MobiGraph: TOWARDS A PRACTICAL EVALUATION PARADIGM FOR MOBILE AGENTS VIA TRAJECTORY-FUSED STATE GRAPHS

Anonymous authors
Paper under double-blind review

ABSTRACT

Mobile agents can autonomously complete user-assigned tasks through GUI interactions. However, existing mainstream evaluation benchmarks, such as AndroidWorld, operate by connecting to a system-level Android emulator and provide evaluation signals based on the state of system resources. In real-world mobile agent scenarios, many third-party applications do not expose system-level APIs to determine whether a task has succeeded, leading to a mismatch between benchmarks and real-world usage. To address these issues, we propose *MobiGraph*, an evaluation framework that supports task construction for arbitrary third-party apps. Using an efficient graph-construction algorithm based on multi-trajectory fusion, *MobiGraph* can effectively compress the state space, support dynamic interaction, and better align with real-world third-party application scenarios. *MobiGraph* covers 20 widely used third-party applications and comprises 240 diverse real-world tasks, with enriched evaluation metrics. Compared with AndroidWorld, *MobiGraph*'s evaluation results show higher alignment with human assessments and can guide the evolution of future GUI-based models under real workloads. The **code**&**data** can be found here: code/Anony-MobiGraph & data/Anony-MobiGraph.

1 INTRODUCTION

With the advancement of artificial intelligence technology, Graphic User Interface (GUI) agents (Ye et al., 2025a), driven by multimodal models(Ma et al., 2024), are emerging as a key technology poised to transform next-generation terminal applications. These agents, guided by human instructions, automatically accomplish daily and professional tasks across diverse device environments, thereby enhancing production efficiency and improving user operational experience. However, how to conduct fast and accurate evaluations of GUI agents' actual potential based on real-world usage scenarios remains a significant challenge(Rawles et al., 2024).

Existing mobile-use evaluation benchmarks are primarily evaluated through online and offline methods. Online benchmark enables interaction with agents via Android emulators and extracts evaluation signals from system-level resource states(Toyama et al., 2021; Rawles et al., 2024). However, the vast majority of third-party applications do not expose system-level interfaces, making it difficult to obtain accurate evaluation signals and consequently limiting the range of applications that can be evaluated. For the Apps that can be evaluated, the environmental factors also make experimental reproduction difficult. Moreover, for vendor applications that do provide system-level APIs, mobile agents operating through GUI interactions are of limited practical relevance. As a result, such evaluation settings inevitably exhibit a gap from real-world usage scenarios.

Offline benchmarks rely on pre-collected human interaction trajectories and assess agents by comparing their action sequences against these reference trajectories(Zhang et al., 2024a; Xu et al., 2025a). While this approach can be applied to arbitrary Apps, the state space of mobile user interfaces is nearly infinite, making it difficult for offline data collection to cover all possible states. Consequently, existing offline datasets typically provide only one or a few trajectories per task. Some studies(Song et al., 2025) attempt to construct a complete interface state transition graph by traversing all interactive UI elements; however, as the graph depth increases, the state space grows exponentially. Consequently,

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

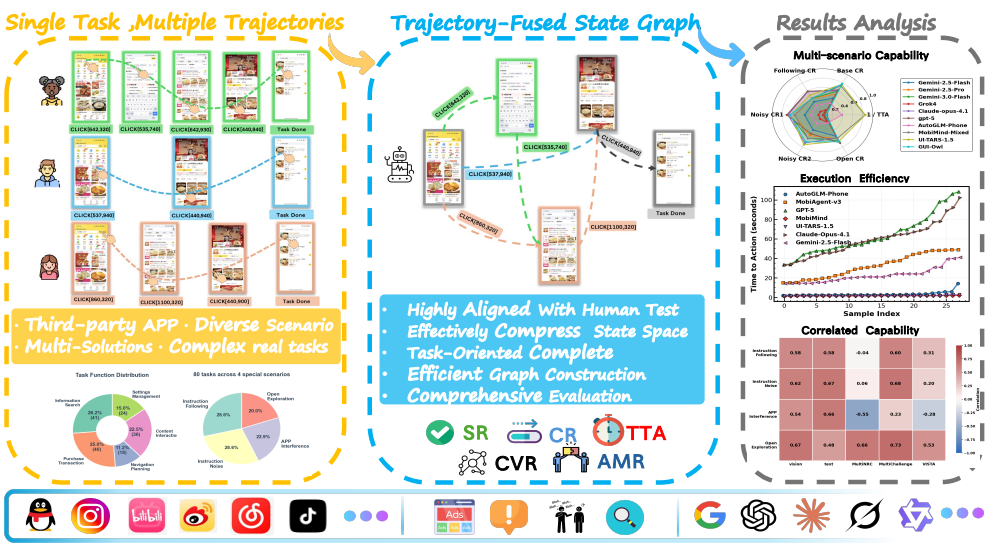

Figure 1: **The framework of *MobiGraph*.** It builds trajectory-fused state graphs from third-party Apps, covering 20 apps and 240 tasks with expanded metrics for multi-perspective model analysis.

existing mobile usage benchmarks still struggle to assess agent performance on real-world third-party applications comprehensively.

To address the dilemma that online benchmarks fail to cover the full range of applications, while offline benchmarks fail to capture all task completion trajectories. We observe that for each task, the set of independent UI objects relevant to the task on a single screen is finite (usually < 3), and different trajectories often converge to the same node after only a few steps. Based on this insight, we propose *MobiGraph*, which compresses the state space by constructing **Trajectory-Fused State Graphs**, enabling effective evaluation signals for arbitrary third-party applications while preserving complete state coverage, as shown in Fig.1.

We efficiently collect human operation trajectories from real-world task scenarios using front-end tools, including interface screenshots, actions, and annotation information,as shown in Appendix A. By merging nodes with identical annotation information and sharing their action transitions, we consolidate multiple real trajectories into a state transition graph capable of simulating realistic tasks. This construction effectively models real-world interaction environments while compressing state complexity, enabling a comprehensive evaluation of mobile agents' practical capabilities.

MobiGraph covers 20 widely used mobile applications and includes 240 real-world tasks designed to comprehensively evaluate the performance of current mainstream general-purpose models and GUI-specialized models on practical tasks(Ye et al., 2025a; Qin et al., 2025; Wang et al., 2025; Zhang et al., 2025). We abstract the agent's task execution process as state transitions on a graph, which enables more precise evaluation(Bu et al., 2025). We establish new metrics, such as coverage rate and completion rate, to comprehensively measure model capabilities. In addition, we design specialized scenarios to assess specific model capabilities, including instruction-following, noise interference, etc.[1]

Through the aforementioned definitions and scenario design, we show that the metrics derived from the graph structure exhibit strong interpretability. On our benchmark, UI-TARS and GUI-OWL achieve success rates of 60.4% and 55.7%, respectively, with single-action execution times of 1.75 s and 20.02 s. Compared to other benchmarks, our evaluation results align more closely with human judgment. Moreover, *MobiGraph* has been deployed to evaluate the capabilities of GUI agent models operating on over one million smartphones, with thousands of user feedback records collected to further refine the evaluation framework.

In summary, our main contributions are as follows:

[1]We prioritize quality over quantity: each task includes multiple valid completion trajectories (often 10+), yielding rich per-task evaluation signal. The 20 apps span diverse daily-use categories (social, entertainment,knowledge, productivity) selected by popularity ranking.

- **An efficient multi-trajectory fusion-based graph construction algorithm**. Our algorithm effectively compresses the state space size, reduces complexity, and ensures generalization to real-world tasks.
- **Diverse evaluation metrics and test scenarios**. Our method evaluates multiple dimensions, including time efficiency and coverage, and designs specialized scenarios, such as instruction following and noise interference.
- **Effective guidance for the future development of mobile agents.** We conduct comparative evaluations between general-purpose models and specialized models on the evaluation set, along with an attribution analysis of their performance across different scenarios.

2 Related Work

This section review existing evaluation frameworks, which can be broadly categorized into two types: offline evaluation based on pre-collected human interaction trajectories and online evaluation based on system-level emulators.

Current offline evaluation methods perform step-by-step evaluation by comparing the predicted actions against reference action trajectories. AITW(Rawles et al., 2023) introduced large-scale data for training and evaluation. AITZ(Zhang et al., 2024c) refined AITW's data, resulting in a more concise dataset. ANDROIDCONTROL(Leung et al., 2025) and AMEX(Chai et al., 2025a) focused on single-step action accuracy via element or coordinate matching. Mobile-Bench-v2(Xu et al., 2025a) covers a wide range of real-world tasks by collecting multiple trajectories and assessing them individually, which fails to support interaction. ColorBench(Song et al., 2025) constructs the complete state transition space of an application, leading to an explosion of the state space. Although existing offline evaluation methods(Lee et al., 2024; Chai et al., 2025b) can cover arbitrary scenarios and conveniently provide valid evaluation signals, they still fail to effectively encompass diverse dynamic interactions.

Several researchers have proposed online evaluation systems that enable interaction with agents via Android emulators and extract evaluation signals from system-level resources state. Mobile-Env(Zhang et al., 2024b) covers only 74 tasks, which limits its diversity. AndroidArena(Xing et al., 2024) and AndroidWorld(Rawles et al., 2024) expand the range of tasks; they are confined to built-in system applications whose designs often differ significantly from mainstream apps. AndroidLab (Xu et al., 2025b) is similarly constrained by its reliance on specific applications. Other works, such as SPA-Bench(Chen et al., 2024), AndroidDaily(Lee et al., 2025), MobileWorld(Kong et al., 2025),and OmniBench(Bu et al., 2025) still suffer from several limitations, including unreleased data, unstable environments or highly limited evaluation signals.

To address the dilemma that online benchmarks fail to cover the full range of applications, while offline benchmarks fail to capture all task completion trajectories, we aim to propose an evaluation framework that ensures coverage of diverse interactions while encompassing arbitrary applications and achieving high alignment with human evaluation.

Table 1: **Comparison of different datasets and environments for benchmarking Mobile GUI agents.** Column definitions: #Task. (number of tasks), #Apps (number of applications), Interactive (Support environmental interaction), Real App (Including any third-party apps), No Dependencies (Does not need additional software dependencies, such as an Android emulator), Multi-Solution (supports solving tasks in multiple ways), Evaluation (Evaluation Strategy)

Benchmark	#APP	#Task	Interactive	Real App	No Dependencies	Multi-Solution	Evaluation
Mobile-Bench-v2(Xu et al., 2025a)	49	12,856	✗	✓	✓	✓	Trajectory-based
AndroidControl(Leung et al., 2025)	-	14,548	✗	✓	✓	✗	Trajectory-based
MobileAgentBench(Wang et al., 2024)	100	10	✓	✗	✗	✓	Result-based
SPA-Bench (Chen et al., 2024)	66	340	✓	✓	✗	✓	Result-based
AndroidLab(Xu et al., 2025b)	9	138	✓	✗	✗	✓	Result-based
AndroidWorld(Rawles et al., 2024)	20	116	✓	✗	✗	✓	Result-based
MobiGraph	20	240	✓	✓	✓	✓	Graph-based

3 Formulation

We provide a formal description of how a Mobile agent accomplishes a given task. Let $\mathbf{G}$ denote the set of all tasks. Although the interaction space of a full mobile application is approximately

unbounded, for a specific task, if we focus on non-redundant correct completion paths and fix the task content to ensure reproducible evaluation, the resulting state-transition space is finite, as analyzed in §4.1. For each task $g \in \mathcal{G}$, we model the task relevant interaction space induced by correct task-completion trajectories as an observable state machine.

$$\mathcal{M}_{\mathbf{g}} = (\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{T}, \mathcal{R}), \tag{1}$$

where $\mathcal{S}$ is a finite set of states and $\mathcal{A}$ is a finite set of actions.[2] Each action $a \in \mathcal{A}$ corresponds to a basic UI operation, such as clicking or text input. $\mathcal{O}$ denotes the observation space, where the observation $o \in \mathcal{O}$ is partitioned by the current state $s \in \mathcal{S}$. The evaluation interaction environment receives agent's action and updates state according to the state transition function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$,and $\mathcal{R}$ represents the environment reward function.

- The transition function T can induce a directed graph $G = (V, E)$, where $T(s, a) = s'$ determines the existence of edge $e = (s, s') \in E$. In evaluation construction, each action a is derived from trajectories that complete the specific task.

We define a Mobile agent as a decision-making entity equipped with internal reasoning and memory mechanisms:

$$\mathcal{AG} = (\mathbf{\Sigma}, \mathbf{\Pi}, \mathcal{H}), \tag{2}$$

where $\mathbf{\Sigma}$ denotes the reasoning space, updated according to $\sigma_t \sim \mathbf{\Sigma}(\cdot \mid h_{t-1}, o_t)$; $\mathbf{\Pi}$ denotes the decision space, from which actions are sampled via $a_t \sim \mathbf{\Pi}(\cdot \mid h_{t-1}, \sigma_t, o_t)$; and $\mathcal{H}$ denotes the memory space, which is updated as $h_t = h_{t-1} \cup \{o_t, \sigma_t, a_t\}$.

- In practice, many agents integrate the reasoning and decision spaces into a single module, and the memory space of some agents may not include all observations, reasoning states, and actions. We present them in this form for conceptual clarity.
- The evaluation framework also supports multi-agent workflows, as the interaction environment only considers action inputs and observation outputs, regardless of the agents' specific working form.

4 MOBIGRAPH

To accurately evaluate agent performance under diverse real-world scenarios, and enable in-depth attribution analysis of agent capabilities, we propose the *MobiGraph* framework. This section is organized into three parts, detailing our innovations in evaluation environment construction, metric design, and specialized scenario development, respectively.

4.1 TASK GRAPH CONSTRUCTION

Observations Constructing a transition graph by exhaustively searching interactions across the entire app leads to state-space explosion and incurs extremely high construction complexity and human labor costs. Through a statistical analysis of the top 30 mainstream apps,[3] we find that the number of independent interactive elements on an app's interface approximately follows a normal distribution with a mean of 55.8(see Fig.2). In contrast, for a single task, the number of task-relevant elements approximately follows a Gumbel extreme value distribution, with a mean of only 1.7. This indicates that task-oriented graph construction yields a state space complexity($\sim 1.7^d$, where d is action depth) that is far smaller than that of App-based approaches($\sim 55.8^d$). Moreover, we observe that most trajectories converge to states with similar transition structures after only a few steps.

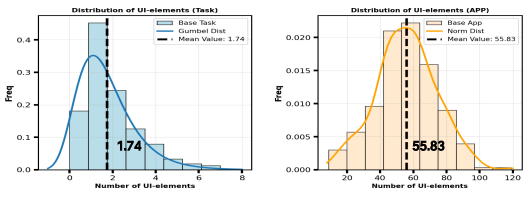

Figure 2: **Comparison** of the Distribution of Interactive Element Counts Across **Task-based** and **APP-based** Modeling Approaches

Correctness and completeness. Considering the requirements of industrial deployment and the characteristics of real-world applications, we aim to construct a **Task-Oriented Complete State-**

[2]The full action space are provided in Appendix B.

[3]Static details in AppendixD

Transition Graph (TCSG), which includes all correct, task-relevant state transitions while excluding irrelevant and redundant states. In real-world deployments, task success rates are typically required to exceed 95%, and triggering incorrect, task-irrelevant transitions is undesirable, as additional action paths can reduce both execution efficiency and success rates. Accordingly, we focus on whether an agent can sample correct actions to complete the task. For incorrect actions, we handle them by keeping the interface unchanged, transitioning to a blank screen, or presenting task-specific UI prompt interfaces.

Trajectory fusion. We observe that different trajectories often converge to states with highly similar transition structures after only a few steps.[4] This indicates the existence of many reusable state transitions, and such reuse can further reduce the complexity of the state space while increasing the diversity of task-completion paths. For a single task, we collect all directly completed trajectories and assign consistent labels to states that share identical transition structures across these trajectories. States with the same labels are then merged to share transition relations, thereby enabling **TCSG** construction, as shown in Fig.3. For arbitrary Apps and cross-Apps tasks, we can likewise construct the graph by collecting and merging trajectories and connecting multiple graphs.

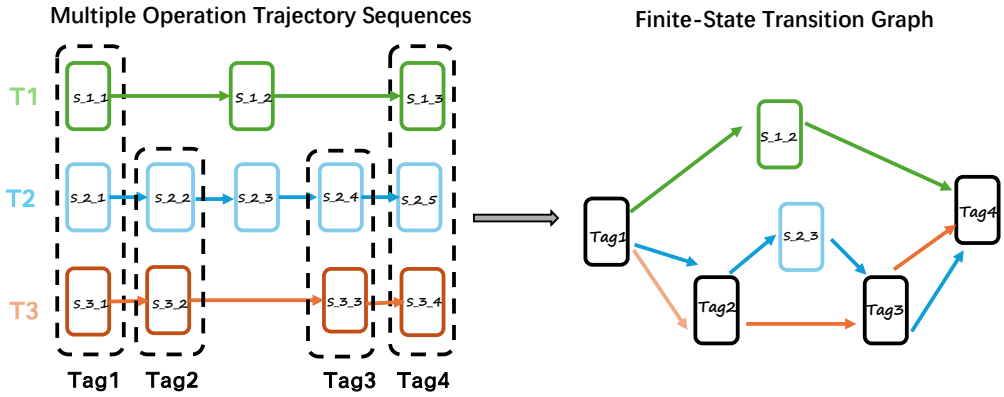

Figure 3: **Algorithm workflow illustration:** For states with identical labels, we merge them and allow the merged state to share their transition conditions, thereby compressing the state complexity.

Efficient graph construction. The construction of the TCSG can be mainly divided into two stages1. First, we traverse the trajectories, initialize chain-structured state transitions, and assign labels to states. If labels are provided during data collection, the labeling step can be omitted. Then, we expand transition diversity by merging the transition spaces of states with the same labels, thereby inducing the state-transition graph. Let N, L, S, and A denote the number of trajectories, average trajectory length, number of unique states, and average out-degree per state, respectively. The construction scans trajectories once and then fuses transitions by semantic labels, yielding a time complexity of $O(NL + SA)$ and a space complexity of $O(S + SA)$. Since A is small in practice(mean=1.74), the construction is nearly linear.[5]

4.2 Comprehensive Evaluation

Based on the aforementioned construction algorithm, a TCSG encompassing all reasonable behavioral trajectories can be rapidly generated for a given task. As the graph is constructed from a global perspective, it enables a more comprehensive evaluation of model behavior. In contrast, most previous evaluation methods, constrained by paradigms that rely solely on outcome signals or single-trajectory analysis, primarily focus on task success rate (SR)(Xie et al., 2024). This metric considers only task outcomes and ignores the execution process, which limits its ability to fully assess the model's capabilities.

To more accurately evaluate task completion on the TCSG, we introduce four complementary metrics. Completion Rate (CR) measures the agent's progress toward the goal state, while Coverage Rate

[4]This similarity mainly appears at the action and successor-state levels: different interfaces may provide equivalent task-relevant controls, such as "Search" or "Place Order" and lead to similar subsequent states. It can be manually identified during data collection or detected through XML-structure comparison and icon matching.

[5]Total runtime generally is < 1s per task(one-time offline cost).

Algorithm 1 Construction of the Trajectory-Fused State Graph

Require: Task trajectories $\mathcal{T} = \{\tau_1, \ldots, \tau_N\}$
Ensure: Trajectory-fused state transition graph $\mathcal{G}$
Initialize state set $\mathcal{S}$, transition space $\mathcal{E}$, and label clusters $\mathcal{C}$
for each trajectory $\tau \in \mathcal{T}$ **do**
    Parse τ into an ordered sequence $(s_0, a_0, s_1, \ldots, a_{L-1}, s_L)$
    Assign labels to states using provided tags; mark s_0 as `START` and s_L as `DONE` if needed
    for $i = 0$ to $L - 1$ **do**
        Add the observed transition (s_i, a_i, s_{i+1}) to $\mathcal{E}(s_i)$
    end for
    Add states $\{s_0, \ldots, s_L\}$ to $\mathcal{S}$
end for
for each labeled cluster $c \in \mathcal{C}$ **do**
    Fuse outgoing transition spaces by union: $\mathcal{E}_c \leftarrow \bigcup_{s \in c} \mathcal{E}(s)$
    for each state $s \in c$ **do**
        Share the fused transition space: $\mathcal{E}(s) \leftarrow \mathcal{E}(s) \cup \mathcal{E}_c$
    end for
end for
return $\mathcal{G} = (\mathcal{S}, \mathcal{E})$

(CVR) quantifies the proportion of explored states, reflecting the agent's exploratory capability. Action Match Rate (AMR) assesses the alignment between the agent's executed actions and the instructed actions, serving as an indicator of instruction compliance and safety. Time to Action (TTA) evaluates execution efficiency. Together, these metrics form a comprehensive evaluation framework spanning four dimensions: task progression, state exploration, instruction compliance, and execution efficiency. A detailed comparison and summary of these metrics is provided in Table2.

4.3 DIVERSE SCENARIO

In this section, we introduce a variety of evaluation scenarios constructed based on task-state transition graphs.(see Fig.4) These scenarios are designed to assess model performance from multiple perspectives, including interference resistance, error recovery, and exploratory capability.

Base scenario: Based on the construction algorithm, we constructed 160 tasks spanning 20 mainstream Apps. To ensure that these tasks reflect real-world usage scenarios, we selected tasks from multiple domains through a small-scale voting, including common activities such as information search, product purchasing, and social chatting. For tasks involving dependencies, the state graphs can be combined to support multi-task evaluation. This scenario enables a comprehensive evaluation of agents' capabilities in handling both everyday tasks and complex multi-task settings.

Special scenario 1 Instruction Following: The agent should be able to adhere to human instructions. In this scenario, specific execution steps are provided in instructions. The action matching rate(AMR) is used to evaluate the agent's ability to follow instructions and its understanding and memory of complex, lengthy instructions. This scenario can also assess the agent's safety performance.

Special scenario 2 Instruction Noise Interference: We introduce noise into task instructions through common typo substitutions, emoji injection, and multilingual mixing, aiming to evaluate the model's resistance to interference and the robustness of its instruction understanding.

Special scenario 3 APP Interference: In real-world applications, software often encounters various interferences, such as pop-up notifications from other apps, ad pushes, unintended touches, or interface jumps caused by system anomalies. Therefore, we separately collect instances containing these interferences to evaluate the model's ability to handle specific APP disturbances.

Special scenario 4 Open Exploration: A knowledgeable agent should possess exploration and error recovery capabilities. To this end, we constructed a set of special transition graphs allowing us to observe whether the model can detect issues within incorrect paths and reasonably backtrack to accomplish the designated task.

5 EXPERIMENTS

This section analyzes the evaluation results of several mainstream general and specialized models on our evaluation benchmark(see Table3).

Table 2: **Evaluation metrics defined on the state-transition graph.**

Metric	Formula	Notation Explanation	Description
Success Rate (SR)	$\frac{1}{N}\sum_{i=1}^{N}\text{suc}(i)$	N: number of tasks; $\text{suc}(i)=1$ if g_i is completed, else 0.	Indicates whether tasks are finally completed.
Completion Rate (CR)	$\max\left\{\frac{d(s_{\text{start}},s_{\text{visited}})}{d(s_{\text{start}},s_{\text{goal}})}\right\}$	s_{start}, s_{goal}: initial and goal states; $d(\cdot,\cdot)$: shortest-path distance.	Measures the maximum progress achieved toward the goal state.
Coverage Rate (CVR)	$\frac{\sum_i \lvert\mathcal{S}_{\text{visited}}\rvert_i}{\sum_i \lvert\mathcal{S}\rvert_i}$	$\lvert\mathcal{S}_{\text{visited}}\rvert_i$: visited states in task i; $\lvert\mathcal{S}\rvert_i$: total states.	Evaluates exploration by the proportion of states observed.
Action Match Rate (AMR)	$\frac{1}{N}\sum_i \frac{\sum_j am[i][j]}{\lvert IF_i\rvert}$	$\lvert IF_i\rvert$: number of instructions; $am[i][j]=1$ if instruction j is satisfied.	Measures instruction-following ability and safety.
Time to Action (TTA)	$\frac{1}{N}\sum_i \frac{\sum_j t_{ij}}{\lvert\tau_i\rvert}$	$\lvert\tau_i\rvert$: number of interactions; t_{ij}: time to generate action j.	Captures execution efficiency via average action latency.

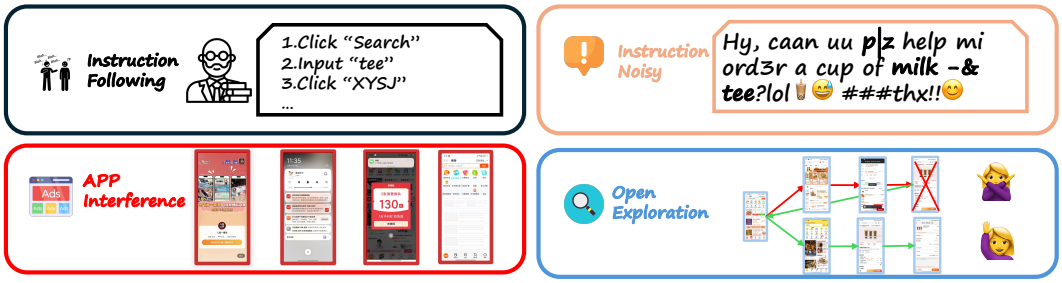

Figure 4: **Special scenarios** including instruction following, instruction interference, application interference, and open exploration (observe whether the model can recover from erroneous paths).

5.1 Experimental Setup

We evaluated a series of advanced models, including general-purpose models and GUI-specific models: OpenAI GPT-5.2, Claude-opus-4.1, Grok4, Gemini-2.5-flash, Gemini-2.5-Pro, Gemini-3-flash(Comanici et al., 2025), UI-TARS-1.5(Qin et al., 2025), AutoGLM-Phone(Dec.8, 2025)(Liu et al., 2024), GUI-Owl(Ye et al., 2025b), and MobiMind(Zhang et al., 2025; Liu et al., 2025; Feng et al., 2025). To ensure a consistent comparison, we adopted the same execution framework for general models, incorporating OmniParser-v2(Lu et al., 2024) to annotate localize actionable UI elements, reducing the complexity of coordinate-based interface operations and ensuring a consistent localization standard[6]. For GUI-specific agents, we employed their official execution frameworks. The maximum number of interaction steps allowed was set to 50. Additionally, the screen resolutions of the evaluation data were primarily 1080×2400 and 1200×2670.

For specialized models, we standardized deployment and resource allocation, conducting inference deployment on NVIDIA A100-SXM4-80GB GPU using the vLLM(Kwon et al., 2023) inference framework. For general models, we used the same prompt framework and issued requests through the OpenRouter API under the same network environment, with the temperature parameter consistently set to 0.

5.2 Main Results

Task completion. In the base scenario and the APP interference scenario, GUI agents achieve superior performance. Specifically, GUI-Owl attains the highest completion rate (CR) of 77.9 in the base scenario, while UI-TARS obtains the highest CRs of 40.5 and 74.5 in the multi-task and APP-interference scenarios, respectively. However, GUI models do not exhibit a comprehensive advantage over general models. In instruction-following, instruction-noise interference, and open exploration scenarios, general-purpose models (e.g., OpenAI GPT-5) demonstrate stronger capabilities.

Execution efficiency. We sample tasks of varying complexity and analyze agent execution efficiency, as shown in Fig.12 in AppendixJ. Single-action latency is mainly affected by network latency, model size, decoding length, and the number of model calls per action. GUI-OWL is slower because each

[6]Detailed implementation can be found in the AppendixF

Table 3: **Main results on *MobiGraph*.** Includes the results of general and GUI models across different scenarios. CR1 and CR2 denote the results under instruction-noise and app-interference settings, respectively. MTR refers to the completion rate for multi-tasks, and HFS denotes the average user feedback score.

			Base Scenario				Following		Noisy Scenario		Open Exploration		
Model	Size(B)	TTA(s)↓	SR(%)↑	CR(%)↑	CVR(%)	MTR(%)↑	AMR(%)↑	SR(%)↑	CR1(%)↑	CR2(%)↑	CR(%)↑	CVR(%)	HFS↑
General Model													
Gemini-2.5-Flash	-	17.03	30.7	41.7	17.4	17.2	35.5	33.3	35.5	37.7	53.2	21.1	2.4
Gemini-2.5-Pro	-	30.12	38.7	52.1	26.9	25.5	55.3	41.6	56.4	51.7	59.7	28.1	2.5
Gemini-3.0-Flash	-	33.31	40.2	59.7	30.2	24.8	56.4	41.6	64.7	58.4	65	34.3	2.7
Grok4	-	142.21	8.3	12.3	8.41	8.4	4.1	8.3	8.1	10.0	17.8	10.2	1.7
Claude-opus-4.1	-	46.68	43.6	60.3	34.6	25.6	62.2	58.3	68.6	70.1	57.9	37.7	3.1
GPT-5.2	-	55.49	49.7	62.7	35.3	26.6	**71.4**	**59.3**	**71.1**	46.7	**70.0**	39.3	3.2
GUI Model													
AutoGLM-Phone	9B	3.85	16.7	30.2	10.6	11.3	30.5	24.2	24.2	22.5	20.1	15.1	3.0
MobiMind-Mixed	4B	**1.03**	59.6	72.6	31.9	29.7	59.9	42.3	55.1	58.3	58.6	35.1	3.8
UI-TARS-1.5	8B†	1.75	**60.4**	76.3	34.4	**40.5**	63.1	50.0	63.1	**74.5**	65.7	37.1	**4.2**
GUI-Owl	8B†	20.02	55.7	**77.9**	37.5	32.1	65.4	41.6	64.7	67.4	67.1	38.1	3.9

†Model size includes the vision encoder (ViT)

action requires multiple model calls, whereas MobiMind and UI-TARS are more efficient due to lightweight architectures, single-call execution, and shorter outputs. In contrast, general models show higher latency due to larger model sizes, network overhead, and task-dependent decoding lengths.[7].

Performance fluctuations. We observe that, across multiple sampling attempts for the same task, GUI models tend to follow relatively consistent execution paths, whereas general models exhibit more diverse paths but show higher variance in completion outcomes, as shown in AppendixG.

Trajectory coverage. Across multiple model evaluations, coverage and completion rates consistently show a positive correlation. Models with higher coverage also demonstrate better performance in open exploration scenarios.

Resolution scaling. We analyze the impact of input image resolution on model performance. For images with a resolution of 1080×2400, the model's task completion rate drops significantly when the resolution is scaled below 0.2×, as shown in AppendixH.

5.3 In-Depth Analysis

Noise robustness. We observe that general models exhibit stronger robustness. As detailed in AppendixI, Table5, one possible explanation is that specific interference may increase the model's attention to the task-relevant parts of long-context instructions.

Lightweight deployment. General models show strong generalization ability and outperform GUI models in open exploration tasks. However, lightweight specialized models support on-device deployment, and their stable accuracy in completing a large number of tasks further facilitates practical deployment and real-world application.

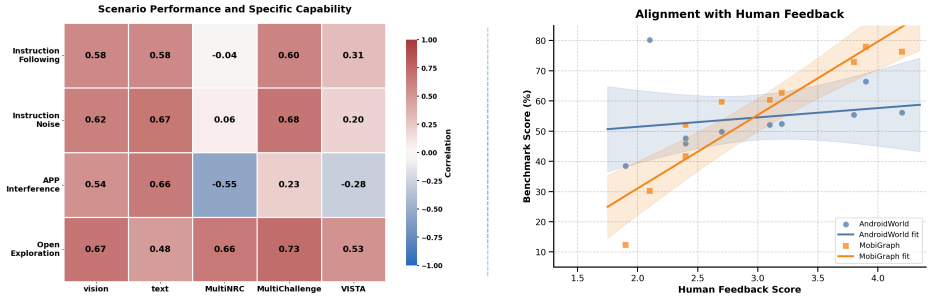

Figure 5: **Correlation Analysis**. **Left**: analysis of scenario-specific evaluation capabilities. **Right**: comparison of alignment with human judgments.

Attribution analysis. Across different scenarios, we correlate model scores with established benchmarks[8] to identify the capabilities reflected by each scenario(see Fig.5 Left). We find that all scenarios are strongly associated with visual input understanding and cross-cultural contextual comprehension. Open exploration and instruction following show particularly high correlations with multi-turn dialogue ability, while app interference exhibits a slight negative correlation with logical reasoning. In addition, open exploration is strongly correlated with causal reasoning ability.

[7]All experiments followed the settings in the §5.1

[8]Text&VisionArena(`https://lmarena.ai`), MultiNRC(Fabbri et al., 2025), MultiChallenge(He et al., 2024), and VISTA(Zheng et al., 2023)(`https://scale.com/leaderboard`)

Human alignment. We correlate scores from different evaluation frameworks with user feedback[9], showing that *MobiGraph* achieves stronger alignment with real-world usage than AndroidWorld(see Fig.5 right).

5.4 SPECIAL ISSUES

Thinking and observation become desynchronized. GUI models sometimes fail to trigger the expected page transition due to erroneous actions, yet assume during reasoning that the state has been updated and continue with subsequent operations.

Insufficient generalization of reasoning. We observe that some models follow fixed reasoning patterns. For unconventional tasks, they often default to keyword search and overlook information already available on the page.

Interface comprehension deviation. Many models can be misled by interface cues. For instance, in search scenarios, light-colored placeholder text displayed in search boxes are often mistaken for actual user input. As a result, models may attempt to clear these suggestions before proceeding, leading to unnecessary or even erroneous actions.

5.5 TRADE-OFFS

Balancing agent execution and system boundaries. Granting the agent greater autonomy over certain system operations can expand its action space and enhance overall capability. For instance, combining clicking and text-input actions can improve action efficiency and prevent the issue of repeated input field activation in search scenarios. However, such delegation must be carefully managed to ensure system security and stability.

Balancing generalization and reliability. Strengthening an agent's generalization ability leads to more diverse execution trajectories, which enriches behavioral flexibility. Yet, this often comes at the cost of reduced determinism and reliability in task completion.

6 DISCUSSION

Balancing Agent Execution and System Boundaries. Granting the agent greater autonomy over certain system operations can expand its action space and enhance overall capability. For instance, combining clicking and text-input actions can prevent the issue of repeated input field activation in search scenarios. However, such delegation must be carefully managed to ensure system security and stability.

Balancing Generalization and Reliability. Strengthening an agent's generalization ability leads to more diverse execution trajectories, which enriches behavioral flexibility. Yet, this often comes at the cost of reduced determinism and reliability in task completion. In real-world deployment, a deliberate and nuanced balance must be struck to maintain both adaptability and consistent performance.

7 CONCLUSION

In summary, we introduce *MobiGraph*, a mobile-use benchmark built on arbitrary third-party APPs. By constructing state transition graphs, *MobiGraph* effectively compresses the state space of multiple task-completion trajectories. Compared with existing benchmarks, *MobiGraph* more faithfully reflects the practical capabilities of agents. We equip the benchmark with a diverse set of evaluation metrics and specialized evaluation scenarios to enable in-depth analysis of the model's specific capabilities. We analyze the performance of current mainstream models and identify the challenges they face. Our benchmark provides a critical solution for gaining a deeper understanding of the real-world capabilities of mobile agents and facilitating their deployment in practical scenarios.

Limitations. The primary limitations are that the evaluation not fully cover different linguistic and cultural contexts. Future work will focus on more broader evaluation coverage.

Impact statement. This work facilitates the real-world deployment of mobile agents and has been adopted by industry partners. The trajectory-fusion based evaluation approach can also be applied to other types of agents, helping improve their practicality in completing various GUI tasks.

[9]We replaced different models and collected feedback from thousands of internal test users using a five-point rating scale.

AI USE STATEMENT

In this work, we used generative AI tools for polishing the manuscript for readability. All trajectories in the paper are genuine records of human operations, including screenshots and action sequences. We have not used generative AI tools for designing experiments or implementing methods, and the rest of the required disclosure tasks (such as formulating theoretical claims and proofs) are not applicable to this work. Additionally, we used generative AI tools for polishing the manuscript for readability. We have reviewed all AI-assisted work. We checked the LLM-polished text against the original manuscript to ensure that no technical content, claims, or meaning were altered. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We have included the complete code and data in the abstract, and we guarantee the reproducibility of the current experimental results. For the experimental details, please refer to Section 5 of the paper.

REFERENCES

Wendong Bu, Yang Wu, Qifan Yu, Minghe Gao, Bingchen Miao, Zhenkui Zhang, Kaihang Pan, Yunfei Li, Mengze Li, Wei Ji, et al. What limits virtual agent application? omnibench: A scalable multi-dimensional benchmark for essential virtual agent capabilities. *arXiv preprint arXiv:2506.08933*, 2025.

Yuxiang Chai, Siyuan Huang, Yazhe Niu, Han Xiao, Liang Liu, Guozhi Wang, Dingyu Zhang, Shuai Ren, and Hongsheng Li. AMEX: Android multi-annotation expo dataset for mobile GUI agents. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 2138–2156, Vienna, Austria, July 2025a. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.110. URL https://aclanthology.org/2025.findings-acl.110/.

Yuxiang Chai, Hanhao Li, Jiayu Zhang, Liang Liu, Guangyi Liu, Guozhi Wang, Shuai Ren, Siyuan Huang, and Hongsheng Li. A3: Android agent arena for mobile gui agents. *arXiv preprint arXiv:2501.01149*, 2025b.

Jingxuan Chen, Derek Yuen, Bin Xie, Yuhao Yang, Gongwei Chen, Zhihao Wu, Li Yixing, Xurui Zhou, Weiwen Liu, Shuai Wang, et al. Spa-bench: A comprehensive benchmark for smartphone agent evaluation. In *NeurIPS 2024 Workshop on Open-World Agents*, 2024.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Alexander R Fabbri, Diego Mares, Jorge Flores, Meher Mankikar, Ernesto Hernandez, Dean Lee, Bing Liu, and Chen Xing. Multinrc: A challenging and native multilingual reasoning evaluation benchmark for llms. *arXiv preprint arXiv:2507.17476*, 2025.

Erhu Feng, Wenbo Zhou, Zibin Liu, Le Chen, Yunpeng Dong, Cheng Zhang, Yisheng Zhao, Dong Du, Zhichao Hua, Yubin Xia, et al. Get experience from practice: Llm agents with record & replay. *arXiv preprint arXiv:2505.17716*, 2025.

Yun He, Di Jin, Chaoqi Wang, Chloe Bi, Karishma Mandyam, Hejia Zhang, Chen Zhu, Ning Li, Tengyu Xu, Hongjiang Lv, et al. Multi-if: Benchmarking llms on multi-turn and multilingual instructions following. *arXiv preprint arXiv:2410.15553*, 2024.

Quyu Kong, Xu Zhang, Zhenyu Yang, Nolan Gao, Chen Liu, Panrong Tong, Chenglin Cai, Hanzhang Zhou, Jianan Zhang, Liangyu Chen, et al. Mobileworld: Benchmarking autonomous mobile agents in agent-user interactive, and mcp-augmented environments. *arXiv preprint arXiv:2512.19432*, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Juyong Lee, Taywon Min, Minyong An, Dongyoon Hahm, Haeone Lee, Changyeon Kim, and Kimin Lee. Benchmarking mobile device control agents across diverse configurations. *arXiv preprint arXiv:2404.16660*, 2024.

Juyong Lee, Taywon Min, Minyong An, Dongyoon Hahm, Haeone Lee, Changyeon Kim, and Kimin Lee. Benchmarking mobile device control agents across diverse configurations, 2025. URL https://arxiv.org/abs/2404.16660.

Ho Fai Leung, Xiaoyan Xi, and Fei Zuo. Androidcontrol-curated: Revealing the true potential of gui agents through benchmark purification. *arXiv preprint arXiv:2510.18488*, 2025.

Xiao Liu, Bo Qin, Dongzhu Liang, Guang Dong, Hanyu Lai, Hanchen Zhang, Hanlin Zhao, Iat Long Iong, Jiadai Sun, Jiaqi Wang, et al. Autoglm: Autonomous foundation agents for guis. *arXiv preprint arXiv:2411.00820*, 2024.

Zibin Liu, Cheng Zhang, Xi Zhao, Yunfei Feng, Bingyu Bai, Dahu Feng, Erhu Feng, Yubin Xia, and Haibo Chen. Beyond training: Enabling self-evolution of agents with mobimem. *arXiv preprint arXiv:2512.15784*, 2025.

Yadong Lu, Jianwei Yang, Yelong Shen, and Ahmed Awadallah. Omniparser for pure vision based gui agent, 2024. URL https://arxiv.org/abs/2408.00203.

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

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, et al. Ui-tars: Pioneering automated gui interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025.

Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. Androidinthewild: A large-scale dataset for android device control. *Advances in Neural Information Processing Systems*, 36:59708–59728, 2023.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, et al. Androidworld: A dynamic benchmarking environment for autonomous agents. *arXiv preprint arXiv:2405.14573*, 2024.

Yuanyi Song, Heyuan Huang, Qiqiang Lin, Yin Zhao, Xiangmou Qu, Jun Wang, Xingyu Lou, Weiwen Liu, Zhuosheng Zhang, Yong Yu, et al. Colorbench: Benchmarking mobile agents with graph-structured framework for complex long-horizon tasks. *arXiv preprint arXiv:2510.14621*, 2025.

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

Haoming Wang, Haoyang Zou, Huatong Song, Jiazhan Feng, Junjie Fang, Junting Lu, Longxiang Liu, Qinyu Luo, Shihao Liang, Shijue Huang, et al. Ui-tars-2 technical report: Advancing gui agent with multi-turn reinforcement learning. *arXiv preprint arXiv:2509.02544*, 2025.

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

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

Mingzhe Xing, Rongkai Zhang, Hui Xue, Qi Chen, Fan Yang, and Zhen Xiao. 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*, pp. 6061–6072, 2024.

Weikai Xu, Zhizheng Jiang, Yuxuan Liu, Pengzhi Gao, Wei Liu, Jian Luan, Yuanchun Li, Yunxin Liu, Bin Wang, and Bo An. Mobile-bench-v2: A more realistic and comprehensive benchmark for vlm-based mobile agents. *arXiv preprint arXiv:2505.11891*, 2025a.

Yifan Xu, Xiao Liu, Xueqiao Sun, Siyi Cheng, Hao Yu, Hanyu Lai, Shudan Zhang, Dan Zhang, Jie Tang, and Yuxiao Dong. Androidlab: Training and systematic benchmarking of android autonomous agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2144–2166, 2025b.

Jiabo Ye, Xi Zhang, Haiyang Xu, Haowei Liu, Junyang Wang, Zhaoqing Zhu, Ziwei Zheng, Feiyu Gao, Junjie Cao, Zhengxi Lu, Jitong Liao, Qi Zheng, Fei Huang, Jingren Zhou, and Ming Yan. Mobile-agent-v3: Fundamental agents for gui automation, 2025a. URL https://arxiv.org/abs/2508.15144.

Jiabo Ye, Xi Zhang, Haiyang Xu, Haowei Liu, Junyang Wang, Zhaoqing Zhu, Ziwei Zheng, Feiyu Gao, Junjie Cao, Zhengxi Lu, et al. Mobile-agent-v3: Fundamental agents for gui automation. *arXiv preprint arXiv:2508.15144*, 2025b.

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

Cheng Zhang, Erhu Feng, Xi Zhao, Yisheng Zhao, Wangbo Gong, Jiahui Sun, Dong Du, Zhichao Hua, Yubin Xia, and Haibo Chen. Mobiagent: A systematic framework for customizable mobile agents. *arXiv preprint arXiv:2509.00531*, 2025.

Danyang Zhang, Hongshen Xu, Zihan Zhao, Lu Chen, Ruisheng Cao, and Kai Yu. Mobile-env: an evaluation platform and benchmark for llm-gui interaction. *arXiv preprint arXiv:2305.08144*, 2024b.

Jiwen Zhang, Jihao Wu, Teng Yihua, Minghui Liao, Nuo Xu, Xiao Xiao, Zhongyu Wei, and Duyu Tang. Android in the zoo: Chain-of-action-thought for GUI agents. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 12016–12031, Miami, Florida, USA, November 2024c. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.702. URL https://aclanthology.org/2024.findings-emnlp.702/.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.

A Real-world Trajectory Collection

To ensure the collected trajectories accurately reflect real-world user behavior, we have developed a lightweight action recording tool for smartphones(see fig6). During data collection, annotators or agents interact with this tool, which captures and logs every user operation before forwarding it to the device. The tool renders bounding boxes for all interactive on-screen elements based on XML files that describe UI hierarchies. If a bounding box is missing due to incomplete XML data, OmniParser is employed to regenerate the corresponding bounding box. The resulting trajectories can subsequently be used for graph construction, evaluation, and other follow-up steps.

Within the graph construction algorithm, the labeling of observed pages can be performed concurrently. Following predefined labeling rules (which may be specified based on the transition structure of observed interfaces), multiple trajectories are merged into a unified state transition graph. Furthermore, to extend the state transition graph, it suffices to append new trajectories according to the same labeling principles, thereby ensuring strong extensibility.

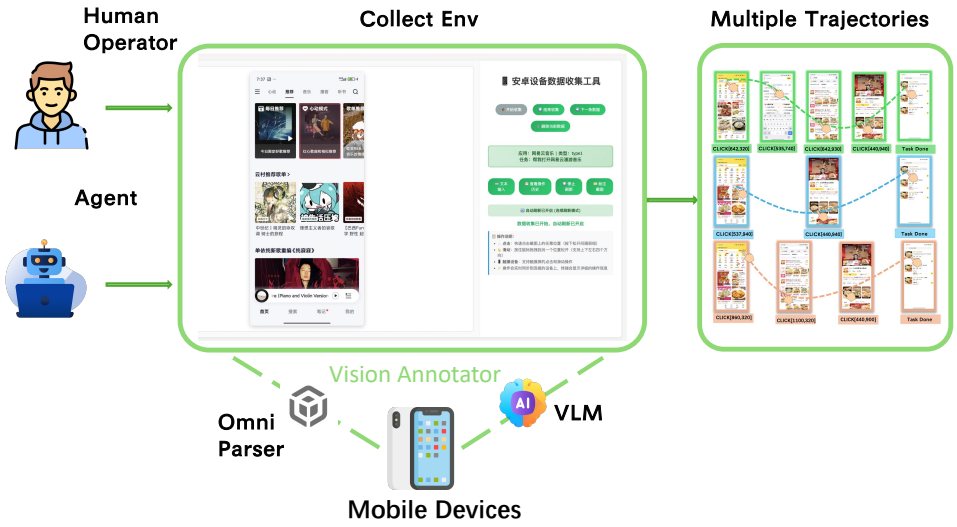

Figure 6: **Efficient trajectory collection tools**. A trajectory collection framework based on real devices and actual applications.

B Action Space

This section presents the action space supported by our environment, which includes operations such as click, swipe, wait, text input, back, and more.

Table 4: Action types and their parameters in the task execution framework.

Action	Parameter	Description
GUI Operation		
click	x,y	Click at the specified coordinates
input	string	Type text into the focused field
swipe	start_x,start_y,end_x,end_y	Swipe from start to end coordinates
wait	–	Wait for update
double_click	x,y	Double Click at the specified coordinates
long_press	x,y	Long press at the specified coordinates
System Operation		
back	–	Navigate to the previous screen
home	–	Navigate to the Home screen
Task Control Operation		
done	–	Task done
failed	–	Task failed

C Task Graph Construction Details

This section introduces the details of the construction. The task graph primarily consists of two elements: node construction and edge construction. For node construction, we provide the following data structure, which describes the attributes of a state, including observation information (State_Info), labeling properties (Tag), and state transition mappings(Transition_Space).

```python
@dataclass
class State:
    '''
        Task Transition Graph state
    '''
    State_Info #screenshots or other(e.g., text or XML)
    Tag # State tagging
    Transition_Space : { #Action-space-based interface transition hashing
        "click": {
            #Box[x1,y1,x2,y2] --> next state
        },
        "swipe": {
            #(Direction,Distance) --> next state
        },
        "input": {
            #Text --> next state
        },
        "wait": {
            #Duration --> next state
        },
        "back": {
            # --> prev state
        },
        "home": {
             # --> home state
        },
        "long Press":{
            #Box[x1,y1,x2,y2] --> next state
        }
        #extend ...    }
```

For edge construction, we first traverse the raw trajectories to assign state-specific labels and record all observed action transitions, thereby populating the transition space of each abstract state. The labels can be manually defined based on the structural distribution of UI elements on the interface or inferred by a model; the key requirement is that states sharing the same label can reuse a common transition space. Specifically, each raw observation is mapped to an abstract state via semantic labeling or hashing, and directed edges are added according to the corresponding (u, a, v) transitions. Subsequently, nodes with identical semantic labels are merged by taking the union of their transition spaces, aggregating historical actions across equivalent states to maximize action coverage while reducing state fragmentation.

D Statistics on the Number of Interface Actions

We employ OmniParserV2 to conduct a statistical analysis of icon elements corresponding to executable actions across various interface types in multiple mainstream applications. Duplicate elements are removed using non-maximum suppression, and element independence is ensured through intersection detection. The resulting statistics are presented in the upper part of Figure7. For multiple feasible

trajectories that accomplish the same task, we further compute the average number of navigation actions per interface, with the results shown in the lower part of Figure7.

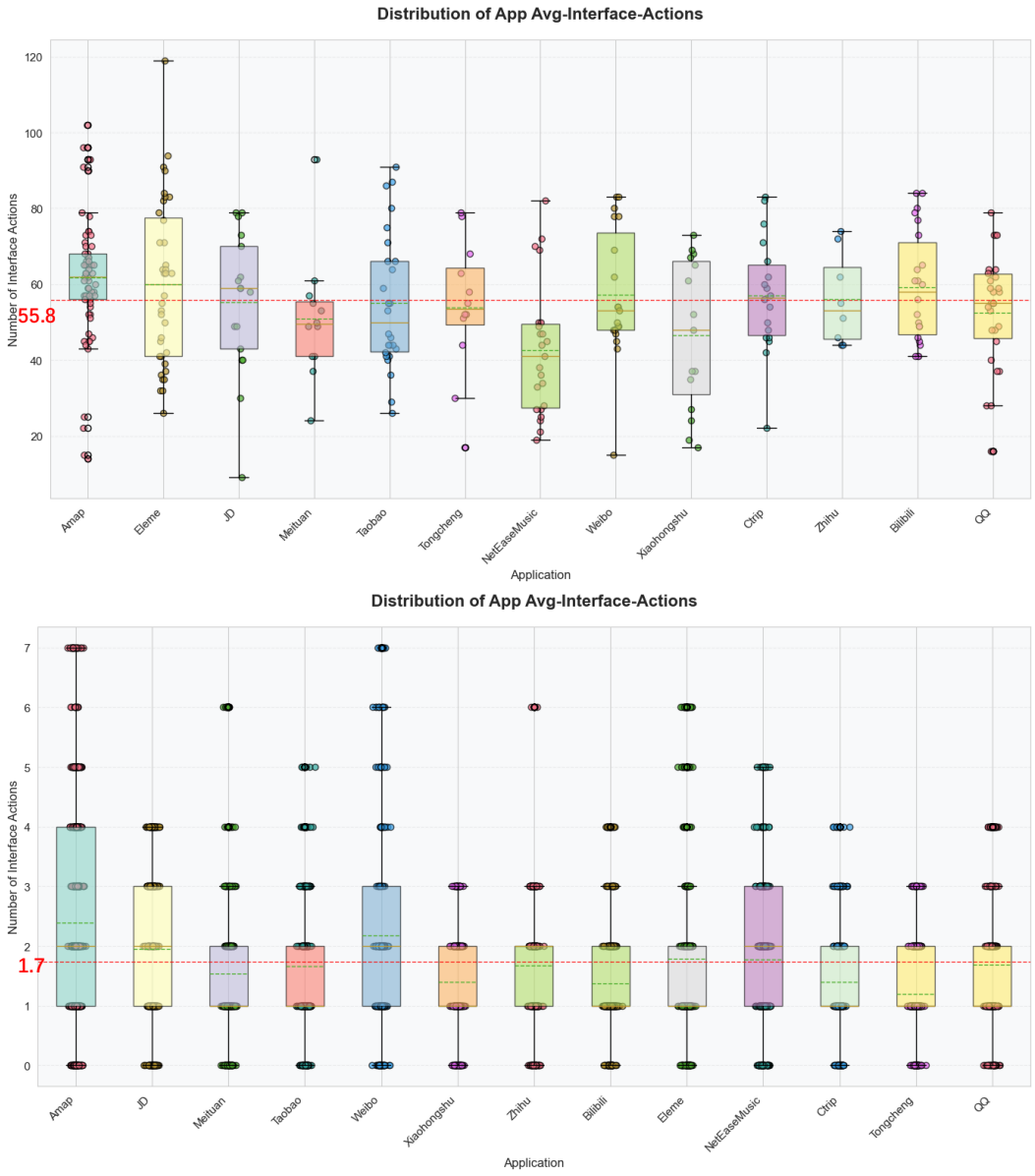

Figure 7: This figure illustrates the statistical distribution of transition action counts on individual interfaces across different mobile applications(Due to space limitations, we present only a subset of the statistical results).

E STATE TRANSITION GRAPH VISUALIZATION

Based on our graph construction algorithm, which merges multiple trajectories into a task state transition graph, we randomly selected four tasks for visualization. The visualization results are presented below.

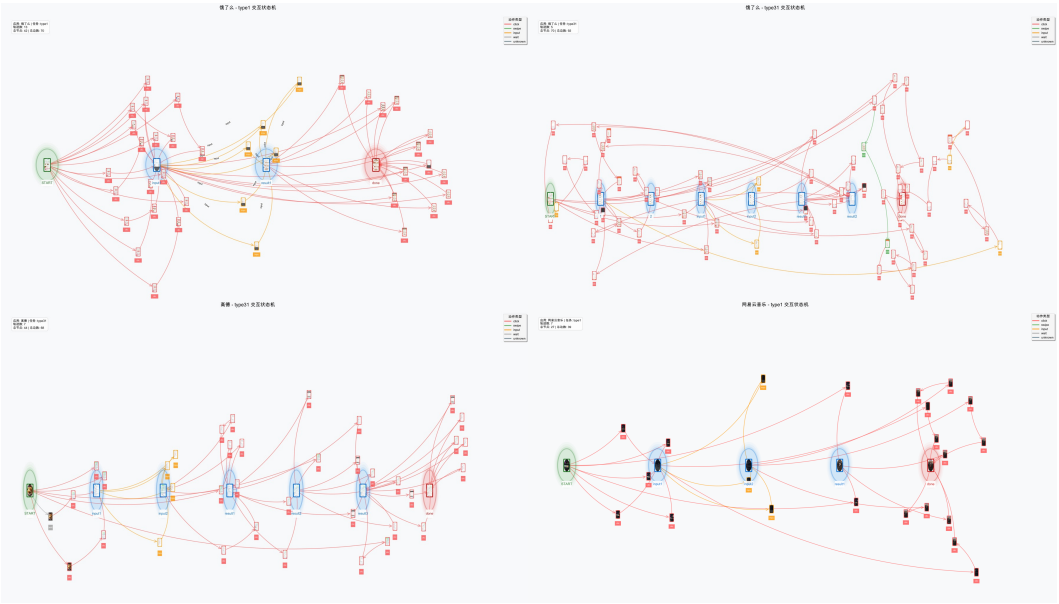

Figure 8: **Visualization Examples of Task State Transition Graphs**, including Different Tasks from Ele.me, AutoNavi, and NetEase Cloud Music Apps.

F Execution Details of General Models

The below constitutes the system prompt designed for task completion by general-purpose models. It defines a constrained action space, provides historical context, and includes both positive and negative examples for in-context learning. Additionally, a style constraint is imposed to require the model to output its reasoning process. Since general-purpose models struggle with generating precise coordinates, we have integrated an auxiliary enhancement tool. This tool utilizes an icon recognition model to overlay bounding boxes and numerical labels directly on the interface image, enabling the model to select from these annotated elements, thereby enhancing its task completion capability.

System Prompt

Role Definition You are a mobile phone operation AI assistant, tasked with helping the user complete the following task: "`{task_description}`".
Input Description I will provide you with:

1. **Action History**: A record of all previous operations.
2. **Screen Screenshot**: A complete screenshot of the current phone screen.
3. **{layer_count} Annotated Screenshots**: Annotation layers of clickable elements generated based on the screen screenshot. To avoid element overlap, all clickable elements are distributed across different layers for display. The union of elements across all layers represents the complete set of clickable elements.
   - Clickable elements are marked with red bounding boxes.
   - Each element's index is displayed inside the top-left corner of its red bounding box as a red-background white-digit number.

Action History `{history}`
Task Requirements Please carefully analyze the current screen state and action history, then determine the most appropriate next action.
Actions Available

1. **Click (click)**

- Parameters: `index` (integer, corresponding to the UI element index in the annotated screenshots)
- Parameters: `target_element` (string, describing the UI element to be clicked)
- **Critical**: You must carefully observe the annotated screenshots, find the red bounding box that best matches the description of the `target_element`, and use the number displayed inside the top-left corner of that box (red background, white digits) as the `index`.
- **Prevent Mis-selection**: The reasoning must explicitly explain why this specific red bounding box was chosen and not an adjacent one.

2. **Swipe (swipe)**
   - Parameters: `direction` (string, must be one of: UP, DOWN, LEFT, RIGHT)
   - **Critical**: Direction Clarification: UP means swiping the finger upward to scroll content upward, revealing content below; DOWN means swiping downward to scroll content downward, revealing content above; LEFT means swiping left to scroll content left; RIGHT means swiping right to scroll content right.
3. **double click**
   - Parameters: `index` (integer, corresponding to the UI element index in the annotated screenshots)
4. **long press**
   - Parameters: `index` (integer, corresponding to the UI element index in the annotated screenshots)
5. **Home**
   - No parameters, navigate to the Home screen.
6. **Home**
   - No parameters, navigate to the Home screen.
7. **Home**
   - No parameters, navigate to the Home screen.
8. **Back (back)**
   - No parameters, indicates returning to the previous state.
9. **Task Complete (done)**
   - No parameters, indicates the task is completed.

Output Format Please output strictly in the following JSON format:

```
{
    "reasoning":
    "Provide a detailed explanation of your analysis and the
     ↪  reason for
    choosing this action. For click actions, it must include
    the following complete process:
    1) Detailed description of the target element: including
     ↪  element
    content, color, shape, size, and other visual features.
    2) Precise location of the target element: its specific
     ↪  position
    on the screen, using surrounding elements as reference.
    3) Annotation map search process: state in which
     ↪  annotation map
    (layer number) the matching red bounding box was found.
    4) Red bounding box verification: confirm that the
     ↪  position,
```

```
    contained content, and boundaries of this red box
    ↪ perfectly
    match the target element.
    5) Index reading confirmation: explicitly state the
    ↪ number found
    inside the top-left corner of the selected red bounding
    ↪ box.
    6) Final confirmation: reiterate that this choice is
    ↪ correct and no
    adjacent element was mistakenly selected.
    For input actions, you must first explicitly state:
    1) Whether a soft keyboard is currently displayed on the
    ↪ screen.
    2) Whether the target input field is already activated
    ↪ (has a cursor
    or is highlighted).
    3) If either of the above checks is negative, you MUST
    ↪ choose a
    click action first to activate the input field, NOT an
    ↪ input action.",
    "action": "action_name(click/swipe/input/back/done)",
    "parameters": {
        "parameter_name": "parameter_value"
    }
}
```

Critical Steps for Index Selection (Mandatory Reading for Click Actions)
Step 1: Precisely describe the visual features of the target element.

- Describe in detail the content, color, shape, and other visual features of the target element.
- Precisely describe the element's position on the screen (e.g., top third of the screen, left edge, bottom-right corner).
- Describe other UI elements surrounding the target element as reference points.
- Example: "Need to click the white input box with the text 'Search', located at the very top of the screen, just below the app title."

Step 2: Systematically search for the red bounding box.

- **You MUST examine each annotation map sequentially, one by one.** Do not skip any map.
- For each map, first observe the overall distribution of all red bounding boxes.
- Focus on finding the red bounding box whose position and features perfectly match the description from Step 1.
- **Critical Requirement: The red bounding box must completely enclose the target element, with boundaries snugly fitting.**

Step 3: Multi-level verification to ensure correct selection (Most Important Step).

- **Position Verification**: Confirm the red bounding box's location matches the position described in Step 1 exactly.
- **Content Verification**: Carefully observe whether the content inside the red bounding box is indeed the target element.
- **Boundary Verification**: The boundaries of the red bounding box should be snug against the target element, not containing excessive blank space.
- **Exclude Interference**: If there are multiple similar red boxes, you MUST choose the one whose position matches most precisely.

- **Avoid Adjacent Selection**: Absolutely DO NOT select a red box next to or near the target element.

Step 4: Read the index number (Final confirmation before execution).

- Re-confirm that the selected red bounding box indeed encloses the correct target element.
- Look at the number inside the **top-left corner** of this red bounding box.
- **Strict Requirement: Must be the top-left corner; the number must be clearly visible.**
- This number is the `index` value to use.

Step 5: Final Verification.

- In the reasoning, explicitly state: "The red bounding box I selected is located at [specific position], contains [specific content], and its top-left number is [X]."
- If you have any uncertainty about the selection, you MUST restart from Step 1.

Critical Steps for Text Input (Mandatory Reading for Input Actions)
Important Prerequisite: Absolutely DO NOT use the input action when the input field is not activated!
Step 1: Mandatory Check of Soft Keyboard Status (Must execute, cannot skip).

- **Must Check**: Carefully observe if a soft keyboard (virtual keyboard interface) is currently displayed at the very bottom of the screen.
- **Judgment Criterion**: If the bottom of the screen does NOT display a soft keyboard interface containing letter and number keys, it means no input field is activated.
- **Key Rule**: **You are only allowed to perform an input action when the soft keyboard is fully displayed at the bottom of the screen.**
- **Must State in Reasoning**: "Check soft keyboard status: [Displayed/Not Displayed]."

Step 2: Activating the Input Field (Must execute if Step 1 check fails).

- **Strictly Forbidden**: If there is no soft keyboard or the input field is not activated, you absolutely CANNOT use the input action.
- **Must Do**: You MUST first use a click action on the target input field to activate it.
- **Must State in Reasoning**: "Soft keyboard not displayed / Input field not activated. Must click to activate the input field first."

Step 3: Handling Existing Content.

- If the input field has default text, you may try to clear it or overwrite it directly.
- Choose the handling method based on the specific situation.

Step 4: Execute Text Input (Only when preconditions are satisfied).

- After confirming the soft keyboard is displayed AND the input field is activated, you may use the input action.
- After inputting, check that the text in the input field is correct, ensuring there are no input errors, omissions, or extra characters.
- In reasoning, you MUST explicitly state: "Confirmed soft keyboard is displayed and input field is activated."

Step 5: Handling Soft Keyboard After Input (Important).

- **Must Check After Input**: Observe the key types on the soft keyboard.
- **Criteria for Hiding Soft Keyboard**:
  - If the soft keyboard has action buttons like "Search", "OK", "Done", "Send", etc., you should click these buttons.

  - If the soft keyboard only has navigation buttons like "Next" or "Enter", AND the keyboard is blocking important UI elements, you should click the "down arrow" button (usually top-right) on the soft keyboard to hide it.
- **Must State in Reasoning**: "Check soft keyboard key types: [Action type/Navigation type]. Is the soft keyboard blocking important elements: [Yes/No]. Decision: [Click action button / Hide soft keyboard / Keep as is]."

Strictly Forbidden Input Action Patterns

1. Inputting directly without a soft keyboard.
2. Inputting directly without describing the check process in reasoning.
3. Inputting directly upon seeing an input field (must check activation status first).
4. Not considering soft keyboard obstruction after input.

The Only Correct Input Action Pattern

- Reasoning includes: "Check soft keyboard status: Displayed. Check input field status: Activated. Confirmed text input is permissible."
- Only reasoning containing this complete check process allows the use of the input action.
- **Post-Input Handling**: After input, you must check soft keyboard key types and whether it blocks important elements to decide if it needs to be hidden.

Important Rules

1. **Position Match Priority**: First determine the element's precise location in the original screenshot, then find the corresponding red bounding box in the annotation maps.
2. **Accurate Number Reading**: The `index` must be the actual number displayed inside the top-left corner of the red bounding box (red background, white digits).
3. **Avoid Mis-selecting Adjacent Elements**: This is the most common mistake! Ensure the chosen red bounding box fully encloses the target element, not a nearby similar element.
4. **Mandatory Adjacent Element Exclusion Check**: Before selecting any index, you must explicitly explain why other red bounding boxes in the vicinity were NOT chosen.
5. **Soft Keyboard Obstruction Handling**: After input, if the soft keyboard is blocking important elements and there is no action button, click the top-right down arrow to hide it.
6. **Multi-step Operations**: For complex selections (like date ranges, time slots, cascading options), multiple consecutive actions are required.
7. **Special Attention for Date Selection**:
8. **Task Completion Judgment**: Use the 'done' action only when the specified task is indeed completed.
9. **Operation Coherence**: Each action should be a logical choice based on the current screen state and task goal.
10. **Page Error Handling**: If you encounter an incorrect page or a loading failure, you can try going back to the previous level (via a swipe gesture from the leftmost screen edge or by clicking a back button).

Index Selection Examples
Incorrect Example 1:

- reasoning: "Need to click the search button."

- Problem: No description of the element's specific location or visual features.

Incorrect Example 2:

- reasoning: "Need to click the search box, located at the top of the screen. Found the search box in the annotation map, choose number 8."
- Problem: Description is too vague, lacks verification process, prone to selecting the wrong adjacent element.

Correct Example:

```
reasoning: "1) Target element detailed description: Need to
 ↪  click the
white input box with the placeholder text 'Search',
 ↪  rectangular in shape,
with a light grey border.
2) Precise location description: This search box is located
 ↪  at the very
top of the screen, approximately 50 pixels below the status
 ↪  bar, occupies
about 80% of the screen width, centered.
3) Annotation map search: In annotation map #2, I found a red
 ↪  bounding
box at the central top position of the screen.
4) Red bounding box verification: This red box completely
 ↪  encloses the
search input box, its boundaries perfectly align with the
 ↪  edges of the
input box, and it indeed contains the white input box with
 ↪  the text
'Search'.
5) Index reading: The top-left inner corner of this red box
 ↪  clearly
shows the number '15'.
6) Final confirmation: Confirmed this box does not contain
 ↪  any irrelevant
elements, nor is it an adjacent UI element; it is precisely
 ↪  the search
box I intend to click."
parameters: {"index": 15, "target_element": "Search input
 ↪  box"}
```



Figure 9: **Execution Example of General Models**: General models leverage icon recognition models for enhancement.

G GENERALIZATION VS. ACCURACY

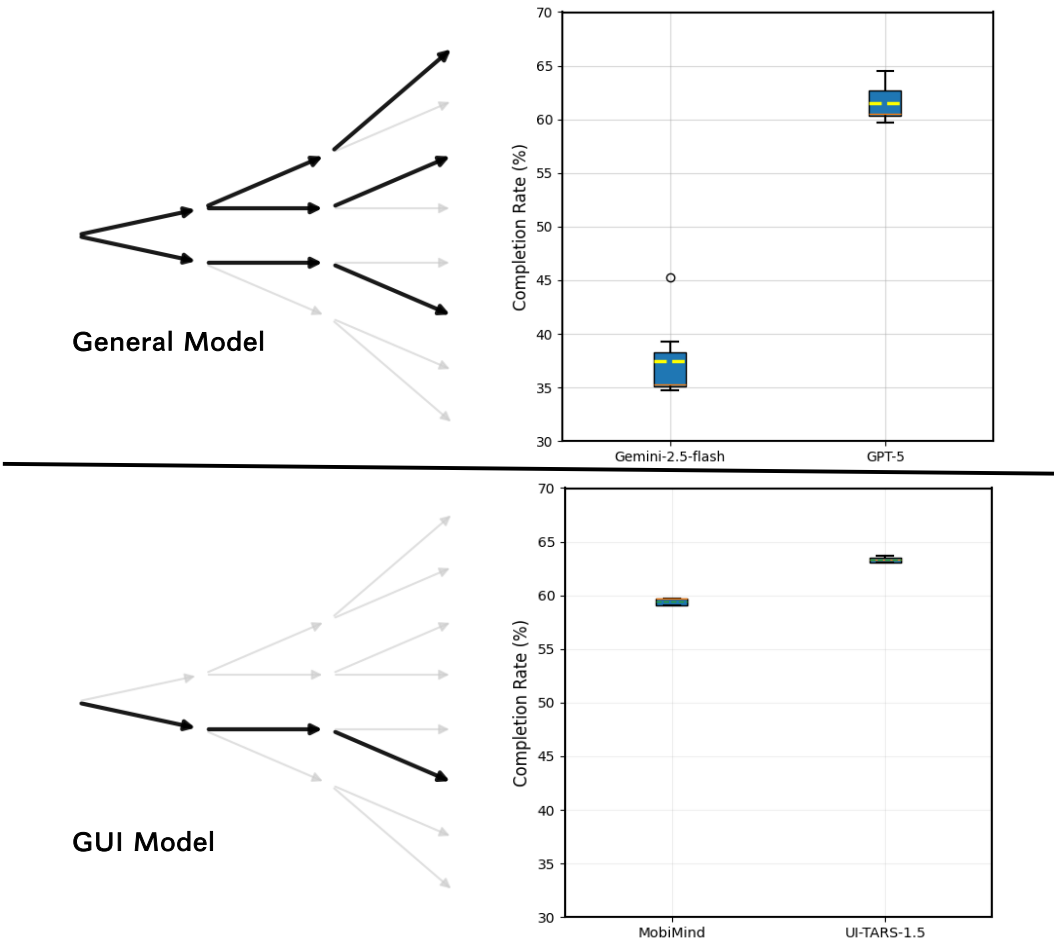

Figure 10: **The tension between generalization capability and determinism**. The upper figure presents the performance scores of a general-purpose model across multiple sampling trials on randomly selected tasks. The lower figure shows the sampling results of a GUI-specialized model on the same set of tasks.

We selected different general-purpose models (GPT-5, Gemini-2.5-flash) and specialized models (UI-TARS-1.5, MobiMInd), randomly chose multiple tasks, and conducted repeated evaluations under identical configurations. The results revealed that the Completion rate (CR) of general-purpose models exhibited significantly greater fluctuations than that of specialized models. The specialized models produced stable outcomes with consistent execution paths, whereas general-purpose models demonstrated more diverse approaches to task completion.

H RESOLUTION SCALING

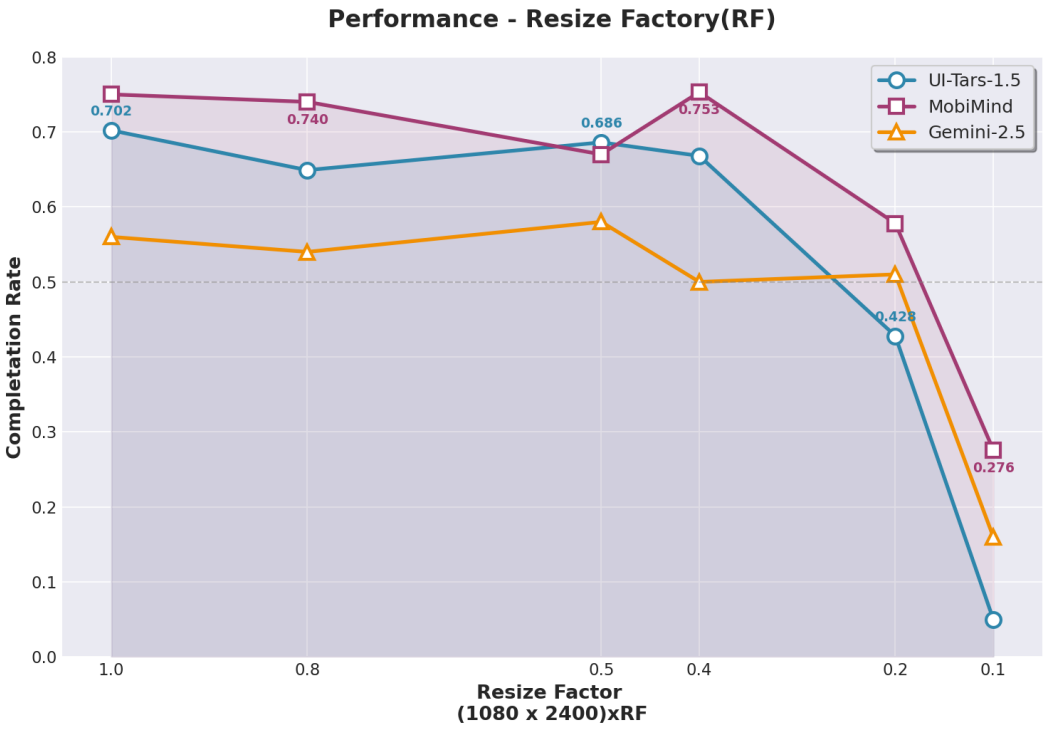

Figure 11: **Completion rate of different models under varying resolution scaling factors**. We randomly sample representative examples and evaluate them at different scaling levels based on a resolution of 1080×2400.

We evaluate UI-TARS-1.5, MobiMind, and Gemini-2.5-Flash on the same evaluation subset by scaling images with an original resolution of 1080×2400 to different factors and assessing task completion performance. We observe that when the scaling factor falls below 0.2, all models exhibit a pronounced drop in completion rate.

I INSTRUCTION NOISE DETAILS

Table 5: **Robustness to instruction noise.** W/O and W/ denote without and with, respectively. AMR measures instruction-following ability.

Model	W/O Noise AMR	W/ Noise AMR
General Model		
Gemini-2.5-Flash	35.55	38.55+3.0
Gemini-2.5-Pro	55.34	56.48+1.1
Claude-opus-4.1	66.38	67.38+1.0
GPT-5	71.38	71.07-0.3
GUI specialized Model		
MobiMind-Mixed	59.89	55.11-4.7
UI-TARS-1.5	63.12	63.12-0.0
GUI-Owl	65.37	64.72-0.4

J EXECUTION EFFICIENCY

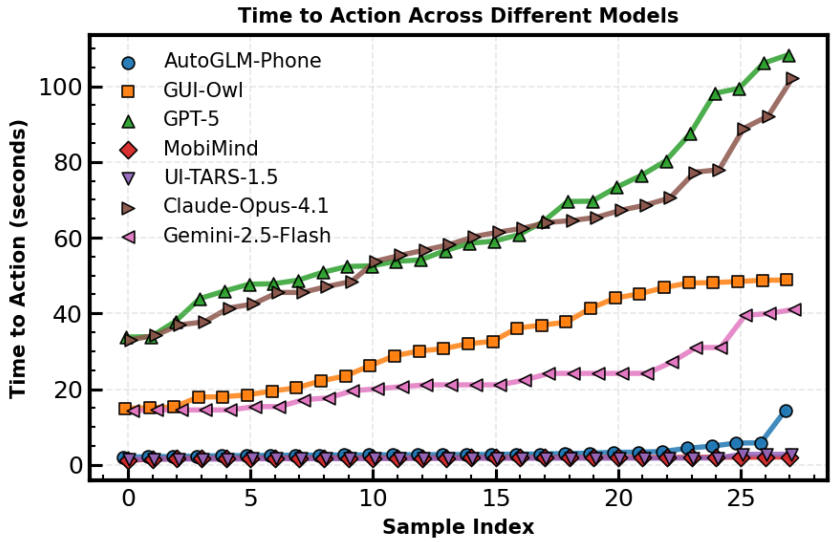

Figure 12: **Model execution efficiency.** General models use the same inference framework and operate under identical network request environments.

K TASK STATISTICS

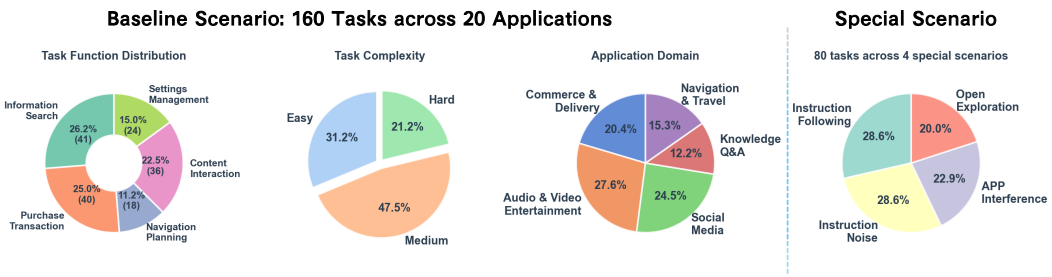

Figure 13: **Task statistics for *MobiGraph***, covering the type, complexity, and application domain of basic-scenario tasks, as well as the quantity of special-scenario tasks.