Prompting Is All You Need: Automated Android Bug Replay with Large Language Models

Sidong Feng
Monash University
Melbourne, Australia
sidong.feng@monash.edu

Chunyang Chen
Monash University
Melbourne, Australia
chunyang.chen@monash.edu

ABSTRACT

Bug reports are vital for software maintenance that allow users to inform developers of the problems encountered while using the software. As such, researchers have committed considerable resources toward automating bug replay to expedite the process of software maintenance. Nonetheless, the success of current automated approaches is largely dictated by the characteristics and quality of bug reports, as they are constrained by the limitations of manually-crafted patterns and pre-defined vocabulary lists. Inspired by the success of Large Language Models (LLMs) in natural language understanding, we propose `AdbGPT`, a new lightweight approach to automatically reproduce the bugs from bug reports through prompt engineering, without any training and hard-coding effort. `AdbGPT` leverages few-shot learning and chain-of-thought reasoning to elicit human knowledge and logical reasoning from LLMs to accomplish the bug replay in a manner similar to a developer. Our evaluations demonstrate the effectiveness and efficiency of our `AdbGPT` to reproduce 81.3% of bug reports in 253.6 seconds, outperforming the state-of-the-art baselines and ablation studies. We also conduct a small-scale user study to confirm the usefulness of `AdbGPT` in enhancing developers' bug replay capabilities.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

automated bug replay, large language model, prompt engineering

ACM Reference Format:
Sidong Feng and Chunyang Chen. 2024. Prompting Is All You Need: Automated Android Bug Replay with Large Language Models. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE 2024), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3608137

1 INTRODUCTION

Mobile applications have gained great popularity in recent years [16, 17, 29]. There have been over 3.8 million Android apps and 2 million iPhone apps striving to gain users on Google Play and Apple App Store [7]. Once an app is released, its quality is largely ensured by continuing maintenance activities and one of the most important maintenance tasks is to handle bug reports, i.e., documents that describe software bugs with information about what happened and what the user expected to happen. Along with the observed and expected behavior, bug reports often go on to contain the steps to reproduce (S2Rs) the bugs that assist developers to replicate and rectify the bugs, albeit with considerable amounts of engineering effort [63].

In an effort to accelerate bug maintenance, numerous researchers have worked toward providing automated solutions for bug reproduction. Previous studies [26, 55, 81, 82] apply natural language processing (NLP) and machine learning (ML) techniques to extract the S2R entities (i.e., action type, target component, input value, and scroll direction) from the bug reports, and employ random or simple guided exploration algorithms for bug reproduction. Unfortunately, in many cases, the S2Rs in the bug reports present significant challenges to previous automated approaches of bug replay, and can even be difficult for professional developers to reproduce manually [10–12, 45, 84]. First, the S2Rs are often unclear, imprecise, and ambiguous, owing to the cognitive and lexical gap between users and developers.

> s1. Open bookmark
> s2. Tap "add new bookmark" and create a name with "a"
> s3. Create another one with name "b"
> s4. Click "a"
> s5. Go back to bookmark after changing name "a" to "b"
> s6. App crash

Given the S2R shown above, extracting entities from the S2R requires a robust semantic and syntactic understanding. For instance, the two "create" words in *s2* and *s3* are not semantically identical; one refers to "input" while the other refers to "repeat" the previous steps. In *s3*, to "create another one", a deep comprehension of the specific creation steps from the previous steps is needed. Furthermore, a step may contain multiple sub-steps, such as *s5*, which can be divided into "go back to bookmark" and "change name 'a' to 'b' ". Last but not least, in *s5*, the conjunction word "after" alters the temporal order of the steps, meaning "change name 'a' to 'b' " should be executed first, followed by "go back to bookmark". These challenges surpass pre-defined word lists and are difficult to address using specific patterns based on previous works. Second, the S2Rs are often incomplete, and developers from more than 1.3k open-source projects wrote a letter to GitHub expressing their frustration of the S2Rs are often missing in the bug reports [8], and requested a solution that would encourage users to include them. So much so

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE 2024, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3608137

that the previous automated approaches are unable to replicate the bugs.

Emerging Large Language Models (LLMs), such as GPT-3 [13], PaLM [19], RoBERTa [56], T5 [64], have been trained on ultra-large-scale corpora and exhibit promising performance in natural language understanding and logical reasoning. For example, GPT-3 [13] (Generative Pre-trained Transformer3) from OpenAI with 175 billion parameters trained on billions of resources can smartly retrieve media information [52, 70], guide cloze selection [60], etc. The recent success of ChatGPT [6] based on GPT-3.5 demonstrates its remarkable ability to comprehend human questions and act as a knowledgeable assistant for interacting with users. Inspired by the impressive advancements of LLMs in the fields of information retrieval, selection guidance, and role-playing, we designate the LLMs as expert developers capable of extracting entities from bug reports and guiding replays from a set of dynamic GUI components.

In this paper, we propose a novel approach called `AdbGPT` to **A**utomatically repro**D**uce **B**ugs using LLMs. `AdbGPT` consists of two phases: i) *S2R Entity Extraction* and ii) *Guided Replay*. Notably, the underlying approach of `AdbGPT` for these two phases is prompt engineering, i.e., prompting the tasks to generate desired output, which is extremely lightweight compared to NLP with manually-crafted patterns and ML with massive training data. To extract the S2R entities, we first provide the LLMs with the knowledge of entity specifications, including available actions and action primitives. Since the LLMs are not specifically designed to handle bug reports, we employ few-shot learning by giving a few representative examples to help LLMs recognize the task of S2R entity extraction. Additionally, we provide LLMs with detailed chain-of-thought reasoning from developers, endowing it to extract S2R entities with the same thought process as developers. Given the inferred S2R entity extraction from the LLMs, we next prompt LLMs to dynamically guide the replay according to the GUI screens. To provide the current GUI information to the LLMs, we propose a novel GUI encoding algorithm to ensure the integrity of the information and the effectiveness of encoding. By giving a few examples with intermediate reasoning, LLMs infer the target components to operate on the GUI screen, repeating the steps to trigger the bugs.

We conduct a comprehensive evaluation to measure the effectiveness and efficiency of our `AdbGPT`. First, we evaluate the performance of our `AdbGPT` in extracting S2R entities from 88 bug reports. Compared with two state-of-the-art baselines and two ablation studies, our approach achieves significantly higher accuracy, i.e., 90.4% and 90.8% in step extraction and entity extraction, respectively. Second, we evaluate the performance of our `AdbGPT` in guiding bug replay and show that our approach can successfully reproduce 81.3% of bugs, outperforming the baselines and ablations. In addition, our approach is much more efficient compared with the baselines, saving an average of 1105.17 seconds per bug report. Apart from the performance of our tool, we also assess the perceived usefulness of our `AdbGPT` by conducting a small-scale user study on replaying bugs from 5 real-world bug reports. Through the study, we provide initial evidence of the usefulness of `AdbGPT` in facilitating bug replay. The contributions of this paper are as follows:

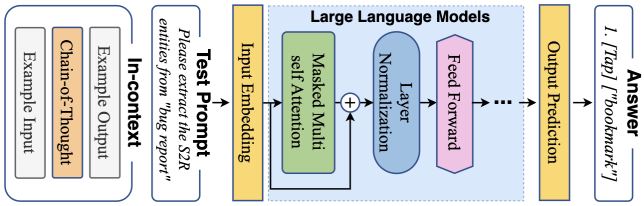

Figure 1: The process of prompt engineering.

- This is the first work to exploit LLMs into bug report analysis and GUI guidance, paving the way for new opportunities in software engineering tasks.
- We propose a lightweight approach, `AdbGPT` [2], that utilizes prompt engineering with few-shot learning and chain-of-thought reasoning to harness LLMs' knowledge for automated bug replay.
- Comprehensive experiments, including the performance evaluation of `AdbGPT` and its detailed qualitative analysis, reveal the capabilities of LLMs in bug replay. A user study to further demonstrate the usefulness of our approach.

2 BACKGROUND

We briefly discuss Large Language Models (LLMs), and in-context learning and chain-of-thought reasoning that we adapt in `AdbGPT`.

2.1 Large Language Models

Pre-trained Large Language Models (LLMs) introduce a new era in natural language understanding. It is trained on ultra-large-scale corpora and can support a wide range of natural language processing tasks with prompt engineering. By simply describing natural language prompts, LLMs can be invoked to perform specific tasks without requiring additional training or hard coding. Figure 1 illustrates the process of prompt engineering. Given an input/test prompt (sentence with task-specific instructions), the input embedding layer encodes it through word embedding. To comprehend prompts and generate corresponding answers, LLMs employ a Transformer model [72]. In detail, the multi-self attention layer is used to divide a whole high-dimensional space into several different subspaces to calculate the similarity. The normalization layer is implemented through a normalization step that fixes the mean and variance of each layer's inputs. The feed-forward layer compiles the data extracted by previous layers to form the final answer. In this work, we prompt the LLMs to extract S2R entities and generate dynamic actionable operations, and decode the LLMs' feedback to reproduce bugs.

2.2 In-context Learning

One common paradigm to master a specific task is to fine-tune the models [50]. The fundamental of model fine-tuning is big data, which is labor-intensive and costly. With the increasing ability of LLMs, in-context learning has shifted to a new paradigm, known as zero-shot learning, where LLMs make predictions by directly describing the desired output. While zero-shot learning shows promising performances in various training tasks by leveraging prior knowledge from training resources, it remains challenging to apply

to unseen tasks [13, 23, 43]. To overcome this challenge, few-shot learning is utilized to augment the context with a few examples of desired inputs and outputs (see Figure 1). This enables LLMs to recognize the input prompt syntax and patterns of the output. In our work, as LLMs are not specifically trained to understand, analyze, organize, and reproduce bug reports, we adopt few-shot learning as the in-context paradigm to help LLMs extract S2R entities and guide bug replay.

2.3 Chain-of-Thought Reasoning

While few-shot learning has proven effective for simple tasks with a few examples of *<input, output>*, it faces difficulties with more complicated tasks that require logical thinking and multiple steps to solve, such as arithmetic or commonsense reasoning questions [47, 54, 73, 83]. To elicit the capability of LLMs' reasoning, many researchers pioneer the idea of using rationales as the intermediate steps in examples. This idea is formulated as chain-of-thought reasoning [75], i.e., *<input, reasons, output>* in Figure 1. Our task of extracting S2R entities and guiding bug replay is not simple, it requires logical thinking to understand the bug reports and a coherent series of intermediate steps to trigger the bugs. As a result, we employ chain-of-thought reasoning to endow the LLMs to reproduce the bugs through the minds of expert developers.

3 ADBGPT APPROACH

Given a bug report and the app, we propose an automated approach to extract the S2R entities and reproduce each step based on the current GUI state to trigger the bugs in the app. The overview of our approach is shown in Figure 2, which is divided into two main phases: (i) the *S2R Entity Extraction* phase, which extracts the S2R entities defining each step to reproduce the bug report, including action types, target components, input values, or scroll directions; (ii) the *Guided Replay* phase that matches the entities in S2R with the GUI states to repeat the bug reproduction steps. While many works [26, 55, 81, 82] attempt to address these two phases using different ad-hoc methods, such as natural language processing for extracting S2R entities and greedy algorithms for guiding replay, our approach is notably lightweight compared to them, we exploit the single LLMs to address both phases through novel prompt engineering.

3.1 S2R Entity Extraction

The first phase of our approach is to understand, analyze, organize, and extract the entities from the S2R text in a bug report. Specifically, we leverage LLMs, equipped with knowledge learned from large-scale training corpora, to provide an in-depth understanding of the potential entities presented in the bug report. As discussed in Section 1, a step can be expressed in various ways and with different words. Therefore, we first provide LLMs with information about entity specifications, including available actions and action primitives, which can be mapped to instructions for device replay. To help LLMs recognize our task and infer S2R entities, we provide a few examples of conditioning, such as an S2R as input, a chain-of-thought as reasoning, and the final entities as the output. An example of the prompts is shown in Table 1. With this in-context

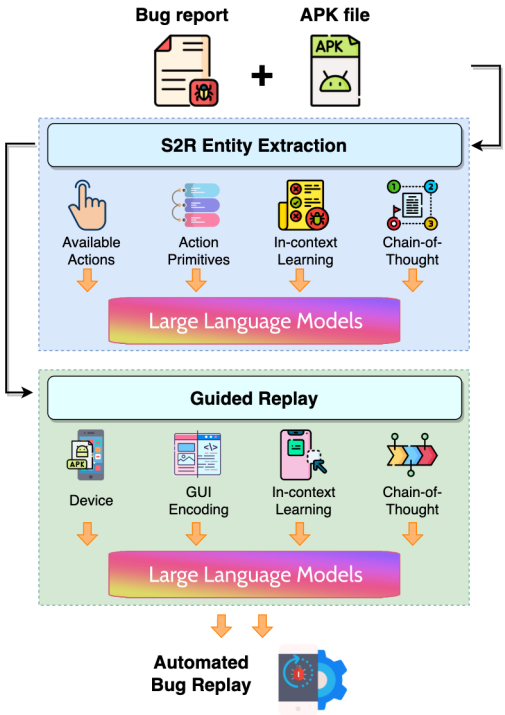

Figure 2: The overview of AdbGPT.

few-shot learning, we prompt the LLMs in the same way to extract the S2R entities from the test bug report.

3.1.1 Available actions. Unlike target components and input values, which are indeterministic and dynamic depending on the current GUI, actions that can interact with the device remain consistent. We identify five standard actions as shown in Table 1, including *Tap, Scroll, Input, Double-tap*, and *Long-tap*. While there are other customized actions, such as pinch, multi-handed gestures, etc., they are less prevalent. For brevity, we focus on the commonly-used actions in this paper.

3.1.2 Action primitives. As the context of action varies, each action requires a different set of primitives to represent entities. For example, the *Tap* action requires a target component to interact with, such as a button in the GUI. As a result, we formulate it as *[Tap] [Component]*. Similarly, for *Double-tap* and *Long-tap* actions, we formulate them using similar linguistic primitives. The *Scroll* action requires indicating the direction of the scrolling effect, such as upward or downward, formulating as *[Scroll] [Direction]*. The *input* action involves the process of entering a specific value into a text field component. To formulate this action, we use the primitive *[Input] [Component] [Value]*.

3.1.3 In-context learning. A representative example helps the model elicit specific knowledge and abstractions needed to complete the task. A common strategy for example selection is to randomly sample from a dataset [74]. However, a random bug report may not encompass all the complexities of the task, thereby limiting the ability of LLMs to gain a comprehensive understanding of the problem. To select representative examples, we recruit three professional

Table 1: The example of prompt engineering of S2R entity extraction.

PROMPT TYPE	INSTANTIATION
Available actions	*Tap, Scroll, Input, Double-tap, Long-tap*
Action primitives	*[Tap] [Component], [Scroll] [Direction], [Input] [Component] [Value], [Double-tap] [Component], [Long-tap] [Component]*
Example input	*1. Open bookmark* *2. Tap "add new bookmark" and create a name with "a"* *3. Create another one with name "b"* *4. Click "a"* *5. Go back to bookmark after changing name "a" to "b"* *6. App crash*
Chain-of-Thought	***1st** step is "Open bookmark". The action is "open" and the target component is "bookmark". However, there is no explicit "open" in the Available actions list. Therefore, we select the closest semantic action "tap". Following the Action primitives, the entity of the first step is [Tap] ["bookmark"].* ***2nd** step is "Tap 'add new bookmark' and create a name with 'a' ". Due to the conjunction word "and", this step can be separated into two sub-steps, "Tap 'add new bookmark' " and "create a name with 'a' " ...* ***3rd** step is "Create another one with name 'b' ". Due to its semantic meaning, this step is meant to repeat the previous steps to add another bookmark with name "b". Therefore, it should actually be the 2nd step ...* ***4th** step is "Click 'a' " ...* ***5th** step is "Go back to bookmark after changing name 'a' to 'b' ". Due to the conjunction word "after", this step can be separated into two sub-steps, "Go back to bookmark" and "change name 'a' to 'b' ". The conjunction word "after" also alters the temporal order of the sub-steps, that "change name 'a' to 'b' " should be executed first, followed by "go back to bookmark" ...* ***6th** step is "App crash". This step does not have any operations.*
Example output	*Overall, the extracted S2R entities are:* *1. [Tap] ["bookmark"]* *2. [Tap] ["add new bookmark"]* *3. [Input] ["name"] ["a"]* *4. [Tap] ["add new bookmark"]* *5. [Input] ["name"] ["b"]* *6. [Tap] ["a"]* *7. [Input] ["name"] ["b"]* *8. [Tap] ["back"]*

developers with four-year experience in bug fixing and triaging by word-of-mouth. First, we give them an introduction to our study and ask them to interact with the LLMs to become familiar with our task. Each developer is then asked to independently review a large-scale of bug reports collected from previous works [27, 28] and select the challenging ones. After 1.5 hours of initial reviewing, each developer on average examines 110 bug reports and selects 7 as challenging. The developers then meet and discuss the representatives of the selection and refine the dataset until a consensus is reached. In total, we collect 15 bug reports as our representative dataset, with an example in Table 1.

3.1.4 Chain-of-thought reasoning. Giving a chain-of-thought reasoning, it endows the LLMs to generate a coherent series of intermediate steps that lead to a reasonable output. To employ this paradigm, we ask the developers to independently write the chain-of-thought reasoning for each bug report in the representative dataset, following the previous study [75]. The three developers and the first author then meet and discuss the optimal reasoning for each bug report, aiming to express a clear step-by-step thinking

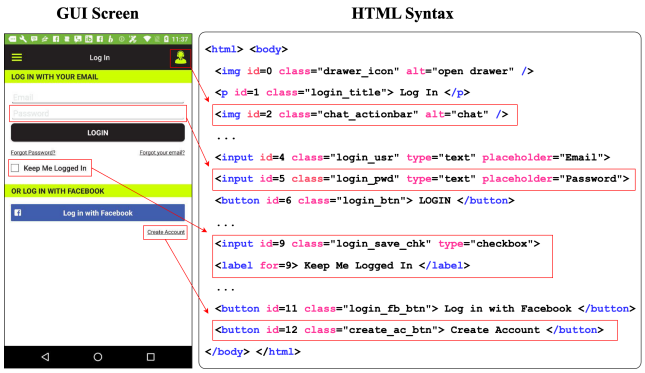

Figure 3: Illustration of GUI encoding.

process that leads to the final S2R entities. An example of chain-of-thought reasoning is shown in Table 1. In the 1st step, we explicitly explain the action ("Open") and target component ("bookmark"). Since "Open" is not listed in the *Available actions*, we map it to the closest semantic action "Tap". For the 2nd and 5th steps, we explicitly explain the conjunction words that lead to multiple sub-steps and alternate temporal order. The 3rd step is more complicated, that we explicitly explain the semantic meaning of the step and its relationship to the previous steps. Overall, this step-by-step thinking leads to the final output of the extracted S2R entities.

3.1.5 Prompt construction. We combine the aforementioned information in Table 1 as the input prompt, i.e., (⟨*Available actions*⟩ + ⟨*Action primitives*⟩ + ⟨*Example input*⟩ + ⟨*Chain-of-Thought*⟩ + ⟨*Example output*⟩). Note that, due to the robustness of the LLMs, the prompt sentence does not need to adhere strictly to grammar rules [13]. Next, we input the test bug report as the test prompt and query for the S2R entities. Due to the advantage of few-shot learning and chain-of-thought reasoning, the LLMs will consistently generate a numeric list to represent the extracted S2R in the same format as our example output, which can be inferred using regular expressions.

3.2 Guided Replay

The second phase is to explore the apps to match the extracted S2R entities to a sequence of GUI events to reproduce the bugs. A seemingly straightforward solution [26, 81, 82] is to use lexical computation to match the extracted components against the displayed text of the components on the current GUI screen. However, as explained in Section 1, this process of component mapping is limited due to the issues of missing steps, i.e., there are no matching components on the GUI screen. To address this, we adopt LLMs with few-shot learning and chain-of-thought reasoning to generate dynamic guidance on the GUI screen, enabling automatic reproduction of the steps even when a step is missing. One challenge in using LLMs for GUI guidance is that they can only process reasonably sized text input. To help LLMs inherently understand GUI screens, we propose a novel method to transfer the GUIs into domain-specific prompts that the LLMs can understand. Given each step and its current GUI screen, we prompt the LLMs to generate the target component to operate on, ultimately triggering the bug.

3.2.1 GUI encoding. To prompt the LLMs to guide bug replay according to the GUIs, we need to dynamically depict the current GUI screen, as well as its contained component information, such as text, type, image, etc. To design our screen encoding, we leverage the insight that if our GUI encoding falls within the training data distribution of LLMs, it is more likely to perform better [18, 53, 85]. Since most of the LLMs' training data is typically scraped from the raw web page [13], we encode a GUI in a text by converting its view hierarchy data into HTML syntax. However, there are two fundamental limitations. First, the native classes in the view hierarchy don't always match HTML tags, for instance, a *<CheckBox>* in the hierarchy corresponds to a combination of *<input type="checkbox">* and *<label>*. Second, including all properties of GUI components will result in excessively long HTML text that may influence the understanding of LLMs.

Therefore, we adopt a heuristic approach to convert the view hierarchy of the GUI screen into HTML text. Note that since the view hierarchy is not designed to be encoded in HTML syntax, a perfect one-to-one conversion does not exist. Our goal is to make the converted view hierarchy look and function similar to the HTML syntax. An example of the GUI screen and the encoded HTML text is illustrated in Figure 3. The encoding is conducted by traversing the view hierarchy tree using a depth-first search. We convert each node into HTML syntax, preserving a related subset of properties from the view hierarchy.

- `resource_id`: the unique resource id of component
- `class`: the native component type
- `text`: the text of component
- `content_desc`: the description of visual component

We first map the classes to HTML tags with similar functionalities using an ad-hoc heuristic method. For example, the *TextView* is mapped to the *<p>* tag as they both present texts. Similarly, the *Button* is mapped to the *<button>* tag; and *ImageView* to the *<img>* tag. For special classes in the view hierarchy (e.g., *<EditText>*, *<CheckBox>*, *<RadioButton>*), we use a combination of *<input>* and *<label>*, aligning with HTML syntax. Note that we focus on the most commonly-used classes for simplicity, and the rest of the classes, such as *<VideoView>*, are mapped to the * * tag.

Then, we insert the text properties of the component in between the opening and closing HTML tags, following the standard syntax of texts in HTML. The resource_id property usually contains additional descriptions of a component's functionality or purpose, written by developers. For example, a resource_id of "submit_btn" refers to the functionality of submit. We insert it in the "class" attributes in HTML to help the models understand the screen context. For the *ImageView* component, we further insert the content_desc as the "alt" attribute in the HTML to express the accessibility to the image in the GUI screen. Lastly, we follow the traversing order in the view hierarchy tree to insert unique numeric indexes to each element as the "id" attribute.

3.2.2 In-context learning. It can be extremely laborious and time-consuming to manually collect representative examples for guided replay, such as checking whether there is a step missing and identifying the target components that may trigger the bugs. To this end, we aim to synthesize a few representative examples to help the LLMs understand the challenges of missing steps in the S2Rs.

Table 2: The examples of prompt engineering of guide replay.

PROMPT TYPE	INSTANTIATION
Example 1	
GUI encoding	*<html> ... </html>*
Example input	*If I need to [Tap] ["Sign in"], which component id should I operate on the GUI?*
Chain-of-Thought	*There is no explicit "Sign in" component in the current GUI screen. However, there is a semantic closest component "Log in" button. The id attribute of "Log in" component is 6. So, we could potentially operate on [id=6] in the screen.*
Example output	*[id=6]*
Example 2	
GUI encoding	*<html> ... </html>*
Example input	*If I need to [Tap] ["darkmode"], which component id should I operate on the GUI?*
Chain-of-Thought	*There is no explicit and semantic similar "darkmode" component in the current GUI screen, so it appears a [MISSING] step. However, "darkmode" could be related to the "display" button in the screen. The id attribute of "display" component is 1. So, we could potentially operate on [id=1] component in the screen.*
Example output	*[MISSING] [id=1]*

To synthesize the missing step, we randomly omit 0 to 2 steps in the bug reports. Given these examples where steps may or may not be missing, we ask the three developers to validate the examples and select the challenging ones following similar procedures in Section 3.1.3, resulting in 5 representative examples as the dataset.

3.2.3 Chain-of-thought reasoning. We further ask the three developers to write their chain-of-thought reasoning, leading to the final output. The documenting process is similar to the previous phase in Section 3.1.4, with independent writing followed by a discussion to select the most representative step-by-step reasoning. We show two particular reasoning in Table 2. In Example 1, we explicitly explain a semantic example of a guiding step that does not exactly component match S2R ("Sign in") and GUI ("Log in"). In Example 2, we explicitly explain an example of guiding a missing step with a potential target component and a missing tag.

3.2.4 Prompt construction. We provide a few examples with their chain-of-thought as input prompt, i.e., (⟨*GUI encoding*⟩ + ⟨*Example input*⟩ + ⟨*Chain-of-Thought*⟩ + ⟨*Example output*⟩). Given each step generated by the S2R entity extraction phase and the current GUI encoding as the test prompt, we then query for the target component selection to reproduce the step. Note that we infer the target component using "id", which is more efficient and space-saving than spelling out the complete HTML tag. In the scenario of missing steps, the output prompt will additionally predict a "MISSING" tag to allow us to iteratively explore the steps in the app until the target component is found. If LLMs fail to explore a potential target component to recover the missing steps (e.g., no "id" in the inference), we go back to the previous step and prompt the models to explore a new component.

3.3 Implementation

For the LLMs, we use the state-of-the-art ChatGPT model [6]. According to a small-scale pilot study, we set the number of few-shot learning examples in the range of 1 to 3. This is how many examples can fit in the LLM's implicit maximum input tokens (i.e., 4,096 for ChatGPT). As the LLMs may generate verbose output (such as

repeat questions, chain-of-though reasons, etc.), we use "[]" to infer the specific prediction, such as entity for S2R entity extraction, and target component id attribute or missing step flag for guided replay. Note that, users may describe input actions vaguely, for example, "enter the name". To address this, we set a "test" value if there is no explicit input value in the step.

Our AdbGPT is implemented as a fully automated bug replay tool. In detail, we use Genymotion [5] for running and controlling the virtual Android device, Android UIAutomator [4] for dumping the GUI view hierarchy, and Android Debug Bridge (ADB) [3] for replaying the steps.

4 EVALUATION

In this section, we describe the procedure we used to evaluate AdbGPT in terms of its performance automatically. Since our approach consists of two main phases, we evaluate each phase of AdbGPT, including S2R Entity Extraction (Section 3.1) and Guided Replay (Section 3.2).

- **RQ1:** How accurate is our approach in extracting S2R entities?
- **RQ2:** How accurate is our approach in guiding bug replay?
- **RQ3:** How efficient is our approach in bug replay?
- **RQ4:** How usefulness is our approach for developers in real-world bug replay?

For **RQ1**, we present the general performance of our approach for S2R entity extraction and the comparison with state-of-the-art baselines. Besides, we also present the performance comparison among the variations of in-context learning (e.g., few-shot vs zero-shot) and the contribution of reasoning by comparing the performance with and without chain-of-thought. For **RQ2**, we carry out experiments to check if our approach can trigger the target components, comparing with the baselines and ablations. For **RQ3**, we evaluate the runtime overhead of our approach in bug replay. For **RQ4**, we conduct a small-scale user study to evaluate the perceived usefulness of AdbGPT for automatically replaying bug reports in real-world development environments.

4.1 RQ1: Performance of S2R Entity Extraction

Experimental Setup. To answer RQ1, we first evaluated the ability of our approach (in Section 3.1) to accurately identify the S2R entities from the bug reports. To avoid potential bias, we collected the bug reports from the artifacts of three existing open-source datasets: (i) the evaluation dataset of ReCDroid [81, 82]; (ii) an empirical study on Android bug report reproduction ANDROR2+ [45]; and (iii) an empirical study of crash bug reports Themis [67]. Due to some overlap across these datasets, we first removed duplicates if the bug reports were from the same issue repository. The users may report the S2Rs using visual information, such as screenshots or videos [28]. In this work, we focused on textual information, i.e., the natural language descriptions of S2Rs, so we further audited the bug reports and retained those only containing textual S2Rs. In total, we collected 88 bug reports as our experimental dataset.

Given the set of experimental bug reports, we needed to manually identify and generate the ground-truth for S2R entities (specific action types, target components, input values, and scroll direction) from the bug reports, as the state-of-the-art automated approaches still cannot 100% accurately infer the entities. We recruited two paid students through the university's internal slack channel and they were compensated with $12 USD per hour. They had experience in Android development and bug reproduction. To ensure accurate annotations, the process began with initial training. First, we provided them with an introduction to our study and an example set of annotated bug reports where the authors had annotated the S2R entities. Then, we asked them to pass an assessment test. The two annotators were then assigned the experimental set of bug reports to label the S2R entities independently, without any discussion. After the initial labeling, the annotators met and sanity corrected any subtle discrepancies. Any disagreements were handed over to the first author for the final decision. In total, we obtained 305 reproduction steps from 88 bug reports, which contained 305 actions, 291 components, 39 inputs, and 14 directions.

Metrics. An extracted S2R was determined to be correct if all of the following conditions matched the ground-truth, including steps (i.e., step ordering, sub-step separation) and entities (i.e., the action types of each step, the possible target components if existed, the possible input values, and the possible scroll directions). Therefore, we employed the accuracy of the S2R extraction as our evaluation metric. The higher the accuracy score, the better the approach can identify the steps in the bug reports.

Baselines. We set up two state-of-the-art methods which are widely used for S2R entity extraction as the baselines to compare with our approach. *ReCDroid* [82] analyzes the dependencies among words and phrases from hundreds of bug reports and defines 22 grammar patterns to extract the S2R entities. For example, a noun phrase (NP) followed by a "click" action should be the target component, etc. We adopted their released repository for evaluation. ReCDroid+ [81] is a later work that scraped bug reports from the issue track websites, which is out of the scope of this study. Since these two approaches perform the same for S2R entity extraction, we used the ReCDroid in the rest of the paper for simplicity. *MaCa* [55] proposes an automated S2R identification tool, which first identifies action words based on natural language processing and machine learning classifier, then extracts its related entities (i.e., target components). Note that we did not adopt Yakusu [26] as a baseline, as it was developed five years ago and many of its libraries are no longer maintained.

Apart from the baselines, we also added two derivatives of our approach to demonstrate the impact of prompt engineering. In Section 3.1.3, we utilized few-shot learning to help the LLMs understand the requirement of our task. Therefore, we set up an ablation study to infer the results without any examples, named AdbGPT *w/o Few*, which is the well-known method called zero-shot learning. To demonstrate the strength of the chain-of-thought outlined in Section 3.1.4, we also conducted an ablation study, namely AdbGPT *w/o CoT*. Specifically, it provides a few examples of bug reports and the outputs, without any intermediate reasons, and then prompts the models to extract the S2R entities from the test bug report.

Results. Table 3 depicts the performance of our approach in extracting the S2R entities from the bug reports. The performance of our approach is significantly better than that of other baselines in all metrics, i.e., on average 39.3%, and 42.2% more accurate in step extraction and entity extraction compared with the best baseline (MaCa). In addition, applying few-shot learning and chain-of-thought which provide examples with intermediate reasons, can

Table 3: Performance comparison of S2R extraction.

Method	Acc. of Step Extraction	Acc. of Entity Extraction			
		Action	Component	Input	Direction
ReCDroid	49.5%	65.9%	45.4%	30.8%	42.9%
MaCa	51.1%	70.1%	41.6%	33.3%	50.0%
AdbGPT w/o Few	83.9%	81.9%	84.5%	87.1%	85.7%
AdbGPT w/o CoT	86.9%	83.6%	87.6%	87.1%	92.9%
AdbGPT	**90.4%**	**91.1%**	**90.0%**	**89.7%**	**92.9%**

endow the LLMs with a better understanding of the task, resulting in a boost of performance from 84.6%, 87.6% to 90.8%.

To fully explore the reason why our approach outperforms other baselines, we carry out a qualitative study by investigating the bug reports which are correctly extracted by our approach but wrongly extracted by the baselines. We summarize three reasons, including inconsistent formats, dependent context, and diversified words. Some representative examples can be seen in Figure 4.

1) Inconsistent formats: The users may employ different formats to compose bug reports depending on their writing preferences. Common formats include numeric indexing, list indexing, or using commas in a line. The baselines can correctly separate the steps following these patterns. However, regarding the users' understanding of the bugs, they may compose the steps under the same state such as "->" and "then" as shown in Figure 4-A, but the baseline approaches fail to interpret them as separate steps.

2) Dependent context. Context dependency can occur from a macro perspective in the bug reports. For example, in Figure 4-B, the users demonstrate an action of "deleting" a value, which is equivalent to "leave the EditText empty", thus, they should be interpreted as one step. In Figure 4-C, the users may omit duplicate steps by describing them as "repeat the previous step" or "create another one", etc. However, without syntactic awareness, the baseline approaches do not monitor the relationships among the words in a sentence, causing inaccurate step extraction.

Context dependency can also occur at a more granular level. The users may use conjunction words to order the reproduction steps that differ from their syntactic order. The baseline MaCa proposes a heuristic approach based on constituency grammar to deal with common connective words (i.e., "before", "after", "when", etc.), but it is limited to less commonly-used words. For instance, in Figure 4-D, the step "Favorites settings" should happen first, and then the step "Show favorites above search bar", due to the semantic of the connective word "under" between the steps, which the baselines fail to recognize.

3) Diversified words: The users may use different words or phrases to describe the same concept, making it difficult for baseline approaches to recognize semantic patterns. This can include synonyms, alternative phrasing, or even colloquial language. For example, as shown in Figure 4-E, the users colloquially use "go to", "click", "choose", "open", and "select" to refer to the action of "tap". Even for the same word, the user's choice of expression can affect the recognition, e.g., "click the button" and "the button is clicked" in Figure 4-F. While the baselines have identified some common vocabulary of words, the inconsistency of human language still poses challenges in recognizing diversified word patterns in bug reports.

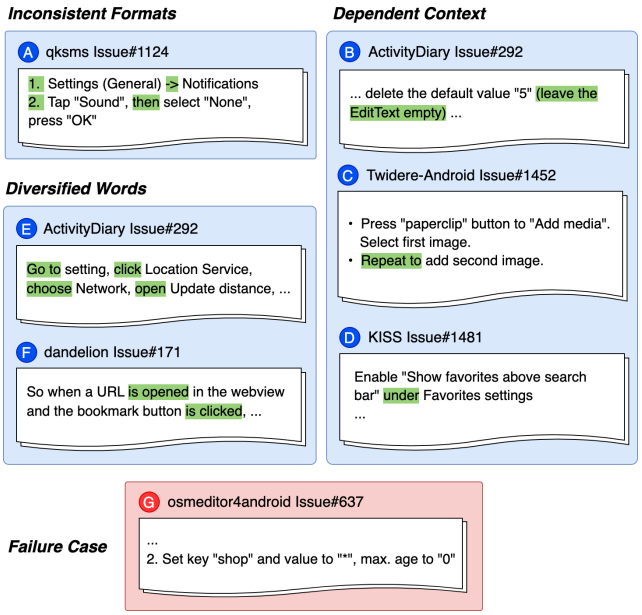

Figure 4: Examples of S2R extraction.

Albeit the good performance of our approach, we still make the wrong extraction for some bug reports. We manually check those incorrect S2R entity extractions and identify one common cause. The users may write the S2Rs ambiguously, causing the LLMs to fail to capture the user intent. For example, in Figure 4-G, the users aim to set the key **to** "shop" (i.e., [Input] ["key"] ["shop"]) and value to "*" (i.e., [Input] ["value"] ["*"]). However, due to the missing **to** keyword, LLMs misunderstand the S2R and extract the step as [Input] ["shop"] ["*"]. Note that the LLMs can correctly extract the S2Rs by adding the **to** keyword, suggesting that our performance could be further improved with well-written bug reports.

4.2 RQ2: Performance of Guided Replay

Experimental Setup. To answer RQ2, we evaluated the ability of our approach (in Section 3.2) to accurately map the entities in the S2Rs to the components in the GUIs for bug reproduction. We used our experimental dataset (outlined in Section 4.1), containing 88 distinct textual S2Rs. However, many S2Rs are no longer reproducible due to two reasons. First, many real-world bugs have been fixed and the apps have been patched, making it difficult to find the corresponding previous versions of the apps for reproduction. Second, some bugs (e.g., financial, social apps) require extensive information like authentication, database, or specific hardware, which are beyond the scope of this study. To that end, we asked the two students to locate the corresponding apps and interact with them to ensure the S2Rs were reproducible. If the students could not reproduce a bug report, the first author would further assess its reproducibility. As a result, we collected 48 reproducible S2Rs from the bug reports.

Metrics. To measure the performance of our approach, we employed reproducibility as the evaluation metric, i.e., whether the

Table 4: Performance comparison of guided replay.

Bug	No. Steps	Method			
		ReCDroid	AdbGPT w/o Few	AdbGPT w/o CoT	**AdbGPT**
Anki#4586	3	✓	✓	✓	✓
Anki#5638	4	✗	✗	✓	✓
Anki#2564	3	✗	✓	✓	✓
Anki#3224	2	✗	✗	✗	✓
Anki#10584	1	✓	✓	✓	✓
AnyMemo#440	5	✓	✓	✓	✓
Birthdroid#13	1	✓	✗	✗	✓
NewsBlur#1053	5	✓	✓	✓	✓
LibreNews#22	5	✓	✓	✓	✓
LibreNews#23	7	✓	✓	✓	✓
LibreNews#27	4	✗	✗	✗	✗
Transistor#63	5	✓	✓	✓	✓
ScreenCam#25	4	✓	✓	✓	✓
News#487	2	✓	✗	✗	✓
k9mail#3255	4	✓	✓	✓	✓
k9mail#2612	3	✓	✗	✗	✗
k9mail#2019	1	✓	✓	✓	✓
openMF#734	2	✓	✓	✓	✓
FamilyFinance#1	6	✗	✗	✗	✓
trickytripper#42	4	✗	✗	✗	✓
NoadPlayer#1	4	✓	✓	✓	✓
calendula#134	2	✓	✓	✓	✓
StreetComplete#1093	3	✗	✗	✗	✗
OmniNotes#592	3	✓	✓	✓	✓
Markor#1020	2	✗	✓	✓	✓
Markor#331	6	✗	✗	✗	✗
KISS#1481	5	✗	✓	✓	✓
ultrasonic#187	3	✗	✓	✓	✓
andOTP#500	3	✗	✗	✗	✗
k9mail#3971	3	✗	✓	✓	✓
kiwix#1414	4	✓	✓	✓	✓
kiwix#555	2	✓	✓	✓	✓
qksms#1155	3	✗	✗	✗	✗
Aegis#287	3	✗	✗	✗	✓
AmazeManager#1796	5	✗	✗	✗	✓
ActivityDiary#285	6	✗	✗	✓	✓
APhotoManager#116	3	✗	✗	✗	✗
collect#3222	2	✗	✓	✓	✓
commons#2123	3	✗	✗	✗	✗
FirefoxLite#5085	4	✗	✓	✓	✓
nextcloud#1918	5	✓	✓	✓	✓
nextcloud#5173	4	✗	✓	✓	✓
osmeditor#729	2	✓	✓	✓	✓
osmeditor#637	4	✗	✗	✗	✗
WordPress#11135	3	✗	✓	✓	✓
WordPress#10302	3	✓	✓	✓	✓
CineLog#60	5	✗	✗	✗	✓
AndrOBD#144	3	✗	✗	✗	✓
Reproducibility	-	45.8%	58.3%	62.5%	**81.3%**

method can successfully reproduce the bugs. The higher the reproducibility score, the better the approach can replicate the S2Rs on the GUIs to trigger the bugs.

Baselines. We adopted the state-of-the-art method *ReCDroid* [82] as the baseline to compare with our method. Specifically, ReCDroid employs a greedy algorithm to search for the matched components on the GUIs by using the word embedding similarity metric Word2Vec. In addition, we adopted two derivations of our approach as the ablation baselines to demonstrate the advantages of AdbGPT, including without few-shot learning (AdbGPT *w/o Few*), and without chain-of-thought (AdbGPT *w/o CoT*). Note that we did not adopt MaCa [55] as the baseline, as it solely focuses on S2R entity extraction without including bug replay.

Results. Table 4 shows detailed results of the reproducibility for each bug. Our approach outperforms all the baselines and ablations, i.e., on average 81.3% compared with 45.8%, 58.3%, and 62.5%, for ReCDroid, AdbGPT w/o Few, and AdbGPT w/o CoT, respectively. We observe that chain-of-thought leads to a substantial improvement

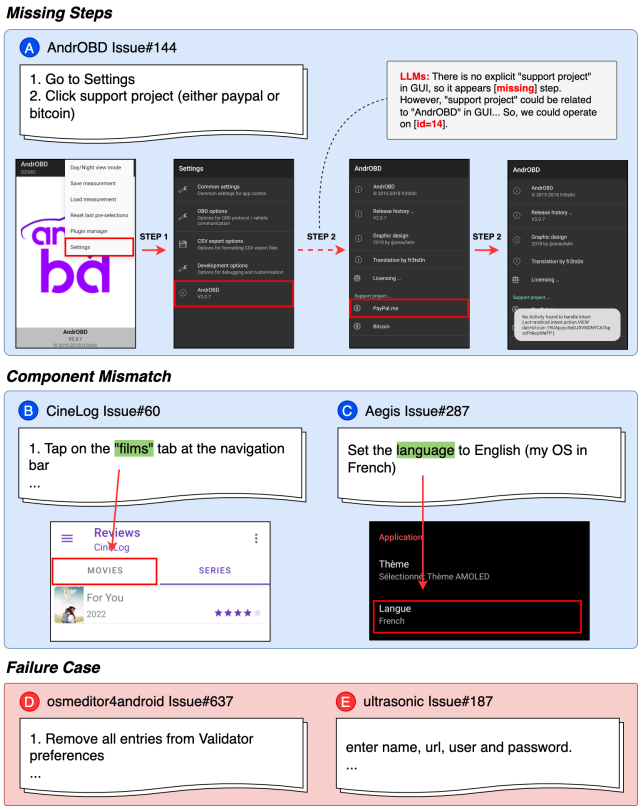

Figure 5: Examples of guided replay.

(i.e., 18.8% boost) in the performance of AdbGPT, indicating that the LLMs can better understand the task by processing it step-by-step.

We further conduct a qualitative analysis to compare the capabilities of our approach with the baselines. Overall, we summarize two common reasons which can be seen in Figure 5.

1) Missing steps: It is a key factor in the failure of automated bug reproduction in baselines. This issue may arise when users overlook some "unimportant" steps. Without a complete set of instructions, automated approaches fail to reproduce the steps needed to trigger the bugs. In contrast, our approach exploits the semantic understanding of subsequent steps and the components in the current GUI to potentially identify the most probable actions that can restore the missing steps. For instance, in Figure 5-A, since there is no explicit target component ("support project") in the current GUI, the LLMs can identify the potential component ("AndrOBD", which is the project name) due to their semantic correlation.

2) Component mismatch: Another common reason that makes it difficult for baselines to repeat the step is the component mismatch. There may be two possible reasons why an S2R cannot directly match components on the GUI. First, the users may describe the component imprecisely. For instance, in Figure 5-A, the "movies" button is described as "films", making it challenging for automated approaches to identify the exact GUI component the S2R refers to. Second, for multi-lingual apps, the languages of S2Rs and GUIs may be different. Figure 5-B shows an example that the user reports

the bug in English while it is reproduced on a GUI in French, causing the reproduction to fail. In contrast, our approach leverages LLMs, which capture a large-scale corpus and language resources, to understand the morphological forms of components including abbreviations, synonyms, misspellings, and languages, and search for the most relevant component in the GUI.

Although our approach demonstrates good bug reproducibility, we still fail to reproduce some unclear S2Rs. For example, the users may not describe the specific values for input fields, as seen in Figure 5-D. To address it, we generate random text for the input fields, i.e., the "test" value for the name field. However, some input fields require valid values, i.e., the URL field requires an input value starting with "https://". This could be achieved by understanding the context information of the input fields and designing linguistic patterns to prompt LLMs to generate valid input values. We would like to put it as our future work.

4.3 RQ3: Efficiency Performance

Experimental Setup. To answer RQ3, we evaluated the overhead of our approach by calculating the average time it takes for a bug report to pass through each of the two phases of the AdbGPT approach. All of our experiments were conducted on the commodity hardware, a 2.6 GHz Macbook Pro with 6 dedicated CPU Intel Core as the AdbGPT engine, and the official Android x86-64 emulator as the device server. Our approach could perform substantially faster on specialized hardware. Note that, inferring the entities and guidance from ChatGPT (our underlying LLM implementation) can be unstable and highly dependent on the ChatGPT server and user volume. Therefore, we ran the inference three times for our approach and the ablation baselines, using the average time for performance evaluation.

Metrics. To measure the overhead of AdbGPT, we employed the time it takes for both extracting S2R entities from the bug report and guiding the bug replay. The less time it takes, the more efficiently the method can reproduce the bugs.

Baselines. We used *ReCDroid* and *MaCa* as the baselines, as well as AdbGPT without few-shot and AdbGPT without chain-of-thought as the ablation studies of our approach.

Results. Table 5 shows the performance comparison with the baselines. Our approach takes AdbGPT 255.75 seconds on average to reproduce a bug report, i.e., 2.11 seconds to extract the S2R entities from the bug report and 253.64 seconds to replay the S2R in the device. In comparison, it takes the ReCDroid method on average 1360.92 seconds, indicating the efficiency of our approach which saves nearly 5x time per bug replay. Notably, our approach accelerates 435% time in guiding replay in the GUI screen compared with ReCDroid. This is primarily due to missing steps in the S2Rs, which cause ReCDroid to randomly explore the app with repeated back-and-forth exploration, which is a time-consuming process. In contrast, our approach elicits the LLMs to understand the semantics of the S2R context and current GUI to predict the most probable components for bridging the missing steps. We expect that the overhead of our approach could be further accelerated by more advanced hardware.

Table 5: Performance comparison of average runtime per bug report.

Method	S2R Entity Extraction (sec)	Guided Replay (sec)
ReCDroid	3.75	1357.17
MaCa	2.18	-
AdbGPT	**2.11**	**253.64**

Table 6: Performance comparison of usefulness evaluation.

Bug	No. Steps	Participants		AdbGPT	
		Success	Time (sec)	Success	Time (sec)
KISS	2	75%	423.1	100%	262.9
NeuroLab	2	100%	396.5	100%	228.0
Alibaba	5	87.5%	545.7	100%	301.2
Wechat	7	100%	493.3	100%	205.1
Anki	8	100%	544.8	100%	349.7
Average	-	92.5%	480.7	**100%**	**269.4**

4.4 RQ4: Usefulness of AdbGPT

Experimental Setup. To investigate the perceived usefulness of AdbGPT, we conducted a small-scale user study with eight participants, including five graduate students and three app developers. The participants were recruited from the university's internal slack channel or through direct contact with the authors. Given that the graduate students all have at least 1.5 years of experience in Android app development and bug replay. The three app developers are more professional and have more than two-year working experience in Android development in the industry.

Procedure: To mitigate the threat of user distraction, we conducted the experiment in a quiet room individually without mutual discussion. We interviewed the participants using a set of questions organized in two sessions. The first session aimed to collect information on participants' backgrounds, including their role at the company, the frequency of addressing bug reports and replaying bugs, the challenges they face in their practices, etc. The second session aimed to assess AdbGPT 's potential usefulness in replaying the S2Rs from bug reports. Each participant was asked to reproduce the same set of 5 randomly selected bug reports from issue trackers and GitHub. These bug reports were of diverse difficulty ranging from 2 to 8 S2Rs in the bug reports. To minimize the impact of stress, participants were allowed to take a short break between each bug replay. We only recorded the time used to reproduce the bug reports, including understanding the textual S2Rs and replicating the steps in Android. Note that all the experiment apps were pre-installed. Participants had up to 10 minutes for each bug replay. At the end of the tasks, we showcased the bug replay automatically generated by our tool AdbGPT and asked participants for feedback, including a 5-Likert scale question about their preference for using our tool, suggestions for improvement, etc.

Results. Table 6 shows the detailed experiment results. Although most participants can successfully finish the bug replay on time, AdbGPT reproduces the bug report much faster than that of participants (with an average of 480.7 seconds versus 269.4 seconds). In fact, the average time for participants' reproduction is underestimated, because they fail to reproduce 3 bugs within 10 minutes, which means that participants may need more time for bug replay. In contrast, our automated approach finishes all the tasks within 7 minutes.

After observing the bug replay automatically generated by AdbGPT, all participants strongly favored using AdbGPT in practice, with an average preference score of 4.5 out of 5.0. To gain insight into the usefulness of our approach, we collect feedback from the participants and summarise two practical challenges of manual replay. First, understanding the S2Rs from the bug reports is surprisingly time-consuming, as it involves grasping the context, reordering the steps, analyzing the potential actions and components, etc. Second, it is difficult to determine the trigger for the missing steps, resulting in participants' guesses of the action for triggering the next steps. That trial-and-error makes the bug replay process tedious and time-consuming. It is especially severe for junior developers who are not familiar with the app.

Participants also provide valuable feedback to help improve AdbGPT. For example, one participant suggests a strategy for interaction between developers and LLMs for bug replay by adding a confidence value to the LLMs. If the inference is unreliable (i.e., the confidence value is low), there will be an interactive conversation to ask developers to confirm the operations. This strategy could efficiently and effectively improve the performance of bug reproduction with human-AI collaboration. We will investigate the possible solution in our future work.

5 THREATS TO VALIDITY

In our experiments evaluating our model, threats to internal validity may arise from the randomness of LLMs generation, which may generate different results for different runs. Namely, across different runs of the same prompt, the obtained metrics could vary. To mitigate this threat, we ran the LLM-related approaches (i.e., AdbGPT and ablation baselines) three times and the metrics were then from the aggregation of the three runs. Another potential confounding factor concerns the manual labeling of the ground-truth for S2R entity extraction (RQ1). To mitigate any potential subjectivity or errors, we gave the annotators a training session and a passing test before labeling. We asked them to independently annotate without any discussion and came to a consensus on the finalized ground-truth.

The main external threat to the validity of our work is the representative of the testing dataset selected to evaluate our approach. To mitigate this threat, we collected bug reports from previous related research, representing an unbiased testing dataset for our study. In addition, we randomly collected five practical bug reports to further evaluate the usefulness of our tool with five graduate students and three developers in real-world development environments. A potential confounding factor concerns the representation of graduate students in the user study. To mitigate this threat, five graduate students have at least 1.5-year experience in Android app development and bug reproduction, so they are recognized as substitutes for developers in software engineering research experiments [65].

6 RELATED WORK

We review the related work in two main areas: 1) bug record and replay, and 2) large language models for software engineering.

6.1 Bug Record and Replay

Nurmuradov et al. [59] introduced a record and replay tool for Android applications that captures user interactions by displaying the device screen in a web browser. It used event data captured during the recording process to generate a heatmap that allows developers to reproduce how users are interacting with an application. Additional tools including ECHO [69], Reran [37], Barista [46], and WeReplay [32] are program-analysis-related applications for the developers to record and replay user interactions. However, they required the installation of underlying frameworks such as replaykit [1], troyd [42], or instrumenting apps which are too heavy for end users. In contrast to these approaches, our AdbGPT is rather lightweight which just requires natural language descriptions of the bug reproduction steps.

Many previous research works have focused on processing bug reproduction steps to automate bug replay. For example, Fazzini et al. [26] proposed Yakusu, a program analysis and natural language processing tool to generate executable test cases from bug reports. Zhao et al. [82] further improved the tool namely ReCDroid, by leveraging the lexical knowledge in the bug reports to automatically reproduce crashes. However, these works did not consider the temporal order of S2Rs (e.g., when, after, etc.). Liu et al. [55] proposed MaCa, a natural language processing (NLP) and machine learning (ML) approach to normalize the S2Rs in the bug reports and extract the S2R entities more effectively. However, these approaches highly depended on the S2R writing in the bug report including formatting, word usage, and granularity [11, 14, 22, 49], which constrains its deployment in the real world. In contrast, our approach does not construct any sophisticated manually-crafted patterns and pre-defined vocabulary of words. We leverage a single model LLMs with novel prompt engineering to elicit the natural language understanding and logical reasoning to extract S2R entities and reproduce the bugs.

Some works attempted to assist reporting process [31, 33, 68, 78, 79] and improve the bug report quality [30, 34, 35, 58, 76]. For example, Chaparro et al. [15] developed DeMIBuD to detect missing information from bug reports. Fazzini et al. [25] proposed a mobile application EBug, suggesting accurate reproduction steps while the users are writing them. More recently, Yang et al. [66] proposed an interactive chatbot system BURT to guide users to report essential bug report elements. These approaches are complementary to our approach, as they increase bug report quality at the time of reporting which in turn, could potentially improve the capabilities of LLMs' understanding.

6.2 Large Language Models for Software Engineering

Following the success of LLMs in many natural language processing tasks, researchers have started exploring their potential in software engineering. Several studies have focused on automated code completion [21, 39, 41, 51, 80]. More recently, Codex [9] employed by Github Copilot has shown promising results in generating code from natural language comments, which has the potential to significantly reduce developers' coding efforts, although some minor issues still exist. This has sparked interest in using LLMs to resolve code issues through program patching [24, 36, 44, 48]. For instance,

AlphaRepair, introduced by Xia et al. [77], effectively encoded the intricate relations between patches and the contexts of codes using LLMs. Additionally, some studies have examined the impact of LLMs on software development [40, 57], developer productivity [20, 71, 86], and security vulnerabilities [38, 61, 62]. LLMs remain an ongoing research topic in the software engineering community. Our work opens up a new possibility for integrating LLMs and GUI understanding, enabling LLMs to understand S2Rs and guide dynamic GUI screens to automated bug reproduction.

7 CONCLUSION

Inspired by the success of Large Language Models (LLMs) in natural language understanding, this paper proposes AdbGPT, a lightweight approach to automatically reproduce the bugs from the S2Rs based on prompt engineering. In detail, we first prompt the LLMs by giving entity specifications, a few representative exemplars, and developers' step-by-step reasoning, to help LLMs elicit the knowledge to extract the entities from the S2Rs in the bug report like a developer expert. Given each step entity, we prompt the LLMs by giving the current GUI screen and few-shot learning with chain-of-thought reasoning to dynamically guide the probable target components to reproduce the bugs. The experiments and user study demonstrate the effectiveness, efficiency, and usefulness of our approach in accelerating bug reproduction.

In the future, we will keep improving our AdbGPT for better effectiveness. For example, a bug report usually contains a wealth of information, including stack trace, error logs, screenshots, screen recordings, etc. We could take this information into the consideration to enhance the understanding of LLMs to the bugs.

REFERENCES

[1] 2021. Command line tools for recording, replaying and mirroring touchscreen events for Android. https://github.com/appetizerio/replaykit.
[2] 2023. AdbGPT. https://github.com/sidongfeng/AdbGPT.
[3] 2023. Android Debug Bridge (adb) - Android Developers. https://developer.android.com/studio/command-line/adb.
[4] 2023. Android Uiautomator2 Python Wrapper. https://github.com/openatx/uiautomator2.
[5] 2023. Genymotion – Android Emulator for app testing. https://www.genymotion.com/.
[6] 2023. Introducing ChatGPT. https://chat.openai.com/.
[7] 2023. Number of apps available in leading app stores. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.
[8] 2023. An open letter to GitHub from the maintainers of open source projects. https://github.com/dear-github/dear-github.
[9] 2023. OpenAI Codex. https://openai.com/blog/openai-codex.
[10] Jorge Aranda and Gina Venolia. 2009. The secret life of bugs: Going past the errors and omissions in software repositories. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 298–308.
[11] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What makes a good bug report?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. 308–318.
[12] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. 2008. Extracting structural information from bug reports. In *Proceedings of the 2008 international working conference on Mining software repositories*. 27–30.
[13] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
[14] Yu Cao, Hongyu Zhang, and Sun Ding. 2014. Symcrash: Selective recording for reproducing crashes. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 791–802.
[15] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. 2017. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 396–407.
[16] Chunyang Chen, Sidong Feng, Zhengyang Liu, Zhenchang Xing, and Shengdong Zhao. 2020. From lost to found: Discover missing ui design semantics through recovering missing tags. *Proceedings of the ACM on Human-Computer Interaction* 4, CSCW2 (2020), 1–22.
[17] Chunyang Chen, Sidong Feng, Zhenchang Xing, Linda Liu, Shengdong Zhao, and Jinshui Wang. 2019. Gallery dc: Design search and knowledge discovery through auto-created gui component gallery. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (2019), 1–22.
[18] 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.
[19] 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).
[20] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Zhen Ming Jiang. 2023. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software* (2023), 111734.
[21] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.
[22] Mona Erfani Joorabchi, Mehdi Mirzaaghaei, and Ali Mesbah. 2014. Works for me! characterizing non-reproducible bug reports. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 62–71.
[23] Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. 2022. Automated Repair of Programs from Large Language Models. ICSE.
[24] Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Improving automatically generated code from Codex via Automated Program Repair. In *Proceedings of the 45th International Conference on Software Engineering*.
[25] Mattia Fazzini, Kevin Patrick Moran, Carlos Bernal-Cardenas, Tyler Wendland, Alessandro Orso, and Denys Poshyvanyk. 2022. Enhancing Mobile App Bug Reporting via Real-time Understanding of Reproduction Steps. *IEEE Transactions on Software Engineering* (2022).
[26] Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. 2018. Automatically translating bug reports into test cases for mobile apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 141–152.
[27] Sidong Feng and Chunyang Chen. 2022. GIFdroid: an automated light-weight tool for replaying visual bug reports. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 95–99.
[28] Sidong Feng and Chunyang Chen. 2022. GIFdroid: automated replay of visual bug reports for Android apps. In *Proceedings of the 44th International Conference on Software Engineering*. 1045–1057.
[29] Sidong Feng, Chunyang Chen, and Zhenchang Xing. 2022. Gallery DC: Auto-created GUI component gallery for design search and knowledge discovery. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 80–84.
[30] Sidong Feng, Chunyang Chen, and Zhenchang Xing. 2023. Video2Action: Reducing Human Interactions in Action Annotation of App Tutorial Videos. *arXiv preprint arXiv:2308.03252* (2023).
[31] Sidong Feng, Minmin Jiang, Tingting Zhou, Yankun Zhen, and Chunyang Chen. 2022. Auto-Icon+: An Automated End-to-End Code Generation Tool for Icon Designs in UI Development. *ACM Transactions on Interactive Intelligent Systems* 12, 4 (2022), 1–26.
[32] Sidong Feng, Haochuan Lu, Ting Xiong, Yuetang Deng, and Chunyang Chen. 2023. Towards Efficient Record and Replay: A Case Study in WeChat. *arXiv preprint arXiv:2308.06657* (2023).
[33] Sidong Feng, Suyu Ma, Jinzhong Yu, Chunyang Chen, Tingting Zhou, and Yankun Zhen. 2021. Auto-icon: An automated code generation tool for icon designs assisting in ui development. In *26th International Conference on Intelligent User Interfaces*. 59–69.
[34] Sidong Feng, Mulong Xie, and Chunyang Chen. 2023. Efficiency Matters: Speeding Up Automated Testing with GUI Rendering Inference. In *Proceedings of the 45th International Conference on Software Engineering*.
[35] Sidong Feng, Mulong Xie, Yinxing Xue, and Chunyang Chen. 2023. Read It, Don't Watch It: Captioning Bug Recordings Automatically. In *Proceedings of the 45th International Conference on Software Engineering*.
[36] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 935–947.
[37] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. 2013. Reran: Timing-and touch-sensitive record and replay for android. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 72–81.

[38] Anastasiia Grishina. 2022. Enabling automatic repair of source code vulnerabilities using data-driven methods. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 275–277.

[39] Qing Huang, Zhiqiang Yuan, Zhenchang Xing, Xiwei Xu, Liming Zhu, and Qinghua Lu. 2022. Prompt-tuned Code Language Model as a Neural Knowledge Base for Type Inference in Statically-Typed Partial Code. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[40] Saki Imai. 2022. Is GitHub copilot a substitute for human pair-programming? An empirical study. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 319–321.

[41] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*. 1219–1231.

[42] Jinseong Jeon and Jeffrey S Foster. 2012. *Troyd: Integration testing for android*. Technical Report.

[43] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring program transformations from singular examples via big code. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 255–266.

[44] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.

[45] Jack Johnson, Junayed Mahmud, Tyler Wendland, Kevin Moran, Julia Rubin, and Mattia Fazzini. 2022. An empirical investigation into the reproduction of bug reports for android apps. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 321–322.

[46] Andrew J Ko and Brad A Myers. 2006. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*. 387–396.

[47] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large Language Models are Zero-Shot Reasoners. In *Advances in Neural Information Processing Systems*.

[48] Sophia D Kolak, Ruben Martins, Claire Le Goues, and Vincent Josua Hellendoorn. 2022. Patch Generation with Language Models: Feasibility and Scaling Behavior. In *Deep Learning for Code Workshop*.

[49] A Gunes Koru and Jeff Tian. 2004. Defect handling in medium and large open source projects. *IEEE software* 21, 4 (2004), 54–61.

[50] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[51] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[52] Yafu Li, Yongjing Yin, Jing Li, and Yue Zhang. 2022. Prompt-Driven Neural Machine Translation. In *Findings of the Association for Computational Linguistics: ACL 2022*. 2579–2590.

[53] 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.

[54] Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. 2017. Program Induction by Rationale Generation: Learning to Solve and Explain Algebraic Word Problems. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 158–167.

[55] Hui Liu, Mingzhu Shen, Jiahao Jin, and Yanjie Jiang. 2020. Automated classification of actions in bug reports of mobile apps. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 128–140.

[56] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[57] 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. (2023).

[58] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2015. Auto-completing bug reports for android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 673–686.

[59] Dmitry Nurmuradov and Renee Bryce. 2017. Caret-HM: recording and replaying Android user sessions with heat map generation using UI state clustering. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 400–403.

[60] Yasumasa Onoe, Michael Zhang, Eunsol Choi, and Greg Durrett. 2022. Entity Cloze By Date: What LMs Know About Unseen Entities. In *Findings of the Association for Computational Linguistics: NAACL 2022*. 693–702.

[61] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2021. An empirical cybersecurity evaluation of github copilot's code contributions. *ArXiv abs/2108.09293* (2021).

[62] 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.

[63] Strategic Planning. 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology* (2002).

[64] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[65] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. 2015. Are students representatives of professionals in software engineering experiments?. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 666–676.

[66] Yang Song, Junayed Mahmud, Ying Zhou, Oscar Chaparro, Kevin Moran, Andrian Marcus, and Denys Poshyvanyk. 2022. Toward interactive bug reporting for (android app) end-users. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 344–356.

[67] Ting Su, Jue Wang, and Zhendong Su. 2021. 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*. 119–130.

[68] Yuhui Su, Chunyang Chen, Junjie Wang, Zhe Liu, Dandan Wang, Shoubin Li, and Qing Wang. 2022. The Metamorphosis: Automatic Detection of Scaling Issues for Mobile Apps. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[69] Yulei Sui, Yifei Zhang, Wei Zheng, Manqing Zhang, and Jingling Xue. 2019. Event trace reduction for effective bug replay of Android apps via differential GUI state analysis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1095–1099.

[70] Viriya Taecharungroj. 2023. "What Can ChatGPT Do?" Analyzing Early Reactions to the Innovative AI Chatbot on Twitter. *Big Data and Cognitive Computing* 7, 1 (2023), 35.

[71] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*. 1–7.

[72] 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* 30 (2017).

[73] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, and Denny Zhou. 2022. Rationale-augmented ensembles in language models. *arXiv preprint arXiv:2207.00747* (2022).

[74] Yaqing Wang, Quanming Yao, James T Kwok, and Lionel M Ni. 2020. Generalizing from a few examples: A survey on few-shot learning. *ACM computing surveys (csur)* 53, 3 (2020), 1–34.

[75] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed H Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*.

[76] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *2014 IEEE international conference on software maintenance and evolution*. IEEE, 181–190.

[77] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 959–971.

[78] Mulong Xie, Sidong Feng, Zhenchang Xing, Jieshan Chen, and Chunyang Chen. 2020. UIED: a hybrid tool for GUI element detection. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1655–1659.

[79] Mulong Xie, Zhenchang Xing, Sidong Feng, Xiwei Xu, Liming Zhu, and Chunyang Chen. 2022. Psychologically-inspired, unsupervised inference of perceptual groups of GUI widgets from GUI images. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 332–343.

[80] 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.

[81] Yu Zhao, Ting Su, Yang Liu, Wei Zheng, Xiaoxue Wu, Ramakanth Kavuluru, William GJ Halfond, and Tingting Yu. 2022. ReCDroid+: Automated End-to-End Crash Reproduction from Bug Reports for Android Apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–33.

[82] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William GJ Halfond. 2019. Recdroid: automatically reproducing android application crashes from bug reports. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 128–139.

[83] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Olivier Bousquet, Quoc Le, and Ed Chi. 2023. Least-to-most prompting enables complex reasoning in large language models. In *2023 11th International Conference on Learning Representations (ICLR)*.

[84] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International conference on software engineering (ICSE)*. IEEE, 14–24.

[85] Kaiyang Zhou, Jingkang Yang, Chen Change Loy, and Ziwei Liu. 2022. Learning to prompt for vision-language models. *International Journal of Computer Vision* 130, 9 (2022), 2337–2348.

[86] Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 21–29.