UICOMPASS: UI Map Guided Mobile Task Automation via Adaptive Action Generation

Yuanzhang Lin , Zhe Zhang , Rui He , Qingao Dong ,
Mingyi Zhou , Jing Zhang , Xiang Gao* , Hailong Sun*
Beihang University, Hangzhou Innovation Institute of Beihang University
{yuanzhanglin, xiang_gao, sunhl}@buaa.edu.cn

Abstract

Mobile task automation is an emerging technology that leverages AI to automatically execute routine tasks by users' commands on mobile devices like Android, thus enhancing efficiency and productivity. While large language models (LLMs) excel at general mobile tasks through training on massive datasets, they struggle with app-specific workflows. To solve this problem, we designed UI Map, a structured representation of target app's UI information. We further propose a UI Map-guided LLM-based approach UICOMPASS to automate mobile tasks. Specifically, UICOMPASS first leverages static analysis and LLMs to automatically build UI Map from either source codes of apps or byte codes (*i.e.,* APK packages). During task execution, UICOMPASS mines the task-relevant information from UI Map to feed into the LLMs, generates a planned path, and adaptively adjusts the path based on the actual app state and action history. Experimental results demonstrate that UICOMPASS achieves a 14.52% higher task executing success rate than SOTA approaches. Even when only APK is available, UICOMPASS maintains superior performance, demonstrating its applicability to closed-source apps.

1 Introduction

Automating mobile tasks is crucial as it has the potential to significantly enhance user experience, particularly in situations where manual interaction with devices is inconvenient or unsafe—such as for people with disabilities or drivers who need to focus on the road. However, modern mobile apps, despite offering valuable functionalities, often feature complex user interfaces that impose significant difficulties for mobile task automation.

There are two primary approaches to automating mobile tasks. The first relies on predefined templates to create customized workflows for specific tasks. Tools like Siri Shortcuts (Shortcuts, 2025) and Google Assistant Routines (Routines, 2025) allow users to design automation sequences across apps, services, and device settings, which can be triggered with a tap or voice command. While this method offers flexibility, it requires manual setup of each workflow, making it time-consuming and cumbersome—particularly for complex or frequently updated tasks—limiting its accessibility and scalability. The second approach leverages the powerful understanding capabilities of LLMs (Wen et al., 2024; Ran et al., 2024; Lee et al., 2024; Wang et al., 2024a; Guan et al., 2024) and visual language models (VLMs) (Wang et al., 2024b, 2025, 2024b, 2025; Song et al., 2024; Zhang et al., 2024c). These tools automate mobile tasks using exploration-based methods that gather UI information and predict the next UI action. Although achieving promising results, these blind-exploring methods have two key limitations: (1) a lack of knowledge about the app's global UI structure and functional logic leads to inefficient task execution, and (2) exploration can trigger many irrelevant actions on the user's device, which is unsafe.

To address the limitations of blind exploration in existing tools, inspired by navigation systems that rely on maps to improve driving efficiency, we propose UI Map, a high-level UI structure of the target app. We design UI Map to outline app activities (corresponding to screens), UI elements, and their syntactic and semantic interrelations—similar to how a map includes cities and intersections. By offering high-level guidance, UI Map helps LLMs better understand the global app's UI structure, make more informed decisions, and avoid inefficient or irrelevant actions, thereby enhancing the effectiveness of task automation.

In this paper, we propose UICOMPASS, an approach that automatically generates UI Maps and leverages them to facilitate task execution. UICOMPASS first creates UI Map by combining

*Corresponding authors.

static program analysis and LLMs. Initially, static analysis processes the target application to generate an initial UI Map by analyzing either source codes or byte codes (*i.e.,* APK packages), supporting both open-source and closed-source apps. However, static analysis may overlook certain key elements and relationships, and struggle with understanding the semantics of UI components. Hence, UICOMPASS uses LLMs to enrich UI descriptions and incorporate missing elements. Since UI Map is automatically generated, this process doesn't add extra workload for developers, simplifying the integration of UICOMPASS into the existing workflow.

Subsequently, UICOMPASS leverages UI Map to automate mobile tasks. For a given task, UICOMPASS initially identifies and extracts a task-specific sub-graph from UI Map, which is then analyzed by LLM to devise an initial path for task completion. Since UI Map is statically derived from the application code (*e.g.,* source code or bytecode), it might not accurately reflect the app's dynamic behaviors. Therefore, UICOMPASS employs an adaptive UI action generation strategy. This innovative approach dynamically modifies the planned execution path in response to the current state of the application and the ongoing progress of the task, enabling efficient and accurate task completion.

We conducted experimental evaluations of UICOMPASS on DroidTask (Wen et al., 2024) and AndroidWorld (Rawles et al., 2025) dataset. UICOMPASS achieved task success rates of 68.27% (14.48% improvement) with DeepSeek-V3 as backend LLM and 78.62% (15.87% improvement) with Qwen-Max. In terms of time efficiency, while UICOMPASS incurs higher per-step overhead, it achieves time savings by reducing the total number of steps required for task completion. Given that unrelated steps may lead to unpredictable behavior, which is undesirable from a user perspective, we argue that this trade-off remains justified. Ablation studies show that UI Map and the adaptive replanning modules are effective in enhancing the agent's mobile task execution capability.

Our contributions can be outlined as follows:

- We propose to automate mobile tasks with the guidance of UI Map, a high-level UI structure of target application.
- We developed UI Map generation approaches based on either source code or APK bytecode, and an adaptive action generation strategy.
- Experimental results show UICOMPASS achieves 14.52% (source code-based) and 13.45% (byte code-based) improvement over state-of-the-art approaches in terms of task success rates. Our tool and experimental results are open available [1]

2 Background and Related Work

In this section, we will provide an introduction to Android programming and mobile task automation.

2.1 Background of Android App Code

At the highest level, every Android app declares its activities (each representing a single screen) and the partial transfer relationships between its activities in the AndroidManifest.xml file. An activity is organized by lifecycle states (e.g., `created`, `resumed`) governing their visibility and interaction logic. The vast majority of interactive UI elements (e.g., `Button`, `TextView`) and their static attributes (such as `@+id/submit_button`) are defined in XML layout files. Developers assign specific functionality to these elements by adding event listeners (such as `setOnClickListener`) within the code.

App structure, behavior, and interaction logic are crucial for tasks. While LLMs learn general patterns, they lack app-specific knowledge. Thus, extracting implementation logic from code has the potential to enhance LLMs' app task capabilities.

2.2 Mobile Task Automation

Mobile task automation aims to complete a user-described task (expressed in natural language without specific instructions) on the device. The agent needs to determine the actions to be performed based on the given task.

Traditional tools (e.g., Siri, Google Assistant) rely on rigid templates, limiting their ability to handle complex tasks and requiring significant developer effort. Supervised (Burns et al., 2022; Li et al., 2020; Sun et al., 2022; Xu et al., 2021) or reinforcement learning (Humphreys et al., 2022; Li and Riva, 2021; Toyama et al., 2021) demanded extensive training data and costs while remaining inflexible for real-world mobile scenarios.

LLMs (Wen et al., 2024; Ran et al., 2024; Lee et al., 2024; Zhang et al., 2023b, 2024a; Wang et al., 2024c; Zhang et al., 2024b) and visual models (Zhang et al., 2023a) (such as multimodal models (Wang et al., 2024b, 2025; Song et al., 2024;

[1] https://github.com/YuanzhangLin/UICompass

Zhang et al., 2024c; Li et al., 2025; Yan et al., 2023; Hoscilowicz et al., 2024; Zhu et al., 2025; Christianos et al., 2025; Ma et al., 2024)) excel in mobile task automation due to their advanced understanding and reasoning capabilities. However, their decisions often follow general practices rather than app-specific considerations, necessitating the provision of app-specific information to enable tailored decision-making. Existing approaches have developed exploration-based frameworks that systematically investigate app user interfaces and archive exploration outcomes to facilitate subsequent action generation. However, due to an insufficient understanding of the app, these methods generate a large number of trial actions that are unacceptable.

3 Method

Figure 1 presents the UICOMPASS framework, which completes tasks in three steps: *UI Map generation* (Section 3.1), *UI path planning* (Section 3.2), and *adaptive action generation* (Section 3.3).

3.1 UI Map Generation

To enable the LLM to understand the usage logic of an app quickly, we propose UI Map, a graph used to describe the app's UI and interaction logic. Just as maps in a navigation task, the UI Map serves to assist LLM in swiftly identifying potential paths prior to the execution of a task. Formally, UI Map is defined as $G = (V, E)$, where V represents the set of nodes and E represents the set of edges. The node set V is partitioned into two distinct subsets:

- **Activity Nodes $\mathcal{N}_a$:** represent the set of activities in the app. Each node is associated with an attribute a_{sum}, describing the functionality summarization of this activity.
- **Element Nodes $\mathcal{N}_{el}$:** represent basic elements like Button, Textfield, CheckBox, and etc. Each node is associated with two attributes a_{static} and a_{dyn}, specifying its static features (e.g., id) and dynamic features (e.g., jump to *setting* activity when this button is clicked), respectively.

The set of edges E is composed of two types:

- **Containment Edges E_c:** for any $u \in \mathcal{N}_a$ and $v \in \mathcal{N}_{el}$, $(u, v) \in E_c$ if u contains v.
- **Transition Edges E_t:** edge $(s, t) \in E_t$, if there is a transition path from a source activity $s \in \mathcal{N}_a$ to target activity $t \in \mathcal{N}_a$.

Figure 2 depicts a portion of the UI Map for the app `Gallery` generated by UICOMPASS. With this structure, each activity node contains its own semantic attributes, such as "viewing files, adjusting settings", which helps quickly understand the functionality. The transition edges between activities (e.g., MainActivity to MediaActivity) aid in quickly understanding potential paths to the target activity. Each activity node is explicitly linked to a set of element nodes via containment edges. Each element comprises static attributes (e.g., @id/media_grid) and dynamic attributes (e.g., clicking this element to open media in a new activity). These attributes help understand the functionality of elements and infer the transitions they might trigger.

To automatically generate UI Map, UICOMPASS first statically analyzes the target application to extract G's nodes and edges. However, considering traditional static program analysis may miss some important nodes/edges (Rountev and Yan, 2014) and struggles to model dynamic behavior, we propose to integrate static program analysis with LLM. As a complement strategy, LLM infers the implicitly defined nodes/edges in the code and extract dynamic program behaviors, which can help UICOMPASS generate a comprehensive UI Map.

Static analysis for UI Map. In the first phase, static analysis is utilized to construct the initial UI Map by parsing three key components of the Android app: AndroidManifest.xml file, layout files, and code. The details of how each component contributes to the UI Map are as follows:

1) **AndroidManifest.xml File:** This essential XML document describes all activities $\mathcal{N}_a$ and partial transition edges E_t between activities.

2) **Layout Files:** Layout files define the visual structure of UI elements for an activity. By analyzing these files, UICOMPASS initializes the set of element nodes $\mathcal{N}_{el}$, extracts their static attributes a_{static}, and links them to the corresponding activity. Once an activity is associated with a layout, all its UI elements are assigned to that activity.

3) **Code Analysis:** Source code or bytecode from APK files is also utilized to expand the activity transition edges E_t and containment edges E_c. For activity transition edges E_t, UICOMPASS focuses on functions `startActivity()` and `startActivityForResult()` and performs data flow analysis to obtain the source

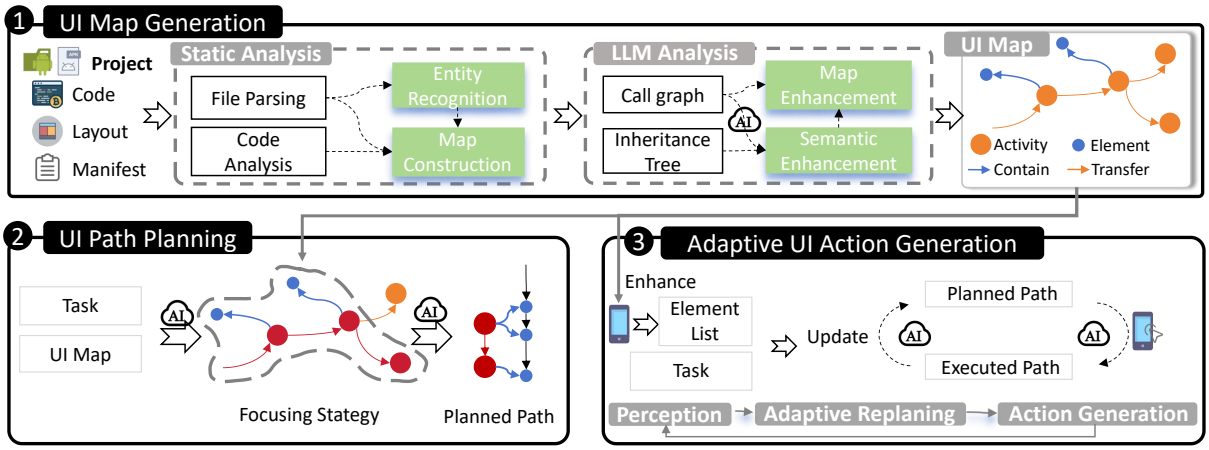

Figure 1: The overall workflow of UICOMPASS

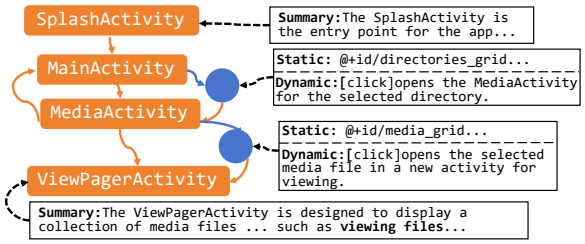

Figure 2: Part of the UI Map of the app named `Gallery`.

activity s and the target activity t, adding a directed edge (s, t). For containment edges E_c, UICOMPASS analyzes the layout loading functions `setContentView()` and `LayoutInflater.inflate()` and identifies the layout files specified in their parameters, thereby inferring the containment relationships between activities and elements.

LLM-based Semantic Enrichment. To address the limitations of static analysis in building a comprehensive UI Map, we use an LLM to semantically enrich the initial static graph. The LLM's strong code understanding enables it to handle complex scenarios like third-party, implicitly defined relations, and custom elements. To improve the LLM's comprehension, UICOMPASS performs call graph and data flow analyses, and builds an activity inheritance tree. Following the topological order of the call graph and providing crucial information such as called custom method summaries and variable definitions/declarations (highly useful for linking element IDs and event handlers), UICOMPASS prompts the LLM to perform the code analysis tasks, ultimately yielding four key outputs:

1) **Custom method summaries:** generate natural language functionality description of methods.

2) **Semantically enriched UI element nodes:** generates element's dynamic properties a_{dyn} by understanding the event handlers of elements.

3) **Enhanced activity transition edges (E_t):** uncover potential activity transition edges.

4) **Enhanced containment edges (E_c):** complement containment edges by leveraging the LLM's semantic understanding capabilities.

Additionally, UICOMPASS performs functional summarization of activity nodes in the UI Map. To capture inherited behaviors, it processes activities in topological order based on the inheritance tree. For each activity, the LLM combines summaries from its own class and parent class summaries to generate a comprehensive activity's functional summary a_{sum}. This LLM-driven step enhances the initial UI Map, making it semantically richer.

3.2 UI Path Planning

Just as in navigation task, the navigation system offers possible routes for reference while the actual driving process is decided by the driver according to real world situations. Inspired by this, UICOMPASS also provides the initial path planning and adjust it based on the state of the app during the actual execution.

UICOMPASS generates a planned path in natural language rather than specific actions. Actions typically include an action type (e.g., click, input), an element locator (e.g., element ID), and parameters (e.g., input text). However, locating elements from the static UI Map can be challenging because 1) some elements lack specific location information; 2) some elements are dynamically loaded. For

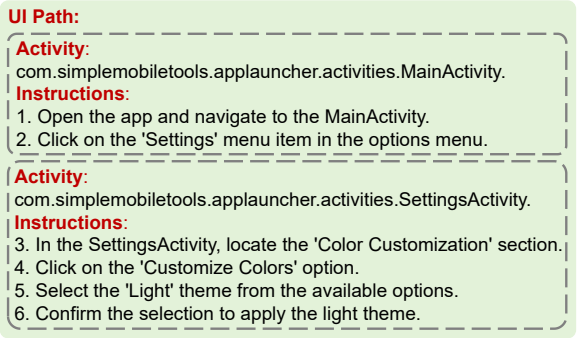

Figure 3: The generated UI path for task "Change theme color to light" in the "App Launcher" app.

instance, contact lists in a contact app are generated after data loads. Asking the LLM to generate concrete actions could lead to invalid locators, disrupting execution. Thus, UICOMPASS first creates a high-level plan using natural language instructions to outline necessary steps, and then generates concrete actions according to runtime UI states.

UICOMPASS generates a UI path by analyzing the relationship between the task and UI Map using LLM. To avoid overwhelming the LLM with excessive information, especially when the UI Map is large, UICOMPASS employs a focusing strategy. First, it summarizes all activities in the UI Map and identifies those relevant to the task. Then, it builds a partial graph of these focused activities and computes the shortest paths from the entry activity to all of them, further expanding the focus set to include intermediate activities. If the graph is small, all activities are considered focused to ensure comprehensive understanding. Specific prompt is provided in Appendix B.2. Finally, UICOMPASS generates a UI path $\mathcal{I}$ by referring to the partial UI Map. $\mathcal{I}$ comprises multiple action blocks, with each containing an activity ID $activity_i$ and a list of instructions I_i, indicating executing I_i in $activity_i$. Formally,

$$\mathcal{I} = \{(activity_1, I_1), \cdots, (activity_m, I_m)\}$$

For instance, Figure 3 illustrates the UI path for the task "Change theme color to light" in the "App Launcher" app. To complete this task, the generated UI path: open the `MainActivity`, click the `setting` button to transition to `SettingActivity`, and select `Light` theme.

This two-layer design helps UICOMPASS more intuitively understand the activities it will go through and the actions it needs to perform. UICOMPASS can determine whether the current path is incorrect from a global perspective by judging whether the current activity matches the activity in the UI Path. At the same time, the specific instructions assist UICOMPASS in quickly identifying what the current operation should be.

3.3 Adaptive UI Action Generation

To generate concrete UI actions, we propose an adaptive mechanism that combines initial natural language instructions with real-time program states. It operates in a loop with three stages: Perception, Adaptive Replanning, and Action Generation, up to a maximum number of iterations.

Perception: UICOMPASS first runs the target app and captures its screen information s in XML format, which details the UI elements and their static properties, such as IDs, text, and positions. This runtime screen information s is then aligned with the UI Map to determine the corresponding activity node n_s. Subsequently, s and n_s are merged into s', combining the full element list from s with the dynamic attributes a_{dyn} from n_s.

Adaptive Replanning: The UI path $\mathcal{I}$ obtained from Section 3.2 may not precisely model App's behaviors because of dynamically loaded elements. Hence, UICOMPASS conducts an adaptive replanning according to both UI Map and dynamic app states. Specifically, given s' and history instructions that have been executed, LLM is prompted to update UI path $\mathcal{I}$, and return the most appropriate next instruction i_n. When i_n is empty, it signifies that the LLM considers the task to be completed successfully. UICOMPASS consolidates this entire process into a single interaction with the aim of reducing the number of interactions and thereby shortening the time required for decision-making. Specific prompt and examples can be found in Appendix B.3.

Action Generation: Given that instructions are expressed in natural language and thus not directly executable, UICOMPASS must translate them into concrete actions. Specifically, UICOMPASS converts the elements mentioned in i_n to candidate executable actions based on their types, such as 'scroll', 'click', or 'input'. If an action type is 'input', UICOMPASS will prompt the LLM to provide the specific input value. For each action in the candidate list, LLM is provided with the static and dynamic attributes of the element to facilitate the LLM's deeper understanding of its functionalities and enable more informed decision-making. We provide prompt and examples in Appendix B.4. Once LLM selects a specific action, UICOMPASS

converts it into the corresponding Android Debug Bridge (ADB) command for execution.

4 Experiments

We evaluated UICOMPASS's performance experimentally.

4.1 Experimental Settings

Datasets. We evaluated UICOMPASS and baseline tools on DroidTask (Wen et al., 2024) and AndroidWorld (Rawles et al., 2025) dataset, which are commonly used to evaluate mobile task automation. We obtained 145 tasks from 12 different apps in DroidTask and 44 tasks from 9 apps in AndroidWorld. We primarily filtered out tasks that were unsupported by baseline tools or no longer executable. Selection criteria are detailed in Appendix A.2.

Baseline Methods. We chose AutoDroid (Wen et al., 2024), Guardian (Ran et al., 2024) and Mobile-Agent-v2 (Wang et al., 2024b) as baseline tools for experimental comparison. Both AutoDroid and Guardian utilize LLMs and exploration-based mechanisms to facilitate task execution. Mobile-Agent-v2 uses a Multi-agent architecture and VLM to assist in task completion.

Model Selection. Considering both cost efficiency and performance, we selected DeepSeek-V3 (DeepSeek-AI et al., 2025), Qwen-Max (Bai et al., 2023a), and Qwen-VL-max (Bai et al., 2023b; Qwen, 2025) as models for the evaluation.

Metrics. Following existing work (Wen et al., 2024; Ran et al., 2024), we measure the following metrics: 1) Success Rate (SR): The ratio of successfully completed tasks to the total number of tasks. 2) Average Completion Proportion (ACP): The proportion of the executed action sequence that matches the prefix of the ground truth action sequence. 3) Correct Termination Rate (CTR): The rate of successfully stopping exploration when the task is completed. 4) 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 Overall Results of Task Completion

We implemented two versions of UICOMPASS that extract UI Map from source code and byte code, respectively. The experimental results are shown in Table 1. Across experiments conducted with three models, two datasets, and four metrics, UICOMPASS's two versions achieved the best performance in 20 out of a total of 24 metric results. In terms of SR, UICOMPASS's source code version showed an average improvement of 14.52% compared to AutoDroid (the best-performing baseline) across all three models on both the DroidTask and AndroidWorld datasets, while its byte code version exhibited an average increase of 13.45%. In contrast, the Guardian and Mobile-Agent-v2 have trouble to understand app-specific information, leading to a comparatively lower task execution accuracy. In terms of ACP, CTR and SPL metrics, UICOMPASS also outperforms almost all the baseline tools. UICOMPASS outperforms existing tools since UI Map facilitates task understanding. Taking the task "Disable showing the dial pad button on the main screen" as an example, existing tools attempt to operate the *MainActivity* to disable the dial pad. However, this task is configured within the *SettingsActivity*. UI Map enables UICOMPASS to accurately comprehend the task's intent with efficiency. Overall, the experimental findings indicate that UICOMPASS, leveraging a UI Map-based approach, can effectively enhance the performance of mobile task automation.

Comparative analysis of different dataset: On the AndroidWorld dataset, performance was degraded compared to DroidTask across all evaluated tools. For instance, the success rate (SR) decreased to 37.21% for UICOMPASS (Byte code) and 41.86% for UICOMPASS (Source code). This performance decrease can be attributed to several key factors in the AndroidWorld dataset: (1) tasks often include more detailed and specific requirements, (2) there are more complex multi-step actions such as deleting multiple data items in a single task, and (3) tasks require deeper understanding, such as identifying duplicate data items.

Comparative analysis of different models. To investigate the impact of different LLMs on tools' performance, we conducted a comparative analysis between DeepSeek-V3 and Qwen-Max. Our experiments reveal two key insights: 1) Model selection significantly impacts tool performance, where stronger LLMs (e.g., Qwen-Max) yield higher task completion rates (+10.35% for UICOMPASS and +8.96% for AutoDroid vs. DeepSeek-v3). Due to enhanced reasoning and contextual understanding—a capability of Qwen-Max, UICOMPASS effectively completes more tasks then DeepSeek-v3. 2) UICOMPASS demonstrates consistent superiority across all metrics (SR/ACP/CTR/SPL) than AutoDroid, which conclusively establishes UICOM-

Benchmark	Methods	DeepSeek-v3				Qwen-Max				Qwen-VL-Max			
		SR↑	ACP↑	CTR↑	SPL↑	SR↑	ACP↑	CTR↑	SPL↑	SR↑	ACP↑	CTR↑	SPL↑
DroidTask	AutoDroid	53.79%	71.72%	**76.92%**	15.87%	62.75%	77.71%	64.35%	17.92%	56.55%	74.24%	**68.29%**	17.02%
	Guardian	45.20%	71.83%	0.0%	1.70%	53.10%	75.78%	2.59%	1.94%	40.69%	64.41%	0.00%	1.53%
	Mobile-Agent-v2	-	-	-	-	-	-	-	-	13.79%	26.17%	60.00%	4.63%
	UICompass(Byte code)	**63.44%**	**80.63%**	60.86%	**20.34%**	**74.48%**	**85.16%**	**75.00%**	**22.08%**	**72.41%**	**83.62%**	61.90%	**19.31%**
	UICompass(Source code)	**68.27%**	**81.96%**	**80.80%**	**20.92%**	**78.62%**	**87.07%**	**76.31%**	**24.15%**	**73.79%**	**82.79%**	**68.22%**	**20.76%**
AndroidWorld	AutoDroid	25.58%	**57.47%**	**63.63%**	1.12%	20.93%	**53.93%**	33.33%	1.48%	16.28%	**48.44%**	28.57%	0.85%
	Guardian	18.60%	31.99%	0.00%	0.18%	20.93%	35.40%	11.11%	0.41%	18.60%	37.29%	0.00%	0.16%
	Mobile-Agent-v2	-	-	-	-	-	-	-	-	6.81%	22.94%	33.33%	0.47%
	UICompass(Byte code)	**37.21%**	**48.86%**	50.00%	**2.15%**	**37.21%**	**46.15%**	**62.50%**	**2.28%**	**31.82%**	45.85%	**42.85%**	**1.39%**
	UICompass(Source code)	**30.23%**	41.86%	**53.84%**	**1.62%**	**30.23%**	43.37%	**69.23%**	**1.92%**	**41.86%**	**66.24%**	**55.56%**	**2.18%**

Table 1: Effectiveness of Task Completion (**Red** indicates 1st place, **blue** indicates 2nd place).

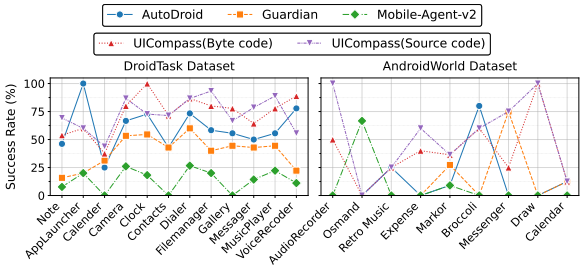

Figure 4: Success Rate across Different Apps

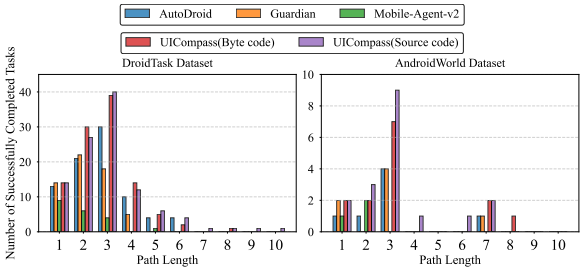

Figure 5: Success Rate across Different Path Length.

PASS's model-agnostic robustness.

Comparative analysis of different apps. To further explore UICOMPASS's performance across different apps, we present comparative experimental results in Figure 4. Our cross-app analysis considers Qwen-VL-Max as base model, as Mobile-Agent-v2 requires VLMs. Experiment results shows that UICOMPASS (either source code or bytecode version) outperforms all baselines in 84% (16/19) of apps. AutoDroid performs best on AppLauncher and Broccoli, and Mobile-Agent-v2 outperforms all tools on Osmand. This may be because: 1) Mobile-Agent-v2's vision module is optimized for map apps like Osmand, enabling direct coordinate clicks. 2) the failure of UICOMPASS to parse elements from third-party libraries causes incomplete UI Map, which leads to inferior performance compared to AutoDroid on two apps. Nonetheless, UICOMPASS performs robustly in most apps, demonstrating strong generalization.

Comparative analysis of different path lengths. We evaluate tools' performance with different task difficulties, with path length as a complexity metric. As shown in Figure 5, for simple tasks (1–2 steps), all the baseline tools and UICOMPASS perform consistently good, with UICOMPASS produces slightly better results. However, when automating medium-level complex tasks (3–6 steps), both Guardian and Mobile-Agent-v2 's performances are much worse than UICOMPASS. For longer tasks (7–10 steps), AutoDroid and Guardian complete only 1 and 2 tasks, respectively, whereas Mobile-Agent-v2 fails entirely. In contrast, UICOMPASS still complete 7 tasks. All tools failed to complete tasks that required more than 10 steps, so these tasks are not included in the figure. The evaluation results show that as tasks become more complex, UICOMPASS consistently achieves strong performance, while the performance of baseline tools drops significantly.

Comparison of source code and bytecode. Compared to the bytecode version, the source code version of UICOMPASS generally performs better on 9 apps. This is primarily because bytecode, affected by obfuscation, loses semantic information. In contrast, source code retains more semantic details, enabling UICOMPASS to better understand the app's logic and plan more accurate execution paths. However, in certain apps, e.g., *Clock*, the bytecode version may have an advantage. For example, bytecode includes compiled third-party libraries, which may contain UI design or Activity logic that is not included from the app's source code.

4.3 Analyzing Decision-Making Efficiency

To objectively compare the efficiency of different tools, we focus exclusively on measuring the duration of their decision-making phases (denoted as time_d). In UICOMPASS, this phase corresponds to the Adaptive Replanning process. Other phases

	Guardian	AutoDroid	UICompass
Average Step Time (s)	**0.89**	4.21	5.49
Step Efficiency	30	4.65	**3.43**
time_d$_{avg}$	26.91	19.6	**18.9**

Table 2: Time Efficiency Comparison.

(e.g., action execution) are susceptible to external noise factors such as device performance fluctuations and UI loading latency. These extrinsic variations could otherwise obscure the genuine disparities in tools' strategic capabilities. To ensure a fair comparison, we only analyzed tasks that were successfully completed by all tools except MobileAgent-v2 , denoted as $task_{\cap}$. We excluded MobileAgent-v2 due to its low number of completed tasks. Therefore, the average decision time of valid tasks (denoted as time_d$_{avg}$) can be calculated using the following formula:

$$\text{time\_d}_{\text{avg}} = \frac{1}{|task_{\cap}|} \sum_{t \in task_{\cap}} \text{time\_d}_t \quad (1)$$

We evaluated all three tools on 41 identical tasks, measuring both task completion efficiency $time\_d_{avg}$ and step efficiency (averaged steps per task). As shown in Table 2, UICOMPASS demonstrated superior step economy, requiring only 3.43 steps/task (mean) – 1.22 fewer than AutoDroid (4.65 steps) and 2.1 fewer than Guardian (5.53 steps). In terms of time, although UICOMPASS took longer for each decision-making step, its use of fewer steps resulted in less average time spent per task overall. Although Guardian consumes minimal time per interface (as it only outputs action indices without requiring the LLM to generate any reasoning content), the excessive number of steps substantially impacts its average decision time. For users, an agent performing a series of irrelevant actions on an app could pose significant risks.

4.4 Evaluating the Time and Cost of UI Map Generation

To assess the method's feasibility, we analyzed the computational overhead of the UI Map generation phase. It is crucial to note that this is a one-time, offline preprocessing step.

In the experiment, this process is performed by the deepseek-v3 model. Figure 6 illustrates the two key metrics measured for each of the 19 applications: the total generation time (in seconds) and the corresponding API monetary cost (in US dollars), sorted in descending order. Across all

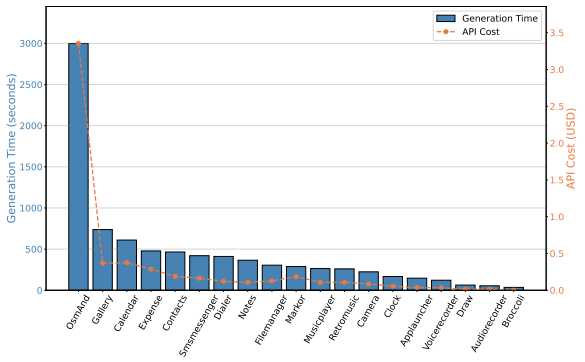

Figure 6: Analysis of UI Map Generation Time and API Cost per Application.

Config	UI Map	Adaptive	SR↑	ACP↑	CTR↑	SPL↑
C-1	×	×	37.24%	64.57%	60.00%	1.77%
C-2	✓	×	42.75%	64.43%	66.67%	15.00%
C-3	×	✓	55.86%	72.62%	53.84%	17.66%
C-4	✓	✓	**68.27%**	**81.96%**	**80.80%**	**30.85%**

Table 3: Ablation Results of UICOMPASS on DroidTask

applications, the average generation time was approximately 509 seconds (8.48 minutes), with an average monetary cost of $0.317. Given that this is a one-time preprocessing cost, we think that the required computational overhead is acceptable.

4.5 Ablation Study

To evaluate the effectiveness of UI Map and the adaptive replanning, we further conducted ablation experiments. When UI Map is omitted, UICOMPASS generates an initial UI path based solely on the remaining information. When adaptive replanning is disabled, UICOMPASS follows the UI path strictly in its original sequence.

The effectiveness of UI Map. The experimental results in Table 3 demonstrate that UI Map significantly improves task automation performance, both with (Config C-2 and C-4) and without (Config C-1 and C-3) UI Map. Specifically, task success rates (SR) increased by 5.51% and 12.41% respectively, proving that global information effectively guides task execution. Task termination (CTR) accuracy improved by 6.67% and 26.96%, attributable to optimized initial route planning; Concurrent improvements in ACP and SPL metrics indicate the system enhances completion rates while reducing redundant exploration steps.

The effectiveness of adapting replanning. The experimental results demonstrate that UI Map's Adapting module significantly enhances task completion capabilities: Task success rates improved by

18.62% (Config C-3 over C-1) and 25.52% (Config C-4 over C-1), while ACP metrics increased by 8.05% and 17.53% respectively. This confirms the effectiveness of adaptive replanning in dynamically adjusting execution paths based on real-time application states. The CTR decreased by 6.16% (Config C-3 over C-1) but increased by 14.13% (Config C-4 over C-1), indicating that UI Map's guidance is more crucial for recognizing task completion. The substantial performance gap observed between C-2 and C-4 stems from the gap of static code and dynamic runtime behaviors. A representative case occurs when the application dynamically skips onboarding screens while the predicted path continues to include these unnecessary instructions. These findings compellingly demonstrate the importance of the adaptive model in UICOMPASS.

5 Conclusion

In this paper, we propose a method for mobile task automation using a UI Map extracted from the code, called UICOMPASS. UICOMPASS leverages LLM and static analysis to analyze the code and generates the UI Map. UICOMPASS can then use this UI Map and the given task to generate an initial UI path. During task execution, we introduce adaptive replanning that combines action history and UI to continuously replan the UI path. Through extensive experiments, we demonstrate the effectiveness of UICOMPASS in task completion capability, achieving state-of-the-art performance.

Limitation

Although our work demonstrates UICOMPASS achieves excellent performance, it still has some limitations. 1) The integrity of the UI Map based on code analysis is insufficient. Due to the complex implementation of programs (such as dynamically loaded elements), code parsing is hard to obtain all elements and their functionalities, leading to the possibility that the UI Map may miss some elements. These missing elements affect the task success rate of UICOMPASS. Although UICOMPASS uses adaptive replanning to mitigate this issue, we will still further explore better methods to obtain a more complete UI Map in the future. 2) UICOMPASS only uses the UI Map of a single app as a reference. When facing cross-app tasks, UICOMPASS relies on LLM to infer the actions that should be performed on other apps. This limits the performance of UICOMPASS in cross-app tasks. In the future, we will explore how to combine the UI Maps of multiple apps to help improve the performance of cross-app tasks.

Acknowledgement

This work was supported by National Natural Science Foundation of China under Grant No 62202026.

References

Jinze Bai, Shuai Bai, and etc. 2023a. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Jinze Bai, Shuai Bai, Shusheng Yang, Shijie Wang, Sinan Tan, Peng Wang, Junyang Lin, Chang Zhou, and Jingren Zhou. 2023b. Qwen-vl: A versatile vision-language model for understanding, localization, text reading, and beyond. *arXiv preprint arXiv:2308.12966*.

Max Brunsfeld and Contributors. 2025. Tree-sitter: An incremental parsing system for programming tools. Accessed: 2025-02-01.

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.

Filippos Christianos, Georgios Papoudakis, Thomas Coste, Jianye Hao, Jun Wang, and Kun Shao. 2025. Lightweight neural app control. *Preprint*, arXiv:2410.17883.

DeepSeek-AI, Aixin Liu, Bei Feng, and etc. 2025. Deepseek-v3 technical report. *Preprint*, arXiv:2412.19437.

Shihan Deng, Weikai Xu, Hongda Sun, Wei Liu, Tao Tan, Jianfeng Liu, Ang Li, Jian Luan, Bin Wang, Rui Yan, and Shuo Shang. 2024. Mobile-bench: An evaluation benchmark for llm-based mobile agents. *Preprint*, arXiv:2407.00993.

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.

Jakub Hoscilowicz, Bartosz Maj, Bartosz Kozakiewicz, Oleksii Tymoshchuk, and Artur Janicki. 2024. Clickagent: Enhancing ui location capabilities of autonomous agents. *Preprint*, arXiv:2410.11872.

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.

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.

Hongxin Li, Jingfan Chen, Jingran Su, Yuntao Chen, Qing Li, and Zhaoxiang Zhang. 2025. Autogui: Scaling gui grounding with automatic functionality annotations from llms. *Preprint*, arXiv:2502.01977.

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.

Xinbei Ma, Zhuosheng Zhang, and Hai Zhao. 2024. Coco-agent: A comprehensive cognitive mllm agent for smartphone gui automation. *Preprint*, arXiv:2402.11941.

Qwen. 2025. Introducing qwen-vl.

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.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy Lillicrap, and Oriana Riva. 2025. Androidworld: A dynamic benchmarking environment for autonomous agents. *Preprint*, arXiv:2405.14573.

Atanas Rountev and Dacong Yan. 2014. Static reference analysis for gui objects in android software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 143–153.

Google Assistant Routines. 2025. Automate daily routines & tasks with google assistant.

Apple Shortcuts. 2025. Run shortcuts with siri, the shortcuts app or siri suggestions.

Skylot. 2025. Jadx: Dex to java decompiler. Accessed: 2025-02-01.

Yunpeng Song, Yiheng Bian, Yongtao Tang, Guiyu Ma, and Zhongmin Cai. 2024. Visiontasker: Mobile task automation using vision based ui understanding and llm task planning. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, UIST '24, page 1–17. ACM.

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. 2024a. Mobile-agent-v2: Mobile device operation assistant with effective navigation via multi-agent collaboration. *arXiv preprint arXiv:2406.01014*.

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

Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024c. Mobile-agent: Autonomous multi-modal mobile device agent with visual perception. *Preprint*, arXiv:2401.16158.

Luyuan Wang, Yongyu Deng, Yiwei Zha, Guodong Mao, Qinmin Wang, Tianchen Min, Wei Chen, and Shoufa Chen. 2024d. Mobileagentbench: An efficient and user-friendly benchmark for mobile llm agents. *Preprint*, arXiv:2406.08184.

Wenhao Wang, Zijie Yu, William Liu, Rui Ye, Tian Jin, Siheng Chen, and Yanfeng Wang. 2025. Fedmobileagent: Training mobile agents using decentralized self-sourced data from diverse users. *Preprint*, arXiv:2502.02982.

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.

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.

An Yan, Zhengyuan Yang, Wanrong Zhu, Kevin Lin, Linjie Li, Jianfeng Wang, Jianwei Yang, Yiwu Zhong, Julian McAuley, Jianfeng Gao, Zicheng Liu, and Lijuan Wang. 2023. Gpt-4v in wonderland: Large multimodal models for zero-shot smartphone gui navigation. *Preprint*, arXiv:2311.07562.

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*.

Chi Zhang, Zhao Yang, Jiaxuan Liu, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2023a. Appagent: Multimodal agents as smartphone users. *Preprint*, arXiv:2312.13771.

Jiayi Zhang, Chuang Zhao, Yihan Zhao, Zhaoyang Yu, Ming He, and Jianping Fan. 2024b. Mobileexperts: A dynamic tool-enabled agent team in mobile devices. *Preprint*, arXiv:2407.03913.

Jiwen Zhang, Jihao Wu, Yihua Teng, Minghui Liao, Nuo Xu, Xiao Xiao, Zhongyu Wei, and Duyu Tang. 2024c. Android in the zoo: Chain-of-action-thought for gui agents. *Preprint*, arXiv:2403.02713.

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

Zichen Zhu, Hao Tang, Yansi Li, Dingye Liu, Hongshen Xu, Kunyao Lan, Danyang Zhang, Yixuan Jiang, Hao Zhou, Chenrun Wang, Situo Zhang, Liangtai Sun, Yixiao Wang, Yuheng Sun, Lu Chen, and Kai Yu. 2025. Moba: Multifaceted memory-enhanced adaptive planning for efficient mobile task automation. *Preprint*, arXiv:2410.13757.

Appendices

A Further Experimental Details and Extended Investigations

In this section, we provide further elaboration on our experimental details and present additional experiments to demonstrate the performance of UICOMPASS. Finally, we illustrate the execution process of UICOMPASS on a specific task to facilitate understanding of adaptive UI action generation.

A.1 Implementation Detail

We implement UICOMPASS by using the following key components.

- **Decompilation:** We use JADX (Skylot, 2025) to decompile APKs into Java code. This enables direct analysis of app byte code.
- **Program Analysis:** We leverage Tree-Sitter (Brunsfeld and Contributors, 2025) to parse Java code and Kotlin code. From these parsed result, we extract call graphs, data flows, and inheritance trees.

The bytecode version enables easier analysis of third-party libraries, but at the cost of increased analysis overhead. Following the source code version, we restrict analysis to the current project's code (filtered by package names). For potential third-party activities, we limit analysis to the activity class level.

A.2 Detail of Benchmark.

We selected DroidTask (Wen et al., 2024) as the benchmark dataset, which include 12 open-source apps and 149 tasks. DroidTask covers all projects and tasks presented in (Wang et al., 2024d)'s study. During evaluation, four tasks were found to be no longer executable, resulting in a final total of 145 usable tasks.

We also selected AndroidWorld (Rawles et al., 2025) as another benchmark. AndroidWorld comprises 116 distinct tasks across 20 real-world apps. However, 28 of these tasks involve system applications, for which APKs are unavailable. An additional 26 tasks are question-answering based, making them incompatible with all baseline tools in our experiments. Furthermore, 18 tasks were excluded due to reasons such as inaccessible login requirements, redundancy with other tasks, or being vision-only tasks (e.g., image recognition). After

this filtering process, our final experimental dataset consists of 44 tasks from 9 applications.

We hope to conduct experimental comparisons between the source code version and the byte version of UICOMPASS to uncover more interesting findings. Several other datasets (Deng et al., 2024) were not included due to the non-open-source nature of their apps.

A.3 Experimental Evaluation Methods.

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) and AndroidWorld (Rawles et al., 2025). 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. For each task, we conducted three trial runs. Successful attempts were recorded for performance analysis. For failed tasks, only the most successful attempt (e.g., highest completion progress) among three trials is recorded, while others are discarded.

A.4 Detail of Experimental Results on Task Completion

As presented in Table 4, the experimental results reveal significant disparities in the task completion performance of various tools across different applications and datasets. In the DroidTask dataset, UICOMPASS (source code) and UICOMPASS (byte code) demonstrate a distinct advantage, successfully completing a larger number of tasks in multiple applications such as SimpleNote and Camera. Within the AndroidWorld dataset, the number of tasks completed by all tools is generally lower, yet UICOMPASS still stands out with relatively better performance in certain applications. Overall, UICOMPASS (source code) accomplishes 125 tasks, while UICOMPASS (byte code) completes 119 tasks. These figures are significantly higher than those of other tools, such as Guardian (66 tasks) and Mobile-Agent-v2 (23 tasks), thereby shows the efficiency and superiority of UICOMPASS in task completion.

A.5 Case Study

We analyze a representative DroidTask case (Fig. 7) demonstrating UICOMPASS's effectiveness, particularly its UI Map-guided process and adaptive instruction 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 7 depict the correct steps generated by UICOMPASS. The task requires navigating to the "Customize colors" module in Settings (Steps 1-3) and switching the theme to light before saving (Steps 4-6). This example presents a challenge for the existing mobile agent. While the content of tasks shows that it is a setting operation, users face confusion because the settings activity neither directly displays theme color options nor makes it clear that 'Customize colors' (Step 3) can modify theme colors.

UICOMPASS generates the initial UI path using the UI Map (shown in the top-left of Figure 7). Due to the complexity of the application, the UI path generated by UICOMPASS is not entirely correct. In this example, we can see that most of the UI path 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 7), UICOMPASS uses the adaptive replanning mechanism to correct the errors in the initial UI path based on the actual execution context. Therefore, the initial UI path generated by the UI Map, combined with the adaptive replanning mechanism, can effectively guide the LLM to complete tasks in the target application.

B More Details and Example of Prompts

In this section, we will detail several of the most important prompts used in the interaction between UICOMPASS and the LLM in this paper. We will provide the templates and specific examples for these prompts to facilitate understanding.

B.1 The Prompt for the Semantic Enhance for UI Map via LLM

Following UICOMPASS constructs the initial UI Map using static analysis techniques, UICOMPASS further enhances it by leveraging an LLM. This enhancement primarily aimed to enable UICOMPASS to extract information from more complex

Table 4: Number of Tasks Completed by Different Tools Across Applications

Dataset	Application	Total	AutoDroid	Guardian	Mobile-Agent-v2	UICOMPASS (Byte code)	UICOMPASS (Source code)
DroidTask	SimpleNote	13	6	2	1	7	9
	AppLauncher	5	5	1	2	6	6
	Calender	16	4	5	0	6	7
	Camera	15	10	8	4	12	13
	Clock	11	8	6	2	11	8
	Contacts	14	6	6	0	10	10
	Dialer	15	11	9	4	13	13
	Filemanager	15	8	6	3	12	14
	Gallery	9	5	4	0	7	6
	Messager	14	7	6	2	9	11
	MusicPlayer	9	5	4	2	7	8
	VoiceRecoder	9	7	2	1	8	5
AndroidWorld	AudioRecorder	2	0	0	0	1	2
	Osmand	3	0	0	2	0	0
	Retro Music	4	1	0	0	1	1
	Pro Expense	5	0	0	0	2	3
	Markor	11	1	3	1	4	4
	Broccoli	5	4	0	0	3	3
	Messager	4	0	3	0	1	3
	Draw	1	0	0	0	1	1
	Calender	9	1	1	0	1	1
Total	19	189	89	66	23	119	125

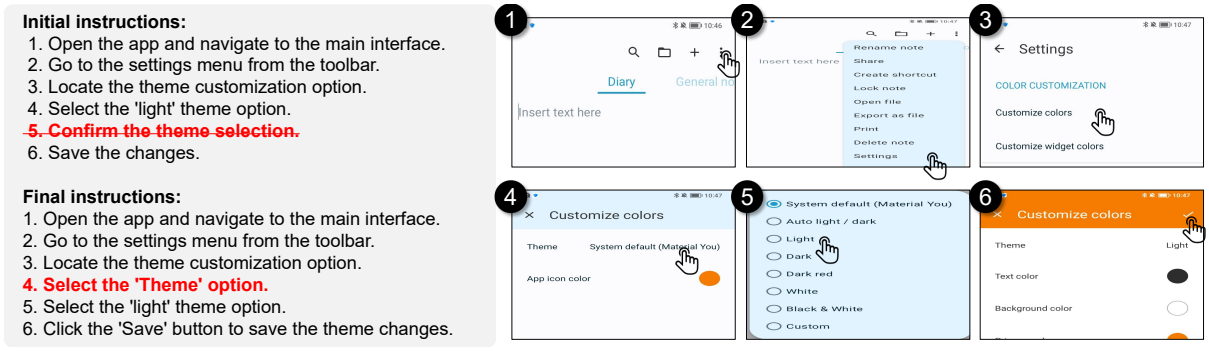

Figure 7: A successful case on DroidTask demonstrates the effectiveness of UICOMPASS's UI Map and adaptive UI action generation.

Table 5: Prompt Template for Semantic Enhance for UI Map via LLM

 # System
You are an Android source code analysis assistant.

 # User
[Background]
You are an Android analyst. I will give you a method from the class **{Class name}**.
Here is the method from the given Android source code:
{Method}
[Method Variable Context]
{Global variable definition and assignment}
[Invoked Method Summary]
Here is the explanation of the method named **{Method name}** that is called within the given method:
{Description of these methods}
[Target]
Based on your analysis, please provide the following information:
1. **Method Summary:** Provide a concise summary of the functionality of this method. Describe what the method does.
2. **Dynamic Attributes of UI Element Analysis:** Identify all **UI elements** present in this method. For each identified UI element, provide the following information:
* **Type:** (e.g., Button, TextView, ImageView, MenuItem)
* **ID:** (if available, e.g., R.id.button_login)
* **Function:** Describe the effects of interacting with or the execution related to this UI element (e.g., UI update, navigation to a new activity, data modification, triggering a function call).
3. **Activity Transfer Relationship:** Summarize any activity transfer relationships in the code.
4. **Activity-Fragment Relationships:** Identify and describe any dependencies or relationships between activities and fragments within the code (e.g., a fragment being added, replaced, or associated with an activity using 'FragmentTransaction').
5. **Layout Relationships:** Identify any relationships between activities or fragments and XML layout files (starting with 'R.layout') or menu files (starting with 'R.menu'). Specify which layout files are inflated or referenced by which activities or fragments (e.g., using 'setContentView()', 'LayoutInflater', 'FragmentTransaction.replace()').
Output Example
{Output Example}

programming scenarios and augment the UI Map with semantic information to facilitate understanding, thereby making the enhanced UI Map more robust. Table 5 presents the prompt template employed in this stage. During this process, UICOMPASS first provides the source code of a method along with its contextual information, including variable definitions and assignments, as well as summaries of the methods it calls. Given that this stage processes each method according to the topological order of the call graph, the summaries of the methods called by the current method are already available. Based on this information, UICOMPASS prompts the LLM to accomplish our intended objectives. As shown in the main text, UICOMPASS requests the LLM to provide a summary of the method, the dynamic attributes of the involved elements, activity transition relationships, and layout relationships. Considering that many applications reuse layouts through fragments, we still account for fragment scenarios. If a fragment is attached to an activity, then all elements within the layout corresponding to that fragment are added to the activity. Finally, UICOMPASS instructs the LLM to output the information according to a specified output format, which facilitates subsequent processing by UICOMPASS.

B.2 The Prompt for the UI Path Planning

The main purpose of the prompt in the UI path planning stage is to guide the LLM in generating a possible execution path for a given application and

Table 6: Prompt Template for UI Path Planning in UICOMPASS

 # System
You are a helpful AI mobile phone operating assistant.

 # User
[Background]
I need to execute a target task within the application. Could you assist in designing the step-by-step instructions to achieve it? I will provide you with the UI Map for the application. The UI Map is a graph used to describe the application's user interface and interaction logic. Your task is to speculate on what instructions are used to execute the given task.
[UI Map]
Here is the UI Map of this app:
Activity list:
$\mathcal{N}_a$
Information about these activities:
Activity name:
{Activity name}
The summary of **{Activity name}**: {a_{sum}}
This activity can be transferred to other activities: $E_t.t$,
{$\mathcal{N}_{el}$}
[Task Description]
Based on the aforementioned application information, our goal is to execute the task: $\{t\}$
[Output Example]
Here's a reference output example. Based on this format, list all activities involved in the task and the corresponding instructions per activity. Output must adhere to the following JSON format.
{Output Example}

target task t.

Table 6 shows the template for the prompt at this stage, which illustrates how UICOMPASS converts the UI Map into easily understandable text format. Within the template, UICOMPASS first informs the LLM about the information that will be provided, and specifies the expectation that the LLM should generate a UI path capable of completing the task. Subsequently, the UI Map is converted into a text format. In this part, UICOMPASS first provides all activity nodes $\mathcal{N}_a$ within the application, and then elaborates on the information for each Activity. For each activity, UICOMPASS provides three key pieces of information: 1) the name of the activity and the functional summary a_{sum}, which helps the LLM quickly understand the activity; 2) the activities that can be navigated to from the current activity, clarifying relationships between activities; and 3) information about all elements contained within that activity. As illustrated by the prompt example for this stage in Table 7, UICOMPASS displays not only the static properties of elements (such as tag and id), which helps quickly identify element types, but also provides dynamic property information. This significantly assists the LLM in understanding element functions and determining their relevance to the task. Finally, UICOMPASS clearly defines the specific task requirements and provides an output example, requesting the LLM to output in the specified format.

The prompt at this stage effectively extracts and organizes the key information about app UI interaction, enabling the LLM to quickly generate effective UI paths.

B.3 The Prompt for the UICOMPASS's Adaptive Decision Making.

The adaptive decision-making component of UICOMPASS is responsible for adjusting the planned path (represented as an instruction list) and determining the next instruction. Table 8 presents the prompt template for this component. Initially, the 'Background' section introduces the task to be executed to the LLM. It informs the LLM that UICOMPASS will provide a UI path for the task t but notes that potential errors may exist, requiring the LLM to identify and correct them. Subsequently, essential information is provided, includ-

ing UI path $\mathcal{I}$, screen S', and history $\mathcal{A}$. In the history section, UICOMPASS furnishes the history of executed actions and the history of instructions. This facilitates the LLM's assessment of the task completion status from both action and instruction perspectives. Finally, UICOMPASS specifies the desired output data and its corresponding explanation to the LLM, and requires the LLM to generate the output in the designated format. Table 9 offers a concrete example of a prompt. It can be observed that UICOMPASS provides substantial relevant information to aid the LLM in understanding the task execution state. As demonstrated by the LLM's response in Table 10, the screen information supplied by UICOMPASS effectively assists the LLM in recognizing that the interface display indicates the task objective is complete (e.g., the font size has been adjusted). Based on the historical information, the LLM can further infer that all instructions have been completed and accurately determine that the next instruction should be 'none', signifying the task's conclusion. From the template, example, and the LLM's response, the effectiveness of the prompt design for UICOMPASS's adaptive decision-making module is evident.

B.4 UICOMPASS Action Selection Prompt

Table 11 presents the prompt template used by UICOMPASS for selecting corresponding candidate elements based on a given instruction I_n and task t. Inspired by the design of the Guardian tool, we designed the prompt template for selecting action candidates. UICOMPASS automatically identifies the type of each element and analyzes the possible actions that can be performed on it based on its type and properties. These actions are then added to an action candidate list. Consequently, in the prompt, UICOMPASS lists all available action candidates. This list may include dynamic attributes to describe the element's functionality. For instance, the element at index 4 in table 12 might indicate its ability to open a new window. Subsequently, UICOMPASS provides the current task t and the instruction I_n to be executed, explicitly stating the required output format. The LLM is only required to output a single number, such as 4, corresponding to the index in the candidate list. Suppose none of these candidate elements is related. In that case, the LLM should output "index-none", indicating that no relevant element is present on the current user interface, in which case UICOMPASS will perform a return operation. For input actions, UICOMPASS's handling them is similar to Guardian's design - it will query the LLM again for the text to be input. Table 12 provides a concrete example, illustrating that we provide various attributes of the element along with the actions to be executed. The element attributes information is extracted from the XML interface description obtained from the Android device. This information can reflect the element's state during execution (e.g., selected=true indicates it is selected), which is crucial for UICOMPASS to determine whether the task has been successfully completed.

C License and Terms for Derived Artifacts

Use of Existing Artifacts:

- **Guardian Tool (Ran et al., 2024)**: Licensed under Apache 2.0, which imposes no restrictions on intended use. Our application of Guardian for developing a mobile automation tool is consistent with its open-source purpose. We confirm there are no specified use restrictions in Guardian's original license or documentation that our usage violates.
- **AndroidWorld Dataset (Rawles et al., 2025)**: Released under Apache License 2.0, permitting free use. Our usage complies with all license requirements.
- **DroidTask Dataset (Wen et al., 2024)**: Licensed under MIT License, allowing free use.
- **Tree-sitter (Brunsfeld and Contributors, 2025)**: Utilized under MIT License, which permits free use in our project with minimal restrictions.
- **JADX (Skylot, 2025)**: Used under Apache-2.0 license, compatible with our project's licensing and usage requirements.

Derived Artifact Compliance:

- Our tool is released under the same Apache License 2.0 (full text included in the repository's LICENSE file), maintaining compatibility with all incorporated components' licenses.

Table 7: Prompt Example for UI Path Planning in UICOMPASS

 # System

You are a helpful AI mobile phone operating assistant.

 # User

[Background]

I need to execute a target task within the application. Could you assist in designing the step-by-step instructions to achieve it? I will provide you with the UI Map for the application. The UI Map is a graph used to describe the application's user interface and interaction logic. Your task is to speculate on what instructions are used to execute the given task.

[UI Map]

Here is the UI Map of this app:

Activity list:

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

Information about these activities:

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..."`

This activity can be transferred to other activities: `SplashActivity, AboutActivity, SettingsActivity,`

`index-1: 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.`

`index-2: tag:include, id:@+id/search_wrapper, action:include, effect:Embeds the search bar layout, enabling search functionality within the activity.`

...

[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]

Here's a reference output example. Based on this format, list all activities involved in the task and the corresponding instructions per activity. Output must adhere to the following JSON format.

```
{"task": "Book a flight", "UI path": [ {"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."]}]}
```

Table 8: The prompt template for the adaptiveDecisionMaking process of UICOMPASS.

 # System
You are a helpful AI mobile phone operating assistant.

 # User
[Background]
I currently have a task **{Task}**, 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 user interface.
[Instrctions]
Instructions:
$\{\mathcal{I}\}$
[Screen]
Here is the information about the screen we are currently on.
$\{S'\}$
[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.):
{Action List}
Here is the history of executed instructions:
{Executed Instructions}
[Output Explanation and Example]
Based on this information, please tell me <current state>, <finished instruction>, <error reason>, <next_instruction>, <updated instructions>.
Note that:
+ current state: Summarize the current program state according to the given widgets.
+ finished instruction: Summarize the just-completed instruction.
+ error reason: Please analyze if there are any errors in the UI path. When the UI path is correct, the error reason should be output as empty.
+ next_instruction: next_instruction should correspond to only one action. If not, please split the instruction, ensuring that the next instruction corresponds to a single action. If the Task is finished, next_instruction = none.
This is an output example:
{Output Example}
Warning:
You should tell me the updated instructions according to this format. (**Do not output any else except the JSON format.**)

Table 9: The prompt example for the adaptiveDecisionMaking process of UICOMPASS.

 # System

You are a helpful AI mobile phone operating assistant.

 # User

[Background]

I currently have a task `Adjust the fontsize of the Notes app to 125%`, 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 user interface.

[Instrctions]

Instructions:

```
{'task': 'Adjust the fontsize of the Notes app to 125%', 'UI_path': [{'activity':
'MainActivity', 'steps': ['1. Open the Notes app.', '2. Navigate to the settings
menu.']}, {'activity': 'SettingsActivity', 'steps': ["3. Locate the 'Font Size'
option.", "4. Click on the '125%' option to adjust the font size.", "5. Click 'OK'
to confirm the font size adjustment"]}]}
```

[Screen]

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

```
index-0: a View (accessibility information: Open note, resource_id "open_note",
content-desc: "Open note", text: "", selected: "false", checked: "false", text: )
to click
index-1: a View (accessibility information: Create a new note, resource_id
"new_note", content-desc: "Create a new note", text: "", selected: "false", checked:
"false", text: ) to click
```

...

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

...

Here is the history of executed instructions:

```
index-1:"Navigate to the settings menu.".
index-2:"Locate the 'Font Size' option.".
```

...

[Output Explanation and Example]

{Output Explanation.}

This is an output example:

{ "task": "Book a flight", "explanation": { "current state": "The booking is complete but not confirmed", "finished instruction": "All booking steps are done", "error reason": "Missing confirmation click on 'Confirm' or 'Book Now' button", "next_instruction": "Click 'Confirm' to finalize the booking" }, "updated_UI_path": [{ "activity": "LoginActivity", "steps": ["1. Open the application Booking", "2. Input the account.", "3. Submit the login form."] }, { "activity": "MainActivity", "steps": ["4. Search for available flights based on your preferences.", "5. Select the flight that suits your needs."] }, { "activity": "BookingActivity", "steps": ["6. Enter the required passenger details for booking.", "7. Make the payment for the selected flight.", "8. Receive a confirmation of the flight booking." "9. Click 'OK' to confirm login"] }] }

Warning: You should tell me the updated instructions according to this format. (**Do not output any else except the JSON format.**)

Table 10: The LLM's Answer for the Prompt in UICOMPASS's the Adaptive Decision Making process.

 # LLM's Answer:


```
{
  "task": "Adjust the fontsize of the Notes app to 125%",
  "explanation": {
    "current state": "The font size adjustment is complete and confirmed",
    "finished instruction": "All steps for adjusting the font size are done",
    "error reason": "",
    "next_instruction": "none"
  },
  "updated_UI_path": [
  {
    "activity": "MainActivity",
    "steps": [
      "1. Open the Notes app.",
      "2. Navigate to the settings menu."
    ]
  }, {
    "activity": "SettingsActivity",
    "steps": [
      "3. Locate the 'Font Size' option.",
      "4. Click on the '125%' option to adjust the font size.",
      "5. Click 'OK' to confirm the font size adjustment"
    ]
  }
  ]
}
```


Table 11: Prompt Template for Action Selection in UICOMPASS

 # System

You are a helpful AI mobile phone operating assistant.

 # User

[Action Candidates]

Currently we have **{number}** widgets, namely:

{List of candidate widget actions}

We now want to execute this instruction **$\{I_n\}$**, which is part of our test target is to **{t}**.

[Output Requirement]

Please choose only one UI element with its index, such that the element can bring us closer to our test target.

If none of the UI elements can do so, respond with index-none.

Table 12: Prompt Example for Action Selection in UICOMPASS

 # System
You are a helpful AI mobile phone operating assistant.

 # User
Currently we have 12 widgets, namely:
index-0: a View (accessibility information: Back, text:) to click
index-1: a View (accessibility information: , resource_id "settings_nested_scrollview", content-desc: "", text: "", selected: "false", checked: "false", text:) to vertical_scroll
index-2: a View (accessibility information: , resource_id "settings_nested_scrollview", content-desc: "", text: "", selected: "false", checked: "false", text:) to horizontal_scroll
index-3: a View (accessibility information: , resource_id "settings_color_customization_holder", content-desc: "", text: "Customize colors", selected: "false", checked: "false", text: Customize colors) to click
index-4: a View (accessibility information: , resource_id "settings_change_date_time_format_holder", content-desc: "", text: "Change date and time format", selected: "false", checked: "false", text: Change date and time format. This element is used for: <When clicked, it triggers the display of a dialog for changing the date and time format.>.) to click
...
We now want to execute this instruction **"`3. In the SettingsActivity, locate the 'Color Customization' section.`"**, which is part of our test target to `change the theme to light on Simple-File-Manager.`
Please choose only one UI element with its index, such that the element can bring us closer to our test target.
If none of the UI elements can do so, respond with index-none.