow the same procedure in Section 2.2 to derive these prompt patterns. The details and examples of the prompts are shown in Table 3.

Pattern related to explored functionalities (Table 3-Id 1). This describes which functions have been explored, the number of explorations, and whether it is finished in testing the function. Since the GUI prompt in the previous Section only demonstrates the widgets and activities in the current GUI page, it could not provide the high-level viewpoints of the functionality which is accomplished by a sequence of operations on the widgets/activities. Therefore, this prompt can remind the LLM about the functional aspects of the app, and facilitate the LLM in deciding the meaningful operation sequence to explore the app's functionality. Specifically, `GPTDroid` extracts the tested functions from the testing sequence memorizer, including the visit times of function pages, and the testing status of the functions.

Pattern related to covered activities. (Table 3-Id 2) This describes the sequence of covered activities during the testing process. Since the basic component of the mobile app is the activity which is recorded in the *AndroidManifest.xml* file, this prompt aims at providing the activity viewpoints of testing history to enable the

Table 3: The example of linguistic patterns of functionality-aware memory prompts and generation rules.

Id	Type	Sample of linguistic patterns/rules	Instantiation
		Long-term functionality-aware memory patterns:*FunctionMemory*	
1	Explored functionalities	**List of tested functions:** <*Function Name*> + <*Visits Time*> + <*Status*>, <*Function Name*> + <*Visits Time*> + <*Status*> ...	**List of tested functions:** "Function1: Add your income. Visits: 3. Status: Finished", "Function2: Delete information. Visits: 2. Status: Finished", ...
2	Covered activities.	**Path of tested activities:** <*Activity Name*> + <*Visits Time*>, <*Activity Name*> + <*Visits Time*>, ...	**Path of tested activities:** "Activity: Main. Visits: 3", "Activity: Account. Visits: 4", "Activity: AddAccount. Visits: 3", ...
3	Recently tested operations	**History of latest tested pages and operations: Latest 5th step** tested the <*Activity Name*> page. The following widgets with visits time of this page have been tested: <*Widget Name*> + <*Visits Time*>, ... The following executive command achieve the page transition: <*Operation*> + <*Widget Name*>. **Latest 4th step** tested the <*Activity Name*> page. The following ... ... **Latest 1st step** tested the <*Activity Name*> page. The following ...	**History of latest tested pages and operations: Latest 5th step** tested the "Exchange" page. The following widgets with visits time of this page have been tested: "Widget: Add exchange, Visits:2", "Widget: Submit exchange, Visits:1", 'Widget: Cancel, Visits:1" ... . The following executive command achieve the page transition: "Click" the "Exchange". **Latest 4th step** tested the "main" page. The following widgets ... ... **Latest 1st step** tested the "Add Account" page. The following widgets ...
		Function question patterns: *FunctionQuestion*	
4	Functionality inquiry	Functionality inquiry + <*Output Template*>	What is the functions currently being tested? Are we testing a new function? (<*FunctionName*> + <*Status*>)
		Functionality-aware memory prompt generation rules	
1	**Functionality-aware memory Prompt:** *FunctionMemory*[1,2,3] + *FunctionQuestion*[4]		

LLM better capture the tested functionalities and to cover more unexplored areas. Specifically, we merge the adjacent same activity and activity sequence, and update the number of visits accordingly.

Pattern related to recently tested operations. (Table 3-Id 3) This denotes the latest test page with the detailed visiting status of all its contained widgets, as well as the operations leaving the page. This information is the first-hand and fine-grained testing recording, which can facilitate the LLM in capturing the latest status of the testing progress, and make informed decisions about exploring a certain functionality. Specifically, GPTDroid extracts the widgets tested on each GUI page and their visit times from the testing sequence memorizer. For the current test page, we will select the operation pages of the lastest k steps and the corresponding operations. For each page, we provide LLM with the activity name of the current page and the number of visits to each widget when the page is accessed. We set k as 5 based on the empirical experience.

Pattern related to functionality inquiry: (Table 3-Id 4) We ask the LLM what function is currently tested, and also provide the output template in the prompt to enable the LLM to generate the function name and its status, with details in Section 2.3.1.

Prompt Generation Rules. We combine the above three patterns for providing LLM with functionality aspects of testing information from different viewpoints, and generate the prompt rules as shown in Table 3. We provide examples of how the GUI context prompts and memory prompts work to enable app testing.

2.4 Implementation

GPTDroid is implemented as a fully automated GUI app testing tool, which uses or extends the following tools: VirtualBox [9] and the Python library pyvbox [7] for running and controlling the Android-x86 OS, Android UIAutomator [67] for extracting the view hierarchy file, and Android Debug Bridge (ADB) [1] for interacting with the app under test (Section 2.1). For the LLM (Section 2.2), we use the pre-trained ChatGPT model which was released on the OpenAI website[2]. The basic model of ChatGPT is the *gpt-3.5-turbo* model which is extremely powerful and good at answering questions.

[2]https://platform.openai.com/docs/models/gpt-3-5

Table 4: The example of how prompts work in GPTDroid.

Prompt type	Instantiation
	Example 1: For general action operation
GUI context	**Start Prompt:**[We want to test "Money tracker" App, It has the following activities, including ... .] **or Test Prompt:** [We successfully did the above operation.] **or Feedback prompt:** [There is no "Exchange" on the current page, please reselect.] [The current page is "AddAccount", it has ... . The upper part of the app is ... , the lower part ... The "Exchange" is Button ...]
Functionality-aware memory	[**List of tested functions:** "Function1: Add your income. ...", ...] [**Path of tested activities:** "Activity: Main. Visits: 3", ...] [**History of latest tested pages and operations: Latest 5th step** tested the "Exchange" page. The following widgets ... **Latest 1st step** tested the "AddAccount" page. ...]
Function question	What is the functions currently being tested? Are we testing a new function? (<Function name> + <Status>)
Action question	What operation is required? (<*Operation*>[click / double-click / long press / scroll]+<*Widget Name*>)
LLM Answer	Function: "Add income". Status: Yes. Operation: "Click". Widget: "ADD INCOME".
	Example 2: For text input operation
GUI context	**Start Prompt:**[...] **or Test Prompt:**. **or Feedback prompt:** ..
functionality-aware memory	[**List of tested functions:** "Function1: Add ...", ...] [**Path of tested activities:** ...] [**History of latest tested pages ...**]
Function question	What is the functions currently being tested? Are we testing a new function? (<Function name> + <Status>)
Input question	Please generate the input text in sequence, and the operation after input. (<*Widget name*>+<*Input Content*>, ... and provided <*Operation*>+<*Widget name*>)
LLM Answer	Function: "Add income". Status: No. Widget: "Price". Input: "3500". Widget: "Title". Input: "salary". Widget: "Category". Input: "personal". Operation: "Click". Widget: "Submit".

3 EFFECTIVENESS EVALUATION

In order to verify the performance of GPTDroid, we evaluate it by investigating the activity and code coverage (RQ1), as well as the number of detected bugs (RQ2). We also present the ablation study of each module in GPTDroid (RQ3). Note that, this Section utilizes the previously-detected bugs in the app's repositories to demonstrate the effectiveness of GPTDroid, and the next Section will evaluate the usefulness of GPTDroid in detecting new bugs.

3.1 Experimental Setup

The experimental dataset comes from two sources. The first is from the apps in the Themis benchmark [63], which contains 20 open-source apps with 34 bugs in GitHub. Considering the small number of apps in the benchmark, we collect a second dataset following similar procedures as the benchmark.

Table 5: Dataset of effectiveness evaluation.

Statistics	#Activities	#Bugs	#Download	#Update
Min	7	1	50K+	05/22
Max	33	9	100M+	05/23
Median	17	4	5M+	-
Average	15	1.5	10M+	-
All	1398	143	-	-

In detail, we crawl the 50 most popular apps of each category from Google Play [4], and we keep the ones with at least one update after May. 2022, resulting in 407 apps in 12 Google Play categories. Then, we use 10 commonly used and state-of-the-art automated GUI testing tools (details are in Section 3.2) to run these apps, in turn, to ensure that they work properly. We then filter out the unusable apps by the following criteria: (1) They would constantly crash on the emulator. (2) One or more tools can not run on them. (3) The registration and login functions cannot be skipped with scripts [21, 38, 42, 63, 87]. (4) They do not have issue records or pull requests on GitHub.

There are 73 apps (with 109 bugs) remaining for this effectiveness evaluation. Note that, same as the benchmark, all bugs are crash bugs. Specifically, for each app, we select the version in which the bugs are confirmed by developers (merged GitHub pull requests) as our experimental data, following the practice of the benchmark. The details of all 93 experimental apps (20 + 73) and related bugs are shown in Table 5.

Note that, there are 101 apps that are filtered out for effectiveness evaluation, yet can successfully run with our proposed approach. We apply them to the manual prompt generation in Section 2.2.1. And this ensures that there is no overlap between the apps in approach design and evaluation.

We employ activity coverage, code coverage and the number of detected bugs, which are widely used metrics for evaluating GUI testing [11, 27, 34, 36, 70, 72]. We also present the number of covered activities and widgets which are also commonly-used metrics in Table 6. We treat the activities defined in the *AndroidManifest.xml* file of an Android app as the whole set of activities [2, 50, 62]. During the testing process, we collect the unique activity name and widget ID of the GUI page with which the operation interacts, and treat them as the activity number and widget number.

3.2 Baselines

To demonstrate the advantage of GPTDroid, we compare it with 10 common-used and state-of-the-art automated testing techniques. We roughly divide them into random-/rule-based, model-based, and learning-based methods, to facilitate understanding. For random-/rule-based methods, we use Monkey [20] and Droidbot [33], Time-Machine [21]. For model-based methods, we use WCTester[83, 88], Stoat [62], Ape [25], Fastbot [13], ComboDroid [70]. For learning-based methods, we use Humanoid [34] and Q-testing [50].

For a more thorough comparison, we additionally include 5 other baselines, in which the originally proposed techniques aim at enhancing the automated GUI testing, and can be utilized by integrating with the above-mentioned automated testing tools. Specifically, QTypist [37] is used to generate valid text input to enhance the coverage of automated testing tools. Toller [71] is a tool consisting of infrastructure enhancements to the Android operating system. Vet [71] is used to identify exploration tarpits by recognizing their patterns in the UI traces, so as to optimize the exploration sequences. We use the experimental setup as their original paper to derive the following 5 baselines, i.e, QTypist is integrated with Droidbot and Ape (Droidbot+QT, Ape+QT), Toller is integrated with Stoat (Stoat+TO), Vet is integrated with WCTester and Ape (WCTester+VE, Ape+VE).

We deploy the baselines and our approach on a 64-bit Ubuntu 18.04 machine (64 cores, AMD CPU) and evaluate them on Google Android 7.1 emulators. Each emulator is configured with 2GB RAM, 1GB SDCard, 1GB internal storage, and X86 ABI image. Different types of external files (including PNGs / MP3s / PDFs / TXTs / DOCXs) are stored on the SDCard to facilitate file access from apps. Following common practice [25, 33], we registered separate accounts for each bug that requires login and wrote the login scripts, and during testing reset the account data before each run to avoid possible interference. In order to ensure fair and reasonable use of resources, we set up the running time of each tool in one app to 60 minutes, which is widely used in other GUI testing studies [22, 25, 33, 63]. We run each tool three times and obtain the highest performance to mitigate potential bias.

3.3 Results and Analysis

3.3.1 ***Performance of Coverage (RQ1).*** Table 6 shows the number of covered widgets, number of covered activities, and average activity coverage of GPTDroid and the baselines. We can see that GPTDroid covers far more widgets and activities than the baselines, and the average activity coverage achieves 75% and average code coverage achieves 66% across the 93 apps. It is 32% (0.75 vs. 0.57) activity coverage higher even compared with the best baseline (Ape with QTypist). Meanwhile, on the Themis benchmark and Google Play datasets, it was 28% (0.69 vs. 0.54) and 33% (0.77 vs. 0.58) higher than the best baseline. This indicates the effectiveness of GPTDroid in covering more activities and codes, thus bringing higher confidence to the app quality and potentially uncovering more bugs. Section 5 will further analyze why GPTDroid performs well.

Figure 3 additionally demonstrates the average activity coverage with varying times. We can see that, at every time point, GPTDroid achieves higher activity coverage than the baselines, and it achieves high coverage within about 24 minutes. This again indicates the effectiveness and efficiency of GPTDroid in covering more activities with less time, which is valuable considering the testing budget.

Among the baselines, the model-based and learning-based approaches have relatively higher performance. Yet the model-based

Table 6: Performance of activity coverage (RQ1).

Metric	Random-/rule-based				Model-based								Learning-based			
	MK	DB	DB+QT	TM	WC	WC+VE	ST	ST+TO	AP	AP+QT	AP+VE	FB	CD	HM	Q-t	`GPTDroid`
#Widgets	691	1707	2522	3811	1745	2465	2830	3087	2964	3496	3406	2944	3210	3022	2261	**5243**
#Activities	266	461	573	811	545	573	615	671	741	853	811	755	783	657	685	**1049**
Avg. activity coverage	0.25	0.33	0.41	0.56	0.39	0.41	0.44	0.48	0.53	0.57	0.56	0.54	0.55	0.47	0.49	**0.75**
Avg. code coverage	0.17	0.28	0.36	0.53	0.31	0.33	0.40	0.44	0.45	0.55	0.52	0.47	0.48	0.43	0.41	**0.66**

Notes: "MK" is Monkey, "DB" is Droidbot, "DB+QT" is Droidbot with QTypist, "TM" is TimeMachine, "WC" is WCTester, "WC+VE" is WCTester with Vet, "ST" is Stoat, "ST+TO" is Stoat with Toller, "AP" is Ape, "AP+QT" is Ape with QTypist, "AP+VE" is Ape with Vet, "FB" is Fastbot, "CD" is ComboDroid, "HM" is Humanoid, "Q-t" is Q-testing.

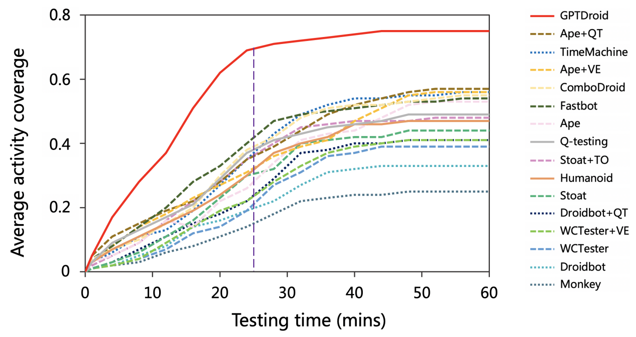


Figure 3: Activity coverage with varying time (RQ1).

approaches can't capture the GUI semantic information and the exploration could not well understand the inherent business logic of the app. Learning-based approaches only use little context information for guiding the exploration, and don't have the mechanism for enabling the model considering the app's functionalities.

We further analyze the potential reasons for the uncovered cases. First, some widgets or inputs do not have meaningful "text" or "resource-id", which hinders the approach of effectively understanding the GUI page. Second, some app requires specific operations, e.g., database connection, long press and drag widgets to a fixed location, which is difficult if not impossible to be automatically achieved.

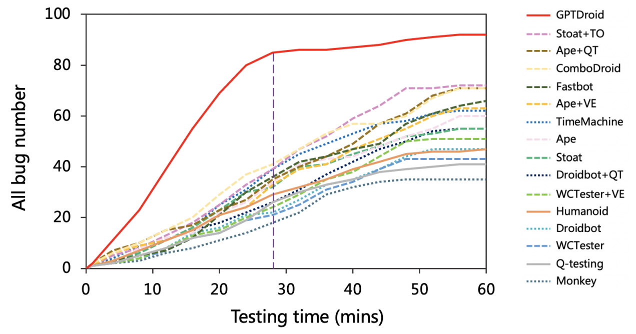


Figure 4: Bug detection with varying time (RQ2).

3.3.2 ***Performance of Bug Detection (RQ2).*** Figure 4 shows the overall number of detected bugs of `GPTDroid` and baselines with varying times. `GPTDroid` detects 95 bugs for the 93 apps, 31% (95 vs. 66) higher than the best baseline (Stoat with Toller). We also compare the similarities and differences of the bugs between Stoat with Toller and our approach, and the results show that all bugs detected by Stoat with Toller are also detected by `GPTDroid`. This indicates the effectiveness of `GPTDroid` in detecting bugs and helps to ensure app quality.

We can also see that, in every time point, `GPTDroid` detects more bugs than the baselines, and reaches the highest value in about 27 minutes, saving 35% (17 vs. 26) of the testing time compared with the best baseline (also with more detected bugs). This again proves the effectiveness and efficiency of `GPTDroid`, which is valuable for saving more time for follow-up bug fixing. We will conduct a further discussion about the reason behind the superior performance in Section 3.3.3.

Table 7: Contribution of different modules (RQ3)

Module	Activity coverage	Code coverage
`GPTDroid`	**0.75**	**0.66**
w/o GUI Context	0.17	0.14
w/o Function Memory	0.34	0.28

3.3.3 ***Ablation Study (RQ3).*** **Contribution of Modules.** Table 7 shows the performance of `GPTDroid` and its 2 variants. In detail, for `GPTDroid` *w/o GUI Context* (Sec 2.1), we replace the GUI context information with the raw view hierarchy file and extracted the widgets name. For `GPTDroid` *w/o Function Memory* (Sec 2.3), we remove the functionality-aware memory prompting.

We can see that `GPTDroid`'s activity and code coverage are much higher than all other variants, indicating the necessity of the designed modules and the advantage of our approach. Compared with `GPTDroid`, `GPTDroid` *w/o GUI Context* results in the largest performance decline, i.e., 77% drop (0.17 vs. 0.75) in activity coverage. This further indicates that the GUI context extraction can help LLM understand the structure and semantic information of GUI pages and make reasonable judgments. `GPTDroid` *w/o Function Memory* also undergoes a big performance decrease, i.e., 55% (0.34 vs. 0.75) in activity coverage. This implies our proposed functionality-aware memory prompt can help retain the knowledge during the testing process and gain global viewpoints to reach the uncovered areas.

Contribution of Sub-modules. Table 5 further demonstrates the performance of `GPTDroid` and its 8 variants. We remove the part of prompt when querying LLM, i.e., the first four variants respectively remove *pattern 1, 2, 3, 5, 6* of Table 2, and the last three variants respectively remove *pattern 1, 2, 3* of Table 3.

The experimental results demonstrate that removing any of the sub-modules would result in a noticeable performance decline, indicating the necessity and effectiveness of the designed sub-modules. Removing the explored functionalities (`GPTDroid` *w/o-Explored Function*) has the greatest impact on the performance, reducing the activity coverage by 51% (0.37 vs. 0.75). This indicates

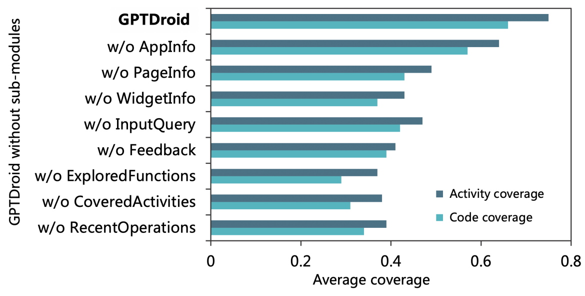


Figure 5: Contribution of different sub-modules (RQ3).

by explicitly querying the LLM about the functionality aspects of the testing progress, the approach can be more aware of what functionality is under test and effectively plan the exploration path to cover more functionalities.

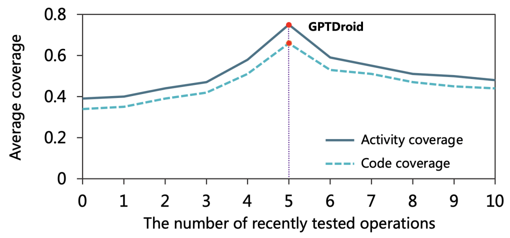


Figure 6: Different number of tested operations (RQ3).

Influence of Different Number of Recent Tested Operations. Figure 6 demonstrates the performance under the different number of latest tested operations. We can see that the activity and code coverage increase with the more tested pages in the latest tested operations, i.e., 5 steps of tested pages and operations. And after that, the performance would gradually decrease even increasing the number of tested pages. It also further verified that the 5 steps selected by our pilot study are effective and valuable.

4 USEFULNESS EVALUATION

4.1 Experimental Setup

This Section further evaluates the usefulness of `GPTDroid` in detecting new crash bugs. We employ a similar experimental setup to the previous section. To make it brief, we only compare the best baselines, i.e., TimeMachine, Stoat with Toller and Humanoid, the best one from each type of method following their bug detection performance as shown in Figure 4. Note that, for coverage, the top three are different sets of baselines, yet this Section is concerned about their capability in detecting bugs, so we use their bug detection performance as the criteria.

We begin with the most popular and recently updated 317 apps from 12 categories as in the previous Section. Then we reuse the four criteria in Section 3.1 for filtering out the unusable apps. Differently, we loosen criteria 4, which only requires the app to have ways for bug reporting, since the issue records or pull requests are not mandatory in this Section. This results in 223 apps for usefulness evaluation. Note that, this Section aims at evaluating whether `GPTDroid` can detect new bugs in the apps, thus the overlap between the apps of this Section and the previous Section is allowed.

We use the same hardware and software configurations as the previous evaluation Section. When the crash bugs are detected, we report them to the development team through online issue reports or emails.

4.2 Results and Analysis

For the 223 apps, `GPTDroid` detects 135 bugs involving 115 apps, of which 53 bugs involving 41 apps are newly-detected bugs. Furthermore, only 9 of these new bugs were detected by three baselines. We submit these 53 bugs to developers, and 35 of them have been fixed/-confirmed so far (20 fixed and 15 confirmed), while the remaining are still pending (none of them is rejected). This further indicates the effectiveness of our proposed `GPTDroid` in bug detection. Due to space limit, Table 8 presents these fixed/confirmed bugs, and the full lists can be found on our website[1].

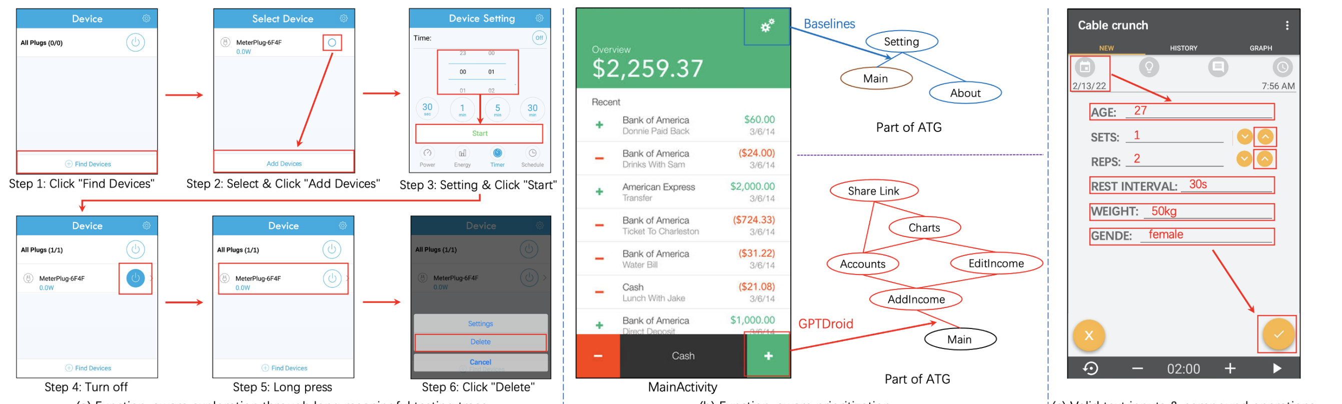
We further analyze the details of these bugs, and 17 of them involve multiple text inputs or compound operations. Besides, we also observe that there are 11 bugs with more than 20 operations before triggering the bug, counting from the *MainActivity* page, which indicates the ability of `GPTDroid` in testing deeper features. Furthermore, we find at least 28 bugs related to the main business logic of the app. This further demonstrates the capabilities of `GPTDroid`, and we provide analysis in Section 5.

Table 8: Confirmed or fixed bugs

Id	APP Name	Category	Download	Status	TM	ST+TO	HM
1	PerfectPia	Music	50M+	confirmed			
2	MusicPlayer	Music	50M+	confirmed			
3	NoxSecu	Tool	10M+	fixed	✓		
4	INSTA	Finance	10M+	fixed		✓	
5	Degoo	Tool	10M+	fixed			
6	Proxy	Tool	10M+	confirmed			
7	Secure	Tool	10M+	fixed			
8	Revolut	Finance	10M+	fixed			
9	Thunder	Tool	10M+	confirmed	✓	✓	
10	ApowerMir	Tool	5M+	fixed			
11	MediaFire	Product	5M+	confirmed		✓	
12	WAVMoney	Finance	1M+	fixed			
13	Postegro	Commun	500K+	fixed			
14	Deezer MP	Music	500K+	fixed	✓		
15	MTG	Utilities	500K+	fixed			
16	Yucata	Tool	500K+	confirmed			
17	ClassySha	Tool	500K+	fixed			
18	Linphone	Commun	500K+	confirmed	✓	✓	
19	OFF	Health	500K+	confirmed			
20	Paytm	Finance	100K+	confirmed			
21	Transdroid	Tool	100K+	confirmed		✓	✓
22	Transistor	Music	10K+	fixed			
23	Onkyo	Music	10K+	fixed			✓
24	Democracy	News	10K+	confirmed			
25	NewPipe	Media	10K+	confirmed	✓		
26	LessPass	Product	10K+	confirmed			
27	CEToolbox	Medical	10K+	confirmed			
28	OSM	Health	10K+	fixed			
29	Monse	Finance	10K+	fixed			✓
30	Fitb	Health	10K+	confirmed			
31	KHAN	Education	10K+	fixed			
32	Leaprt	Tool	10K+	fixed			
33	Penly	Product	10K+	fixed			
34	Rocket	Product	10K+	fixed			
35	Fkowy	Tool	10K+	fixed			

5 INSIGHTS FROM EXPERIMENT RESULTS

This Section summarizes 4 kinds of capabilities of `GPTDroid` including high-level (i.e., long meaningful test trace, test case prioritization) and low-level ones (i.e., valid text input, compound actions), to unveil the mystery of why `GPTDroid` outperforms existing method.



(a) Function-aware exploration through long meaningful testing trace
(b) Function-aware prioritization
(c) Valid text inputs & compound operations

Figure 7: Examples of our insights from experiments.

Functionality-aware exploration through the long meaningful testing trace. GPTDroid can automatically generate the test cases with a long sequence of operations which together accomplish a business logic of the app, and this is quite important for covering the app features and ensuring its quality. As shown in Figure 7 (a), in SmartMeter app [8], to test a commonly-used app feature "delete equipment", the automated tool first needs to click "Find Devices" on the device page, then select a device (Bluetooth is turned on and there are candidate devices) and click "Add Devices" for adding it in the device page, input the related information and click "Start" to start the device, then turn off this device in the device page, long press it and click "Delete" from the pop-up menu. Only with this long sequence of operations that touches the "deleting equipment" feature, a crash can be revealed. Our functionality-aware memory prompt can enable the LLM to capture the long-term dependencies among GUI pages to conduct the functionality-guided exploration.

Function-aware prioritization. We also observe that GPTDroid usually prioritizes testing the "important" functions, which is valuable for reaching a higher activity coverage and covering more key activities with relatively less time. As shown in Figure 7 (b), on the Main page of the Moni app [5], the baseline tools tend to first click the "Setting" button following the exploration order from upper to lower, which leads the testing easily trapped into the setting page cycle. GPTDroid chooses to first click the "AddIncome" button to explore the add income functionality which is the key feature of the app. This is facilitated by the semantic understanding of the GUI page and the functionality-aware memory designed in GPTDroid.

Valid text inputs. GPTDroid can automatically fill in valid text content to the input widget which is essentially the key for passing the page as seen in Figure 7 (c). Similar to prior work [37], our model can generate semantic text input (e.g., income, date, etc) accordingly. Besides single text input, it can also successfully fill in multiple input widgets at the same time which are correlated to each other like the departure and arrival cities and dates in the flight booking app. We have designed a prompt, especially for querying the text input and utilize the few-shot learning by providing demonstrations with the output template to facilitate the LLM generating desired executive commands, which can be accurately mapped to the GUI widgets of text inputs to enable it to execute automatically.

Compound actions. GPTDroid can conduct complex compound operations guided by the LLM. As shown in (Figure 7 (c), to add the "Cable crunch" information, it first inputs the text, selects the date, sets the "SETS" and "REPS" by clicking the upper or lower button, then click the submit button in the lower right corner. Thanks to our designed executive command generation method with few-shot learning and output template, GPTDroid can accurately map the LLM's output into the actions related to GUI widgets.

6 RELATED WORK

Automated GUI testing. To ensure the quality of mobile apps, many researchers study the automatic generation of large-scale test scripts to test apps [74]. Monkey [20] is the popular random-based automated GUI testing tool, which emits pseudo-random streams of UI events and some system events. However, the random-based testing strategy cannot formulate a reasonable testing path according to the characteristics of the app, resulting in low test coverage. To improve the test coverage, researchers propose model-based [21, 25, 44, 47, 62, 70, 77, 78, 83] automated GUI testing methods, design corresponding models through the research and analysis of large-scale apps. Although model-based automated GUI testing tools can improve test coverage, the coverage is still low because it does not consider the semantic information of the app's GUI and Page. Researchers further proposed human-like testing strategies and designed learning-based [34, 50] automated GUI testing methods. Although the learning-based approach can improve the test coverage by learning the interactive processes or using the idea of reinforcement learning. However, it is still unable to better understand the semantic information of the page and plan the path according to the actual situation of the app. We aim at proposing a more effective approach to generate human-like actions for thoroughly and more effectively testing the app, accomplishing it with LLM. There are also studies that tried to find bugs in similar apps to move beyond crashes [39–41, 65], yet it cannot reveal the crashes automatically as this work. Techniques related to test migration [12, 64] can generate meaningful operation sequences borrowed from the source app, yet it is quite demanding to require the test cases of an app, and by comparison, GPTDroid can generate more meaningful test traces from scratch.

LLM for Software Engineering. Considering the powerful performance of LLM, researchers have successfully leveraged it to solve various tasks in the field of software engineering [10, 29, 45, 46, 48, 49, 51, 57, 59, 66, 75, 80–82, 85]. Supported by code naturalness [28], researchers applied the LLMs to code writing in different programming languages [15, 23, 24, 76]. In testing, LLMFuzz [19] used LLMs to generate input programs for fuzzing Deep Learning libraries. Xia et al. [73] applied LLM for automatic program repair to improve the accuracy of the generated repair patches. Lemieux et al. [32] leveraged LLM in escaping the coverage plateaus in test generation. Kang et al. [30] explored the LLM-based bug reproduction. A similar work QTypist [37] leveraged the LLM to generate the text inputs for passing a GUI page in order to improve the testing coverage of mobile testing. Different from its sole focus on text input generation to boost existing GUI testing tools, `GPTDroid` is a complete GUI testing tool in asking LLM to propose different actions to interact with the target app including clicking buttons, filling in text and even more complicated compound actions. It makes this work more generalized for wild mobile app testing.

7 CONCLUSION

Automated GUI testing has made much progress, yet still suffers from low activity coverage and may miss critical bugs. This paper aims at generating human-like actions to facilitate app testing more thoroughly and effectively. Inspired by ChatGPT, we formulate the GUI testing problem as a Q&A task and propose `GPTDroid`. It extracts the GUI context and functionality-aware memory, encodes them into prompt questions to ask the LLM, decodes the LLM's feedback answer into actionable operations to execute the app, and iterates the whole process. Results on 93 popular apps demonstrate that `GPTDroid` can achieve 75% activity coverage, with 32% higher than the best baseline, and can detect 31% more bugs with faster speed than the best baseline. `GPTDroid` also detects 53 new bugs on Google Play with 35 of them being confirmed/fixed. In the future, we plan to explore more advanced prompt engineering design to better exploit the power of LLM, and may also fine-tune open-source LLM for this specific tasks for better performance.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China Grant No.62232016, No.62072442 and No.62272445, Youth Innovation Promotion Association Chinese Academy of Sciences, Basic Research Program of ISCAS Grant No. ISCAS-JCZD-202304, and Major Program of ISCAS Grant No. ISCAS-ZD-202302.

REFERENCES

[1] 2023. Android Debug Bridge (adb). https://developer.android.com/studio/command-line/adb.html#forwardports.

[2] 2023. Android development. http://developer.android.com/reference/android.

[3] 2023. App Store. https://www.apple.com.cn/app-store/.

[4] 2023. Google play. https://play.google.com/store/apps/.

[5] 2023. Moni. https://play.google.com/store/apps/details?id=Moni.

[6] 2023. pascal case. https://en.wikipedia.org/wiki/Camel_case.

[7] 2023. pyvbox. https://pypi.org/project/pyvbox/.

[8] 2023. SmartMeter. https://play.google.com/store/apps/details?id=SmartMeter.

[9] 2023. virtualbox. https://www.virtualbox.org/.

[10] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2023. A3Test: Assertion-Augmented Automated Test Case Generation. *arXiv preprint arXiv:2302.10352* (2023).

[11] Yauhen Leanidavich Arnatovich, Lipo Wang, Ngoc Minh Ngo, and Charlie Soh. 2018. Mobolic: An automated approach to exercising mobile application GUIs using symbiosis of online testing technique and customated input generation. *Software: Practice and Experience* 48, 5 (2018), 1107–1142.

[12] Farnaz Behrang and Alessandro Orso. 2019. Test Migration Between Mobile Apps with Similar Functionality. In *ASE 2019*.

[13] Tianqin Cai, Zhao Zhang, and Ping Yang. 2020. Fastbot: A Multi-Agent Model-Based Test Generation System Beijing Bytedance Network Technology Co., Ltd.. In *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test.* 93–96.

[14] Andrew Cantino. 2016. Prompt Engineering Tips and Tricks with GPT-3. https://blog.andrewcantino.com/blog/2021/04/21/prompt-engineering-tips-and-tricks/.

[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[16] Xiang Chen, Ningyu Zhang, Xin Xie, Shumin Deng, Yunzhi Yao, Chuanqi Tan, Fei Huang, Luo Si, and Huajun Chen. 2022. Knowprompt: Knowledge-aware prompt-tuning with synergistic optimization for relation extraction. In *Proceedings of the ACM Web Conference 2022.* 2778–2788.

[17] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).

[18] 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 *UIST*.

[19] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2022. Fuzzing Deep-Learning Libraries via Large Language Models. *arXiv preprint arXiv:2212.14834* (2022).

[20] Android Developers. 2012. Ui/application exerciser monkey.

[21] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of android apps. In *ICSE*. IEEE.

[22] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018. Large-scale analysis of framework-specific exceptions in android apps. In *ICSE*. IEEE, 408–419.

[23] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *EMNLP* (2020).

[24] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[25] 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*. IEEE.

[26] Yuxian Gu, Xu Han, Zhiyuan Liu, and Minlie Huang. 2021. Ppt: Pre-trained prompt tuning for few-shot learning. (2021).

[27] Yuyu He, Lei Zhang, Zhemin Yang, Yinzhi Cao, Keke Lian, Shuai Li, Wei Yang, Zhibo Zhang, Min Yang, Yuan Zhang, et al. 2020. TextExerciser: feedback-driven text input exercising for android applications. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1071–1087.

[28] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

[29] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. 2023. Explainable Automated Debugging via Large Language Model-driven Scientific Debugging. *arXiv preprint arXiv:2304.02195* (2023).

[30] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2022. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. *CoRR* abs/2209.11515 (2022).

[31] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. 2018. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability* 68, 1 (2018), 45–66.

[32] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. CODAMOSA: Escaping coverage plateaus in test generation with pre-trained large language models. (2023).

[33] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight ui-guided test input generator for android. In *ICSE*. IEEE.

[34] 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.

[35] Jinzhi Liao, Xiang Zhao, Jianming Zheng, Xinyi Li, Fei Cai, and Jiuyang Tang. 2022. PTAU: Prompt Tuning for Attributing Unanswerable Questions. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval.* 1219–1229.

[36] Peng Liu, Xiangyu Zhang, Marco Pistoia, Yunhui Zheng, Manoel Marques, and Lingfei Zeng. 2017. Automatic text input generation for mobile testing. In *ICSE*. IEEE.

[37] 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 *ICSE*.

[38] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2022. Nighthawk: Fully Automated Localizing UI Display Issues via Visual Understanding. *IEEE Transactions on Software Engineering*, 1–16. https://doi.org/10.1109/TSE.2022.3150876

[39] Zhe Liu, Chunyang Chen, Junjie Wang, Yuhui Su, Yuekai Huang, Jun Hu, and Qing Wang. 2023. Ex pede Herculem: Augmenting Activity Transition Graph for Apps via Graph Convolution Network. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1983–1995.

[40] Zhe Liu, Chunyang Chen, Junjie Wang, Yuhui Su, and Qing Wang. 2022. NaviDroid: a tool for guiding manual Android testing via hint moves. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 154–158.

[41] Zhe Liu, Chunyang Chen, Junjie Wang, and Qing Wang. 2022. Guided Bug Crush: Assist Manual GUI Testing of Android Apps via Hint Moves. In *CHI 2022*. https://doi.org/10.1145/3491102.3501903

[42] Zhengwei Lv, Chao Peng, Zhao Zhang, Ting Su, Kai Liu, and Ping Yang. 2022. Fastbot2: Reusable Automated Model-based GUI Testing for Android Enhanced by Reinforcement Learning. In *ICSE*.

[43] 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*. 224–234.

[44] 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*. 94–105.

[45] Antonio Mastropaolo, Nathan Cooper, David Nader-Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2023. Using Transfer Learning for Code-Related Tasks. *IEEE Trans. Software Eng.* 49, 4 (2023), 1580–1598.

[46] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader-Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 336–347.

[47] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing combinatorics in GUI testing of android applications. In *ICSE*. IEEE.

[48] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*.

[49] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J Mooney, and Milos Gligoric. 2023. Learning Deep Semantics for Test Completion. *arXiv preprint arXiv:2302.10166* (2023).

[50] 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*. 153–164.

[51] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2022. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 1–18.

[52] Fabiano Pecorelli, Gemma Catolino, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. 2022. Software testing and Android applications: a large-scale empirical study. *Empirical Software Engineering* 27, 2 (2022), 1–41.

[53] Chao Peng, Zhao Zhang, Zhengwei Lv, and Ping Yang. 2022. MUBot: Learning to Test Large-Scale Commercial Android Apps like a Human. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 543–552.

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

[55] Julia Rubin, Michael I Gordon, Nguyen Nguyen, and Martin Rinard. 2015. Covert communication in mobile applications (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 647–657.

[56] Konstantin Rubinov and Luciano Baresi. 2018. What are we missing when testing our android apps? *Computer* 51, 4 (2018), 60–68.

[57] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. Adaptive test generation using a large language model. *arXiv preprint arXiv:2302.06527* (2023).

[58] J Schulman, B Zoph, C Kim, J Hilton, J Menick, J Weng, JFC Uribe, L Fedus, L Metz, M Pokorny, et al. 2022. ChatGPT: Optimizing language models for dialogue.

[59] Mohammed Latif Siddiq, Joanna Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. 2023. Exploring the Effectiveness of Large Language Models in Generating Unit Tests. *arXiv preprint arXiv:2305.00418* (2023).

[60] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.

[61] Abigale Stangl, Nitin Verma, Kenneth R Fleischmann, Meredith Ringel Morris, and Danna Gurari. 2021. Going Beyond One-Size-Fits-All Image Descriptions to Satisfy the Information Wants of People Who are Blind or Have Low Vision. In *The 23rd International ACM SIGACCESS Conference on Computers and Accessibility*. 1–15.

[62] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 245–256.

[63] Ting Su, Jue Wang, and Zhendong Su. 2021. Benchmarking automated GUI testing for Android against real-world bugs. In *FSE*.

[64] Saghar Talebipour, Yixue Zhao, Luka Dojcilovic, Chenggang Li, and Nenad Medvidovic. 2021. UI Test Migration Across Mobile Platforms. In *ASE*. IEEE, 756–767.

[65] Shin Hwei Tan and Ziqiang Li. 2020. Collaborative bug finding for Android apps. In *ICSE '20*. ACM, 1335–1347.

[66] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. 2022. Generating accurate assert statements for unit test cases using pretrained transformers. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*. 54–64.

[67] UIAutomator. 2021. Python wrapper of Android uiautomator test tool. https://github.com/xiaocong/uiautomator.

[68] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* (2017).

[69] Bryan Wang, Gang Li, Xin Zhou, Zhourong Chen, Tovi Grossman, and Yang Li. 2021. Screen2words: Automatic mobile UI summarization with multimodal learning. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. 498–510.

[70] Jue Wang, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. 2020. Combodroid: generating high-quality test inputs for android apps via use case combinations. In *ICSE*. 469–480.

[71] Wenyu Wang, Wing Lam, and Tao Xie. 2021. An infrastructure approach to improving effectiveness of Android UI testing tools. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 165–176.

[72] Wenyu Wang, Wei Yang, Tianyin Xu, and Tao Xie. 2021. Vet: identifying and avoiding UI exploration tarpits. In *FSE*. 83–94.

[73] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *FSE*. 959–971.

[74] Qing Xie and Atif M Memon. 2007. Designing and comparing automated test oracles for GUI-based software applications. *TOSEM* (2007).

[75] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. ChatUniTest: a ChatGPT-based automated unit test generation tool. *arXiv preprint arXiv:2305.04764* (2023).

[76] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10.

[77] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. 2018. Static window transition graphs for Android. *Automated Software Engineering* 25, 4 (2018), 833–873.

[78] Wei Yang, Mukul R Prasad, and Tao Xie. 2013. A grey-box approach for automated GUI-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 250–265.

[79] 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* (2021).

[80] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. 2021. Automated conformance testing for JavaScript engines via deep compiler fuzzing. In *Proceedings of the 42nd ACM SIGPLAN international conference on programming language design and implementation*. 435–450.

[81] Wei Yuan, Quanjun Zhang, Tieke He, Chunrong Fang, Nguyen Quoc Viet Hung, Xiaodong Hao, and Hongzhi Yin. 2022. CIRCLE: continual repair across programming languages. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 678–690.

[82] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. *arXiv preprint arXiv:2305.04207* (2023).

[83] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2016. Automated test input generation for android: Are we really there yet in an industrial case?. In *FSE*.

[84] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068* (2022).

[85] Ting Zhang, Ivana Clairine Irsan, Ferdian Thung, DongGyun Han, David Lo, and Lingxiao Jiang. 2022. iTiger: an automatic issue title generation tool. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1637–1641.

[86] Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, Jeffrey Nichols, Jason Wu, Chris Fleizach, et al. 2021. Screen recognition: Creating accessibility metadata for mobile applications from pixels. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–15.

[87] Liu Zhe, Chen Chunyang, Wang Junjie, Huang Yuekai, Hu Jun, and Wang Qing. 2020. Owl Eyes: Spotting UI Display Issues via Visual Understanding. In *2020 35rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE.

[88] 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*. IEEE.

[89] Kaiyang Zhou, Jingkang Yang, Chen Change Loy, and Ziwei Liu. 2022. Learning to prompt for vision-language models. *International Journal of Computer Vision* (2022), 1–12.