UICOMPASS: UI Manual Guided Mobile Task Automation via Adaptive Instruction Replanning

Anonymous ACL submission

Abstract

Large language model(LLM)-based agents have been widely applied in the field of mobile task automation. However, LLMs, which are proficient in general task execution practices, often struggle to execute tasks correctly on specific applications due to a lack of application-specific knowledge, leading to confusion and errors. Although existing methods use exploration-memory mechanisms to mitigate this issue, excessive exploration on user devices is unacceptable, and these mechanisms still struggle to handle tasks effectively. In this work, we propose a method for assisting agents in mobile task completion using a User Interface Manual, called UICOMPASS. Specifically, it first automates the extraction of the User Interface Manual from the source code, which describes the application's interface and interaction logic. During execution, it analyzes the User Interface Manual to generate simulation paths for the given task and adaptively adjusts the execution path based on the actual application state. Experiments show that UICOMPASS achieves state-of-the-art performance on the DroidTask dataset, with a success rate improvement of 14.48%, and a reduction in the length of execution paths.

1 Introduction

With the increasing prevalence of mobile devices, a vast array of diverse mobile applications has gradually become an inseparable part of our daily lives(Islam et al., 2010). These applications boast a wide range of functionalities and varied designs, supporting numerous daily tasks such as online ordering, chatting, and reading. However, the complexity of these application designs imposes a significant learning cost on users. For instance, the option to "enable night mode" might be located in a side drawer or within the settings activity, or it might not be supported in the app. Users need to constantly try out these applications to understand them, which severely impacts the user experience. Therefore, designing an intelligent assistant capable of automating the use of applications according to user goals has the potential to enhance user experience and assist users in completing their tasks automatically in inconvenient scenarios (Xing et al., 2024; Wang et al., 2024).

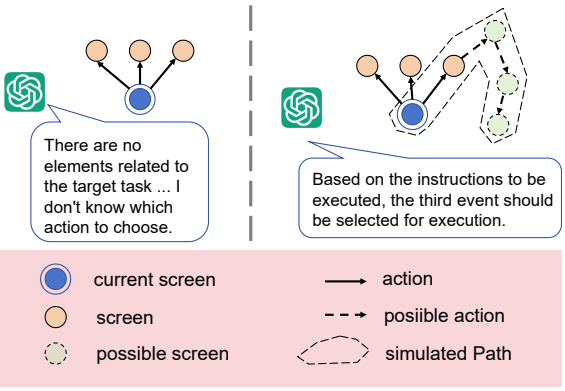

Figure 1: Comparison of Task Execution with and without Guidance in LLMs.

Large Language Models (LLMs) excel in various fields (Xu et al., 2023; Chen et al., 2024; Zhang et al., 2024b,c), with strong natural language understanding, generation, and decision-making abilities. In the field of mobile task automation, a more general method (Wen et al., 2024; Ran et al., 2024; Lee et al., 2024; Guan et al., 2024; Wen et al., 2023) involves providing the LLM with information (such as target task description, historical execution records, user interface information, etc.) and requesting the LLM to return actions. These tools then interact with mobile applications based on the actions returned by the LLM, ultimately accomplishing the given tasks.

LLMs (OpenAI, 2023a,b,c; Zhu et al., 2024) are trained on vast amounts of data, and their decision-making often reflects general practices across applications. This leads to poor performance when dealing with special task event sequence designs

or application-specific tasks. To enable agents to adapt to a wide range of applications, some agents (Ran et al., 2024; Lee et al., 2024; Wen et al., 2024; Guan et al., 2024) adopt an exploration-memory mechanism to explore the application and use the exploration results to assist in task execution. However, these methods has two main limitations: 1) Exploration combined with execution may lead to a large number of task-unrelated actions being generated on the user's device, which is unacceptable to users. 2) When the user interface does not contain directly relevant information for the task, the LLM may become confused and explore aimlessly, leading to highly inefficient exploration. As shown in Figure 1 (left), when elements on the interface are not directly related to the given task, the LLM may mistakenly assume that it has reached the wrong user interface or engage in purposeless exploration.

Considering that LLMs are familiar with general practices for mobile tasks, but different applications may have different design styles, this leads to challenges in executing tasks on mobile devices. Inspired by the vehicle driving scenario, we observe that having a clear map not only significantly enhances the driver's efficiency in reaching their destination but also helps them make more informed decisions when navigating complex road conditions. The presence of a map provides a sense of direction and security, reducing the risk of getting lost or taking unnecessary detours. This concept applies equally to other fields, where clear guidance and effective information greatly improve the likelihood of achieving objectives. As shown in Figure 1 (right), if an application manual serving as a map is used to guide the agent, the agent will be able to clearly understand the steps and sequence of task execution, thereby improving the efficiency and accuracy of task performance. Therefore, we propose a scenario in which application developers should provide an application manual alongside the application APK to assist agents in executing user tasks. Considering that providing such a manual would increase the burden on developers, the goal of this paper is twofold: 1) to define and automate the generation of application manuals, and 2) to design an agent capable of completing target tasks on mobile devices based on the provided application manual.

In this paper, we present UICOMPASS, a system designed to automatically generate a User Interface Manual (an application manual) and use UI Manual to complete user-specified tasks on the target application. First, it includes a source code analysis system that analyzes the application's source code to generate a UI Manual (User Interface Manual). The `UIManual` is a type of application guide that describes the functionality of activities, elements of activities, and transition relationships between activities. Unlike user manuals or other application guides that directly instruct how to execute a task, the UI Manual retains more detailed information to support a wide variety of tasks. To protect the security of the application, the manual undergoes multiple levels of abstraction and retains only the UI-level information. Considering that applications will provide user interfaces and even user documentation, it is acceptable for application vendors to provide the UI Manual to the agent. Secondly, when a user provides a task, UICOMPASS generates instructions based on the UI Manual and the task description. To better adapt to the actual execution state of the program (e.g., some elements may not have an ID), these instructions use natural language to outline potential execution paths. Finally, during the execution phase, considering that the program's dynamic state may differ from the results of static analysis, UICOMPASS adaptively replanning the instructions based on the instructions, historical execution data, and the current state of the application.

We conducted experimental evaluations of UICOMPASS on DroidTask dataset (Wen et al., 2024). The experimental results demonstrate that UICOMPASS outperforms existing LLM-based agents. UICOMPASS achieved a task success rate of 68.27%, representing a 14.48% improvement. Additionally, UICOMPASS demonstrated shorter execution paths and a higher capability for automatic task completion termination during the process. Ablation studies show that the UI Manual and the adaptive instruction replanning modules are effective in enhancing the agent's mobile task execution capability.

Our contributions can be outlined as follows.

- We propose a User Interface Manual-guided task execution method. This method first generates a User Interface Manual based on the mobile application's source code, which contains information about the user interface and operational responses, to assist the LLM in generating an initial instruction list based on the task and using the instruction list to aid in

task execution.

- An adaptive planning mechanism is employed to adjust for discrepancies between the initial instruction list and the actual execution environment of the application, allowing the agent to adapt to different application states.
- We conducted experimental evaluations of UICOMPASS, and the results show that, compared to the baseline, UICOMPASS demonstrated superior performance in executing user tasks. We open-source our code and experimental results [1].

2 Background and Related Work

2.1 Backgroud of Android Program

To facilitate mobile task automation, understanding the source code of the Android application's structure and execution model is essential. An activity [2] is a single screen in an Android app, handling user interaction and transitioning through lifecycle states such as *created*, *resumed*, and *destroyed*. Apps often include multiple activities for different functionalities, and fragments [3] serve as modular UI components within activities, supporting flexible layouts and reuse. Android UI elements (e.g., `Button`, `TextView`) are defined in XML layouts and assigned unique resource identifiers (@+id/...) for programmatic interaction, making accurate identification essential for automation. Each Android app includes an `AndroidManifest.xml` file, defining activities, permissions, and intent filters. Parsing this file helps map the application's structure and extract essential automation-related metadata.

Analyzing the functionality and execution logic of Android applications from source code is challenging, primarily because Android is an event-driven system (Payet and Spoto, 2012; Li et al., 2017). Event handling is distributed across lifecycle methods, user interactions, and system callbacks, with the complete logic depending on the collaboration of multiple components. Additionally, dynamic loading, reflection, and code obfuscation obscure certain logic paths, while third-party libraries and frameworks may conceal critical implementations within external dependencies. These factors make it difficult to fully reconstruct an application's behavior and execution flow solely from source code. However, with the rise of large models, combining static analysis tools with the powerful reasoning capabilities of these models offers a more efficient approach to interpreting UI-layer events and their interaction logic, providing an opportunity to deduce the application's functionality and specific execution steps from its source code.

2.2 Mobile Task Automation

Given a user task T described in natural language, mobile task automation aims to complete the task on the target application. (Wen et al., 2024) An inputted task is a natural language description of a functional request and does not include specific execution instructions, such as "Change theme color to light." The agent needs to determine the actions to be performed on the current user interface based on the given task, along with information about the application, screen details, historical execution actions, and other relevant information.

Traditional tools (such as Siri, Google Assistant, Cortana, etc.) use template-based approaches to perform tasks. These tools are difficult to handle complex and flexible tasks and require developers to engage in extensive programming work (Wen et al., 2024). Before the emergence of large language models, researchers designed supervised learning (Burns et al., 2022; Li et al., 2020; Sun et al., 2022; Xu et al., 2021) and reinforcement learning methods (Humphreys et al., 2022; Li and Riva, 2021; Toyama et al., 2021) to accomplish mobile task automation. These methods not only require a large amount of training data and high training costs but also lack flexibility in the diverse, real-world mobile applications and scenarios.

Large language models (LLMs) have demonstrated excellent performance in mobile task automation due to their powerful language understanding, decision-making, and reasoning capabilities (Wen et al., 2024; Ran et al., 2024; Lee et al., 2024; Zhang et al., 2023, 2024a). However, based on the user interface, LLMs often provide decisions following general practices rather than making decisions specifically tailored to the application itself. To enable LLMs to make decisions for a given app, it is necessary to provide application-specific information to the LLM, allowing it to consider the app's specific implementation. Existing methods (Wen et al., 2024; Ran et al., 2024) have designed exploration-memory mechanisms, which

[1] https://anonymous.4open.science/r/UICompass-B193/

[2] https://developer.android.com/reference/android/app/Activity

[3] https://developer.android.com/guide/fragments

explore the application and store the exploration results. When a task is given, this exploration information is communicated to the LLM in a specific manner. Nevertheless, this approach has inherent limitations: 1) it is difficult to explore and complete the target task within a limited number of attempts, and 2) users are reluctant to accept agents performing extensive exploration on their mobile devices.

3 Method

Figure 2 illustrates the overall framework of UICOMPASS. Basically, the UI tasks are completed in three steps: *user interface (UI) manual generation* (Section 3.1), *UI instruction generation* (Section 3.2), and *adaptive instruction replanning* (Section 3.3).

3.1 User Interface Manual Generation

UI Manual, which is automatically extracted from source code, delineates the interface layout and corresponding functionalities of mobile applications. It serves as a crucial reference for large language models to acquire the application's UI structures, transitions, and detailed logic. For instance, Figure 3 describes an example UI Manual of "App Launcher", which is composed of an activity list with detailed descriptions for each activity. Each activity usually corresponds to a user interface in the mobile apps. UI Manual of each activity includes the activity name, function summary, transition relationships between UIs, and contained UI element list. Each UI element is characterized by both static and dynamic properties. Static property defines layout information including tags, position, and IDs, while dynamic properties describe interactive actions and the resulting impact on which this element is operated (such as interface changes or activity transitions when a button is clicked).

To enable LLM to better understand the application, UICOMPASS first builds a UI Manual by analyzing the source code. Specifically, UICOMPASS generates UI Manual in three steps: method-level manual generation, activity-level manual generation, and application-level manual generation. Starting from the lower levels, UICOMPASS progressively refines and aggregates the information, and ultimately produces a concise and effective application-level UI Manual.

Method-level manual generation. The method-level manual demonstrates the information defined in each method, including UI structure, UI transaction relations, and functionality logic. UI structure primarily describes the hierarchy or containment relationships between components, such as an activity containing a fragment. UI transaction relations describe the transitions between activities, which help the LLM understand how to navigate to the target activity. Functionality logic is reflected via a function summary, describing the functionality of this method.

UICOMPASS combines program analysis and LLMs to generate the method-level manual. The key advantage of program analysis is its ability to provide precise, unambiguous information about method relationships. LLMs, while powerful in generating natural language, often lack the capability to deeply understand code structure and dependencies. Hence, UICOMPASS uses tree-sitter [4] to analyze the source code, extracting information such as global variables, method names, class names, source code of the method, and a Call Graph. Considering that we should provide the LLM with summaries of the invoked methods, UICOMPASS generates the corresponding information block for each method using topological sorting based on the call graph. For the current method, UICOMPASS provides the summary of all methods invoked by the current method, along with the context of the current method (the source code of the method, the name of the class it belongs to, and the definitions of global variables). More details can be found in Appendix A.1. remove variable in the paragram.

Activity-level manual generation. Effective mobile task automation requires an understanding of how user interfaces are structured and how different components interact. While method-level details provide granular information, they often fail to capture the holistic execution flow of an application. To bridge this gap, we construct an **activity-level UI Manual**, which abstracts method-level details into structured representations of activities. This approach enables a comprehensive view of the application's interface, capturing UI transitions and hierarchical relationships, thereby improving task execution efficiency and reducing ambiguity in automated navigation.

The activity list is extracted from "AndroidManifest.xml". In each activity, UICOMPASS collects the relevant UI elements from the layout files called

[4] https://tree-sitter.github.io/tree-sitter/

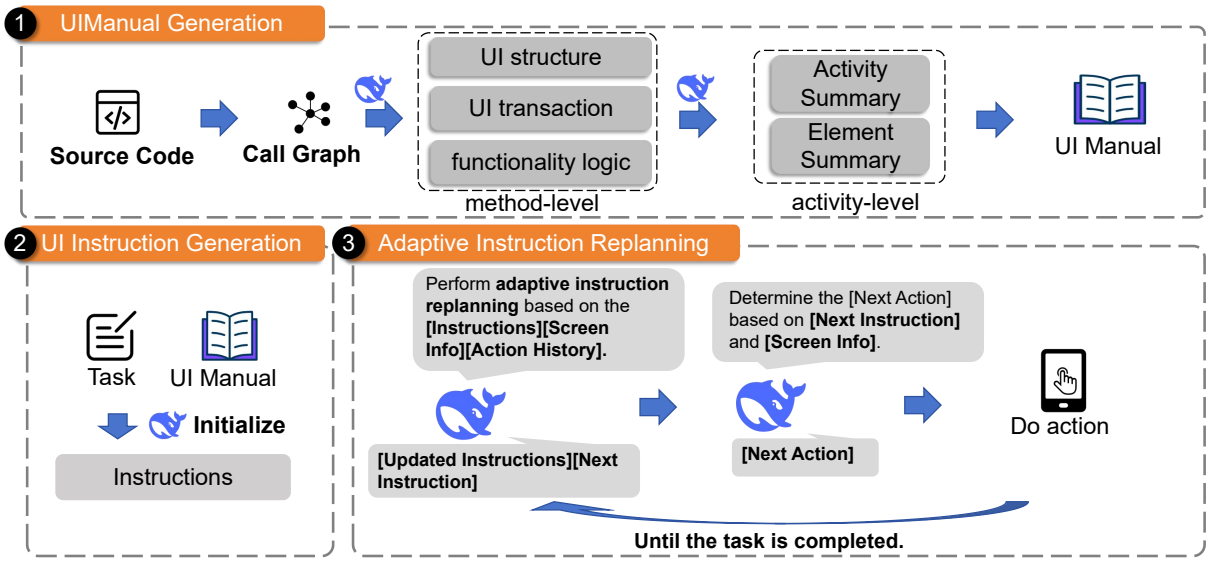

Figure 2: Our tool consists of three components: UI Manual generation, instruction initialization, and adaptive instruction replanning. We analyze the application's source code to generate the UI Manual. During the execution phase (2-3), an initial set of instructions is generated based on the UI Manual. As execution proceeds, the instruction list is continuously adjusted by incorporating the program's state and action history.

Activity list: SplashActivity, MainActivity, SettingsActivity, AboutActivity, CustomizationActivity.
Infomation about these activity:

Activity name:
com.simplemobiletools.applauncher.activities.MainActivity
Summary: The MainActivity is the primary interface for managing app launchers. It allows users to dynamically adjust the column count in a grid layout, check for invalid apps, display new release information, and manage app launchers through a Floating Action Button (FAB)...
This activity can be **transfer to**: AboutActivity, SettingsActivity.
Elements:
element_1:
Static properties: tag:MySearchMenu, id:@+id/main_menu,
Dynamic properties: action:search, effect:provides a search functionality to filter app launchers based on user input..
element_2:...

Figure 3: UI Manual of the mobile App - "APP Launcher".

Instruction list:

Activity:
com.simplemobiletools.applauncher.activities.MainActivity.
Instructions:
1. Open the app and navigate to the MainActivity.
2. Click on the 'Settings' menu item in the options menu.

Activity:
com.simplemobiletools.applauncher.activities.SettingsActivity.
Instructions:
3. In the SettingsActivity, locate the 'Color Customization' section.
4. Click on the 'Customize Colors' option.
5. Select the 'Light' theme from the available options.
6. Confirm the selection to apply the light theme.

Figure 4: The generated instruction list for task "Change theme color to light" in the "App Launcher" application.

by the activity class and its ancestor classes. Each activity's functionality is then characterized by aggregating relevant UI elements and interaction logic. This includes a core functionality summary, a structured list of UI elements with their properties, and the transition relationships between activities. This structured abstraction shifts from method-level details to an **activity-centric perspective**, producing a concise yet expressive UI Manual that enhances LLM-driven task execution.

Application-level manual generation. Building on the activity-level manual, UICOMPASS then generates an application-level UI Manual that provides a unified view of the entire application. This step integrates activity-level information, capturing inter-activity transitions, global UI elements, and application-wide settings. We achieve this by analyzing component interactions, permission requirements, and shared UI elements across activities. The resulting manual offers a high-level representation of the application's navigational structure and interaction logic, enabling LLM-driven agents to perform complex tasks.

3.2 UI Instruction Generation

With the generated UI Manual, UICOMPASS then guides LLM to generate an instruction list $\mathcal{I}$ to complete the given UI tasks. $\mathcal{I}$ comprises multiple action blocks, with each containing an activity ID $activity_i$ and a set of specific execution instructions I_i, indicating executing I_i within $activity_i$.

Formally,

$$\mathcal{I} = \{a_1(activity_i, I_1), \cdots, a_m(activity_m, I_m)\}$$

Figure 4 illustrates the instruction list for the task "Change theme color to light" in the "App Launcher" application. To complete the given task, the generated instruction lists show that we need to open the MainActivity, click setting to transition to SettingActivity, and click 'Light' theme.

3.3 Adaptive Instruction Replanning

A static instruction list generated from the UI Manual may not always be directly executable due to variations in runtime application states. Factors such as dynamically loaded UI elements, state-dependent UI transitions, and missing ids can cause discrepancies between the planned execution path and the actual interface observed during execution. Without an adaptive replanning mechanism, an agent may either fail the task when encountering an unexpected state or perform unnecessary exploratory actions, reducing efficiency and increasing the risk of errors.

To address these challenges, UICOMPASS incorporates an adaptive instruction replanning mechanism. By continuously evaluating execution progress and adjusting the instruction list accordingly, the agent dynamically adapts to UI changes while maintaining a goal-directed execution strategy. This approach minimizes unnecessary interactions, improves task success rates, and ensures robustness against application variability. In contrast to purely exploratory methods, which rely on trial-and-error and may generate redundant actions, our adaptive replanning mechanism leverages structured task execution data to guide decision-making efficiently.

Algorithm 1 demonstrates how UICOMPASS adjusts the instruction list based on the specific application state and tasks. The algorithm takes the initial instruction list $\mathcal{I}$ and the maximum number of attempts max_tries as inputs, and finally outputs the action history. Before the action loop begins, UICOMPASS initializes the number of attempts and an empty action history. Leveraging the action history, screen information, and the instruction list, the LLM analyzes the application's state to update the instruction list and choose the next instruction to be executed (line 4). This data helps the LLM comprehend the application's current state, monitor task execution progress, and select appropriate instructions. If the LLM identifies a need for modifications to the instructions, it makes the necessary adjustments during this phase. If no next instruction needs to be executed, UICOMPASS terminates the task (lines 6-8). Otherwise, the LLM proceeds by executing the instruction and recording it in the action history (lines 9-12). Adaptive instruction list replanning enables UICOMPASS to complete tasks effectively while accounting for the application's dynamic nature.

Algorithm 1: Adaptive Instruction Replanning Algorithm

Input: task t, instruction list $\mathcal{I}$, maximum attempts max_tries
Output: action history $\mathcal{H}$

1 $tries \leftarrow 0$; $\mathcal{H} \leftarrow \emptyset$
2 **while** $tries < max\_tries$ **do**
3 $\quad S \leftarrow$ GetInterfaceInfo();
4 $\quad (\mathcal{I}_{update}, I_{next}) \leftarrow \text{LLM}(t, \mathcal{I}, \mathcal{H}, S)$;
5 $\quad \mathcal{I} \leftarrow \mathcal{I}_{update}$;
6 $\quad$ **if** $I_{next} ==$ *null* **then**
7 $\quad\quad$ **break**;
8 $\quad$ **end**
9 $\quad action \leftarrow \text{LLM}(S, I_{next})$;
10 $\quad$ DoAction($action$) ;
11 $\quad \mathcal{H}.push(action)$;
12 $\quad tries \leftarrow tries + 1$;
13 **end**
14 **return** $\mathcal{H}$;

4 Experiments

UICOMPASS has been developed on the foundation of Guardian (Ran et al., 2024). To investigate the performance of UICOMPASS, we conducted experimental evaluations.

4.1 Experimental Settings

Datasets. We chose the DroidTask (Wen et al., 2024) dataset to validate UICOMPASS because it is based on open-source applications and provides the source code of the applications. DroidTask contains 158 high-level tasks from 13 popular apps. Since one application in the dataset is not open-source, we selected 12 of these open-source applications. To minimize the impact of data leakage, we selected the latest versions along with their corresponding source code. However, four tasks were no longer executable, leaving us with a total of 145 tasks.

Baseline Methods. We chose AutoDroid (Wen

Methods	SR↑	ACP↑	OSR↑	SPL↑
AutoDroid	53.79%	71.72%	76.92%	15.87%
Guardian	45.20%	71.83%	0.0%	1.70%
UICOMPASS	**68.27%**	**81.96%**	**80.80%**	**20.92%**

Table 1: Effectiveness of Task Completion.

UI Manual	Adapting	SR↑	ACP↑	OSR↑	SPL↑
×	×	37.24%	64.57%	60.00%	1.77%
✓	×	42.75%	64.43%	66.67%	15.00%
×	✓	55.86%	72.62%	53.84%	17.66%
✓	✓	**68.27%**	**81.96%**	**80.80%**	**30.85%**

Table 2: Ablation Results of UICOMPASS

et al., 2024) and Guardian (Ran et al., 2024) as baseline tools for experimental comparison. AutoDroid combines the commonsense knowledge of LLMs and domain-specific knowledge of apps through an exploration-memory mechanism. Guardian refines action spaces with domain-specific knowledge and adjusts application exploration through recovery-based replanning. Both tools utilize LLMs for mobile task automation and have demonstrated excellent performance.

Metrics. Following existing work (Wen et al., 2024; Ran et al., 2024), we measure the following metrics:

- Success Rate (SR): The ratio of successfully completed tasks to the total number of tasks.
- Average Completion Proportion (ACP): The maximum proportion of the executed action sequence that matches the prefix of the ground truth action sequence.

To evaluate whether the tools can effectively avoid unnecessary actions and correctly terminate, inspired by robotic navigation tasks (Chen et al., 2024), we introduce two additional metrics:

- Oracle Success Rate (OSR): The rate of successfully stopping exploration when the task is completed.
- Success Rate Penalized by Path Length (SPL): A metric that evaluates the rate that is calculated by the ground truth action sequence length divided by the actual action sequence length.

4.2 Results of task completion

We validated the task execution capabilities of UICOMPASS, Guardian, and AutoDroid on DroidTask. Table 1 summarizes the evaluation results. UICOMPASS achieved a higher task success rate (SR) of 68.27%, representing a 14.48% improvement over the best results (53.79%) from existing tools. This demonstrates that under the guidance of UI Manual, UICOMPASS can effectively complete tasks using the adaptive instruction list replanning strategy. UICOMPASS achieves an ACP of 81.96%, indicating its superior understanding of task execution logic and its ability to perform a greater number of task-related actions. The 3.88% improvement in OSR and the 5.05% enhancement in SPL further demonstrate UICOMPASS's capability to terminate promptly upon task completion. Among these results, Guardian's OSR is 0 because it only stops after reaching the maximum number of operations. These experimental results validate the effectiveness of UICOMPASS, showcasing its ability to accomplish more tasks while utilizing fewer execution steps.

4.3 Ablation Study

To evaluate the effectiveness of UI Manual and the adaptive instruction replanning, we conducted additional ablation experiments. When UI Manual is omitted, the agent generates an initial instruction list based solely on the remaining information. When adaptive instruction replanning is disabled, the agent follows the instruction list strictly in their original sequence.

With UI Manual guidance vs. Without UI Manual guidance. When UI Manual is enabled, the task success rate (SR) increases from 37.24% to 42.75% (without adaptive re-planning) and from 55.86% to 68.27% (with adaptive re-planning). Additionally, the SPL metric improves substantially, rising from 1.77% to 15.00% in the absence of re-planning and from 17.66% to 30.85% when re-planning is applied. These results suggest that UI Manual provides crucial contextual guidance that enhances both the success rate and task efficiency.

Direct Execution with Instructions vs. Adaptive Instruction Replanning. In scenarios without UI Manual, enabling adaptive re-planning increases the SR from 37.24% to 55.86%. Similarly, with UI Manual guidance, adaptive re-planning boosts the SR from 42.75% to 68.27%. These results demonstrate that adaptive re-planning allows the agent to dynamically adjust its strategy, effectively improving task success and overall efficiency.

By combining UI Manual guidance and adaptive instruction re-planning, the agent achieves the best

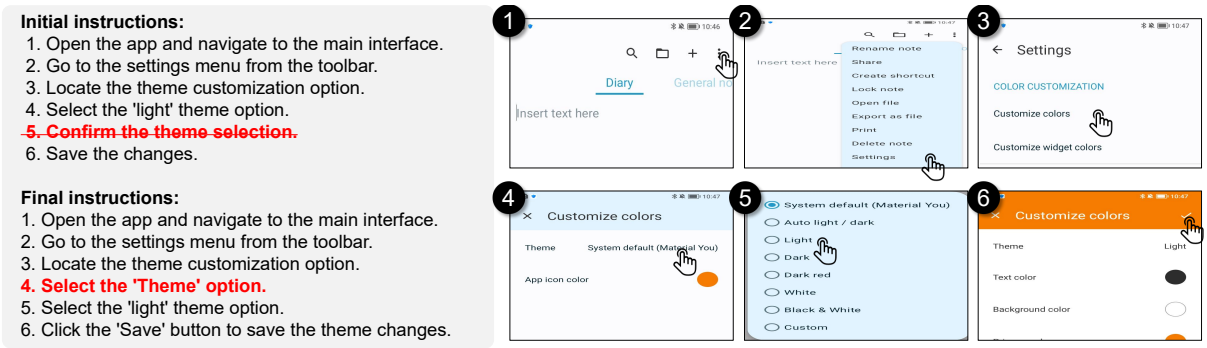

Figure 5: A successful case on DroidTask demonstrates the effectiveness of UICOMPASS's UI Manual and adaptive instruction replanning.

results across all metrics, with the SR improving to 68.27%. This is significantly higher than using either UI Manual (SR: 42.75%) or adaptive replanning alone (SR: 55.86%). These results demonstrate that UI Manual provides essential contextual guidance, while adaptive re-planning enhances flexibility, and their integration effectively maximizes task success and efficiency.

4.4 Case Study

As shown in Figure 5, we present a successful example from the DroidTask dataset, which highlights the effectiveness of the UICOMPASS's UI Manual-guided process and the capability of adaptive instruction list replanning in real-world scenarios. The task in this example is "Set app theme to light and save it," and the six images on the right side of Figure 5 depict the correct steps generated by UICOMPASS. This example presents a challenge for the existing mobile agent. Although the instruction indicates this is a setting operation, the numerous setting options in the settings activity and 'Customize colors' (Step 3) not indicating that it is for modifying the theme colors both contribute to the difficulty of task execution. UICOMPASS generates the initial instruction list using the UI Manual (shown in the top-left of Figure 5). Due to the complexity of the application, the instruction list generated by UICOMPASS is not entirely correct. In this example, we can see that most of the instruction list generated by UICOMPASS are correct, except for the "Select the 'Theme' option" being missing and "Confirm the theme selection" being mistakenly included. As shown in the final instructions (as shown in the bottom left of Figure 5), UICOMPASS uses the adaptive instruction list replanning mechanism to correct the errors in the initial instruction list based on the actual execution context. Therefore, the initial instruction list generated by the UI Manual, combined with the adaptive instruction list replanning mechanism, can effectively guide the LLM to complete tasks in the target application.

5 Conclusion

In this paper, we propose a method for guiding agents in mobile task automation using a User Interface Manual extracted from the source code, called UICOMPASS. UICOMPASS leverages LLM to analyze the source code and ultimately generates the User Interface Manual. UICOMPASS can then use this User Interface Manual along with the task to generate an initial instruction list. Furthermore, to ensure the agent can adapt to different application states, we introduce adaptive instruction list re-planning, which combines action history, and the current interface to continuously adjust the instruction list. Through extensive experiments, we demonstrate the effectiveness of the instruction list generated by UICOMPASS and its task completion capability, achieving state-of-the-art performance.

Limitation

Although UICOMPASS can automatically generate the UI Manual, its reliance on application developers to provide the UI Manual to the agent limits its usage scenarios. However, the UI Manual can still effectively improve task execution capabilities for applications with easily accessible source code (e.g., system applications) or those willing to provide the UI Manual. In the future, we aim to explore methods for extracting the UI Manual directly from APK files, which would significantly expand the usage scenarios of UICOMPASS.

References

Andrea Burns, Deniz Arsan, Sanjna Agrawal, Ranjitha Kumar, Kate Saenko, and Bryan A Plummer. 2022. A dataset for interactive vision-language navigation with unknown command feasibility. In *European Conference on Computer Vision*, pages 312–328. Springer.

Jiaqi Chen, Bingqian Lin, Ran Xu, Zhenhua Chai, Xiaodan Liang, and Kwan-Yee Wong. 2024. Mapgpt: Map-guided prompting with adaptive path planning for vision-and-language navigation. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9796–9810.

Yanchu Guan, Dong Wang, Zhixuan Chu, Shiyu Wang, Feiyue Ni, Ruihua Song, and Chenyi Zhuang. 2024. Intelligent agents with llm-based process automation. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5018–5027.

Peter C Humphreys, David Raposo, Tobias Pohlen, Gregory Thornton, Rachita Chhaparia, Alistair Muldal, Josh Abramson, Petko Georgiev, Adam Santoro, and Timothy Lillicrap. 2022. A data-driven approach for learning to control computers. In *International Conference on Machine Learning*, pages 9466–9482. PMLR.

Rashedul Islam, Rofiqul Islam, and Tohidul Mazumder. 2010. Mobile application and its global impact. *International Journal of Engineering & Technology*, 10(6):72–78.

Sunjae Lee, Junyoung Choi, Jungjae Lee, Munim Hasan Wasi, Hojun Choi, Steve Ko, Sangeun Oh, and Insik Shin. 2024. Mobilegpt: Augmenting llm with human-like app memory for mobile task automation. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, pages 1119–1133.

Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 88:67–95.

Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. 2020. Mapping natural language instructions to mobile ui action sequences. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8198–8210.

Yuanchun Li and Oriana Riva. 2021. Glider: A reinforcement learning approach to extract ui scripts from websites. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1420–1430.

OpenAI. 2023a. Gpt-4 technical report. Technical report, OpenAI. 2023a.

OpenAI. 2023b. Gpt-4v(ision) system card. Technical report, OpenAI. 2023b.

OpenAI. 2023c. Gpt-4v(ision) technical work and authors. Technical report, OpenAI. 2023c.

Étienne Payet and Fausto Spoto. 2012. Static analysis of android programs. *Information and Software Technology*, 54(11):1192–1201.

Dezhi Ran, Hao Wang, Zihe Song, Mengzhou Wu, Yuan Cao, Ying Zhang, Wei Yang, and Tao Xie. 2024. Guardian: A runtime framework for llm-based ui exploration. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 958–970.

Liangtai Sun, Xingyu Chen, Lu Chen, Tianle Dai, Zichen Zhu, and Kai Yu. 2022. Meta-gui: Towards multi-modal conversational agents on mobile gui. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 6699–6712.

Daniel Toyama, Philippe Hamel, Anita Gergely, Gheorghe Comanici, Amelia Glaese, Zafarali Ahmed, Tyler Jackson, Shibl Mourad, and Doina Precup. 2021. Androidenv: A reinforcement learning platform for android. *arXiv preprint arXiv:2105.13231*.

Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024. Mobile-agent-v2: Mobile device operation assistant with effective navigation via multi-agent collaboration. *arXiv preprint arXiv:2406.01014*.

Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2023. Empowering llm to use smartphone for intelligent task automation. *arXiv preprint arXiv:2308.15272*.

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

Mingzhe Xing, Rongkai Zhang, Hui Xue, Qi Chen, Fan Yang, and Zhen Xiao. 2024. Understanding the weakness of large language model agents within a complex android environment. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 6061–6072.

Nancy Xu, Sam Masling, Michael Du, Giovanni Campagna, Larry Heck, James Landay, and Monica Lam. 2021. Grounding open-domain instructions to automate web support tasks. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1022–1032.

Zhuolin Xu, Qiushi Li, and Shin Hwei Tan. 2023. Guiding chatgpt to fix web ui tests via explanation-consistency checking. *arXiv preprint arXiv:2312.05778*.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2024. Wavecoder: Widespread and versatile enhancement for code large language models by instruction tuning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5140–5153.

Chaoyun Zhang, Shilin He, Jiaxu Qian, Bowen Li, Liqun Li, Si Qin, Yu Kang, Minghua Ma, Qingwei Lin, Saravan Rajmohan, et al. 2024a. Large language model-brained gui agents: A survey. *arXiv preprint arXiv:2411.18279*.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024b. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv preprint arXiv:2401.07339*.

Zhe Zhang, Xingyu Liu, Yuanzhang Lin, Xiang Gao, Hailong Sun, and Yuan Yuan. 2024c. Llm-based unit test generation via property retrieval. *arXiv preprint arXiv:2410.13542*.

Zhizheng Zhang, Xiaoyi Zhang, Wenxuan Xie, and Yan Lu. 2023. Responsible task automation: Empowering large language models as responsible task automators. *arXiv preprint arXiv:2306.01242*.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. OpenCodeInterpreter: Integrating code generation with execution and refinement. In *Findings of the Association for Computational Linguistics ACL 2024*, Bangkok, Thailand and virtual meeting.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

Appendices

A More Details

A.1 Prompts

To present our prompts more clearly, we further demonstrate practical examples in the "notes" application. Figure 6 is the prompt used for method-level manual generation. Figure 7 describes the prompt content for generating the initial instruction list when performing the task "Set the app theme to light and save it" in the "notes" application. Figure 8 illustrates the detailed prompt for UICOMPASS in "Adaptive Instruction Replanning," which facilitates adjustments to task execution in response to changing requirements or unforeseen challenges.

A.2 Experimental Details

In this experiment, we utilized DeepSeek-v3 (Zhu et al., 2024) as the LLM. This choice was made not only because of its powerful performance but also due to its high cost-effectiveness. DeepSeek has been widely used and widely accepted in LLM-related work (Zhang et al., 2024c; Zheng et al., 2024; Yu et al., 2024).

To ensure a fair comparison of each tool, we manually annotated the experimental data. The three authors of the paper first familiarized themselves with the applications and referred to the ground truth provided in DroidTask (Wen et al., 2024). For the execution results of each tool on each application, we conducted separate analyses and ultimately engaged in discussions. Different tasks may have multiple implementation approaches. Therefore, for the execution results of each tool, we analyze and evaluate the shortest path chosen to complete the task when calculating the experimental results. In addition, during the experiments, we prepared the data required for each task in advance to ensure that the task was executable.

[Background]

You are an Android analyst. I will give you a method from the class SettingsActivity.

Here is the method from the given Android source code:

```
override fun onResume() {
    super.onResume()
    setupToolbar(binding.settingsToolbar, NavigationIcon.Arrow)
    setupPurchaseThankYou()...}
```

[Invoked Method Summary]

Here is the explanation of the method named com.simplemobiletools.applauncher.activities.SettingsActivity.setupPurchaseThankYou that is called within the given method:

The method `setupPurchaseThankYou` is designed to configure the UI component `settingsPurchaseThankYouHolder` for a thank you screen after a purchase. It hides the holder if a thank you has already been shown, and sets up an onClickListener to launch a new intent if the holder is visible.

...

Please analyze the method above and identify any activity migration relationships that exist.

Specifically, look for `startActivity` calls and their associated `Intent` objects.

[Target]

You need to analyze the functionality of the source code. If there are any actionable elements, provide a description of the element's response.

[Notice]

Additionally, identify all **UI elements** involved in the method. An **element** should be one of the following:

- A view component such as `Button`, `TextView`, `ImageView`, etc.
- An interaction-triggering element, like a button that starts an activity or any clickable UI component.

In addition to activity migrations, please also check for:

1. Any **fragment** and **activity** dependencies or relationships in the code. Specifically, identify if a fragment is associated with an activity (e.g., using `getActivity()`, `FragmentTransaction`, or similar methods).
2. Any **migration relationships** between fragments and activities. This could include cases where a fragment starts an activity, or an activity dynamically replaces a fragment, or any other relationships indicating a transition between a fragment and an activity.
3. Any relationship between **XML files (R.layout，or R.menu)** and **activities** or **fragments**. Look for references to XML layouts that are inflated in an activity or fragment (e.g., using `setContentView()`, `LayoutInflater`, `FragmentTransaction.replace()`, etc.). **You only need to output layout files that start with R.layout and R.menu.**
4. **element_list**: Please list all the elements (normal element or menu item) present in the method. Tell me the type and ID of each element. Additionally, describe the conditions under which the element is executed and the effects it will have. For example, the condition could be that the login button can only be clicked after entering the account password, and the effect could be a transition to MainActivity.

[Output Example]

The output should be in JSON format, for example:

```
{"functionality": "description of the method functionality",
    "element_list": [
        {"type": "element type",
         "element_id": "R.id.xx",
         "action": "description of the action triggered by the element"}],
    "activity_migrations": [
        {"from_activity_or_fragment": "sourceActivityOrFragmentName",
         "to_activity_or_fragment": "targetActivityOrFragmentName",
         "description": "A brief description of how the migration happens."}],
    "fragment_activity_relationships": [
        {"fragment": "fragmentName",
         "activity": "activityName",
         "relationship": "Fragment is attached to Activity using FragmentTransaction methods like add(), replace(), or show()."}],
    "xml_relationships": [
        {"xml_file": "R.layout.xx.xml",
         "associated_with": "activity_or_fragment"}]}
```

Figure 6: The prompt for the method-level manual generation of the “OnResume” method in the “Applaucher” application.

[Background]

You are a user of an application, and I will provide you with the instruction manual for the application. Your task is to speculate on what instructions is used to execute the given task.

[UI Manual]

Infomation about this app:

Activity list:

SplashActivity, MainActivity, WidgetConfigureActivity, AboutActivity, CustomizationActivity, SettingsActivity,

Infomation about these activity:

Activity name: com.simplemobiletools.notes.pro.activities.SplashActivity

The summary of com.simplemobiletools.notes.pro.activities.SplashActivity: "This activity serves as a splash screen that checks for a specific intent extra (`OPEN_NOTE_ID`). If the extra is present, it starts `MainActivity` with the extra data; otherwise, it starts `MainActivity` without any extra data. After starting `MainActivity`, the `SplashActivity` is finished."

This activity can be transfer to other activities: MainActivity,

Activity name: com.simplemobiletools.notes.pro.activities.MainActivity

The summary of com.simplemobiletools.notes.pro.activities.MainActivity: "The activity serves as the main interface for managing notes, including creating, editing, deleting, and viewing notes. It supports various note types (text and checklist), handles file imports/exports, manages search functionality, and integrates with system features like shortcuts, printing, and sharing. Additionally, it provides options for locking/unlocking notes, sorting checklists, and managing app settings."

This activity can be transfer to other activities: MainActivity, SplashActivity, AboutActivity, SettingsActivity,

tag:MaterialToolbar, id:@+id/main_toolbar, action:toolbar, effect:Displays the activity's toolbar, which contains menu items for actions like saving, searching, creating notes, and accessing settings..

tag:include, id:@+id/search_wrapper, action:include, effect:Embeds the search bar layout, enabling search functionality within the activity..

tag:MyViewPager, id:@+id/view_pager, action:viewpager, effect:Manages the display of multiple notes in a swipeable interface, allowing users to navigate between notes..

tag:PagerTabStrip, id:@+id/pager_tab_strip, action:tabstrip, effect:Provides visual indicators for the current note's position within the ViewPager, aiding navigation..

tag:MaterialToolbar, id:@+id/main_toolbar, action:toolbar, effect:Displays the activity's toolbar, which contains menu items for actions like saving, searching, creating notes, and accessing settings..

tag:include, id:@+id/search_wrapper, action:include, effect:Embeds the search bar layout, enabling search functionality within the activity..

tag:MyViewPager, id:@+id/view_pager, action:viewpager, effect:Manages the display of multiple notes in a swipeable interface, allowing users to navigate between notes..

tag:PagerTabStrip, id:@+id/pager_tab_strip, action:tabstrip, effect:Provides visual indicators for the current note's position within the ViewPager, aiding navigation..

...

[Task Description]

Based on the aforementioned application information, our goal is to execute the task: "Set app theme to light and save it".

[Output Example]

```
{"task": "Book a flight",
    "activities_sequence": [
        {"activity": "LoginActivity",
            "steps": [
                "1. Input the account.",
                "2. Submit the login form."]},
        {"activity": "MainActivity",
            "steps": [
                "3. Search for available flights based on your preferences.",
                "4. Select the flight that suits your needs."]},
        {"activity": "BookingActivity",
            "steps": [
                "5. Enter the required passenger details for booking.",
                "6. Make the payment for the selected flight.",
                "7. Receive a confirmation of the flight booking."]}],
    "explanation": "because the BookingActivity has the flight booking button."}
```

Figure 7: In the application “Notes”, the prompt content of UICOMPASS generates the initial instruction list for task “Set app theme to light and save it”.

[Background]

I currently have a task Set app theme to light and save it, and I have a set of instructions for this task, but there may be errors in this set of instructions that need to be adjusted based on the current interface.

[Instructions]

Instructions:

{'task': 'Set app theme to light and save it', 'activities_sequence': [{'activity': 'MainActivity', 'steps': ['1. Open the app and navigate to the main interface.']}, {'activity': 'SettingsActivity', 'steps': ['2. Go to the settings menu from the toolbar.', '3. Locate the theme customization option.', "4. Select the 'light' theme option."]}, {'activity': 'CustomizationActivity', 'steps': ['5. Confirm the theme selection.', '6. Save the changes.']}]}

[Screen Info]

Here is the information about the screen we are currently on.

Widget(text: Rename note, class: android.widget.LinearLayout, position: (539, 220, 1054, 346))

Widget(resource-id: com.simplemobiletools.notes.pro:id/content, class: android.widget.LinearLayout, position: (539, 220, 1054, 346))

Widget(resource-id: com.simplemobiletools.notes.pro:id/title, text: Rename note, class: android.widget.TextView, position: (581, 254, 1012, 311))

Widget(text: Remove done items, class: android.widget.LinearLayout, position: (539, 346, 1054, 472))

...

[Action History]

#History information (You should refer to the historical records to identify which part of the instructions they correspond to, consider the relationship between the current interface and the next step, and then update the instructions accordingly.):{

index-0: open the target application

index-1:Event(action=click, widget=a View (accessibility information: More options, text:))

}

If Task is finished, next_instruction = none

[Output Example]

"task": "Book a flight",

"explanation": {

"current state" : "The current interface indicates that the search has been completed and the search results are displayed, but no flight has been selected yet.",

"finished step" : "Based on the history, a search operation has already been performed. Therefore, 3. Search for available flights based on your preferences.",

"error reason" : "The next action should select the flight",

"revised method" : "add 4. Select the flight that suits your needs",

"next_instruction": "4. Select the flight that suits your needs"},

"updated_activities_sequence": [

{"activity": "LoginActivity",

"steps": [

"1. Input the account.",

"2. Submit the login form."]},

{"activity": "MainActivity",

"steps": [

"3. Search for available flights based on your preferences.",

"4. Select the flight that suits your needs."]},

{"activity": "BookingActivity",

"steps": [

"5. Enter the required passenger details for booking.",

"6. Make the payment for the selected flight.",

"7. Receive a confirmation of the flight booking."]}]}

Figure 8: In application “Notes”, UICOMPASS generates the prompt for the "Adaptive Instruction Replanning" section to perform task A.