VisiDroid: An Approach for Generating Test Scripts from Task Descriptions for Mobile Testing

Hai Phung[1,2][0009−0000−1481−1013], Hao Pham[1,2][0009−0003−7633−6731], Tien Nguyen[4][0009−0006−7962−6090], and Vu Nguyen[1,2,3][0000−0002−0594−4372]✉

[1] Faculty of Information Technology, University of Science, Ho Chi Minh city, Vietnam
[2] Vietnam National Univerisity, Ho Chi Minh city, Vietnam
20127018@student.hcmus.edu.vn, 20127155@student.hcmus.edu.vn, ✉nvu@fit.hcmus.edu.vn
[3] Katalon Inc.
[4] University of Texas at Dallas, Texas, USA
tien.n.nguyen@utdallas.edu

Abstract. Testing user interface and system-level functionality of a mobile app is crucial for ensuring its quality. However, it is becoming increasingly costly due to the complexity of modern applications and the diverse range of devices. Recent approaches have focused on exploring entire applications to test and detect defects in mobile apps. Additionally, they do not consider the ability to guide and restrict large language models (LLMs) based on user-defined rules. This paper introduces VisiDroid, an approach to generating scripts for mobile testing from task goals or natural language descriptions by leveraging the capabilities of LLMs. We evaluate the approach using an open-source dataset consisting of 131 tasks on 11 mobile apps. The results show that VisiDroid can accurately generate actions and achieves a task completion rate of 75.5%, outperforming the state-of-the-art approach. It also successfully generates valid test scripts with an 80.05% success rate overall.

Keywords: Test script generation · Task Automation · Mobile GUI Testing.

1 Introduction

Testing mobile apps at the graphical user interface (GUI) and system levels is essential to ensuring their expected quality and functionality. However, this process is time-consuming due to the complexity of modern applications, the wide variety of devices and platforms, and the frequent changes throughout the software development life cycle.

A previous study shows that the effectiveness of GUI mobile app testing is significantly improved by concentrating on test cases that target specific features and individual use cases [10]. When testing, developers design and perform tests focusing on how users generally use the apps. Test scripts and test cases are often constructed using natural language inputs in practical scenarios.

Leveraging Large Language Models (LLMs) has emerged for test generation as a promising direction to enhance the efficiency and accuracy of GUI mobile app testing, thanks to their unique capabilities like instruction following [15] and step-by-step reasoning [14]. LLMs have the potential to automate Android problem-solving systems by understanding natural language and adapting to the context. However, they require appropriate prompts and an understanding of the context to execute actions accurately.

In response to the challenges associated with GUI testing, significant research efforts have been dedicated to automating various components of mobile GUI testing [18, 13, 16]. Despite this, the complexity and unpredictability of mobile applications make it challenging to determine task completion based only on GUI descriptions.

In this paper, we propose VisiDroid, an agent designed to generate test scripts based on task goal provided through natural language input, thereby minimizing the need for manual effort. Utilizing LLMs with self-improvement techniques, our approach enhances the testing process through two distinct phases, leveraging a Vision API of OpenAI [11] to analyze actual device screen images rather than relying solely on the Android application's Document Object Model (DOM). In the training phase, the agent explores the operational environment, identifying buttons, observing patterns, and extracting rules and optimized steps, with all acquired data stored in a persistent memory. During the evaluation phase, the agent utilizes the relevant data obtained from the training phase to efficiently complete tasks without redundant steps.

In our approach, a memory module is introduced to store data between runs, enabling a self-improving mechanism. This approach aims to streamline the testing process, reducing the time and costs associated with manual testing.

To determine whether a task is completed or not, we use the Vision API [11], which enables our agent to visually inspect the screen and precisely confirm that the task has been completed.

We evaluate the proposed approach using an open-source dataset consisting of 131 tasks on 11 mobile apps. The results show that VisiDroid can accurately generate actions and achieves a task completion rate of 75.5%, outperforming the state-of-the-art approach [5]. It also successfully generates valid test scripts with an 80.05% success rate overall.

The remaining of the paper is organized as follows: Section 2 summarizes related studies. Section 3 presents our proposed approach. Section 4 describes our experiment to evaluate the proposed approach. Sections 5 and 6 offer discussions and threats to validity, and Section 7 presents conclusions of the paper.

2 Related Work

Android GUI testing. Several studies present strategies for generating scripts for Android GUI testing. One strategy is to explore mobile apps randomly, as proposed in Google's Monkey tool [3]. Humanoid [7] represents a learning-based strategy. Additionally, model-based strategies are employed [2].

LLMs integrated. Recently, approaches have shifted towards using LLMs to enhance the effectiveness of test script generation [5, 6, 4]. For instance, GPT-Droid [9] demonstrated that large language models (LLMs) can select a human-like next action for continued exploration, using a summary of previous exploration and descriptions of the current GUI state. Lately, DroidAgent [6] employed GPT-4 with long-term and short-term memory to systematically explore apps and generate GUI test cases. It effectively creates test cases and achieves high activity coverage, but its capacity to address developer-specific tasks is still somewhat limited. The other method, AutoDroid [5] gathers app-specific knowledge offline by exploring UI relations and synthesizing tasks, and then uses memory-augmented LLMs online to guide actions. However, its ability to determine task completion remains limited because it relies on UI descriptions, which can vary significantly among developers [17, 12, 8].

3 Methodology

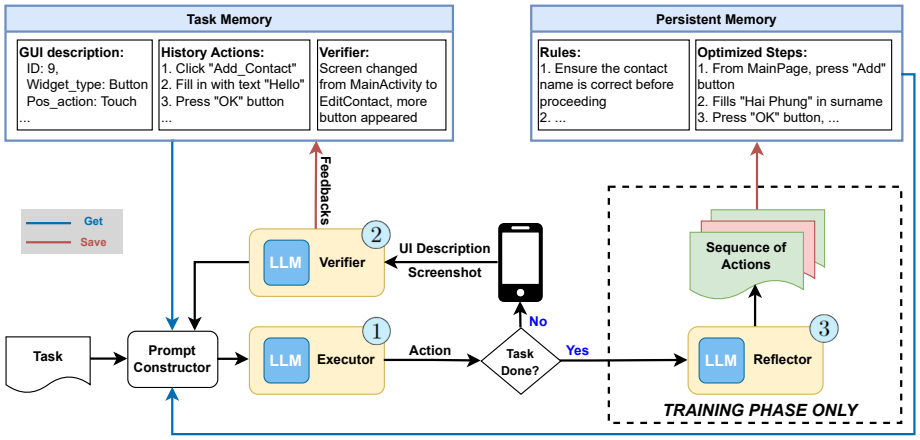


Fig. 1: Overview of VisiDroid's Architecture

Figure 1 presents an overview architecture of VisiDroid. The architecture includes modules for both training and evaluation phases. It comprises three LLM-based components: Executor, Verifier, and Reflector. VisiDroid maintains experience-based learning by using the Persistent Memory component to store knowledge gained from previously executed similar tasks and the Task Memory component to retain the history and feedback of each step.

3.1 Memory

VisiDroid includes two main types of memory, Task Memory and Persistent Memory, to increase the overall knowledge of the current task and improve performance.

Task Memory: This memory contains the knowledge and information of the currently performed task. It includes the current GUI description and the history of actions taken. This memory is reset after each run to ensure its distinctiveness.

Persistent Memory: Since LLMs can self-learn and continuously improve their performance from past mistakes, we use Persistent Memory to leverage this capability. This memory stores rules and steps refined by the Reflector component during the training phase. These memory is used to guide the agent to follow the most accurate path based on all the previous runs.

3.2 Executor

This component determines appropriate actions to interact with the mobile app.

Prompt Constructor: We utilize the chain-of-thought prompting technique [14], which requires the LLM to reason through the problem before making decisions. Executor is prompted with the current GUI state, a history of actions, feedback from Verifier about previous actions, and the current screen (detailed in the next section). This information is stored in Task Memory. Information about rules and steps is retrieved from Persistent Memory to guide Executor in performing and following the correct path.

Action Decision: The Executor component determines the next action to perform. Possible actions include scrolling, touching, long-touching, filling forms, navigating back, or ending a task. It can also pause if the LLM detects a loading screen.

3.3 Verifier

The Verifier component is responsible for monitoring the results of actions and providing feedback to Executor. Figure 2 illustrates its capability.

Vision Capability: This component can monitor changes in the GUI description and the screenshot state, leveraging OpenAI's Vision API. A previous study in Android automated tasks [5] indicates the challenge of accurately determining task completion. One issue is that LLMs can be misled by changes that do not directly affect the GUI description, such as theme or font size changes. The Verifier component in VisiDroid addresses this limitation by observing both the GUI description in JSON format and the actual screen. This dual observation ensures more accurate task completion decisions, reduces excessive queries to LLMs, and ultimately lowers costs.

Task done determination: Verifier is responsible for determining if the task is completed successfully by examining the screen and the GUI description. This approach enables the LLM to make more accurate decisions. If the task is still incomplete, Verifier provides suggestions for the next action. This feedback

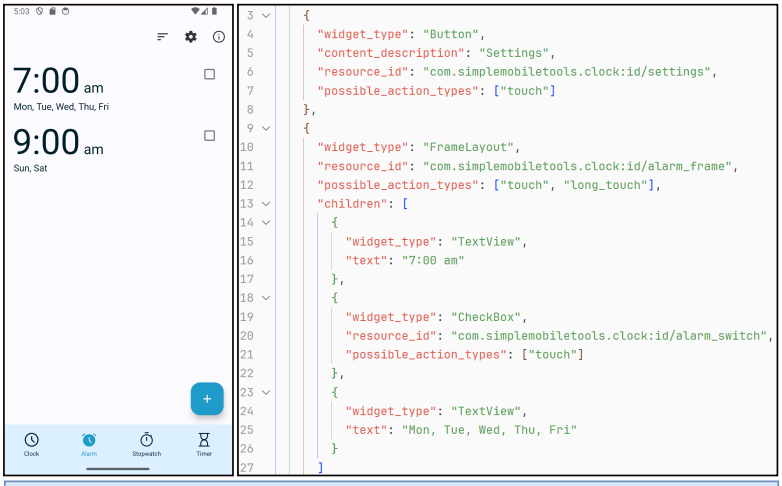


Fig. 2: An example of how the Verifier component gives feedback based on the observation of both the GUI description using JSON and the screenshot.

is essential as it can identify mistakes from the action history and execute a back action to correct the task. Without the Verifier's monitoring, the Executor component would be unaware of deviations, leading to unnecessary exploration.

3.4 Reflector

The Reflector component is responsible for summarizing the entire task execution history. Reflector's importance lies in its ability to determine whether previously performed tasks succeeded or failed and derive Rules and Optimized Steps for VisiDroid to follow in subsequent runs.

Training phase: Reflector updates Persistent Memory automatically upon completing each iterative planning cycle. Generally, if an agent's run is successful, it is appended to the set of Optimized Steps for future use, as LLMs can follow established patterns. While failed attempts are not explicitly used to prevent LLMs from learning incorrect patterns, they are analyzed to extract Rules expressed in natural language. The extracted Rules and Optimized Steps are then used to guide the VisiDroid, helping it avoid previous mistakes.

Evaluation phase: In this phase, Persistent Memory remains fixed, and the LLM operates solely based on the most helpful Rules and Optimized Steps that are finalized during the training phase. This elaborate reflection from the

training phase can increase the overall success rate per task as well as reduce unnecessary and excessive steps.

3.5 Test script generation

VisiDroid aims to generate test cases and test scripts based on task descriptions in natural language. Although tools like DroidAgent [6] help generate mobile interaction scripts, they lack assertion capabilities.

Automating assertion generation presents significant challenges, given the diverse nature of Android tasks. We classify tasks into three main groups:

- GUI changes, text assertionable: Example: *'Create a new contact for Stephen Bob, with the mobile number 12345678900'*. The new contact appears on the main screen.
- No GUI changes, screenshot assertionable: Example: *'Change the theme color to light'* The Verifier component analyzes screenshots to confirm completion.
- No observable changes: Example: *'Export all messages'*. Verifying these tasks is challenging, as there is no visible outcome or popup confirmation.

To handle the high variability of mobile tasks, our approach uses **screenshot assertions** to validate test cases instead of relying on text assertions. VisiDroid utilizes data from Persistent Memory and the evaluation phase to generate test scripts, enabling cross-platform testing on various devices and platforms.

VisiDroid only uses one successful run by analyzing the final screenshot of the task to determine the most suitable one. Afterward, the agent takes a screenshot upon completion of the final step in the script. This screenshot is then compared to the final screenshot of the selected run using the Vision API[11]. If the comparison result is positive, it indicates that the script has been executed successfully; otherwise, it indicates failure.

4 Evaluation

4.1 Experimental Design

Research Questions.

Our evaluation of VisiDroid focuses on exploring the following research questions:

RQ1. [Action Sequence Generation Accuracy]. How well does VisiDroid generate task-specific action sequences in comparison with other methods?

RQ2. [Analysis of Experience-Based Learning and Visual Perception]. To what degree do the experience-based learning and visual perception capabilities improve the agent's effectiveness?

RQ3. [Test Script Generation Capability]. How effective is VisiDroid in producing test scripts?

Dataset. We evaluate VisiDroid using the DroidTask dataset, which is publicly available and provided in [5]. However, due to VisiDroid's runtime focus on

executing tasks within a single application's activity, certain tasks and applications were deemed unsuitable for evaluation. In this experiment, we test on 131 high-level tasks exrracted from 11 popular applications found on the F-Droid repository [1].

Hardware. To evaluate VisiDroid's performance, we use the Android emulator in Android Studio. The emulator is configured to mimic a Pixel 3a device running Android 14.0 (UpsideDownCake API level 34). The emulator runs on a 64-bit Windows 11 machine with an R7-7840HS CPU (8 cores) and 32GB memory.

Setup. To optimize cost-efficiency, the Executor component uses GPT-3.5. The Verifier component also employs GPT-3.5 for interpreting GUI descriptions and GPT Vision for the visual processing of screenshots while the Reflector component uses GPT-4.0.

Baselines. We compare VisiDroid with DroidBot-GPT [4] and AutoDroid [5], both using LLMs for mobile task automation. We chose AutoDroid and DroidBot-GPT as they use task descriptions as input but they do not include visual perception capabilities. AutoDroid also explores all available paths within an app while VisiDroid focuses on only the target task and gathers information only relevant to the specific path completing the task.

Metrics. We utilize the Action Accuracy and Completion Rate metrics, as introduced in the AutoDroid paper [5]. Using these established metrics, we maintain consistency in our evaluation approach and enable direct comparisons with prior research. These metrics are calculated based on a sequence of user interfaces (UIs) $\{U_1, U_2, \ldots, U_n\}$ where human annotators performed actions $A = \{A_1, A_2, \ldots, A_n\}$ to complete a task T. For an agent making a sequence of decisions $\hat{A} = \{\hat{A}_1, \hat{A}_2, \ldots, \hat{A}_n\}$ on the same sequence of UIs, the metrics are defined as follows:

- **Action Accuracy**: This metric is the ratio of actions $\hat{A}_i$ that match the ground-truth actions A_i, denoted as $P(\hat{A}_i = A_i)$. An action is considered correct only if both the target UI element and the input text are accurate. Action Accuracy measures the agent's ability to make correct decisions based on the given information.
- **Completion Rate**: This metric is the probability of the agent completing all actions in the sequence correctly, denoted as $P(\hat{A} = A)$. It reflects the likelihood of the agent consistently and successfully completing an entire task without errors.

4.2 Action Sequence Generation Accuracy (RQ1)

Table 1 shows task completion rates of the methods including ours, DroidBot-GPT, and AutoDroid. The results show that VisiDroid GPT-3.5 outperforms AutoDroid GPT-3.5 and DroidBot-GPT by 35.2% and 48.1%, respectively. It also results in a slightly higher rate than does AutoDroid using GPT-4.0. Our further analysis reveals that these improvements are a result of VisiDroid's approach focusing on self-reflection. For example, in the task *'Create a new contact*

Agent	Completion Rate
DroidBot-GPT	27.4%
AutoDroid GPT-3.5	40.3%
AutoDroid GPT-4.0	71.3%
VisiDroid	**75.5%**

Table 1: Comparison of Completion Rate of VisiDroid with baseline methods

Stephen Bob, mobile number 12345678900', VisiDroid initially fills in excessive fields that are not needed for the task. However, VisiDroid learns to correct these mistakes after the training phase.

Metric	Touch	Input	Complete	Overall
AutoDroid GPT-3.5	72.10%	62.50%	41.80%	65.10%
AutoDroid GPT-4.0	91.20%	82.50%	93.70%	90.90%
VisiDroid	87.10%	81.30%	**96.20%**	89.45%

Table 2: Comparison of Action Accuracy of VisiDroid and AutoDroid

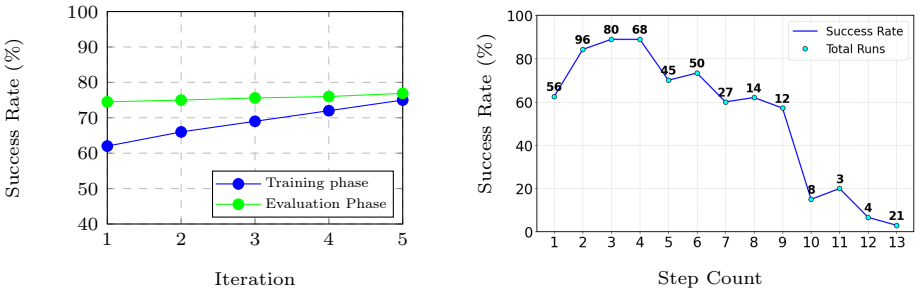
Action Accuracy is presented in Table 2. Three most common interactions on mobile devices are included: Touch, Input, and Complete (determining whether the task is completed). As shown in Table 2, VisiDroid significantly outperforms AutoDroid GPT-3.5 while its action accuracy is comparable to AutoDroid GPT-4.0 (although VisiDroid employs mainly GPT-3.5).

We conducted an analysis of failure cases, noting that while VisiDroid can determine actions using both UI description and screenshots, it has limitations with applications that have complex user interfaces. For instance, in the task *add a new event on Jan 1st* within *Calendar* applications, the agent must repeatedly navigate through the month or week panels, depending on the settings. However, the UI description includes numerous elements, leading to confusion after each navigation, and in certain scenarios, the agent may exceed the step limit before reaching the January month.

4.3 Experience-Based Learning and Visual Perception (RQ2)

Figure 3a shows VisiDroid's utilization of Persistent Memory to enhance performance across multiple runs. The moving average in the training phase depicts a consistent upward trend, indicating ongoing improvement, while the evaluation phase remains stable. Additionally, Figure 3b shows that tasks with fewer steps generally achieved higher success rates than more complex tasks, which are often associated with a greater number of steps. VisiDroid supports real-world usage

without requiring application reinstalls after each iteration, enhancing its vision capabilities. Interestingly, single-step executions showed lower success rates compared to multi-step tasks (2, 3, or 4 steps). For example, Task *'Change bitrate to 96 kbps'* involved accessing the settings to adjust the bitrate to 96 kbps. After successfully completing the task on the first attempt, subsequent runs had the Verifier check the settings, recognize they were already configured, and terminate the task early.



(a) Moving average calculated across two phases

(b) Average success rate per run at a specific step count

Fig. 3: Comparison of success rate trends across different phases and step counts

Furthermore, visual analysis significantly reduced unnecessary and excessive steps, particularly for tasks where changes are not directly reflected in the GUI description, such as theme changes or toggling favorites. This reduction is evident in the average steps measured, with VisiDroid requiring only **4.2** steps on average compared to **6.8** steps without applying the visual perception capabilities.

4.4 Test Script Generation Capability (RQ3)

Test Script	**Success Rate**
AutoDroid w/ GPT assertion	61.20%
VisiDroid - Valid	**80.05%**
VisiDroid - Optimal	**71.75%**

Table 3: Comparison of Success Rate for Test Scripts generated by VisiDroid

AutoDroid does not support generating test scripts with assertions. Therefore, we used its execution history to prompt GPT to generate assertions. After generating these assertions, we evaluated their effectiveness.

As presented in Table 3, out of 131 test scripts generated by VisiDroid, 80.05% were valid and executed successfully. The bottom row (VisiDroid - Optimal) shows a lower success rate of 71.75% for optimal test scripts, which means all steps were correct, with no unnecessary actions, and the final screenshot assertion was valid.

We examined the causes of failed test scripts, revealing that one important cause was permission pop-ups occasionally appearing during test script execution, resulting in script failure. Another cause is that preconditions were not satisfied to complete a task. For example, for tasks involving data interactions like *'Find a Contact Alice'*, Alice needs to be present before executing the test script for the task.

4.5 Cost and execution time

We analyzed GPT's token usage for both prompt requests and responses. The number of tokens depends on the complexity of the GUI layout and the task at hand.

Our analysis shows that a single task execution typically requires between 15,000 and 70,000 tokens, with an average of 33,000 tokens. Using the GPT-4.0 model, this translates to approximately $0.12 per run. In contrast, using GPT-3.5 reduces the cost to $0.005 per run.

In terms of execution time, a single run typically takes between 0.5 and 2 minutes, with longer durations for more challenging tasks. The training phase generally takes more time than the evaluation phase due to the need for exploration.

5 Discussions

Challenges of screenshot assertions. VisiDroid uses screenshots to determine if the script operates as expected. Nevertheless, this approach may be unreliable because of the preconditions of each test script and the dynamic data within the app during execution.

Runtime Cost. While the Verifier component can analyze app screenshots and eliminate the need to manually determine task completion, integrating the Vision API from OpenAI can be costly. After each action performed by the Executor, the Verifier is triggered, which adds to the overall cost. Currently, we query the LLMs using a low-detail option, provided by the API, to reduce costs. Further research is needed to minimize the constant reliance on screenshot observations, such as implementing strategies to observe only after several steps.

CI/CD Integration. As a potential future enhancement, we envision integrating our approach into Continuous Integration/Continuous Deployment (CI/CD) pipelines, which could further minimize the need for manual testing in mobile app development. This integration would enable automated and consistent test execution, aligning with the goals of efficient and reliable software delivery.

6 Threats to validity

Internal validity. Our approach is susceptible to hallucinations inherent in LLMs. Since we rely on API calls to these models, we have limited control over the process, which may introduce bias. Additionally, VisiDroid may generate an incorrect sequence of steps when dealing with applications that require permissions or involve navigating to external applications.

External validity. Our approach relies on a limited dataset of tasks and applications, which may hinder its ability to generalize across diverse scenarios. We attempted to run VisiDroid on real-world applications like Messenger and Telegram, but the excessive number of elements in these applications can push the context limits of LLMs. Future work should focus on developing techniques to compress the UI elements into more concise UI descriptions to address this challenge.

7 Conclusions

In this paper, we have proposed VisiDroid, an agent designed to generate test scripts from task descriptions for mobile app testing. Utilizing LLMs with self-improvement techniques, our approach consists of two phases: training and evaluation. It leverages OpenAI's Vision API to analyze application screenshots, rather than relying solely on the DOM. Our experiments on 11 apps shows that VisiDroid could complete tasks of varying complexity. The results also indicate that LLM-based agents could self-improve during the training phase by acquiring helpful information. For VisiDroid, this information consists of steps and rules, which enhance performance in the evaluation phase. These results indicate that autonomous agents have the potential to advance GUI testing automation and reduce the need for manual effort.

8 Acknowledgements

This research is partially supported by research funding from the Faculty of Information Technology, University of Science, VNU-HCM, Vietnam.

References

1. Fdroid: Free and open source android app repository, https://f-droid.org/en/
2. Amalfitano, D., Fasolino, A.R., Tramontana, P., Ta, B.D., Memon, A.M.: Mobiguitar: Automated model-based testing of mobile apps. IEEE Software **32**(5), 53–59 (2014)
3. Google: https://developer.android.com/studio/test/other-testing-tools/monkey (2022)
4. Hao, Wen, Hongmingand Wang, J., Liu, Y., Li: Droidbot-gpt: Gpt-powered ui automation for android. arXiv:2304.07061v5 (2024)

5. Hao, Wen, Y., Li, G., Liu, S., Zhao, T., Yu, T., Jia-Jun Li, S., Jiang, Y., Liu, Y., Zhang, Y., Liu: Empowering llm to use smartphone for intelligent task automation. arXiv:2308.15272 (2023)
6. Juyeon, Yoon, R., Feldt, S., Yoo: Autonomous large language model agents enabling intent-driven mobile gui testing. arXiv:2311.08649v1 (2023)
7. Li, Y., Yang, Z., Guo, Y., Chen, X.: 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). pp. 1070–1073. IEEE (2019)
8. Lin, J.W., Malek, S.: Gui test transfer from web to android. In: 2022 IEEE Conference on Software Testing, Verification and Validation. pp. 1–11. IEEE (2022)
9. Liu, Z., Chen, C., Wang, J., Chen, M., Wu, B., Che, X., Wang, D., Wang, Q.: Chatting with gpt-3 for zero-shot human-like mobile automated gui testing. arXiv preprint arXiv:2305.09434 (2023)
10. Mario, Linares-Vasquez, C., Bernal-Cardenas, K., Moran, D., Poshyvany: How do developers test android applications. 2017 IEEE International Conference on Software Maintenance and Evolution (2017)
11. OpenAI: Openai vision api. https://platform.openai.com/docs/guides/vision (2024)
12. Qin, X., Zhong, H., Wang, X.: Testmig: Migrating gui test cases from ios to android. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 284–295 (2019)
13. Shengcheng, Yu, C., Fang, Y., Ling, C., Wu, Z., Chen: Llm for test script generation and migration: Challenges, capabilities, and opportunities. arXiv:2309.13574 (2023)
14. Shunyu, Yao, D., Yu, J., Zhao, I., Shafran, T., L. Griffiths, Y., Cao, K., Narasimhan: Tree of thoughts: Deliberate problem solving with large language models. arXiv:2305.10601 (2023)
15. Taori, R., Gulrajani, I., Zhang, T., et al.: Stanford alpaca: An instruction-following llama model. GitHub repository (2023)
16. Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., et al.: A survey on large language model based autonomous agents. Frontiers of Computer Science **18**(6), 1–26 (2024)
17. Yu, S., Fang, C., Yun, Y., Feng, Y.: Layout and image recognition driving cross-platform automated mobile testing. In: IEEE/ACM 43rd International Conference on Software Engineering. pp. 1561–1571. IEEE (2021)
18. Zhe, Liu, C., Chen, J., Wang, M., Chen, B., Wu, X., Che, D., Wang, Q., Wang: Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions. arXiv:2310.15780v1 (2023)