Integrating LLM-Based Text Generation with Dynamic Context Retrieval for GUI Testing

Juyeon Yoon
KAIST
Daejeon, Korea
juyeon.yoon@kaist.ac.kr

Seah Kim
Samsung Research
Seoul, Korea
sseah.kim@samsung.com

Somin Kim
KAIST
Daejeon, Korea
somin.kim@kaist.ac.kr

Sukchul Jung
Samsung Research
Seoul, Korea
sukchul.jung@samsung.com

Shin Yoo
KAIST
Daejeon, Korea
shin.yoo@kaist.ac.kr

***Abstract*—Automated GUI testing plays a crucial role for smartphone vendors who have to ensure that the widely used mobile apps–that are not essentially developed by the vendors–are compatible with new devices and system updates. While existing testing techniques can automatically generate event sequences to reach different GUI views, inputs such as strings and numbers remain difficult to generate, as their generation often involves semantic understanding of the app functionality. Recently, Large Language Models (LLMs) have been successfully adopted to generate string inputs that are semantically relevant to the test case. This paper evaluates the LLM-based input generation in the industrial context of vendor testing of both in-house and 3rd party mobile apps. We present DROIDFILLER, an LLM based input generation technique that builds upon existing work with more sophisticated prompt engineering and customisable context retrieval. DROIDFILLER is empirically evaluated using a total of 120 textfields collected from a total of 45 apps, including both in-house and 3rd party ones. The results show that DROIDFILLER can outperform both vanilla LLM based input generation as well as the existing resource pool approach. We integrate DROIDFILLER into the existing GUI testing framework used at Samsung, evaluate its performance, and discuss the challenges and considerations for practical adoption of LLM-based input generation in the industry.**

***Index Terms*—GUI Testing, Large Language Models, Test Automation**

I. INTRODUCTION

Android mobile platform is a massive ecosystem that accounts for approximately 70% of worldwide smartphone market [1]. The platform consists of multiple smartphone vendors and a diverse array of devices that are often customised to add value to the users. The success of the ecosystem and its resulting size, however, also contributed to the technical challenge known as device fragmentation [2], i.e., the combinatorial explosion in the number of vendor specific customisations, different hardware configurations, and rapidly evolving operating systems.

Device fragmentation poses a challenge not only to the individual app developers (who have to test their apps against a diverse array of configuration) but also for the smartphone vendors who develop software customisation layers, as none of the vendors control the app development environment exclusively. Rather, each vendor needs to ensure that the widely used, popular apps run well on their customised environment, resulting in their need to test 3rd party apps on their devices and customised configurations. Automated GUI testing can be of significant help to smartphone vendors, as it can reduce the cost of testing evolving 3rd party mobile apps. Samsung depends on its own GUI testing framework, and showed that some of the challenges from testing evolving apps developed by 3rd party developers, namely the View Identification Failure (VIF) issues [3], can be effectively addressed with the use of machine learning [4].

However, the challenges in automated GUI testing of 3rd party mobile apps do not stop at VIF issues. One of the longstanding challenges in automated GUI testing is how to generate meaningful inputs to emulate human users [5], [6], [7], [8]. Unlike other GUI events such as button clicks or scrolling, textfield inputs need to be semantically relevant to the app functionality as well as the test case. Randomly generated inputs may not pass the input validation, or may not achieve the testing objective even if it passes the validation. Further, certain functionalities require very specific input texts: for example, coupon codes (required to test store apps) or device model numbers (required to test specific search functionalities) are strictly dependant on external knowledge and test intentions. Samsung has developed a resource pool based technique for efficient GUI testing automation: we prepare a categorised pool of text inputs, and try to match GUI elements to existing input categories based on the context of the widget.

Recently, QTypist, a Large Language Models (LLMs) based input generation for GUI testing, has been proposed [9]. QTypist extracts local context from the GUI view that requires the input, and generates prompts based on the context as well as the type of the widget. Compared to the existing resource pool approach, an LLM-based approach has the benefit of not having to pre-generate all input values: they are instead generated on the fly by the underlying LLM. However, we also note that a purely LLM-based approach cannot *generate* the specific text inputs such as coupon codes, and it lacks customisability upon target user profile and domain knowledge. Especially for the smartphone vendors, the target domain of the apps to be tested often lies on the large scale of various domains, so the LLM-based approach should be able to adapt

This work has been supported by Samsung Electronics, the National Research Foundation of Korea (NRF) funded by the Korean government MSIT (RS-2023-00208998), as well as the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (RS-2022-II220995).

979-8-3315-0814-2/25 © 2025 IEEE

ICST 2025, Naples, Italy
Industry Track

Accepted for publication by IEEE. © 2025 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

to the diverse domains of target apps.

This paper presents DROIDFILLER, an LLM-based text input generation technique for GUI testing. DROIDFILLER advances the state-of-the-art LLM based input generation techniques in two ways. First, its prompt is derived not only from the local context of the input field, but also the general characteristics of the AUT (App Under Test): we prompt the LLM to reason about the app functionality, and subsequently asks it to generate a suitable text input based on its own reasoning. Second, following the underlying idea of the ReAct prompting technique [10], we allow DROIDFILLER to provide specifically customised text inputs, such as coupon codes and device serials, by calling external functions and retrieving the context based on the return values.

We have evaluated DROIDFILLER against two baselines: the existing resource pool approach, and a vanilla LLM-based technique that only uses local GUI context information. We have collected 120 text input fields from a total of 45 apps, including those developed in-house by Samsung as well as 3rd party apps. Based on human labelling of whether the generated input is semantically relevant or not, DROIDFILLER outperforms both baselines. An ablation study shows that both the more sophisticated reasoning based prompt strategy and the function-call based context retrieval contribute to the performance of DROIDFILLER.

Technical contributions of this paper are as follows:

- We show that advanced prompting and customisable dynamic context retrieval based on function calling can improve LLM based text input generation for GUI testing. DROIDFILLER can generate text inputs based on predetermined user profiles, or store specific coupon codes.
- We conduct an ablation study to show that each component of DROIDFILLER contributes to its performance.
- We report our experience of integrating an LLM-based text input generation technique with an industry-scale automated GUI testing framework with an empirical evaluation.
- We publicly release the implementation of DROIDFILLER[1] that can be run with DroidBot [11], an open-source GUI exploration tool. It provides customisable configurations for user profiles and context retrieval functions, allowing easy adaptation to the specific usage scenario. To the best of our knowledge, no such tool for GUI text input generation has been made publicly available.

The rest of the paper is organised as follows. Section II presents background information about the industrial GUI testing tool, and the relevant LLM prompt engineering techniques. Section III describes methodology, Section IV presents evaluation results, and Section V discusses remaining challenges for practical adoption of DROIDFILLER. Section VI discusses threats to validity, Section VII presents the related work, and Section VIII concludes the paper.

[1] https://github.com/coinse/droidfiller-android

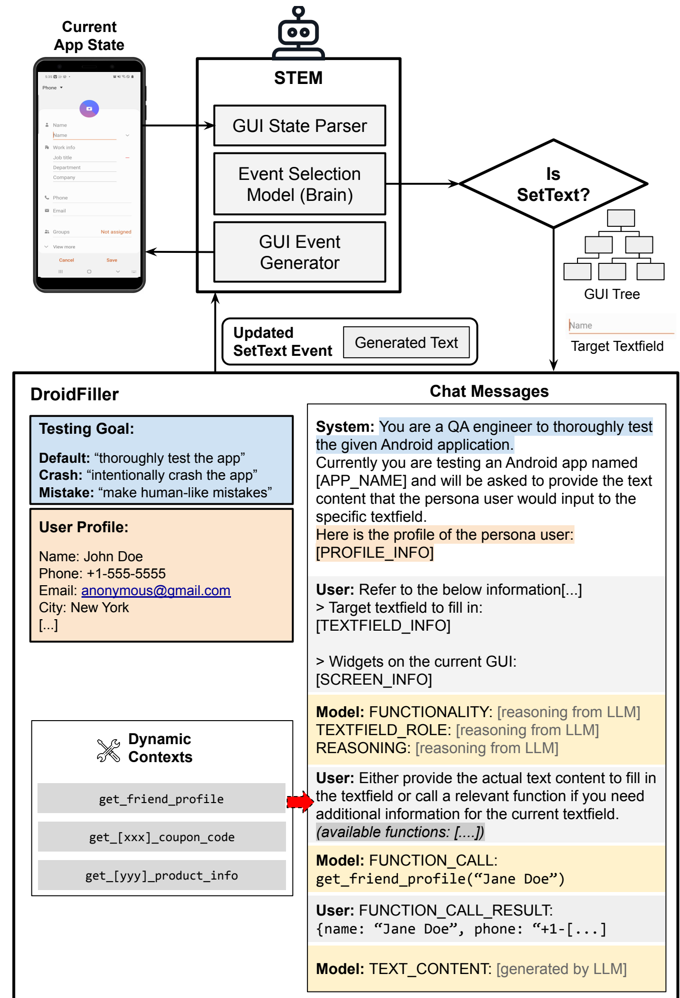

Fig. 1. Overview of DROIDFILLER integrated with STEM

II. BACKGROUND

This section describes the background of our work.

A. STEM: *Scenario-learnt Test Execution Model*

1) Learning-based GUI Exploration Strategy: STEM operates by selecting next UI event based on the current UI state and recent transitions. The basic exploration strategy of STEM refers to a public implementation of DroidBot [11], which systematically explores the GUI of an app. DroidBot employs a simple heuristic for selecting the next UI event: it prioritises UI elements that have not yet been interacted with, either in breadth-first or depth-first order. However, in Samsung's testing environment that typically involves testing a large number of diverse apps (for ensuring compatibility with the new device), this simple heuristic falls short in efficiently covering each app's "core" features within a fixed time budget.

Li et al. [12] have shown that deep-learning models trained on the UI transition data from human users can predict next events that a human user likely to trigger and the learning-based strategy help reaching more "important" states of AUT. Inspired by the technique, STEM incorporates a deep-learning model

that combines LSTM and CNN layers, totalling approximately 7 million parameters. The model is trained on usage patterns of the testers, which are internally collected during the manual GUI testing process in Samsung.

As the decision of prediction model is based on the current GUI state and previous interactions, STEM captures the screenshot of the current screen, and continuously records the transition history namely the sequence of UI events, throughout the exploration process. The model predicts both the location of the UI element to interact with and the type of the event. Currently, STEM supports the following types of UI events: Click, Scroll, SetText, and LongClick.

At every iteration of selecting the next UI event, STEM prioritises the event that the trained model assigns the highest probability. However, when the current GUI state differs significantly from the training data, the model may struggle to predict the next event with high confidence. In these instances, STEM falls back to the heuristic-based strategy. Initially, STEM checks if any UI elements on the current screen trigger a transition to a new screen. If so, STEM prioritises the element that facilitates this transition. If there is no such UI element, STEM randomly selects one of the UI elements that have not been interacted with yet.

STEM applies additional optimisation for the pages that include textfields. If the target GUI state contains one or more textfields requiring input, and thus necessitates a SetText event, STEM gives priority to textfields that have not yet been filled in. This approach is adopted because textfields often need specific text content for subsequent UI transitions, such as search queries, sign-in forms, etc.

2) Text Input Generation Policy: STEM is equipped with a separate module for text input generation activated each time a SetText event is generated. Currently, STEM supports two types of text generation policies: *random* and *context*. Random policy generates an arbitrary string without considering any context information, whereas context policy is based on manually crafted textfield categories (e.g., email, name, search, etc.) and the similarity-based category matching scheme.

Specifically, each category is defined with a set of representative words and a corresponding value pool. For example, a category "username" is represented by keywords `[user, ID, username]`, and any textfield matching this category is filled with content from the value pool, such as an arbitrary username like "tester123". Category matching is basically ranking all available categories with respect to a target textfield based on their similarity. The textfield is represented as its textual properties: for Android, text, content_description, and resource_id attributes. Since both the category and textfield are represented as a set of tokens, the two directions of pairwise similarity sets can be obtained: $S^i_{c_x}$ and $S^i_{t_y}$, which are basically sets of maximum similarities between a token from the target tokenset and the tokens from the counterpart.

In detail, each token is vectorised with FastText [13] language model with an aim of capturing *semantic* relevance (e.g., "place" and "location" is lexically far from each other but the semantic embeddings of two words are expected to be similar), and their cosine similarities (denoted as cos_sim) are calculated. Formally, where τ_c denotes a set of keyword tokens of c_x, and τ_t denotes a set of tokens extracted from t_y,

$$S^i_{c_x} = sort_desc(\{max_{t_j \in \tau_t} cos_sim(vec(t_i), vec(t_j) | t_i \in \tau_c\})$$

and

$$S^i_{t_y} = sort_desc(\{max_{t_i \in \tau_c} cos_sim(vec(t_i), vec(t_j) | t_j \in \tau_t\})$$

.

Then, the similarity between c_x and t_y is defined as an aggregation of similarity values in both directions, as described in Equation 1. Additionally, we prune the pairwise similarities with K highest values.

$$sim_{agg}(c_x, t_y) = \frac{1}{2} \times (\frac{\sum_{i=1}^{K} S^i_{c_x}}{min(|\tau_c|, K)} + \frac{\sum_{i=1}^{K} S^i_{t_y}}{min(|\tau_t|, K)}) \quad (1)$$

STEM currently incorporates the category matching scheme as an additional mode to generate meaningful rather than random text inputs: it maintains 57 categories identified from the subset of their AUTs.

B. Large Language Models

Large Language Models (LLMs) have shown promising results in generating human-like reasoning and answers in wide range of domains, including software testing automation [14], [15], [9]. The GPT-3 and GPT-4 models, including its versions fine-tuned for source code [16], [17], [18] has demonstrated proficiency in generating human-like text based on given contexts. This capability has been leveraged in GUI testing, where the model is fine-tuned with specific input-output examples to generate text contents from natural language screen descriptions [9], [19]. In this line of research, fine-tuning is often considered essential for executing domain-specific tasks. However, this approach may limit the model's generalisability across various apps and usage scenarios.

Modern designs of Large Language Model-based architectures increasingly emphasise in-context learning. This approach offers greater flexibility in adapting the model to a target domain by incorporating task descriptions and examples directly into the prompt [16], [20]. Within this concept, even large-scale external databases can be adopted and we can construct prompts by retrieving relevant information for the target task; Retrieval-Augmented Generation, as described in Lewis et al. [21] enhances tasks that demand specific and factual knowledge. It achieves this by dynamically retrieving relevant documents from sources like Wikipedia based on the given context. Chain-of-Thought [22], a prompting technique encourages the articulation of intermediate reasoning steps, has gained wide adoption and has been proven effective in enhancing the quality of answers across various tasks [23]

Recent efforts have focused on integrating external tools, like search engines and APIs, into the iterations of LLM prompting. ReAct [10], for example, combines LLMs' reasoning and acting capabilities in an interleaved fashion. The aforementioned prompting strategies can be implemented with "function calling"

features provided by the latest models from OpenAI [24], or several LLM-powered application building frameworks, such as AutoGPT [25], and LangChain [26]. For our text input generation task, we employ a combination of prompt engineering techniques and frameworks associated with LLMs, integrating the concepts of Chain-of-Thought and Retrieval-Augmented Generation alongside the function call feature. Compared to the fine-tuning based approaches, the retrieval-augmented generation offers more flexibility, especially in tailoring the model to predefined user profiles and domain-specific data, which may vary across AUTs. Properly guiding reasoning steps enables the model to generate accurate text contents in the desired formats, eliminating the need for fine-tuning with pre-labelled data.

III. Methodology

In this section, we describe the structure of prompt template used in DroidFiller and the overall workflow, which is illustrated in Figure 1.

A. Initialising System Prompt with User Profile

A real-world usage of mobile apps commonly requires a user to provide its personal information to the app, such as name, email, phone number, typically through the sign-in process. However, most of the existing GUI testing techniques do not explicitly consider this aspect and usually bypass the sign-in process with manually crafted scripts [27], [9], [28], [29], [12] or a snapshot of the intermediate app state [30].

Instead, DroidFiller incorporates a target user profile, encompassing personal information, account credentials, and personality descriptions in its dynamically constructed prompt, to guide the model in using this persona information when generating text inputs. This simple strategy naturally enables the model to correctly fill textfields requiring personal information like usernames and passwords, thus successfully passing initial input validation. Compared to using prepared login scripts, DroidFiller mitigates the risk of inconsistent app behaviour in the login process and eliminates the manual effort needed to prepare login scripts for each AUT.

The user profile is contained in a system message that is sent to the chat-based model, OpenAI gpt-3.5-turbo model [31] in our experiment. In Table I, we provide the structure of the system message: tester description, and profile information. The resulting system message is the concatenation of these two parts.

B. Generating Reasoning Steps for Screen Comprehension

The first user message sent to the model includes information about the target textfield and the current GUI state, and the model is asked to provide the reasoning about the text content to be inputted to the textfield. We provide the message structure in Table II. The model is expected to generate an output following the given reasoning template, which calls for an inference of the app functionality, and the role of the target textfield based on the given GUI information.

TABLE I
Structure of DroidFiller's System Message

Context Type	Example
Tester Description	You are a QA engineer to thoroughly test the application. Currently you are testing an android app named X
Profile Information	Here is the profile of the persona user you are going to adopt for testing: - name : John Doe - phone: (646)555-3890 - email: anonymous@gmail.com - username: anonymous - password: Pwd123456@[...]

TABLE II
Structure of DroidFiller's Initial User Message

Context Type	Example ('Recipe Keeper' App)
Instruction	Refer to the below information and follow the provided steps to fill in the given textfield.
Target TextField Description	> Target textfield to fill in: a textview that has resource_id " txtTitle" (Widget ID: 23)
GUI State Description	> Widgets on the current GUI state: I see the following widgets from top to bottom: a textview that has text "Add meal" (Widget ID: 9)a button that has content description "Navigate up" (Widget ID: 8)[...]
Reasoning Template	=== Below is the template for your answer === FUNCTIONALITY: < briefly describe the functionality of the Recipe Keeper that the current GUI state is about> TEXTFIELD_ROLE: <briefly describe the role of the target textfield> REASONING: <briefly describe the reasoning process (1-2 sentences) to fill in the textfield. Consider following questions: What is the textfield for? Is there any function that can provide relevant information for the textfield?>

1) GUI State Description: DroidFiller provides the textual description of the GUI state to the model. First, it assigns a unique integer ID to all the visible widgets on the current GUI state, and then it describes each widget with its textual properties: text, content_description, and resource_id. In addition, DroidFiller partially gets use of the widget type to indicate each widgets; for example, EditText type widgets are referred as "textfield" in the description, Button type widgets are referred as "button", CheckBox type widgets are referred as "checkbox", and so on. The undefined widget types are referred as the default indicator, "widget".

The natural language descriptions of the visible widgets are concatenated in the order of their vertical positions on the screen (top to bottom), and the resulting description is provided to the model as a part of the user message.

2) Reasoning Template: A naive way to query a language model to generate text contents would be to provide the information about the target textfield and the current GUI state with a simple command, such as "Generate a suitable text content to

fill in the target textfield". However, one issue of this zero-shot prompting is that the outputs of the model are not guaranteed to be in the consistent format. For example, the model may add some redundant comments along with the generated text content in its output, such as, `The text input should be "John Doe" (The profile name)`; here, it becomes challenging to automatically parse the pure text content.

Moreover, because of the lengthy prompt containing the entire GUI screen description, the model may struggle focusing on the relevant information for the target textfield and misunderstand the overall GUI context; this occasionally results in the generation of irrelevant text content. Hence, we employ a step-by-step reasoning template to guide the model to follow the rational thinking process and generate output in a consistent format. This template is inspired by the Chain-of-Thought prompting strategy [23], but is specifically tailored for our task of generating text content related to the GUI context, ensuring that the model follows the designated reasoning steps. In response to the the first user message, the LLM instance in DROIDFILLER generates the reasoning following the given template; till this point, the model has not confirmed the final text content to be inputted to the textfield yet.

```
FUNCTIONALITY: The Recipe_Keeper app allows users to create
     and save meal recipes with detailed information such as
      title, date, meal type, serving size, and notes.

TEXTFIELD_ROLE: The target textfield with the text "
     Delicious Pasta Carb..." and resource_id "txtTitle" is
     used to enter the title of the recipe.

REASONING: To fill in the target textfield, I will provide a
      unique title for the recipe.
```

Then, a followed user message, structured as Table III, is added to the conversation thread, asking the model to either provide the final text content or call one of the given functions. Details about the function call-based context retrieval is described in Section III-C.

TABLE III
STRUCTURE OF DROIDFILLER'S FOLLOWED USER MESSAGE

Context Type	Example ('Recipe Keeper' App)
Instruction	`Now, either provide the actual text content to fill in the textfield or call a relevant function if you need additional information for the current textfield, and there is a function that can provide the information.`
Enforcing Output Template	`If you are going to immediately generate the text content, provide the text content with the prefix "TEXT_CONTENT:" in a single line, and do not include anything else in your answer except the text content.`

In this example with the 'Recipe_Keeper' app, the model instantly generates the final text content in the requested format, which is the title of the recipe:

```
TEXT_CONTENT: Creamy Chicken Alfredo
```

C. Dynamic Context Retrieval

Large language models are known to be suitable for generating answers that follow common knowledge [32], but not for generating domain-specific information that requires external knowledge [33], [10]. For example, the model may not be able to generate a specific coupon code for a store app, if the valid coupon codes are not included in the prompt or the part of training data (which is not even desirable to be included in the training data if we regard privacy concerns).

Moreover, including all the possible domain-specific information in the prompt is not feasible with the limited length of the prompt, and may confuse the model to focus on the relevant information for the target textfield. DROIDFILLER instead adopts the function call-based context retrieval, which allows the model to selectively and dynamically refer to the external information sources when generating text contents.

For the original text generation policy of STEM described in Section II-A2, a category value pool has been curated by internal engineers by Samsung based on their general target AUTs so that there are several categories that require certain predefined ingredients (e.g., coupon codes, device model numbers, etc.). Based on them, we define a set of functions as Table IV that can provide domain-specific information for our experiment on in-house apps. Note that in our publicised implementation, the function set can be customised by users, as they can write their own function definitions as part of the configuration file.

TABLE IV
LIST OF THE DEFINED FUNCTIONS THAT CAN BE INVOKED BY LLM USED IN OUR EXPERIMENT FOR IN-HOUSE APPS.

Function Name/Arguments	Description
`get_friend_profile(name)`	Returns the profile of a specific person with given name
`get_samsung_product_info(type)`	Returns an example product information of Samsung
`get_galaxy_store_coupon_code()`	Returns a set of valid coupon codes of store Y (an internal app of Samsung)

The description (i.e., required arguments, textual explanation about the returned data) about the available functions are given along with the followed user message described in Table III., which is after the generation of intermediate reasoning from the model. Note that the model is not guaranteed to call a function with the correct arguments although it is provided with the full function specification. In such cases, DROIDFILLER detects the ill-formed function call and discards the response. Subsequently it re-tries the function call or the final text input until a valid function call response is generated.

D. Integration to Industrial GUI Testing Tool

To integrate DROIDFILLER into the STEM, the GUI testing tool developed by Samsung as described in Section II-A, we

target the text input generation policy of STEM. We implement DroidFiller as a separate module that can be plugged into the existing STEM framework, and the module is activated when STEM generates a SetText event. Note that the SetText events are prioritised over other types of events when STEM encounters a GUI state that includes one or more textfields to be filled in. The DroidFiller module receives the information about the target textfield and the current GUI state from STEM, specifically in the format of dictionary, and then it constructs a prompt by applying the information to the predefined prompt template. As the resulting prompt (in the form of chat messages) queries the LLM instance, the LLM instance generates the text content to be inputted into the textfield, and DroidFiller sends the generated text content back to STEM. Finally, the SetText event with the generated text content is executed on the testing device.

IV. Evaluation

This section describes our experimental setup and presents the results of our evaluation on DroidFiller.

A. Research Questions

We aim to answer the following research questions.

RQ1. Text Generation Quality: How realistic are the text contents generated by DroidFiller compared to existing methods? RQ1 aims to assess the quality of generated text inputs from DroidFiller and conventional approaches, determining whether each text is likely to be entered by a human.

RQ2. Dynamic Context Retrieval How effective is the function call based context retrieval? With RQ2, we aim to examine the efficacy of function call-based context retrieval. This involves verifying if domain-specific contexts are accurately retrieved and utilised to generate meaningful text contents, potentially leading to the discovery of new app states.

RQ3. Ablation How does each component of DroidFiller impact the text generation quality? With RQ3, we aim to assess the contribution of reasoning template prompting and function call components to the overall text generation quality.

RQ4. Testing Enhancement How does DroidFiller enhance an industrial GUI exploration tool? With RQ4, we aim to investigate the impact of integrating human-like text inputs from DroidFiller into an industrial GUI exploration tool. We assess whether STEM, when integrated with DroidFiller, covers a broader range of screens compared to its original text input setup.

B. Experimental Setup

1) Subjects: To evaluate the quality of generated text contents by DroidFiller (RQ1-RQ3), we collect a total of 120 textfield from 45 apps including 11 internal apps from Samsung. The 3rd party apps are collected from Google Play Store and FDroid [34]. The full list of the apps and the dataset is available in our public repository. The textfields are collected by manually exploring the apps and identifying the textfields that require specific text contents to fill in. Specifically, we collect pairs of the textfield information and the entire GUI screen states. For RQ4, we use a separate set of apps that satisfy the following conditions: an app should be runnable in our target device as STEM does not support emulators, and should have more than three textfields. We select total of 10 Android apps; detailed information are provided in Table V.

The *Dynamic Context Retrieval* strategy in DroidFiller allows for customisation of the function set based on the domain-specific requirements of the target AUT. For a subset of the subject apps, we defined additional context-providing functions: `get_ampache_server_info` for AmpachePlayer and `get_nextcloud_server_info` for NextCloudTalk, which include feasible server addresses and credentials. Both apps require connection to a dedicated server by specifying a unique address. We also added this server address information and credentials to the original STEM's value pool to ensure a fair comparison.

2) Metrics: We measure the quality of generated text contents by the ratio of realistic text generations. We consider generated text as realistic if, after manual assessment, we decide that a human could have entered the same text. Three of the authors have independently labelled the generated texts, and discussed to reach a consensus.

For the function call-based context retrieval (RQ2), we assess how accurately function call invocations are generated by LLMs. We manually label whether a textfield requires calling one of the provided function, and check whether DroidFiller actually invoked the required function during the text generation process. Further, we report the precision, recall, and F1-score of each function and the aggregated scores for all the functions.

To assess the impact of DroidFiller integrated with STEM on the test effectiveness (RQ4), we adopt UI-based coverage metrics: the number of reached activities [35] and the number of unique states [36] discovered during exploration. When counting covered activities and states, we only consider the activities that belong to the target app, as well as and the states from those internal activities.

Given the latency of querying an LLM, it may not be fair to compare results under the same time budget: STEM+DroidFiller may only trigger a smaller number of events due to the latency. Instead, we have matched the number of UI events created. Both configurations ran for 90 minutes, and we synchronised the length of the exploration sequences by trimming them to the length of the shorter one. In the future, such latency may be shortened by replacing API calls with a more lightweight local LLM. To reduce potential bias, we run each configuration three times and record the highest performance in terms of reached activities as done in previous studies [37].

3) Implementation: We implement DroidFiller as a separate Python module that can be plugged into the existing STEM framework. As explained, the module is activated when STEM generates a SetText event. We use the official OpenAI API to get access to the GPT-3.5-turbo-0613 model [31], which is a chat-based completion model that supports to receive definitions of the custom function definitions and pretrained to

TABLE V
ANDROID APPS USED IN DROIDFILLER'S TESTING EFFECTIVENESS EVALUATION.

App ID	Simplified Name	Category	# of Activities	App ID	Simplified Name	Category	# of Activities
oss.krtirtho.spotube	Spotube	Multimedia	7	org.tasks	Tasks	Writing	46
luci.sixsixsix.powerampache2.fdroid	AmpachePlayer	Multimedia	1	com.ichi2.anki	Anki	Education	29
com.money.manager.ex	MoneyManager	Money	48	com.moimob.drinkable	Drinkable	Writing	1
com.nextcloud.talk2	NextCloudTalk	Connectivity	30	moe.dic1911.urlsanitizer	URLSanitizer	System	4
lying.fengfeng.foodrecords	FoodRecords	Health	2	ru.aeroflot	Aeroflot	Flight Management	33
org.asafonov.monly	Monly	Money	1	io.github.zwieback.familyfinance	FamilyFinance	Money	30

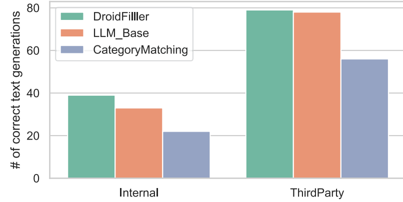

Fig. 2. Comparison on the number of realistic text generations between DROIDFILLER and baseline techniques.

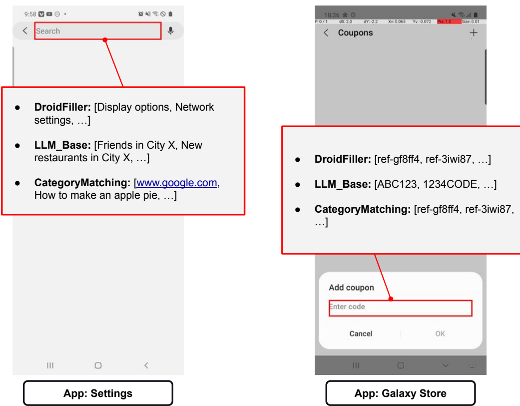

Fig. 3. Example text generations from different techniques.

invoke the functions when necessary. We use Galaxy Fold Z 2 as a device for running STEM.

C. Results

1) Text Generation Quality (RQ1): Among the total 120 investigated textfields, our manual assessment confirms that DROIDFILLER generates the correct (i.e., whether the text is likely to be entered by a human using the app) text contents for the 118 cases (98.3%), while the basic LLM prompting succeeds to generate correct inputs for 111 cases (92.5%) and the category matching baseline produces correct inputs for 78 cases (65%). Figure 2 compares the performance on realistic text generation for the two dataset types: Internal, ThirdParty. For textfields from 3rd party apps, DROIDFILLER and basic LLM prompting show similar performances, but DROIDFILLER generates meaningful contents where the basic LLM prompting does not. We provide such cases in Figure 3.

For the search field of the "Settings" app, DROIDFILLER correctly generates a search query for finding corresponding setting options, but the two other techniques generate general search queries, which do not yield any search results. Although the app name, "Settings" is provided as a prompting context both for DROIDFILLER and LLM_base, the result implies that enforcing proper reasoning steps help refer to the relevant information in the long prompting context. The other example textfield, the coupon code box of the "Galaxy Store" app, receives a specific set of codes; the codes cannot be inferred via general knowledge. Different from LLM_base, DROIDFILLER is able to call a function which loads the available coupon codes. DROIDFILLER correctly invokes the function and outputs the predefined coupon codes based on the returned result of the function. In this case, CategoryMatching baseline also produces correct answers because the "coupon code" category and its value pool is defined and the textfield is predicted to be a correct category.

TABLE VI
ACCURACY OF EACH FUNCTION CALL BY LLM USED IN OUR EXPERIMENT.

Function Name / Aggregation	Precision	Recall	F1-score
`get_friend_profile`	0.31	0.65	0.42
`get_samsung_product_info`	0.14	1.00	0.25
`get_galaxy_store_coupon_code`	1.00	1.00	1.00
Micro-averaged	0.27	0.71	0.39
Macro-averaged	0.48	0.88	0.56

2) Dynamic Context Retrieval (RQ2): In our experiment, we provide three functions that provide additional context information to infer correct text inputs. We first manually label whether a textfield requires calling one of the provided function, and check whether DROIDFILLER invoked any function during the text generation process for each textfield (total 120 textfields). We treat this as a ternary classification problem and report the accuracy of function calling in Table VI.

The relatively low precision for the functions "get_friend _profile" and "get_samsung_product_info" indicates that there are significant number of textfields that unnecessarily invoked those functions. However, the low precision does not necessarily mean that DROIDFILLER generates inappropriate texts when an irrelevant function is falsely called; in fact, DROIDFILLER eventually succeeds to generate at least one correct text input for every cases that called irrelevant function calls. For example, although DROIDFILLER attempts to call "get_friend_profile" on the textfield that requires a TV show name to be inputted, it finally generates a correct show title, "Friends".

However, we also observe several cases that the incorrect function calling leads to an incorrect text input as well; we plan to reinforce the reasoning process on function calling,

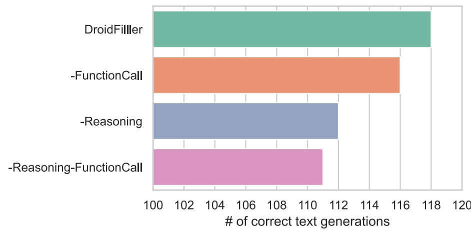

Fig. 4. Comparison on the number of realistic text generations among DROIDFILLER and its ablation settings

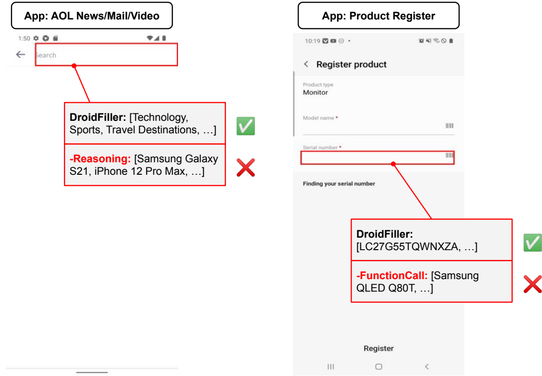

Fig. 5. Example text generations from different ablation settings.

possibly by enforcing LLMs to provide corresponding reason about why the additional information is needed. In contrast, the recall is relatively high, which highlights the capability of DROIDFILLER detecting a necessity of additional information for the target textfield and the overall screen state.

3) Ablation (RQ3): Figure 4 compares the number of realistic text generations among DROIDFILLER and its ablation settings. The ablation settings are as follows: *-Reasoning* removes the reasoning step template from the prompt and directly asks the model to either generate the text content or invoke a function call. *-FunctionCall* removes the function call-based context retrieval so it has a reasoning step but does not aware of any functions that can provide additional information. *-Reasoning-FunctionCall* removes both the reasoning step template and the function call-based context retrieval.

Compared to the original DROIDFILLER setting (generates 118 realistic text contents among total 120 textfields), DROIDFILLER without the reasoning step template generates 112 realistic text contents, DROIDFILLER without the function call-based context retrieval generates 116 realistic text contents, and DROIDFILLER without both the reasoning step template and the function call-based context retrieval generates 111 realistic text contents. The result implies that both the reasoning step template and the function call-based context retrieval contribute to the overall text generation quality, but the reasoning step template seems to be a more crucial component on generating better quality text contents.

In Figure 5, DROIDFILLER without reasoning step tends to

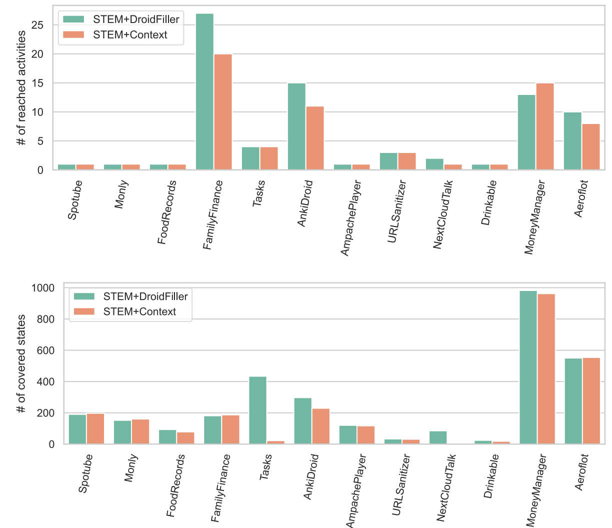

Fig. 6. Comparison on the number of reached activities (top) and discovered unique states (bottom) between DROIDFILLER and STEM with its original text generation configuration, *context*.

struggle to generate valid search queries, because this requires capturing the overall app context (search for any news, mail, or video). On the other hand, DROIDFILLER without function call-based context retrieval fails to generate a valid serial number for the target product, which is only given as an additional context delivered by the result of function call (specifically, `get_samsung_product_info` function).

4) Testing Enhancement (RQ4): RQ4 concerns the actual impact on test effectiveness brought forward by DROIDFILLER integrated with the underlying industrial GUI exploration tool, STEM. Our initial hypothesis is that DROIDFILLER will contribute to increasing test coverage by helping to pass input validation process, in turn allowing STEM to discover more screens. For brevity, we will henceforth refer to STEM integrated with DROIDFILLER as DROIDFILLER.

Figure 6 shows the number of activities and states covered by each scheme. For specific apps, such as FamilyFinance, AnkiDroid, NextCloudTalk and Aeroflot, DROIDFILLER generally covers more activities, ranging from 25% to 100% more than the original STEM. A Wilcoxon signed rank test (one-sided) indicates that the number of activities covered by DROIDFILLER was statistically significantly higher than those covered by original STEM ($p = 0.015$). However, interestingly, the statistical test does not confirm that DROIDFILLER can cover more states when compared to the original ($p = 0.124$). In cases where both configurations cover the equal number of activities, DROIDFILLER discovered more states in five apps, but fewer states in two apps. There are also cases where DROIDFILLER covers more activities but discovers fewer states. This occurs when STEM triggers early activity transitions by reaching the necessary state to pass the current screen—for example, when all required text inputs are correctly filled—rather than exhaustively exploring state variations within the same activity. In such cases, activity coverage more accurately

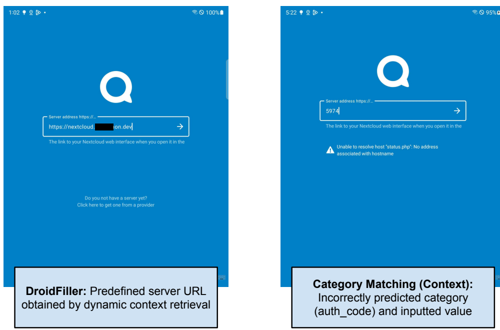

Fig. 7. Comparison on the filled textfield contents during the actual GUI testing process between the original STEM and DROIDFILLER.

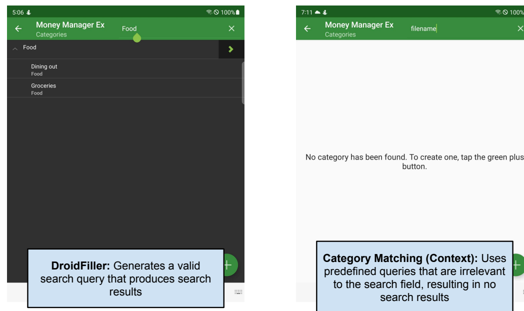

Fig. 8. Comparison on the filled textfield contents between the original STEM and DROIDFILLER in terms of search query generation.

reflects the true diversity of exploration. Additionally, we note that the number of unique states discovered can be significantly inflated by UI components that introduce minor variations, such as calendar pickers.

Overall, we observe partial improvements in testing effectiveness with DROIDFILLER according to the metrics we adopted. The results suggest that a careful orchestration is needed when integrating LLM-based input generator to automated GUI testing tools. We refer readers to Section V for discussions on how DROIDFILLER facilitates accesses to more screens, as well as on the cases for which DROIDFILLER produces both correct and incorrect text content.

V. DISCUSSION

Here we discuss the results of empirical evaluation in depth.

A. *How* DROIDFILLER *Assists the Testing Process*

Figure 7 demonstrates the robustness of DROIDFILLER. Although we provide the same server URL to the value pool of the original STEM in the existing category name "URL", the category predicted by the *context* strategy is, incorrectly, "Auth Code". This is because the scope of the surrounding texts around the target textfield can vary according to the layout design, and contain irrelevant tokens as well. It is worth noting that, even if the category is correctly predicted, the original STEM would not be able to handle more sophisticated apps that receive multiple URLS. DROIDFILLER, on the other hand, generates an appropriate URL using the provided functions.

Figure 8 shows that DROIDFILLER successfully fills in the search boxes within a budget planner app, with a valid search query that yields the available expense category list, whereas the original STEM fails to do so. Search query are particularly challenging for STEM to generate, as it is not feasible to maintain a diverse value pool for all search functionalities.

Some textfield inputs require a specific format for better parsing, such as phone numbers with/without the country code prefixes, with/without dash (-) splitters, and datetime values in various possible formats (YYYY/MM/DD or DD-MM-YY). The original STEM is limited to using a predefined value in the corresponding category as it is, and cannot re-format the content to adapt to the target textfield. On the other hand, DROIDFILLER can flexibly adapt the data to the desired format. Figure 9 shows that DROIDFILLER can generate date inputs in different formats according to the existing placeholder values, whereas STEM tries to use the provided value "2021-01-01" as it is, resulting in the textfield remaining empty after attempting to use the incorrectly formatted value.

B. *Challenges of Integrating* DROIDFILLER

Analysis of 981 text inputs generated for 12 apps reveals that 857 of them (87.4%) are valid. The remaining invalid inputs typically due to the misunderstanding of the intended use of the text field. For example, DROIDFILLER may use a dollar sign ($) when only numbers are expected. Often, the app either discards these incorrect inputs (i.e., the target textfield remains unchanged), or displays error messages. One future improvement would be a separate component of DROIDFILLER that exclusively monitors app responses to input errors, and feed them back to the LLM. This may allow for subsequent attempts by DROIDFILLER to be more accurately based on observed app feedback. Another common cause is that DROIDFILLER misses the larger context, such as the previously visited screens. We envision automated summarisation of previous navigation path to improve DROIDFILLER against these cases.

Despite the high proportion of valid inputs (87.4%), there are cases when DROIDFILLER still fails to result in better GUI exploration. Analysis of such exploration reveals further challenges. First, STEM sometimes makes repeated requests to DROIDFILLER to generate inputs for the same textfield. This is because sometimes minor screen changes cause STEM to believe that there is a new screen. To address this, we need to improve STEM's screen identification. Second, even after filling in all the required textfields, the app might not progress if the subsequent event generated by STEM does not process the entered texts. For example, while the app might require a "submit" button to progress, STEM might inadvertently press a "cancel" button, discarding all entered text. In order to avoid this, we envision a dedicated module that captures and maintains exploration contexts, to avoid meaningless event sequences such as filling in a form only to cancel.

Fig. 9. Comparison on the filled textfield contents between the original STEM and DROIDFILLER in terms of flexible text formatting.

C. Model Capability

In our experiment, we use the GPT-3.5-turbo model from OpenAI; however, our approach is not limited to this specific model. Although the dynamic context retrieval of DROIDFILLER currently relies on the function call feature of OpenAI API [38], this function-calling capability is also available with other LLMs [39], and external frameworks like LangChain [40] provide a tool call feature for various chat-based LLMs where function calls are not natively supported.

We have also preferentially considered textual descriptions of the GUI state as input to the model. However, recently released large language models, such as GPT-4V, can also accept visual inputs [41]. While providing accurate positional information for each widgets on the screen can be helpful, "hidden" textual information, like resource IDs or content descriptions, may still contribute to text generation. As future work, we plan to explore the potential of visual inputs.

VI. THREATS TO VALIDITY

Threats to internal validity concern whether the experimental results support the claims about cause and effect. While we empirically demonstrate that the LLM based text input generation is more realistic for GUI testing of Android apps, the criteria for the realism remains inherently human and subjective. We tried to mitigate this concern by labelling the results collectively based on multiple inputs. Threats to external validity concern whether the results warrant generalisation. The findings are limited to the apps we studied and the specific LLM instance used in our technique. However, since DROIDFILLER is drawing from common knowledge found in LLMs and not any app specific analysis, we expect its ability to generalise to other target apps as long as they are common apps used by the majority of users. Further evaluations using a wider selection of apps, as well as more LLMs including open source models, will help mitigating this concern.

VII. RELATED WORK

This section presents existing related work.

A. String Input Generation for Testing

Generating string test input has been a longstanding challenge in automated testing in general. McMinn et al. [42] used web search results to find meaningful string values in the context of automated unit test generation for Java. Afshan et al. [43] later applied an entropy-based language model to generate readable strings, as the raw generated strings tend to be sequences of random characters and symbols. We also use a language model, but specifically a large pretrained one that allows us to prompt with much longer context.

TextExerciser [44] introduces an interesting twist to text input generation: it combines UI analysis with machine learning to identify textfield constraints, then generates inputs using a mutation-based strategy. It applies an iterative approach which allows for continuous input generation, adapting to new feedback, even after failures.

B. GUI Testing with Large Language Models

LLMs have demonstrated feasibility in automating or assisting GUI testing so far. QTypist [9] proposed an approach to filling in the textfields on the current GUI state by fine-tuning the GPT-3 model [16], [45], and InputBlaster [46] adopted mutation strategy to generate crash-inducing inputs from LLMs. However, the generated texts solely rely on the training data, and the tester has no control over the generated text contents and varying target apps; DROIDFILLER enables the tester to provide domain-specific knowledge based on the target AUTs or inject the user persona information.

Wen et al. [36] extended DroidBot [11] to generate a sequence of GUI actions from a task description through LLM prompting, and GPTDroid [37] proposed that a chat-like interaction with an LLM can be used to generate a sequence of plausible GUI actions. DroidAgent [47] suggested a multi-agent architecture with diverse prompts to generate high-level testing scenarios and the low-level GUI actions at once.

Note that, despite the advances described above, we think integrating LLM-based text generation with an existing GUI testing tool is still a valid approach towards GUI testing. LLM usage in industry context brings the additional challenges of time and monetary cost, as well as data sensitivity. A more focused use of LLM for a task it can perform the best, i.e., text generation, can help alleviating these issues.

VIII. CONCLUSION

We present DROIDFILLER, an LLM based text input generation technique designed for GUI testing of Android apps. By exploiting the emergent behaviour of LLMs, DROIDFILLER can generate realistic text inputs based on contextual information about the current GUI states. DROIDFILLER adopts advanced prompting strategies, such as Chain-of-Thoughts and ReAct, to provide accurate text inputs as well as domain specific information that cannot be generated solely by LLMs. An evaluation of DROIDFILLER using inhouse apps developed by a smartphone vendor, Samsung, as well as 3rd party apps, suggest that DROIDFILLER can effectively generate realistic textual inputs for GUI testing.

REFERENCES

[1] IDC, "Smartphone market share https://www.idc.com/promo/smartphone-market-share/os (Last checked: 23 October 2024)," October 2024.

[2] L. Wei, Y. Liu, and S.-C. Cheung, "Taming android fragmentation: Characterizing and detecting compatibility issues for android apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 226–237. [Online]. Available: https://doi.org/10.1145/2970276.2970312

[3] R. Coppola, E. Raffero, and M. Torchiano, "Automated mobile ui test fragility: An exploratory assessment study on android," in *Proceedings of the 2nd International Workshop on User Interface Test Automation*, ser. INTUITEST 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 11–20. [Online]. Available: https://doi.org/10.1145/2945404.2945406

[4] J. Yoon, S. Chung, K. Shin, J. Kim, S. Hong, and S. Yoo, "Repairing fragile gui test cases using word and layout embedding," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 291–301.

[5] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?(e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 429–440.

[6] K. S. Said, L. Nie, A. A. Ajibode, and X. Zhou, "Gui testing for mobile applications: objectives, approaches and challenges," in *Proceedings of the 12th Asia-Pacific Symposium on Internetware*, 2020, pp. 51–60.

[7] T. Su, J. Wang, and Z. Su, "Benchmarking automated gui testing for android against real-world bugs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 119–130.

[8] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, "Automatic text input generation for mobile testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 643–653.

[9] Z. Liu, C. Chen, J. Wang, X. Che, Y. Huang, J. Hu, and Q. Wang, "Fill in the blank: Context-aware automated text input generation for mobile gui testing," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1355–1367.

[10] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," *arXiv preprint arXiv:2210.03629*, 2022.

[11] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 23–26.

[12] ——, "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, 2019, pp. 1070–1073.

[13] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov, "Fasttext. zip: Compressing text classification models," *arXiv preprint arXiv:1612.03651*, 2016.

[14] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *International conference on software engineering (ICSE)*, 2023.

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

[16] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[17] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[18] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[19] C. Cui, T. Li, J. Wang, C. Chen, D. Towey, and R. Huang, "Large language models for mobile gui text input generation: An empirical study," *arXiv preprint arXiv:2404.08948*, 2024.

[20] Q. Dong, L. Li, D. Dai, C. Zheng, Z. Wu, B. Chang, X. Sun, J. Xu, and Z. Sui, "A survey for in-context learning," *arXiv preprint arXiv:2301.00234*, 2022.

[21] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[22] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.

[23] S. Feng and C. Chen, "Prompting is all your need: Automated android bug replay with large language models," *arXiv preprint arXiv:2306.01987*, 2023.

[24] "Openai models," https://platform.openai.com/docs/models/.

[25] "Autogpt: An autonomous GPT-4 experiment," https://github.com/Significant-Gravitas/Auto-GPT, 2023.

[26] "Langchain: a framework for developing applications powered by language models," https://www.langchain.com.

[27] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, X. Che, D. Wang, and Q. Wang, "Chatting with gpt-3 for zero-shot human-like mobile automated gui testing," *arXiv preprint arXiv:2305.09434*, 2023.

[28] Y. Zhao, B. Harrison, and T. Yu, "Dinodroid: Testing android apps using deep q-networks," *arXiv preprint arXiv:2210.06307*, 2022.

[29] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, and Y. Liu, "Automatic web testing using curiosity-driven reinforcement learning," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 423–435.

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

[31] "Gpt-3.5," https://platform.openai.com/docs/models/gpt-3-5.

[32] X. Zhou, Y. Zhang, L. Cui, and D. Huang, "Evaluating commonsense in pre-trained language models," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 05, 2020, pp. 9733–9740.

[33] C. Wang, X. Liu, Y. Yue, X. Tang, T. Zhang, C. Jiayang, Y. Yao, W. Gao, X. Hu, Z. Qi *et al.*, "Survey on factuality in large language models: Knowledge, retrieval and domain-specificity," *arXiv preprint arXiv:2310.07521*, 2023.

[34] "Fdroid: Free and open source android app repository," https://f-droid.org/en/.

[35] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013, pp. 641–660.

[36] H. Wen, H. Wang, J. Liu, and Y. Li, "Droidbot-gpt: Gpt-powered ui automation for android," *arXiv preprint arXiv:2304.07061*, 2023.

[37] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, X. Che, D. Wang, and Q. Wang, "Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[38] OpenAI, "Openai function calling https://platform.openai.com/docs/guides/function-calling (Last checked: 23 October 2024)," October 2024.

[39] "Llama documentation: Ai function calling," https://docs.llama-api.com/essentials/function, 2024.

[40] LangChain, "How to use chat models to call tools https://python.langchain.com/docs/how_to/tool_calling/ (Last checked: 23 October 2024)," October 2024.

[41] "Gpt-4," https://openai.com/research/gpt-4.

[42] P. McMinn, M. Shahbaz, and M. Stevenson, "Search-based test input generation for string data types using the results of web queries," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 141–150.

[43] S. Afshan, P. McMinn, and M. Stevenson, "Evolving readable string test inputs using a natural language model to reduce human oracle cost," in *Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST 2013. IEEE, 2013, pp. 352–361.

[44] Y. He, L. Zhang, Z. Yang, Y. Cao, K. Lian, S. Li, W. Yang, Z. Zhang, M. Yang, Y. Zhang *et al.*, "Textexerciser: feedback-driven text input

exercising for android applications," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1071–1087.

[45] Y. Gu, X. Han, Z. Liu, and M. Huang, "Ppt: Pre-trained prompt tuning for few-shot learning," *arXiv preprint arXiv:2109.04332*, 2021.

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

[47] J. Yoon, R. Feldt, and S. Yoo, "Autonomous large language model agents enabling intent-driven mobile gui testing," *arXiv preprint arXiv:2311.08649*, 2023.