MobiFlow: Real-World Mobile Agent Benchmarking through Trajectory Fusion

Yunfei Feng , Xi Zhao , Cheng Zhang , Dahu Feng , Daolin Cheng , Jianqi Yu , Yubin Xia , Erhu Feng *

Posted Date: 17 March 2026

doi: 10.20944/preprints202603.1313.v1

Keywords: GUI Agent; VLM; evaluation

 doi:10.20944/preprints202603.1313.v1

Article

MobiFlow: Real-World Mobile Agent Benchmarking through Trajectory Fusion

Yunfei Feng [1], Xi Zhao [1], Cheng Zhang [1], Dahu Feng [2], Daolin Cheng [1], Jianqi Yu [3], Yubin Xia [1] and Erhu Feng [1,*]

[1] Institute of Parallel and Distributed Systems (IPADS), Shanghai Jiao Tong University
[2] Department of Precision Instrument, Tsinghua University
[3] National Innovation Institute of High-end Smart Appliances
* Correspondence: fengerhu1@sjtu.edu.cn

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, however, 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 and making it difficult to evaluate model performance a ccurately. To address these issues, we propose *MobiFlow*, an evaluation framework built on tasks drawn from arbitrary third-party applications. Using an efficient graph-construction algorithm based on multi-trajectory fusion, *MobiFlow* can effectively compress the state space, support dynamic interaction, and better align with real-world third-party application scenarios. *MobiFlow* covers 20 widely used third-party applications and comprises 240 diverse real-world tasks, with enriched evaluation metrics. Compared with AndroidWorld, *MobiFlow*'s evaluation results show higher alignment with human assessments and can guide the training of future GUI-based models under real workloads.

Keywords: GUI Agent; VLM; evaluation

1. Introduction

With the advancement of artificial intelligence technology, graphic User Interface (GUI) agents (Ye et al. 2025a), driven by multimodal large 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(Rawles et al. 2024; Toyama et al. 2021). 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(Xu et al. 2025a; Zhang et al. 2024a).

 doi:10.20944/preprints202603.1313.v1

While this approach can be applied to arbitrary applications, 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, existing mobile usage evaluation 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 *MobiFlow*, 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.

We efficiently collect human operation trajectories from real-world task scenarios using front-end tools[1], including interface screenshots, actions, and annotation information. 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.

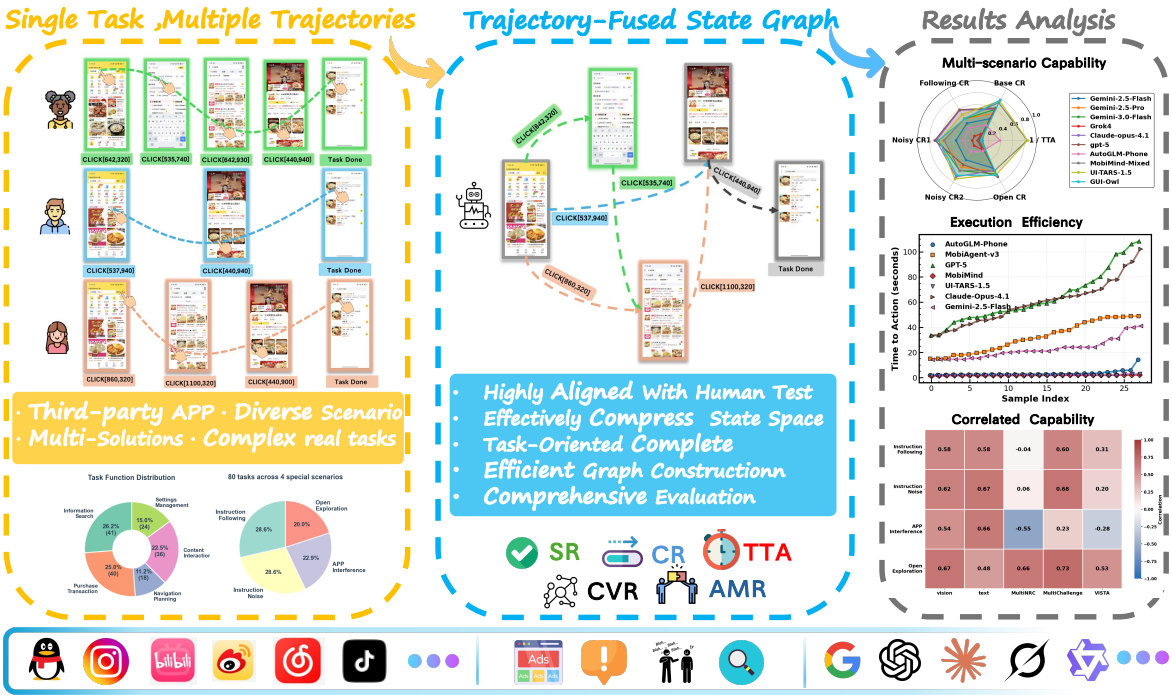

Figure 1. The framework of *MobiFlow*. It constructs state transition graphs from third-party applications and evaluates 20 applications across 240 tasks. The framework extends existing metrics to assess multiple models in terms of task completion, execution efficiency, generalization, and alignment with competency requirements. The code is available at https://github.com/nanookfyf/MobiBench. Our data will be released upon acceptance.

MobiFlow 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(Qin et al. 2025; Wang et al. 2025; Ye et al. 2025a; Zhang et al. 2025). We abstract the agent's task execution process as state transitions on a graph (see Figure

[1]Implementation details are provided in the AppendixA.

 doi:10.20944/preprints202603.1313.v1

2), which enables more precise evaluation(Bu et al. 2025). We can 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, instruction-noise interference, and multi-application interference, etc.

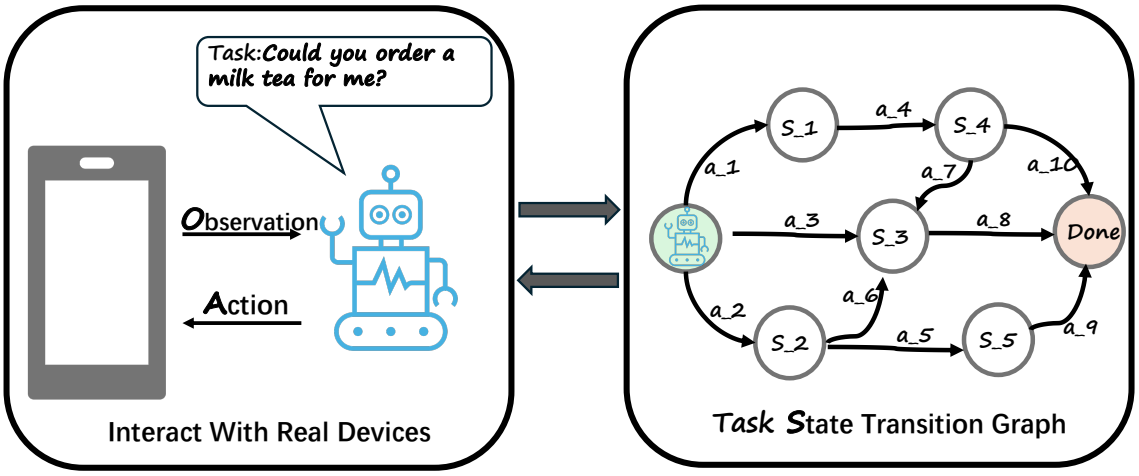

Figure 2. Modeling agent-device interactions with state transition graph. Executing actions triggers state transitions. Completing a task corresponds to reaching a terminal state.

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, *MobiFlow* has been deployed to assess the capabilities of GUI agent models on smartphones with sales volumes exceeding ten million units.

In summary, our main contributions are as follows:

- **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(Detailed comparisons are provided in Table 1), 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 possible trajectories and assessing them individually, which fails to support effective 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(Chai et al. 2025b; Lee et al. 2024) can cover arbitrary scenarios and conveniently provide valid evaluation signals, they still fail to effectively encompass diverse dynamic interactions.

 doi:10.20944/preprints202603.1313.v1

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) and MobileWorld(Kong et al. 2025), still suffer from several limitations, including unstable environments and 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
AndroidWorld(Rawles et al. 2024)	20	116	✓	✗	✗	✓	Result-based
AndroidLab(Xu et al. 2025a)	9	138	✓	✗	✗	✓	Result-based
SPA-Bench (Chen et al. 2024)	66	340	✓	✓	✗	✓	Result-based
MobiFlow	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, and consider a specific task $\mathbf{g} \in \mathbf{G}$. We model the mobile device together with the application software as an observable finite-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, swiping, or text input. $\mathcal{O}$ denotes the observation space, where the observation $o \in \mathcal{O}$ perceived by the agent is determined by the current state $s \in \mathcal{S}$. The state transition function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is assumed to be deterministic(the UI transition structure of APPs remains fixed, even though its content may vary randomly), and $\mathcal{R}$ represents the environment reward function.

- A statistical analysis that motivates the task-oriented environment formulation is presented later in this paper.
- The transition function $\mathcal{T}$ induces a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where an edge $e = (s, s') \in \mathcal{E}$ exists if and only if there is an action $a \in \mathcal{A}$ such that $\mathcal{T}(s, a) = s'$. Each edge is labeled with the corresponding action a.

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

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

[2]Details of the action space are provided in Appendix B.

 doi:10.20944/preprints202603.1313.v1

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, the reasoning and decision spaces are often integrated into a single module in many models; we present them separately here for conceptual clarity.
- Depending on the implementation, the memory update process may not explicitly store all components—observation, reasoning state, and action.

4. MobiFlow

To capture state transitions across diverse tasks, accurately evaluate agent performance under varied scenarios, and enable in-depth attribution analysis of agent capabilities, we propose the *MobiFlow* 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 graphs via exhaustive search leads to state space explosion, while directly building complete graphs incurs extremely high construction complexity and human labor costs. Through an analysis of the top 30 mainstream apps[3], we find that the number of independent interactive elements on an app interface approximately follows a normal distribution with a mean of 55.8(see Figure 3). 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($O(1.7^d)$, where d is action depth) that is far smaller than that of search-based approaches($O(55.8^d)$). Moreover, we observe that most trajectories converge to states with similar transition structures after only a few steps.

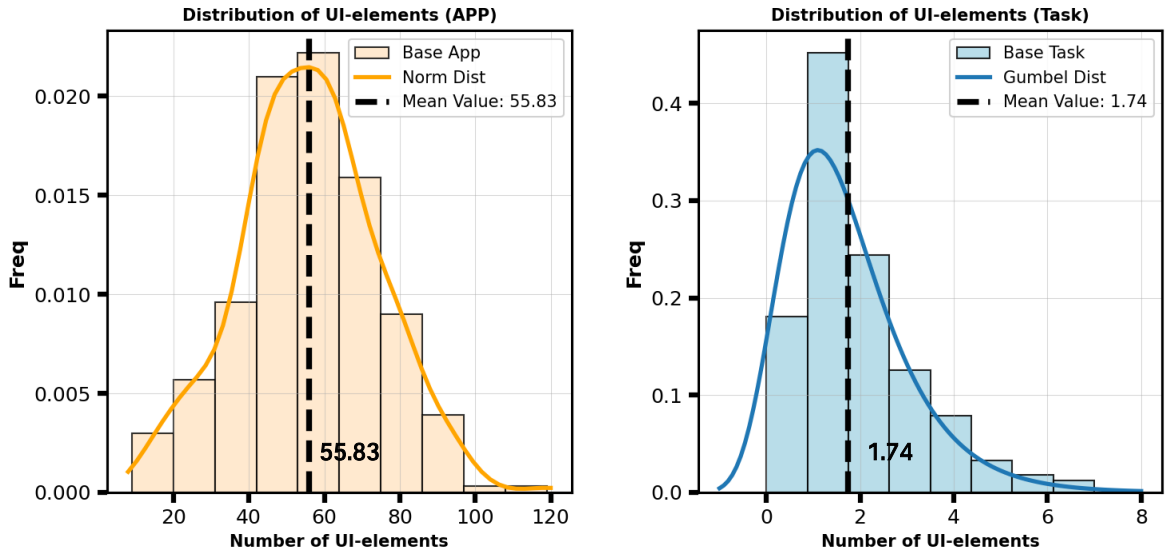

Figure 3. Comparison of the Distribution of Interactive Element Counts Across Different Modeling Approaches. The left figure presents the app-based modeling approach, while the right figure illustrates the task-based modeling approach.

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-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

[3] Static details in Appendix D

 doi:10.20944/preprints202603.1313.v1

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 tend to converge to states with highly similar transition structures after very few steps, indicating the presence of many reusable state transitions. This reuse can further reduce the complexity of the state space. 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 low-complexity TCSG construction (as shown in Figure 4). For cross-application tasks, we can likewise construct via trajectory merging or multi-graph connectivity methods.

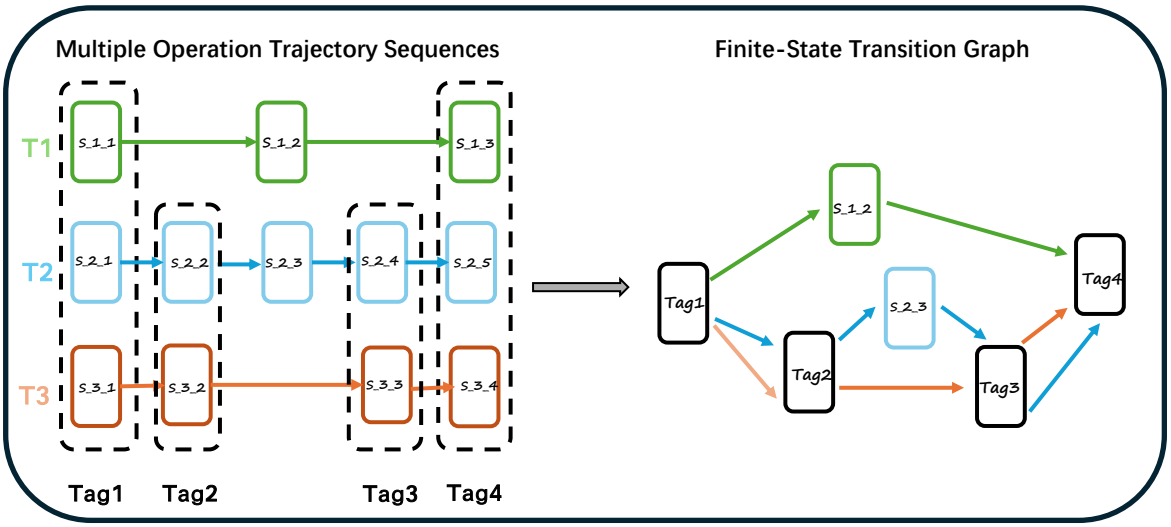

Figure 4. Algorithm workflow illustration: For states with identical labels (or equivalently, similar transition structures), we merge them and allow the merged state to share their transition conditions, thereby compressing the state complexity.

Efficient Graph Construction. The construction of the TCSG can be mainly divided into three stages. First, trajectories are traversed to assign labels to states. Second, chain-structured transitions are initialized. Finally, the transition spaces of states sharing the same labels are merged via union coverage[4]. We achieve fast localization of target UI elements by parsing XML contents. More efficient labeling is enabled through UI structural similarity matching and visual model inference. We employ efficient node representations and maintain a union-find structure to support fast merging of transition spaces. With the techniques described above, the time cost of practical data collection and graph construction can be substantially reduced. For example, a task that involves posting a comment on Weibo typically requires around 15 interaction steps to complete. By collecting 7 trajectories, we can cover all valid task completion paths, and the entire construction process, including manual verification, can be finished in under 10 minutes.

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

[4]The construction algorithm for the transition graph is provided in AppendixC

 doi:10.20944/preprints202603.1313.v1

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 (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 Table 2.

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

Metric	Formula	Notation Explanation	Description
Success Rate (SR)	$\frac{1}{N}\sum_{i=1}^{N}\mathrm{suc}(i)$	N: number of tasks; $\mathrm{suc}(i)=1$ if g_i is completed, else 0.	Indicates whether a task is fully completed, focusing only on the final outcome.
Completion Rate (CR)	$\max\left\{\frac{d(s_{\text{visited}},s_{\text{goal}})}{d(s_{\text{start}},s_{\text{goal}})}\right\}$	$s_{\text{start}}, s_{\text{goal}}$: 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.

4.3. Diverse Scenario

In this section, we introduce a variety of evaluation scenarios constructed based on task-state transition graphs.(see Figure 5) These scenarios are designed to assess model performance from multiple perspectives, including practicality, safety, robustness, and exploratory capability.

Base Scenario: Based on construction algorithms, we constructed 160 distinct tasks spanning 20 mainstream applications. To ensure these tasks reflect real-world usage scenarios, we selected tasks across multiple domains, including common activities like information search, product purchasing, ride-hailing navigation, social chatting, and application settings. Furthermore, we resampled task descriptions to ensure comprehensive alignment with real-world applications. This framework enables a thorough evaluation of agents' capabilities for handling common everyday tasks.

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

Special Scenario 2 Instruction Noise Interference: In daily communication, errors such as typos or the use of emojis and special characters may occur. The agent should be able to identify the true task intent within such noisy instructions and correctly complete the task. In this scenario, noise will be introduced into task instructions through methods such as common typo substitutions, emoji injections, and multilingual mixing. This aims to evaluate the model's resistance to interference and the robustness of its instruction comprehension.

Special Scenario 3 APP Interference: In real-world applications, software often faces various interferences, such as pop-up notifications from other apps, ad pushes, unintended touches, or interface jumps caused by system anomalies. The agent should respond safely when encountering such abnormal situations. Therefore, instances containing these interferences are collected separately to evaluate the model's ability to cope with specific APP disturbances.

 doi:10.20944/preprints202603.1313.v1

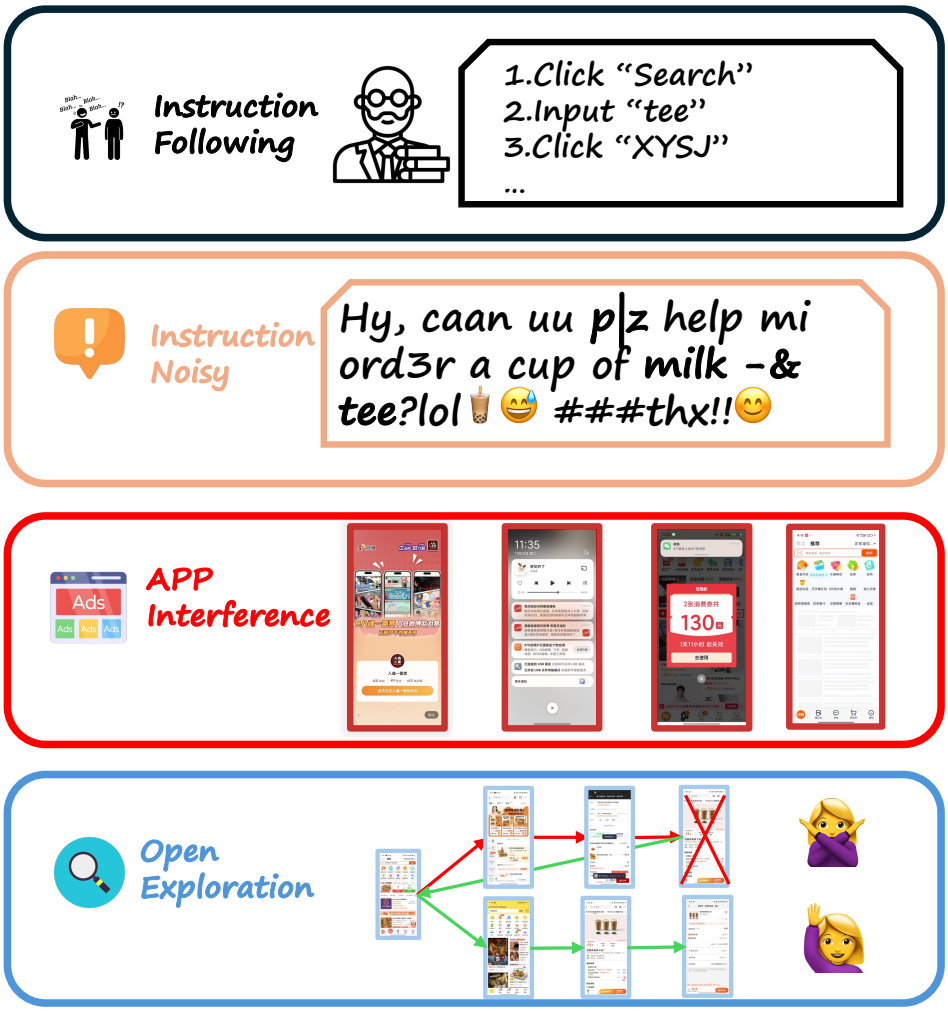

Figure 5. Special scenarios including instruction following, instruction interference, application interference, and open exploration (to observe whether the model can deviate from erroneous paths).

Special Scenario 4 Open Exploration: A knowledgeable agent should possess exploration and autonomous learning capabilities. To this end, we artificially constructed a set of special transition graphs. For example, based on basic scenarios, we introduce erroneous paths that do not satisfy task requirements, allowing us to observe whether the model can detect issues within these incorrect paths and reasonably backtrack to accomplish the designated task.

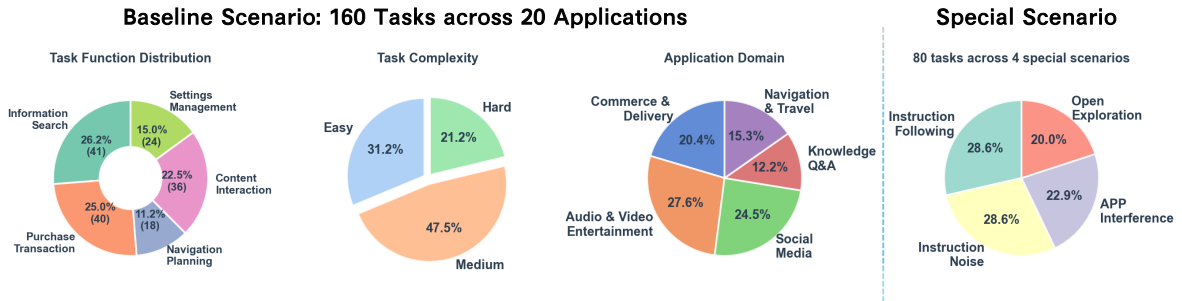

Figure 6. Task statistics for *MobiFlow*, covering the type, complexity, and application domain of basic-scenario tasks, as well as the quantity of special-scenario tasks.

5. Experiments

This section analyzes the evaluation results of several mainstream general and specialized models on our evaluation benchmark(see Table 3).

 doi:10.20944/preprints202603.1313.v1

Table 3. Main Results on *MobiFlow*.Includes general-purpose and specialized models. TTA (Time to Action), SR (Success Rate), CR (Completion Rate), AMR (Action Matching Rate), CVR (Coverage Rate). CR1 (with instruction noise), CR2 (under APP interference).

Model	Size(B)	TTA(s)↓	Base Scenario			Following		Noisy Scenario		Open Exploration	
			SR(%)↑	CR(%)↑	CVR(%)	AMR(%)↑	SR(%)↑	CR1(%)↑	CR2(%)↑	CR(%)↑	CVR(%)
General Model											
Gemini-2.5-Flash	-	17.03	30.7	41.7	17.4	35.5	33.3	35.5	37.7	53.2	21.1
Gemini-2.5-Pro	-	30.12	38.7	52.1	26.9	55.3	41.6	56.4	51.7	59.7	28.1
Gemini-3.0-Flash	-	33.31	40.2	59.7	30.2	56.4	41.6	64.7	58.4	65	34.3
Grok4	-	142.21	8.3	12.3	8.41	4.1	8.3	8.1	10.0	17.8	10.2
Claude-opus-4.1	-	46.68	43.6	60.3	34.6	62.2	58.3	68.6	70.1	57.9	37.7
GPT-5	-	55.49	49.7	62.7	35.3	**71.4**	**59.3**	**71.1**	46.7	**70.0**	39.3
GUI Model											
AutoGLM-Phone	9B	3.85	16.7	30.2	10.6	30.5	24.2	24.2	22.5	20.1	15.1
MobiMind-Mixed	4B	1.76	53.8	72.9	31.9	59.9	42.3	55.1	58.3	58.6	35.1
UI-TARS-1.5	8B	**1.75**	**60.4**	76.3	34.4	63.1	50.0	63.1	**74.5**	65.7	37.1
GUI-Owl	8B	20.02	55.7	**77.9**	37.5	65.4	41.6	64.7	67.4	67.1	38.1

5.1. Experimental Setup

We evaluated a series of advanced models, including general-purpose models and GUI-specific models: OpenAI GPT-5, Claude-opus-4.1, Grok4, Gemini-2.5-flash, Gemini-2.5-Pro, Gemini-3-flash(Anthropic 2025; Comanici et al. 2025; OpenAI 2025), UI-TARS-1.5(Qin et al. 2025), AutoGLM-Phone(Liu et al. 2024), GUI-Owl(Ye et al. 2025b), and MobiMind(Zhang et al. 2025). To ensure a consistent comparison, we adopted the same execution framework for general models, incorporating OminiParser(Lu et al. 2024) to annotate actionable UI elements to reduce the complexity of coordinate-based interface operations[5]. 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 GPUs using the vLLM(Kwon et al. 2023) inference framework. For general models, we uniformly utilized the OpenRouter API for requests, 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 60.4 in the base scenario, while UI-TARS achieves the highest CR of 74.5 under APP interference. However, specialized GUI models do not exhibit a comprehensive advantage over general-purpose 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 sampled tasks of varying complexity and analyzed the execution efficiency of agent interactions, as shown in Figure 7. The factors that affect single-action latency include network latency, model size, the number of decoding tokens, and the number of model invocations per action. Under the same model size, fewer decoding tokens generally lead to higher efficiency. For some models, such as GUI-OWL, a single action requires multiple model calls, which degrades action execution efficiency. In contrast, MobiMind and UI-TARS achieve efficiency advantages due to their lightweight architectures, single-call execution per action, and limited decoding outputs. General models, on the other hand, suffer from higher action latency because of greater network latency, larger model sizes, and decoding outputs that vary substantially with task difficulty[6].

Performance Fluctuations. We observed that the evaluation results of general models exhibit greater fluctuation than those of GUI models. Furthermore, while the execution paths of GUI models

[5]Detailed implementation can be found in the AppendixF

[6]All experiments followed the settings in the section5.1.

 doi:10.20944/preprints202603.1313.v1

remain relatively consistent across multiple sampling attempts for a given task, general models demonstrate diverse paths but unstable completion outcomes, characterized by substantial variance.[7]

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[8]. For images with a resolution of 1080×2400, the model's task completion rate drops significantly when the resolution is scaled below 0.2×.

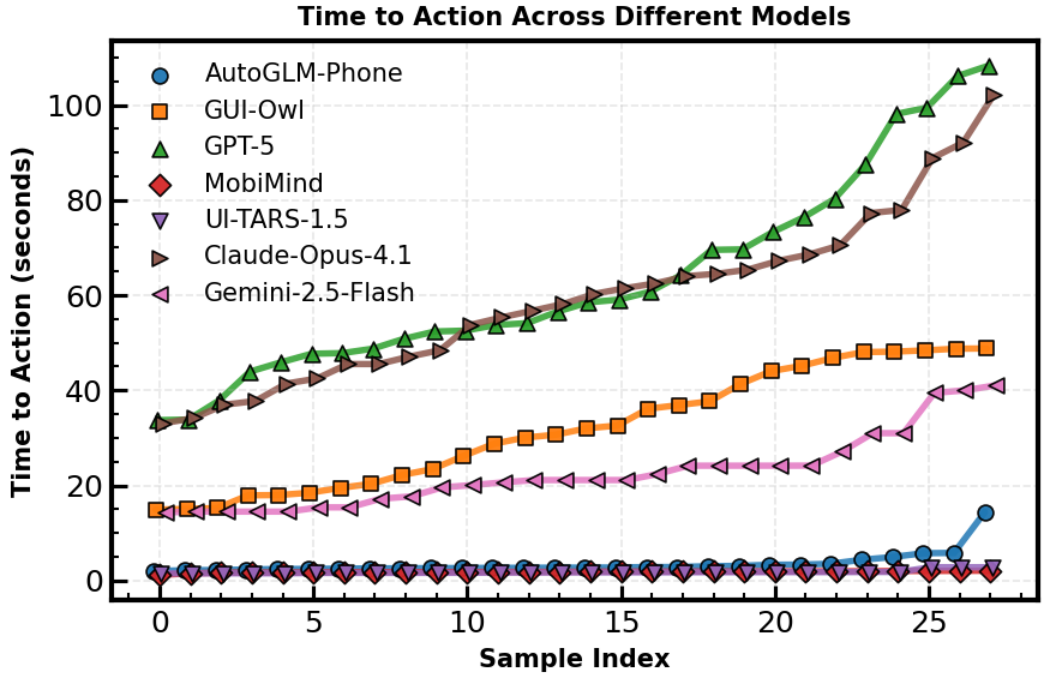

Figure 7. Model execution efficiency. GUI models use the same inference framework and operate under identical network request environments.

5.3. In-Depth Analysis

We observe that general models not only exhibit greater robustness against instruction noise interference compared to specialized GUI models, but also achieve a higher completion rate in instruction-following capability under such conditions. As shown in the Table 4, one possible explanation is that the presence of specific interference may heighten the model's attention to the task-relevant portion of the long-context instructions.

General models exhibit strong generalization capabilities and outperform GUI models in exploratory tasks. However, lightweight specialized models support on-device deployment, while their stable accuracy in completing a large volume of tasks further facilitates practical deployment and application in real-world scenarios.

Across different scenarios, we analyze the performance scores of general models and compare them with established benchmarks(Text&VisionArena[9], MultiNRC(Fabbri et al. 2025), MultiChallenge(He et al. 2024), and VISTA(Zheng et al. 2023)[10]) to investigate which specific capabilities are reflected by each scenario. Our correlation analysis results are illustrated in the Figure 8.

[7] Experiment details are provided in the AppendixG.
[8] Details are provided in the AppendixH
[9] https://lmarena.ai
[10] https://scale.com/leaderboard

 doi:10.20944/preprints202603.1313.v1

Table 4. Robustness to Instruction Noise: A Comparison of General Models vs. GUI Models.

Model	No Instruction Noise AMR	Instruction 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-sonnet-4.5	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

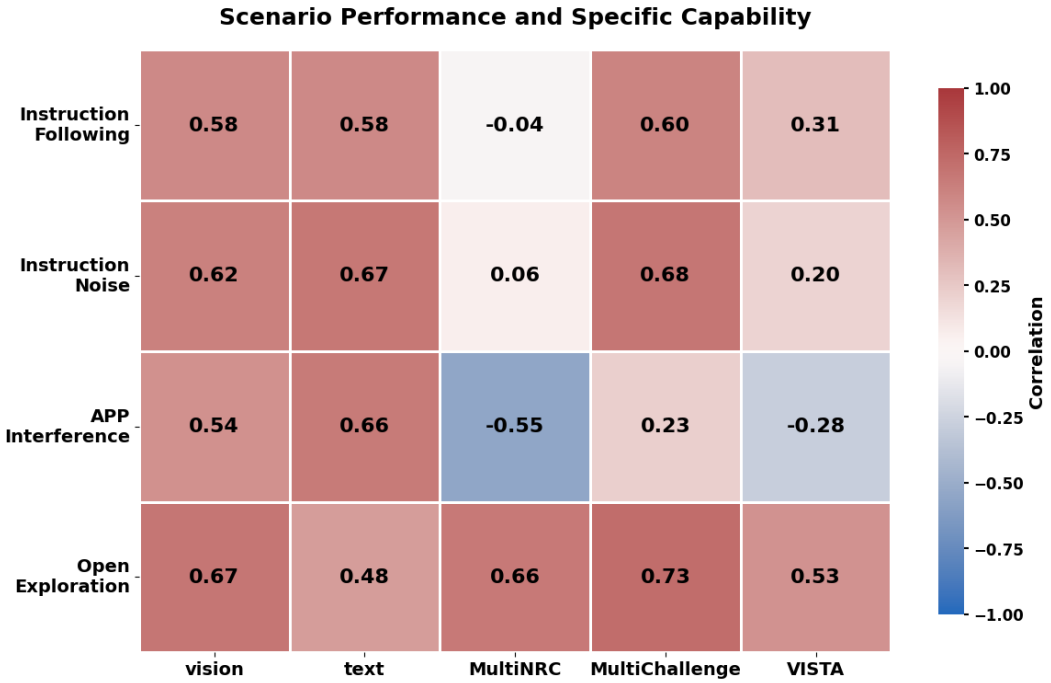

Figure 8. Correlation Analysis. Correlation between General Models' Performance in Specific Scenarios and Different Benchmark Capabilities.

We find that each scenario exhibits relatively high correlations with capabilities in visual input understanding and cross-cultural contextual comprehension. In open exploration and instruction-following scenarios, the correlation with multi-turn dialogue capability is particularly pronounced. Under app interference conditions, a certain negative correlation with logical reasoning capability is observed. Additionally, the open exploration scenario shows a particularly strong correlation with causal reasoning ability.

5.4. Special Issues

Thinking and observation become desynchronized. During the experiments, we observed that GUI models occasionally perform erroneous actions without triggering the corresponding page transition. Yet, the models proceed under the assumption that the transition has been completed, continuing to operate as if they were in the post-transition state.

Insufficient Generalization of Reasoning. We observe that some models tend to operate based on fixed reasoning patterns. For instance, when encountering unconventional tasks, most models default to searching for task-related keywords, occasionally overlooking the information inherently available on the page itself.

 doi:10.20944/preprints202603.1313.v1

Interface Comprehension Deviation. We have observed that models can sometimes be misled by interface information. For instance, in search scenarios, actual application recommendation algorithms often display suggested content in light colors; however, the model frequently misinterprets this as user-input text and attempts to clear it.

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 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.

6. Discussion

Cross-APP Tasks. We support cross-application tasks via trajectory merging or the multi-graph connectivity method. In this paper, we focus on evaluating the intrinsic capabilities of GUI models, and therefore, only single-task results are reported.

Reproduction. Our state graph is configured with deterministic transitions to ensure reproducible and accurate evaluation of third-party applications.

MCP Support. Since this paper focuses more on the evaluation of GUI operations for agents, support for the Model Context Protocol (MCP)(Yan et al. 2025) will be addressed in future work.

7. Conclusion

In summary, we introduce *MobiFlow*, a mobile-use benchmark built on arbitrary third-party APPs. By constructing state transition graphs, *MobiFlow* effectively compresses the state space of multiple task-completion trajectories. Compared with existing benchmarks, *MobiFlow* 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.

Impact Statement

This work facilitates the real-world deployment of mobile agents, which has been adopted in industry. The trajectory-fusion-based assessment approach is also applicable to the evaluation of other types of agents. This work will contribute to enhancing the practicality of agents in accomplishing various GUI tasks.

Appendix 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 Figure 9). 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

 doi:10.20944/preprints202603.1313.v1

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.

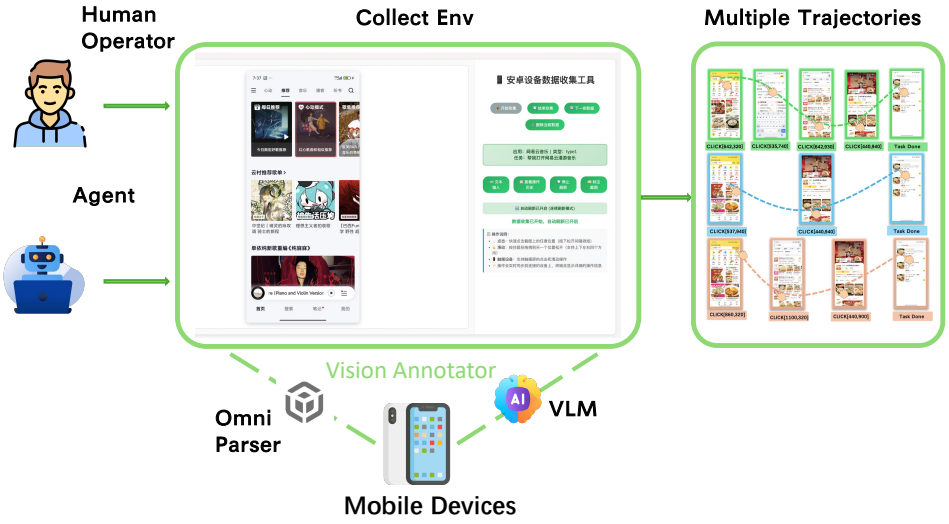

Figure 9. Efficient trajectory collection tools. A trajectory collection framework based on real devices and actual applications.

Appendix 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 5. 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

Appendix 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).

 doi:10.20944/preprints202603.1313.v1


```
@dataclass
class State:
    '''
    Task Transition Graph state
    '''
    State_Info      # screenshots or other (e.g., text or XML)
    Tag             # State tagging

    # Action-space-based interface transition hashing
    Transition_Space = {
        "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.

 doi:10.20944/preprints202603.1313.v1

Algorithm 1: Offline Graph Construction via Trajectory Merging

Input: Set of raw trajectories $\mathcal{D} = \{\tau_1, \ldots, \tau_N\}$, where $\tau_i = \{(o_1, a_1), \ldots, (o_T, \cdot)\}$
Output: State Transition Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
// Core Logic Overview:
// 1. Abstraction: Map raw UI observations to nodes via Semantic Labels or Hashing.
// 2. Construction: Build directed edges based on recorded action transitions.
// 3. Unification: Merge action spaces of identical states to maximize coverage.
Initialize node set $\mathcal{V} \leftarrow \emptyset$, edge map $\mathcal{E} \leftarrow \emptyset$
Define abstraction function $\Phi(o) \rightarrow s_{id}$
{**Phase 1: Trajectory Traversal & Topology Building**}
for each trajectory $\tau \in \mathcal{D}$ **do**
  for each transition step (o_t, a_t, o_{t+1}) in τ **do**
    {Step 1: State Abstraction}
    if o_t has semantic label L **then**
      $u \leftarrow L$
    else
      $u \leftarrow \text{HASH}(o_t)$
    end if
    $v \leftarrow \Phi(o_{t+1})$
    {Step 2: Graph Update}
    $\mathcal{V} \leftarrow \mathcal{V} \cup \{u, v\}$
    if $u \notin \mathcal{E}$ or $a_t \notin \mathcal{E}[u]$ **then**
      $\mathcal{E}[u][a_t] \leftarrow v$
    else
      $\mathcal{E}[u][a_t] \leftarrow \mathcal{E}[u][a_t] \cup \{v\}$
    end if
  end for
end for
{**Phase 2: Transition Space Unification**}
for each abstract state $u \in \mathcal{V}$ **do**
  if u is a labeled node **then**
    $\mathcal{T}_{all} \leftarrow \bigcup_{s \in \Phi^{-1}(u)} \text{Transitions}(s)$
    $\mathcal{E}[u] \leftarrow \mathcal{E}[u] \cup \mathcal{T}_{all}$ {Aggregate all historical actions}
  end if
end for
return $\mathcal{G} = (\mathcal{V}, \mathcal{E})$

Appendix D. Statistics on the Number of Interface Actions

We employ OminiParserV2 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 Figure10. 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 Figure10.

 doi:10.20944/preprints202603.1313.v1

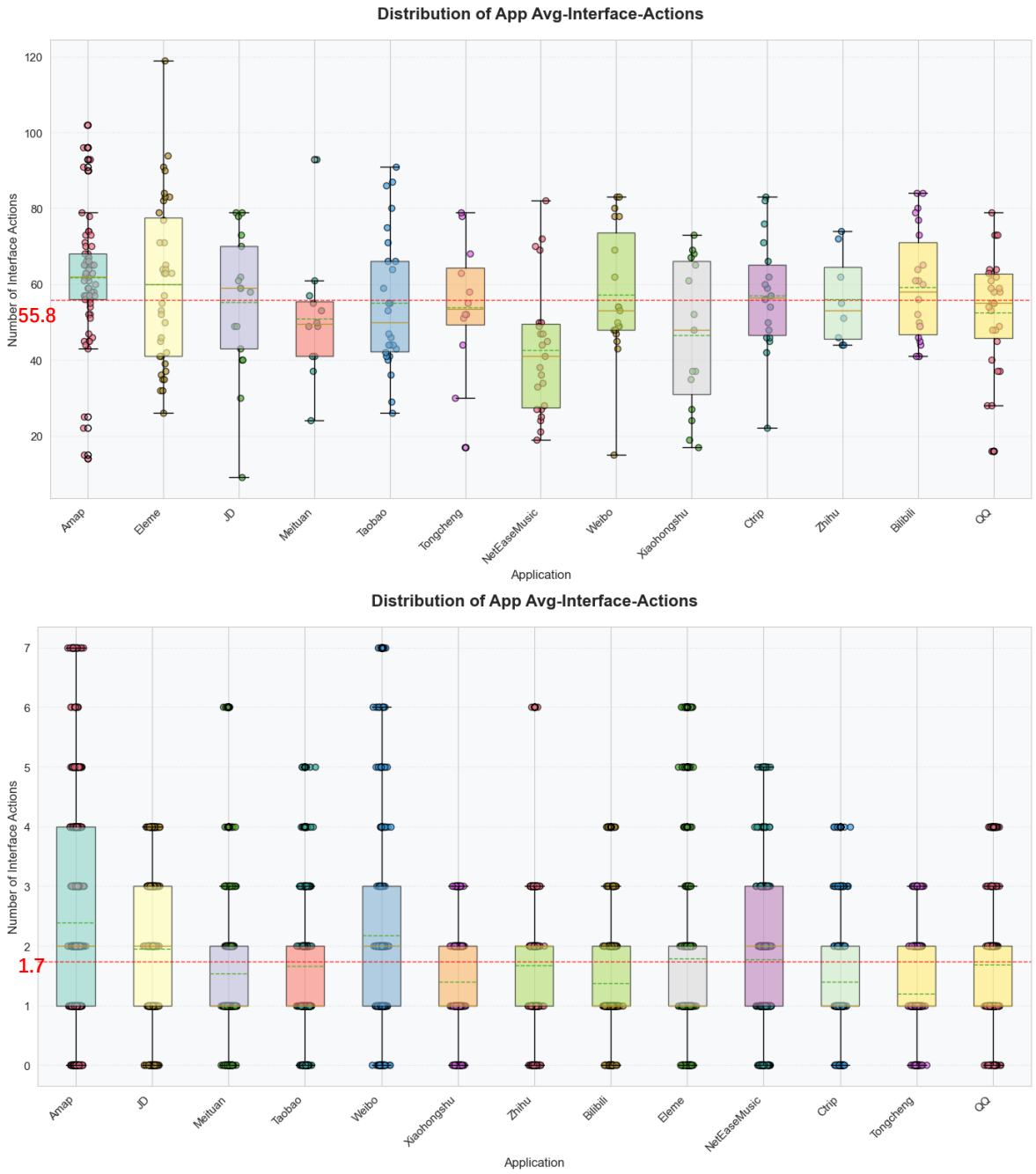

Figure 10. This figure illustrates the statistical distribution of transition action counts on individual interfaces across different mobile applications(Due to space limitations, we present only a subset of the statistical results).

Appendix 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.

 doi:10.20944/preprints202603.1313.v1

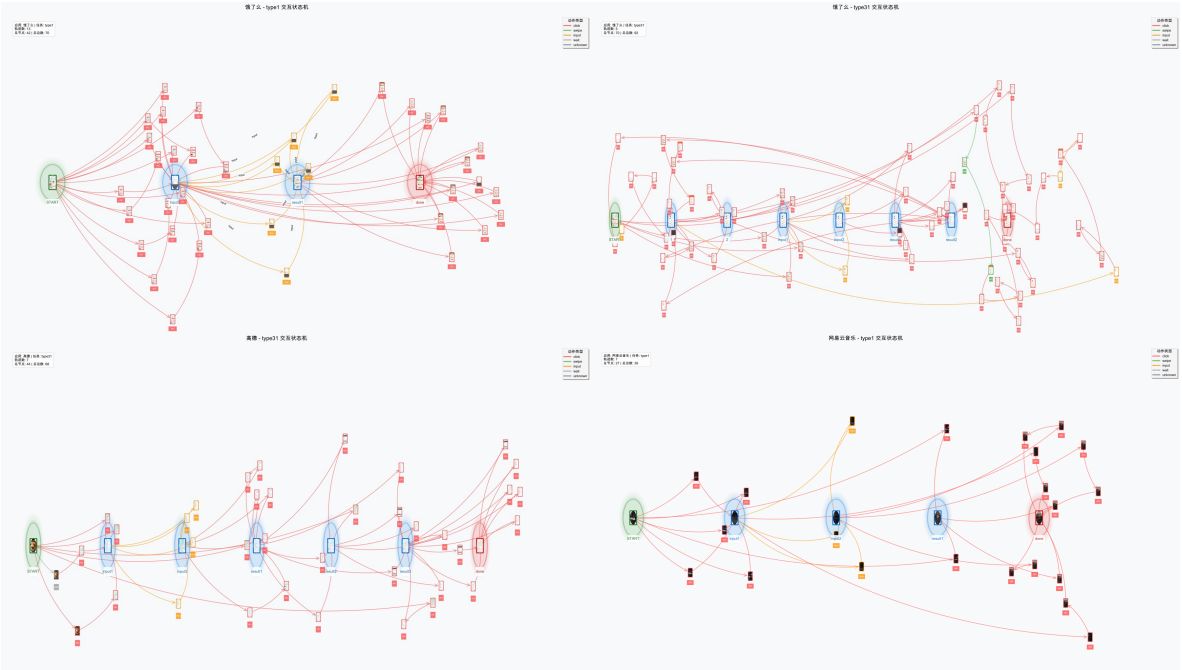

Figure 11. Visualization Examples of Task State Transition Graphs, including Different Tasks from Ele.me, AutoNavi, and NetEase Cloud Music Apps.

Appendix F. Execution Details of General Models

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

 doi:10.20944/preprints202603.1313.v1

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.
```

 doi:10.20944/preprints202603.1313.v1


```
    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]."

 doi:10.20944/preprints202603.1313.v1

- 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.

 doi:10.20944/preprints202603.1313.v1

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**:
   - On a date selection interface, you must first confirm if the currently displayed month is correct.
   - Do not just click an identical date number; you must ensure the month matches the task requirement.
   - If the month is wrong, you need to switch to the correct month first, then select the date.
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
```

 doi:10.20944/preprints202603.1313.v1


```
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"}
```

The above 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.

Figure 12. Execution Example of General Models: General models leverage icon recognition models for enhancement.

 doi:10.20944/preprints202603.1313.v1

Appendix G. Generalization vs. Accuracy

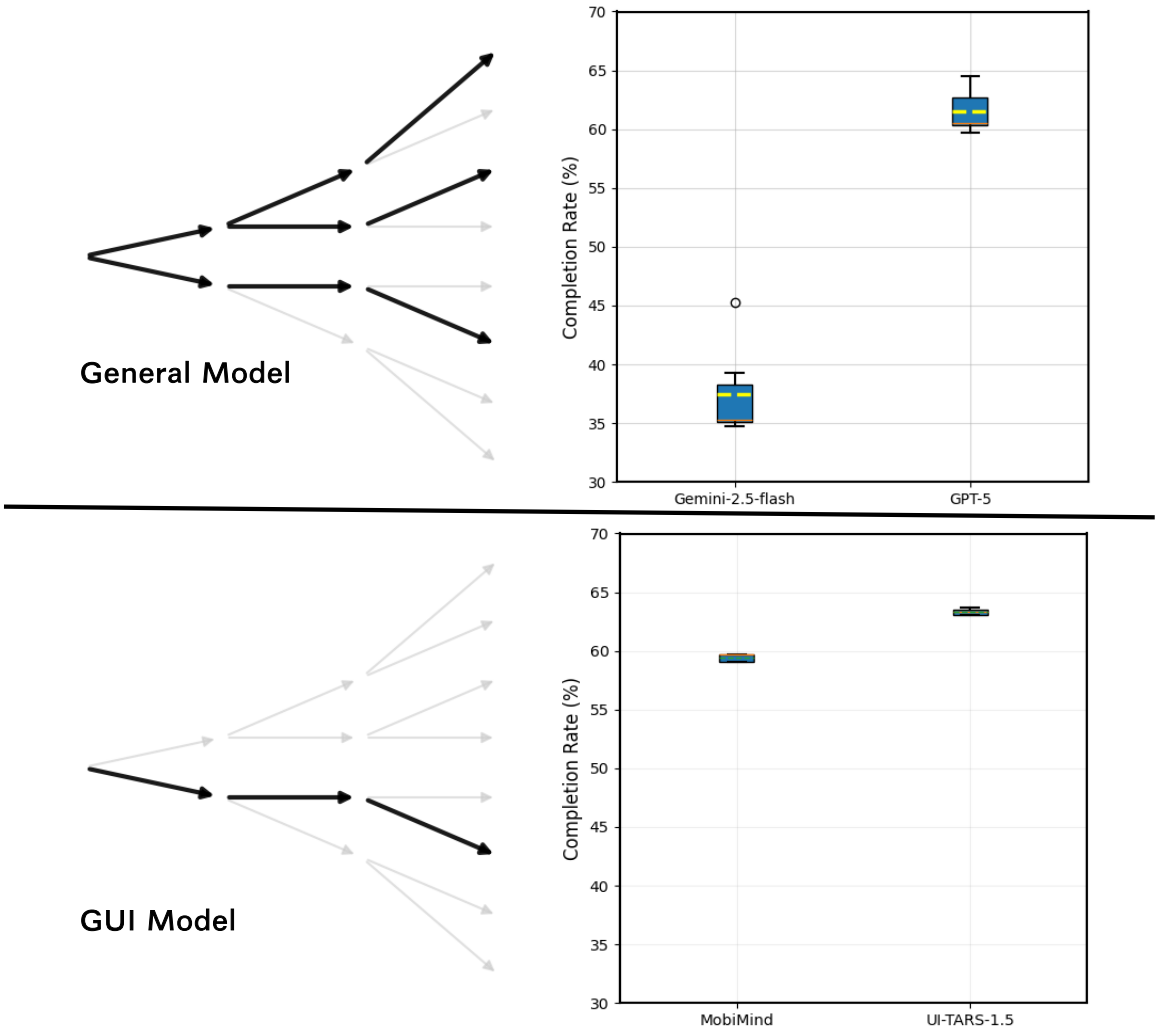

Figure 13. The tension between generalization capability and determinism. The upper figure presents the performance scores of a general-purpose model across multiple sampling trials on randomly selected tasks. The lower figure shows the sampling results of a GUI-specialized model on the same set of tasks.

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.

Appendix H. Resolution Scaling

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.

 doi:10.20944/preprints202603.1313.v1

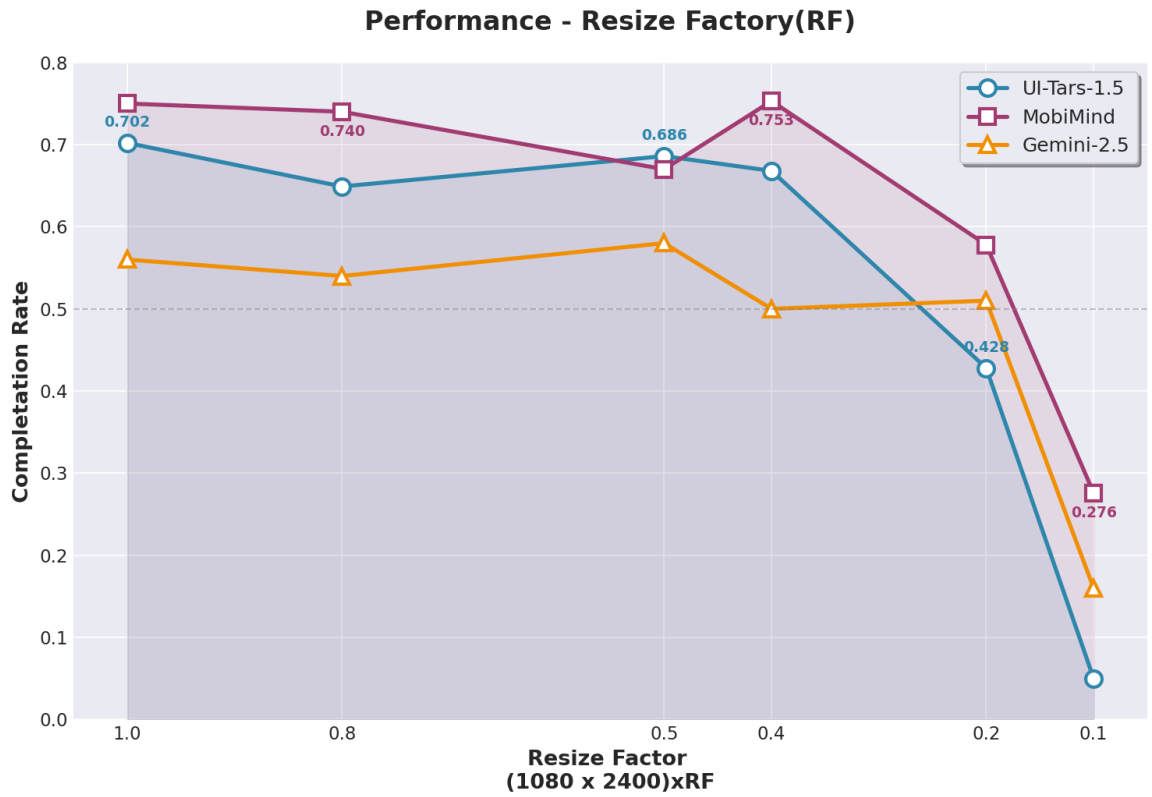

Figure 14. Completion rate of different models under varying resolution scaling factors. We randomly sample representative examples and evaluate them at different scaling levels based on a resolution of 1080×2400.

References

Anthropic. Claude opus 4.1. Large language model, https://www.anthropic.com, 2025. Accessed: [Insert Full Date You Accessed the Model, e.g., 6 Jan. 2026].

Bu, W., Wu, Y., Yu, Q., Gao, M., Miao, B., Zhang, Z., Pan, K., Li, Y., Li, M., Ji, W., et al. What limits virtual agent application? omnibench: A scalable multi-dimensional benchmark for essential virtual agent capabilities. *arXiv preprint arXiv:2506.08933*, 2025.

Chai, Y., Huang, S., Niu, Y., Xiao, H., Liu, L., Wang, G., Zhang, D., Ren, S., and Li, H. AMEX: Android multi-annotation expo dataset for mobile GUI agents. In Che, W., Nabende, J., Shutova, E., and Pilehvar, M. T. (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/.

Chai, Y., Li, H., Zhang, J., Liu, L., Liu, G., Wang, G., Ren, S., Huang, S., and Li, H. A3: Android agent arena for mobile gui agents. *arXiv preprint arXiv:2501.01149*, 2025b.

Chen, J., Yuen, D., Xie, B., Yang, Y., Chen, G., Wu, Z., Yixing, L., Zhou, X., Liu, W., Wang, S., et al. Spa-bench: A comprehensive benchmark for smartphone agent evaluation. In *NeurIPS 2024 Workshop on Open-World Agents*, 2024.

Comanici, G., Bieber, E., Schaekermann, M., Pasupat, I., Sachdeva, N., Dhillon, I., Blistein, M., Ram, O., Zhang, D., Rosen, E., 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.

Fabbri, A. R., Mares, D., Flores, J., Mankikar, M., Hernandez, E., Lee, D., Liu, B., and Xing, C. Multinrc: A challenging and native multilingual reasoning evaluation benchmark for llms. *arXiv preprint arXiv:2507.17476*, 2025.

He, Y., Jin, D., Wang, C., Bi, C., Mandyam, K., Zhang, H., Zhu, C., Li, N., Xu, T., Lv, H., et al. Multi-if: Benchmarking llms on multi-turn and multilingual instructions following. *arXiv preprint arXiv:2410.15553*, 2024.

Kong, Q., Zhang, X., Yang, Z., Gao, N., Liu, C., Tong, P., Cai, C., Zhou, H., Zhang, J., Chen, L., et al. Mobileworld: Benchmarking autonomous mobile agents in agent-user interactive, and mcp-augmented environments. *arXiv preprint arXiv:2512.19432*, 2025.

 doi:10.20944/preprints202603.1313.v1

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Lee, J., Min, T., An, M., Hahm, D., Lee, H., Kim, C., and Lee, K. Benchmarking mobile device control agents across diverse configurations. *arXiv preprint arXiv:2404.16660*, 2024.

Lee, J., Min, T., An, M., Hahm, D., Lee, H., Kim, C., and Lee, K. Benchmarking mobile device control agents across diverse configurations, 2025. URL https://arxiv.org/abs/2404.16660.

Leung, H. F., Xi, X., and Zuo, F. Androidcontrol-curated: Revealing the true potential of gui agents through benchmark purification. *arXiv preprint arXiv:2510.18488*, 2025.

Liu, X., Qin, B., Liang, D., Dong, G., Lai, H., Zhang, H., Zhao, H., Iong, I. L., Sun, J., Wang, J., et al. Autoglm: Autonomous foundation agents for guis. *arXiv preprint arXiv:2411.00820*, 2024.

Lu, Y., Yang, J., Shen, Y., and Awadallah, A. Omniparser for pure vision based gui agent, 2024. URL https://arxiv.org/abs/2408.00203.

Ma, X., Zhang, Z., and Zhao, H. Coco-agent: A comprehensive cognitive mllm agent for smartphone gui automation. *arXiv preprint arXiv:2402.11941*, 2024.

OpenAI. Conversation with an ai model [generative ai chat]. Available at: chat.openai.com, 2025. Accessed: [Date you accessed the tool].

Qin, Y., Ye, Y., Fang, J., Wang, H., Liang, S., Tian, S., Zhang, J., Li, J., Li, Y., Huang, S., et al. Ui-tars: Pioneering automated gui interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025.

Rawles, C., Li, A., Rodriguez, D., Riva, O., and Lillicrap, T. Androidinthewild: A large-scale dataset for android device control. *Advances in Neural Information Processing Systems*, 36:59708–59728, 2023.

Rawles, C., Clinckemaillie, S., Chang, Y., Waltz, J., Lau, G., Fair, M., Li, A., Bishop, W., Li, W., Campbell-Ajala, F., et al. Androidworld: A dynamic benchmarking environment for autonomous agents. *arXiv preprint arXiv:2405.14573*, 2024.

Song, Y., Huang, H., Lin, Q., Zhao, Y., Qu, X., Wang, J., Lou, X., Liu, W., Zhang, Z., Yu, Y., et al. Colorbench: Benchmarking mobile agents with graph-structured framework for complex long-horizon tasks. *arXiv preprint arXiv:2510.14621*, 2025.

Toyama, D., Hamel, P., Gergely, A., Comanici, G., Glaese, A., Ahmed, Z., Jackson, T., Mourad, S., and Precup, D. Androidenv: A reinforcement learning platform for android. *arXiv preprint arXiv:2105.13231*, 2021.

Wang, H., Zou, H., Song, H., Feng, J., Fang, J., Lu, J., Liu, L., Luo, Q., Liang, S., Huang, S., et al. Ui-tars-2 technical report: Advancing gui agent with multi-turn reinforcement learning. *arXiv preprint arXiv:2509.02544*, 2025.

Wang, L., Deng, Y., Zha, Y., Mao, G., Wang, Q., Min, T., Chen, W., and Chen, S. Mobileagentbench: An efficient and user-friendly benchmark for mobile llm agents. *arXiv preprint arXiv:2406.08184*, 2024.

Xie, T., Zhang, D., Chen, J., Li, X., Zhao, S., Cao, R., Hua, T. J., Cheng, Z., Shin, D., Lei, F., et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

Xing, M., Zhang, R., Xue, H., Chen, Q., Yang, F., and Xiao, Z. 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.

Xu, W., Jiang, Z., Liu, Y., Gao, P., Liu, W., Luan, J., Li, Y., Liu, Y., Wang, B., and An, B. Mobile-bench-v2: A more realistic and comprehensive benchmark for vlm-based mobile agents. *arXiv preprint arXiv:2505.11891*, 2025a.

Xu, Y., Liu, X., Sun, X., Cheng, S., Yu, H., Lai, H., Zhang, S., Zhang, D., Tang, J., and Dong, Y. 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.

Yan, Y., Wang, S., Du, J., Yang, Y., Shan, Y., Qiu, Q., Jia, X., Wang, X., Yuan, X., Han, X., et al. Mcpworld: A unified benchmarking testbed for api, gui, and hybrid computer use agents. *arXiv preprint arXiv:2506.07672*, 2025.

Ye, J., Zhang, X., Xu, H., Liu, H., Wang, J., Zhu, Z., Zheng, Z., Gao, F., Cao, J., Lu, Z., Liao, J., Zheng, Q., Huang, F., Zhou, J., and Yan, M. Mobile-agent-v3: Fundamental agents for gui automation, 2025a. URL https://arxiv.org/abs/2508.15144.

Ye, J., Zhang, X., Xu, H., Liu, H., Wang, J., Zhu, Z., Zheng, Z., Gao, F., Cao, J., Lu, Z., et al. Mobile-agent-v3: Fundamental agents for gui automation. *arXiv preprint arXiv:2508.15144*, 2025b.

Zhang, C., He, S., Qian, J., Li, B., Li, L., Qin, S., Kang, Y., Ma, M., Liu, G., Lin, Q., et al. Large language model-brained gui agents: A survey. *arXiv preprint arXiv:2411.18279*, 2024a.

Zhang, C., Feng, E., Zhao, X., Zhao, Y., Gong, W., Sun, J., Du, D., Hua, Z., Xia, Y., and Chen, H. Mobiagent: A systematic framework for customizable mobile agents. *arXiv preprint arXiv:2509.00531*, 2025.

 doi:10.20944/preprints202603.1313.v1

Zhang, D., Xu, H., Zhao, Z., Chen, L., Cao, R., and Yu, K. Mobile-env: an evaluation platform and benchmark for llm-gui interaction. *arXiv preprint arXiv:2305.08144*, 2024b.

Zhang, J., Wu, J., Yihua, T., Liao, M., Xu, N., Xiao, X., Wei, Z., and Tang, D. Android in the zoo: Chain-of-action-thought for GUI agents. In Al-Onaizan, Y., Bansal, M., and Chen, Y.-N. (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/.

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E., et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36: 46595–46623, 2023.