AndroidLab: Developing and Evaluating Android Agents in A Reproducible Environment

Anonymous ACL submission

Abstract

Autonomous agents have become increasingly important for interacting with the real world. Android agents, in particular, have been recently a frequently-mentioned interaction method. However, existing studies for training and evaluating Android agents lack systematic research on both open-source and closed-source models. In this work, we propose ANDROIDLAB as a systematic Android agent framework. It includes an operation environment with different modalities, action space, and a reproducible benchmark. It supports both large language models (LLMs) and multimodal models (LMMs) in the same action space. ANDROIDLAB benchmark includes predefined Android virtual devices and 138 tasks across nine apps built on these devices. By using the ANDROIDLAB environment, we develop an Android Instruction dataset and train six open-source LLMs and LMMs, lifting the average success rates from 5.07% to 25.60% for LLMs and from 1.69% to 14.98% for LMMs. ANDROIDLAB is open-sourced and publicly available at `https://anonymous.4open.science/r/Android-Lab-Reivew-C93E`.

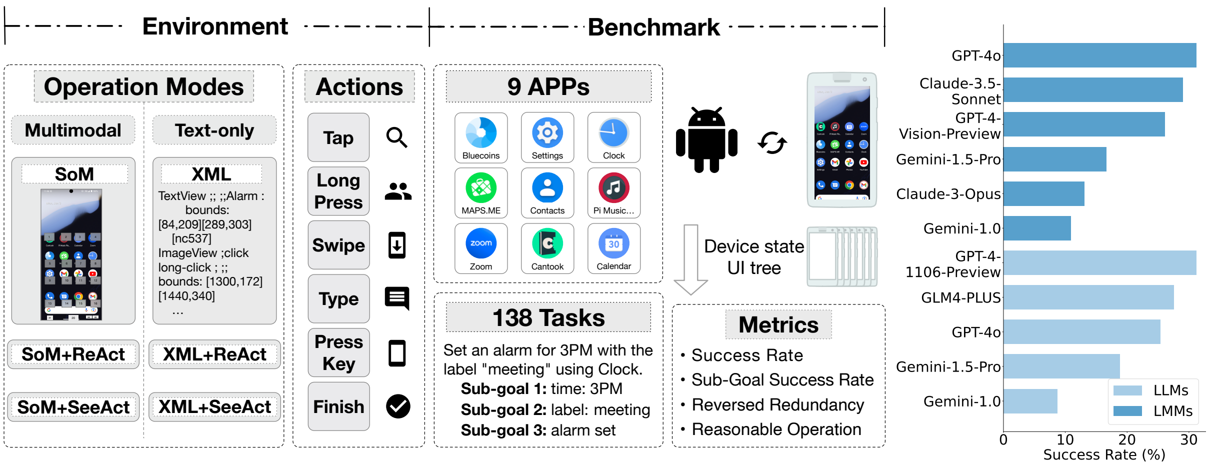
1 Introduction

Developing autonomous agents to execute human instructions within mobile operating systems has long been a goal for researchers (Burns et al., 2021; Yang et al., 2023b; Wang et al., 2023; Hong et al., 2023; Rawles et al., 2023; Li et al., 2020a; Romao et al., 2019; Rai et al., 2019). Recently, a significant line of research has focused on using large language models (LLMs) (Zeng et al., 2022; OpenAI, 2023; Anthropic, 2023; Team et al., 2024) and large multimodal models (LMMs) (OpenAI, 2023; Anthropic, 2023; Hong et al., 2023) as the backbone for these agents (Deng et al., 2023; Rawles et al., 2023; Zhou et al., 2023).

Despite these advancements, the lack of a reasonable and fair benchmark to evaluate mobile agents presents a critical challenge. Previous benchmarks (Rawles et al., 2023; Sun et al., 2022; Li et al., 2020a) usually provide static environments, requiring agents to predict the next action based on screenshots. For example, Android Env (Toyama et al., 2021) defines the agent's action space and state for an operable Android operation environment. Following works (Yang et al., 2023b; Xing et al., 2024; Lee et al., 2024) construct benchmarks based on this environment. However, most of them rely on online software, making the tests non-reproducible. In summary, these benchmarks still have the following issues:

- **Non-reproducibility due to dynamic environments.** Existing benchmarks (Toyama et al., 2021; Kapoor et al., 2024; Li et al., 2020b) set tasks in dynamic environments, such as those involving real-time information or social media, making these benchmarks non-reproducible.
- **Inability to simulate multiple completion paths for a task.** Existing works (Burns et al., 2021; Sun et al., 2022; Rawles et al., 2023; Deng et al., 2023; Xing et al., 2024) provide standard operation sequences or use metrics such as single-step accuracy or similarity of operation sequences, but fail to simulate multiple paths to complete a task.

These issues have motivated us to develop a new Android agent evaluation and training framework. In this paper, we propose **ANDROIDLAB**, which includes a standard operational environment and a benchmark for agents interacting with Android devices. We define basic operation modes across LLMs and LMMs by aligning actions and objects within different observations of the mobile system: XML and screenshots, referred to as XML mode and SoM mode, respectively. Additionally, we introduce two modes for each basic mode, ReAct (Yao et al., 2022) and SeeAct (Zheng et al., 2024). Node information is annotated in the XML for screenshots using set-of-mark (Yang et al.,



(a) Overview of the environment and benchmark of ANDROIDLAB.

(b) Results of Close Models.

Figure 1: (a) We design the SoM mode for the multimodal models (LMMs) and the XML mode for the text-only models (LLMs), ensuring an identical action space. We also implement ReAct and SeeAct frameworks in both modes. Based on the environment, we propose the ANDROIDLAB benchmark. (b) ANDROIDLAB Success Rates of closed-source models. In the XML mode, GPT-4-1106-Preview has the highest success rate at 31.16%, matching GPT-4o's performance in the SoM mode.

2023a), ensuring identical actions across modes for a fair comparison. Based on the environment, the ANDROIDLAB benchmark includes 138 tasks across nine different apps. By using Android virtual devices with preloaded app operation histories and offline data, ANDROIDLAB benchmark ensures reproducibility and eliminates dependencies on external networks or time.

Previous benchmarks had limitations in their evaluation metrics. In the ANDROIDLAB benchmark, each task is divided into multiple required page states as sub-goals. Correct trajectories are verified using UI tree structure matching or device state validation. This approach allows for precise assessments of task completion and progress without being influenced by the specific paths taken to achieve sub-goals, offering flexibility in the sequence of actions. Additionally, we introduce metrics such as reversed redundancy and reasonable operation to evaluate the efficiency of actions.

We have evaluated 17 open-source and closed-source models using the ANDROIDLAB benchmark. Although the GPT series achieved over 30% success rate in both XML and SoM modes, we observed that open-source models performed poorly, with the best reaching only around 5% success rate. Initial attempts to enhance mobile agent performance through more complex reasoning frameworks led to marginal improvements despite significantly increased inference times. Therefore, fine-tuning small-scale open-source models may bridge the gap to closed-source performance, enhancing mobile agent accessibility.

By using ANDROIDLAB's operation modes and action space, we have constructed the Android Instruct dataset. We develop an online annotation tool with the same action space, collecting 10.5k traces and 94.3k steps from annotators. Among these, 6208 steps are derived from the Apps included in the ANDROIDLAB benchmark, and we use this portion of the data to fine-tune the model. This dataset includes tasks, phone screen states, XML information, and operations, which have been used to fine-tune six text-only and multimodal models. As shown in Figure 3, fine-tuning with our dataset raises average success rates from 5.07% to 25.60% for LLMs and from 1.69% to 14.98% for LMMs. Our further analysis reveals that fine-tuning improves operational accuracy, efficiency, and reduces redundancy in Android agents.

The contributions are summarized as follows:

- We design the ANDROIDLAB suite, which includes an operational environment and a benchmark, which unifies the evaluation and development of Android Agents, as shown in Figure 1.
- We develop ANDROIDLAB benchmark, a reproducible and challenging benchmark for evaluating mobile agent. It includes a simulated evaluation environment and 138 tasks, as shown in Figure 2 based on text-only or multimodal inputs. ANDROIDLAB benchmark presents significant challenges, with the leading model

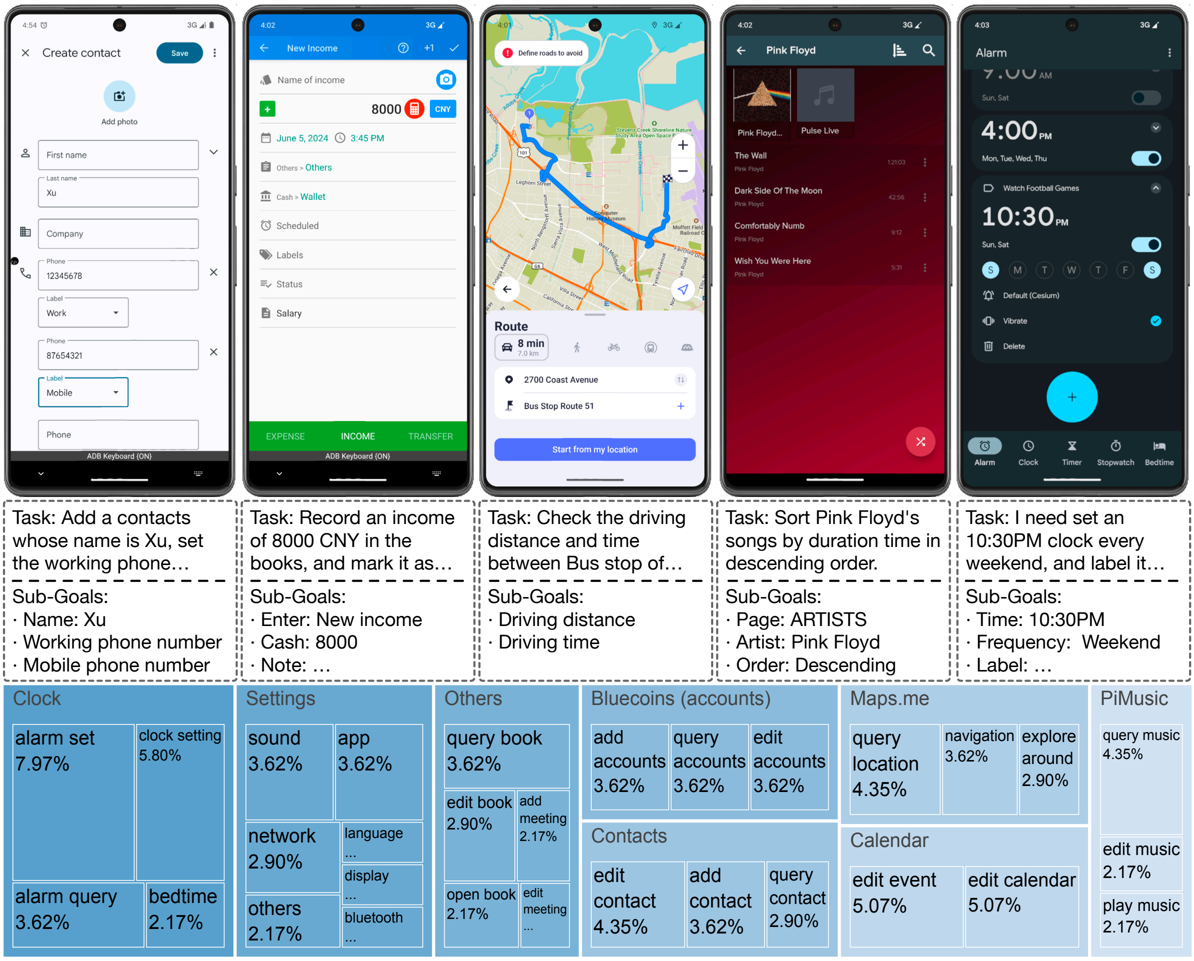


Figure 2: Task examples and the distribution of all apps and subcategories in the BENCHMARK benchmark. We decomposed each task into sub-goals and evaluated them independently. A task is considered complete only if all sub-goals are correctly addressed.

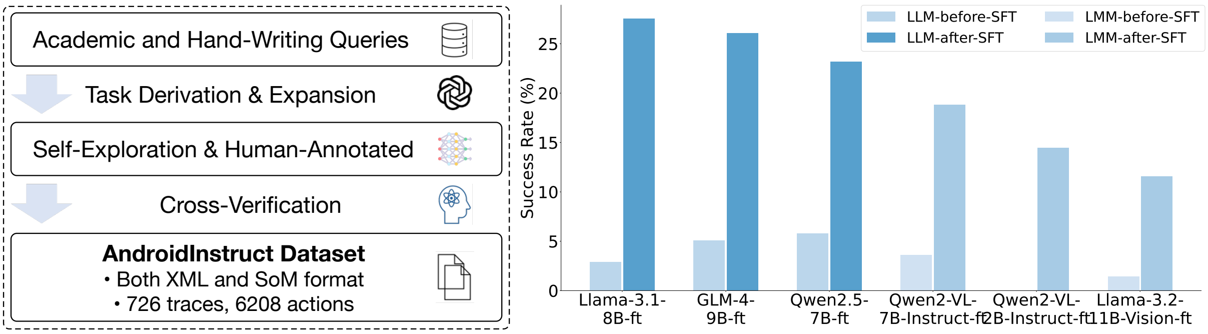
GPT-4o, achieving only 31.16% success.

- We construct an Android Instruct dataset, containing 94.3k operation records for fine-tuning. This dataset supports both text-only and multimodal training, yielding competitive results in LLMs and LMMs, as shown in Table 2. We also demonstrate that fine-tuned models achieve comparable scores and offer the best balance of efficiency and accuracy.

2 Related Work

Benchmarks for Mobile Agents. Mobile benchmarks for Android began with static systems like PixelHelp (Li et al., 2020a) and MetaGUI (Sun et al., 2022) and later expanded through AITW (Rawles et al., 2023), which provided over 5 million images. AndroidEnv (Toyama et al., 2021) introduced dynamic evaluations, while Android Arena (Xing et al., 2024) added cross-app evaluations. Although task diversity was limited, B-MOCA (Lee et al., 2024) standardized the Android Virtual Device. AndroidWorld (Rawles et al., 2024) offers reward signals for 116 tasks across 20 real-world apps, but does not support instruction-tuning data construction. Our benchmark provides a challenging and reproducible environment with direct interaction capabilities. Table 1 compares ANDROIDLAB benchmark to other benchmarks.

Agents for Interactive System. For Web environments, WebGPT (Nakano et al., 2021) and WebGLM (Liu et al., 2023) integrate LLMs for improved question-answering. MindAct (Deng et al., 2023), WebAgent (Gur et al., 2023), and AutoWebGLM (Lai et al., 2024) focus on executing complex interactive tasks. In mobile agents, early work on Android systems utilized multiple execution modules (Burns et al., 2021; Venkatesh et al., 2023; Li et al., 2020a; Zhan and Zhang, 2023). PixelHelp (Li et al., 2020a) mapped actions to images,



(a) Overview of ANDROIDINSTRUCT data collection. (b) Success Rates of before and after fine-tuned by ANDROIDINSTRUCT.

Figure 3: (a) We have collected over 726 trajectories containing more than 6208 fully aligned steps of XML and SoM mode training data. (b) By using ANDROIDINSTRUCT, we trained six open-source text-only and multimodal models, achieving an average increase of 504% and 885%, respectively, reaching a performance level comparable to proprietary models.

while Auto-GUI (Zhan and Zhang, 2023) used image and text encoders with LLMs for CoT (Chain of thoughts)outputs. CogAgent (Hong et al., 2023) achieved SOTA on AITW (Rawles et al., 2023) by combining modules for action prediction. Recent zero-shot mobile agents using GPT-4V (OpenAI, 2023) have shown strong results (Yang et al., 2023b; Zheng et al., 2024; Yan et al., 2023; Wang et al., 2023), but planning complexity limits inference speed and practical deployability due to security restrictions.

3 ANDROIDLAB

3.1 The Operation Environment

ANDROIDLAB defines a set of action space and two operation modes, forming the ANDROIDLAB environment. We adopt the main action space from prior work and add a model return value (finish action). The two basic operation modes are SoM (Yang et al., 2023a) and XML, differing in whether the agent can access a snapshot of the phone screen. For comparison, we also implement ReAct (Yao et al., 2022) and SeeAct (Zheng et al., 2024). This framework supports real and virtual Android devices and is compatible with Android-like mobile operating systems.

Table 1: Comparsion of different Android benchmarks.

	Virtual Env	Reprod-ucibility	Sub-goal Evaluation	Support Query Task	Containing Training Set	Metric
PixelHelp		✓			✓	Sequence match
AITW		✓			✓	Single step ACC
Android Env	✓					Single step ACC
Android Arena	✓					Sequences LCS
B-MOCA	✓	✓				Device state
ANDROIDLAB benchmark	✓	✓	✓	✓	✓	Device state& UI tree

Action Space. Based on the action spaces from AppAgent (Yang et al., 2023b) and Android Env (Toyama et al., 2021), we define four basic phone operations: `Tap`, `Swipe`, `Type`, `Long Press`, along with two shortcut keys, `Home` and `Back`, as the core action space. We add the `Finish` action as the final step, allowing the agent to return execution results or answers. This action space applies to all modes

XML Mode. XML mode is tailored for text-only input models (LLMs). Inspired by Android Arena (Xing et al., 2024), we redesign the XML compression algorithm (Cf. Appendix C) to convey screen information. The LLMs select corresponding elements directly for operations.

SoM Mode. SoM mode is for multimodal input models (LMMs), based on the Set-of-Mark method (Yang et al., 2023a). Each clickable or focusable element is assigned a serial number, and the LMMs select the element by its number. The selected elements in SoM mode align with those in the compressed XML list, allowing both modes to interact with the same action space and objects.

These basic operation modes directly require the agent to output operation commands. Based on these two methods, we further test two novel agent frameworks, ReAct (Yao et al., 2022) and SeeAct (Zheng et al., 2024). These two frameworks allow the agent to observe and reflect on the environment or more easily select specific tasks to execute. Please refer to Appendix B for more details about our operation modes.

ReAct Modes. Based on the above two modes, we follow (Yao et al., 2022) to prompt the model, allowing models to think step by step and output their thought and reasoning process before outputting the action. We name the corresponding two modes as XML+ReAct and SoM+ReAct.

SeeAct Modes. Following (Zheng et al., 2024), we separate the reasoning and element grounding processes. We instruct models to interact for two rounds in a single operation. The models are supposed to generate a detailed description of the desired action and output the real action, respectively. We name these two modes as XML+SeeAct and SoM+SeeAct.

3.2 The Reproducible Benchmark

Based on ANDROIDLAB's environment, ANDROIDLAB benchmark offers a deterministic and reproducible evaluation platform, allowing users to perform fair and challenging comparisons of Android agent capabilities. ANDROIDLAB benchmark introduces the following designs:

- We gathered 138 tasks from nine apps, ensuring reproducibility. These tasks, derived from common mobile scenarios, are divided into two types: (a) Operation Tasks, where agents must complete a series of actions to meet a goal, and (b) Query Tasks, where agents answer queries based on phone information.
- Using UI tree structure in the XML file, we identify screen information that uniquely defines task completion, making task completion our primary metric. Therefore, our approach allows us to directly evaluate the completion status without considering the path to reach them, thus enabling the simulation of multiple completion paths. Additionally, we select auxiliary metrics such as the proportion of valid actions and the redundancy of successful operation sequences.

3.2.1 Task Formulation

We formalize each task input as a 4-tuple: Task(E, I, F, M). Here, E represents the execution environment of the task, which, in the context of benchmark testing, is the pre-packaged AVD (Android virtual device) image. This includes a fixed phone screen size, Android version, API level, and a fixed app usage state. I denotes the specific natural language instruction for the task. To avoid confusion during testing, we specify the app required to complete the task in natural language. F represents the agent testing framework. Finally, M denotes the backbone model used to perform the task, referring primarily to LLMs or LMMs.

Thus, we can formally define the two types of tasks included in ANDROIDLAB benchmark:

Operation Task. $\mathrm{T}(E, I, F, M) \rightarrow (S_1, \ldots, S_n)$. The output of this type of task is a sequence of continuous Android virtual machine states.

Query Task. $\mathrm{T}(E, I, F, M) \rightarrow (S_1, \ldots, S_n, A)$. This type of task assesses the agent's ability to answer specific questions based on the state sequence after exploration. The model must explore the environment to find the answers and output the correct response.

Based on the above formulation, we designed 138 tasks, including 93 Operation Tasks and 45 Query Tasks. Please refer to Appendix A for detailed information.

3.2.2 Reproducible Designs

To ensure our evaluation reflects real-world agent usage scenarios with an appropriate level of difficulty and full reproducibility, we design the tasks with the following considerations:

- **Fixed Evaluation Time and Space**: We use ADB (Android debug bridge) commands at the start of each evaluation to set the machine's time and virtual geolocation to predetermined values.
- **Offline Testing**: All test apps function offline, with preloaded records in the AVD image to ensure usability without an internet connection.
- **Predefined Answers**: For query tasks, we conduct operations on the corresponding apps in advance to guarantee uniquely determined correct results.

3.2.3 Metrics

Previous evaluations with virtual environments have relied on indirect metrics like single-step accuracy and operation path matching, leading to imprecise assessments. In response, ANDROIDLAB benchmark introduces a task-completion-based evaluation system that judges directly from device and screen states. We provide an example of an agent completing all sub-goals of a task in Fig 4. Our key metrics are:

- **Success Rate (SR)**: Measures the overall task completion rate across all tasks, representing the average success rate.
- **Sub-Goal Success Rate (Sub-SR)**: Evaluates the completion of sub-goals within tasks, rewarding models with stronger understanding and operational capabilities.
- **Reversed Redundancy Ratio (RRR)**: Assesses

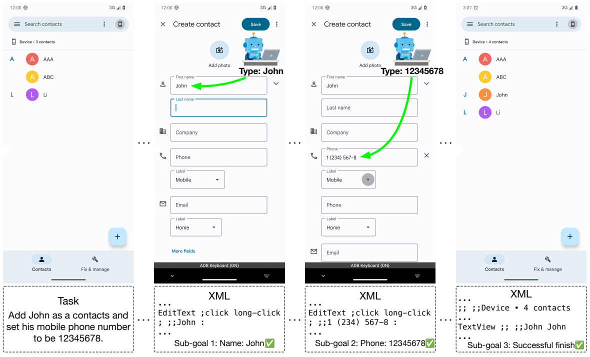


Figure 4: An example of an agent completing all sub-goals of a task, showing only the starting and ending steps, as well as sub-goal completion points. By focusing solely on these points, our method simulates multiple completion paths without tracking how the agent reaches them.

the redundancy of the model's operation path compared to a human operator's path, indicating efficiency.

- **Reasonable Operation Ratio (ROR)**: Measures the proportion of operations that result in a screen change, with unchanged screens considered unreasonable.

Due to the length constraints of the paper, the detailed definitions of our metrics can be found in the Appendix D, where we provide formal definitions and relevant examples. By incorporating these metrics, our evaluation system provides a comprehensive and precise assessment of an agent's performance in completing specified tasks.

4 The Android Instruction Data

Previous work on Android agents focuses on using powerful closed-source models to design interaction logic (Zheng et al., 2024; Yang et al., 2023b; Wang et al., 2023), raising concerns about accessibility and privacy. To address this, we aim to build an open-source mobile agent. The main challenge lies in generating training data for mobile operations to handle open-world tasks in diverse environments.

We propose task derivation and expansion methods for task generation, allowing models to generate tasks for specific apps controllably. ANDROIDLAB connects to devices via ADB, enabling compatibility with various real or virtual devices for data generation. Using self-exploration and manual annotation, we generate example operation trajectories. Our Android Instruction data is built on the $\mathrm{T}(E, I) \rightarrow (S_1, \ldots, S_n, A)$ framework within ANDROIDLAB's environment, but this does not include evaluation scripts and is annotated by human annotators.

4.1 Data Construction

The primary challenges in data construction include generating executable Android instructions and annotating operation path data. Our approach involves three steps:

- **Task Derivation and Expansion**: Tasks were generated using academic datasets and language models, with manual checks to ensure realism and executability.
- **Self-Exploration Reward Model Construction**: Advanced LLMs and LMMs autonomously completed tasks, and a reward model was constructed based on combined image inputs, achieving 87.64% accuracy.
- **Manual Annotation**: Involved four steps: (1) instruction feasibility check, (2) preliminary app exploration, (3) task execution and documentation, and (4) cross-verification by a second annotator and reward model.

Please refer to Appendix I for more details of the data construction process. This combination of autonomous and manual processes resulted in 10.5k trajectories and 94.3k steps, and we use 726 trajectories and 6208 steps derived from the Apps included in the ANDROIDLAB benchmark for training. Each trajectory includes the specific task instruction, the device state at each step (including screenshots and XML files), and the action for the current step. We provide statistics of the Android Instruct dataset in Fig 20.

5 Experiments

5.1 Experiment Setup

Evaluation Settings. In preliminary tests, agents often failed to complete tasks due to issues with launching the specified apps correctly. To avoid this, we started tasks directly within the specified app during formal experiments and then allowed the agent to proceed. We also set a 25-step limit for each task, with a 3-second interval for the virtual machine to respond to each operation. Tasks were generated by greedy search for each model.

Baseline Models. For large language models (LLMs) with text-only input capability, we selected the following closed-source models: GPT-4o (OpenAI, 2023), GPT-4-1106-Preview (OpenAI,

Table 2: **Main Result of XML and SoM modes.** SR, Sub-SR, RRR, and ROR stand for Success Rate, Sub-Goal Success Rate, Reversed Redundancy Ratio, and Reasonable Operation Ratio, respectively. For all these metrics, a higher value means better. **-ft** represents a finetuned model. In each mode, **Bold** represents the best result. We do not report RRR score if SR < 5.

Mode	Model	SR	Sub-SR	RRR	ROR
XML	GPT-4o	25.36	30.56	**107.45**	86.56
	GPT-4-1106-Preview	**31.16**	**38.21**	66.34	86.24
	Gemini-1.5-Pro	18.84	22.40	57.72	83.99
	Gemini-1.0	8.70	10.75	51.80	71.08
	GLM-4-PLUS	18.12	22.66	84.83	83.41
	LLaMA3.1-8B-Instruct	2.90	4.71	23.73	69.85
	Qwen2.5-7B-Instruct	5.07	5.80	22.75	66.96
	GLM4-9B-Chat	7.25	9.06	54.43	58.34
XML+SFT	LLaMA3.1-8B-**ft**	27.54	35.27	77.19	**89.86**
	Qwen2.5-7B-**ft**	26.09	35.31	81.70	89.50
	GLM-4-9B-**ft**	23.19	29.47	75.99	86.76
SoM	GPT-4o	**31.16**	**35.02**	87.32	85.36
	GPT-4-Vision-Preview	26.09	29.53	99.22	78.79
	Gemini-1.5-Pro	16.67	18.48	105.95	91.52
	Gemini-1.0	10.87	12.56	72.52	76.70
	Claude-3.5-Sonnet	28.99	32.66	**113.41**	81.16
	Claude-3-Opus	13.04	15.10	81.41	83.89
	LLaMA3.2-11B-Vision-Instruct	1.45	1.45	-	50.76
	Qwen2-VL-2B-Instruct	0.00	1.09	-	30.25
	Qwen2-VL-7B-Instruct	3.62	4.59	-	84.81
SoM+SFT	LLaMA3.2-11B-Vision-**ft**	11.59	14.01	63.76	86.08
	Qwen2-VL-2B-Instruct-**ft**	14.49	20.53	62.83	92.41
	Qwen2-VL-7B-Instruct-**ft**	18.84	22.58	77.62	**92.42**

2023), Gemini-1.5-Pro (Team et al., 2024), Gemini-1.0 (Team et al., 2024), and GLM-4-PLUS (GLM et al., 2024). The open-source models included as baselines for testing in the XML mode are LLaMA3.1-8B-Instruct (Touvron et al., 2023), GLM-4-9B-Chat (GLM et al., 2024), and Qwen2.5-7B-Instruct (Bai et al., 2023). For large multimodal models (LMMs) with image input capability, we selected the following closed-source models: GPT-4o (OpenAI, 2023), GPT-4-Vision-Preview (OpenAI, 2023), Gemini-1.5-Pro (Team et al., 2024), Gemini-1.0 (Team et al., 2024), Claude-3.5-Sonnet, and Claude-3-Opus. The open-source models in this category included LLaMA3.2-11B-Vision-Instruct (Touvron et al., 2023), Qwen2-VL-7B-Instruct, and Qwen2-VL-2B-Instruct (Wang et al., 2024). Fine-tuned versions of all six open-source models (denoted with "-ft") were also evaluated under the XML or SoM+SFT setting.

Training Settings. To explore the effectiveness of our dataset on lightweight open-source models, we selected all six open-source models above as the training backbones for LLMs and LMMs, respectively. Due to our preliminary experiments showing that training agents from base models yielded better results, we selected the base versions of all models for fine-tuning, except for Qwen2.5-VL-7B-Instruct (as no open-source base model was available). However, we still reported the instruct versions as baselines because the base models could not follow instructions without further tuning. For all training sessions, we used a batch size of 32 and a maximum sequence length of 4096, training for five epochs. The learning rate was set to 1e-5.

5.2 Main Results

As shown in Table 2, in the XML mode, GPT-4-1106-Preview outperforms the other models with a Success Rate (SR) of 31.16%, the highest in this mode while also achieving the best Sub-Goal Success Rate (Sub-SR) at 38.21%. Although GPT-4o exhibits slightly lower SR (25.36%), it achieves the highest Reversed Redundancy Ratio (RRR) at 107.45, indicating its strong ability to reduce unnecessary operations. The ROR metric shows that both models in the GPT-4 series perform comparably, with around 86% of operations being reasonable, though there is room for improvement in efficiency. Other models, such as Gemini-1.5-Pro and GLM-4-PLUS, show moderate performance, with ROR around 84 but lag in SR.

In the SoM mode, GPT-4o again shows dominance, reaching an SR of 31.16% and a Sub-SR of 35.02%, the highest in both categories. GPT-4-Vision-Preview follows closely, but models like Claude-3.5-Sonnet exceed GPT-4o in RRR (113.41), demonstrating higher efficiency in task completion with fewer redundant steps. The Reasonable Operation Ratio (ROR) in SoM mode indicates that models such as fine-tuned Llama3.1-8B achieve the highest ROR at 89.86%, showing the most effectiveness in this mode.

5.3 Additional Findings

Influence of Instruction Tuning. Instruction tuning significantly enhances the performance of models in both XML and SoM modes. In XML mode, the success rates (SR) of three open-source models increase by an average of 440%, demonstrating this approach's substantial impact. Notably, LLaMA3.1-8B-**ft** achieves an SR of 27.54%, dramatically improving from its baseline SR of 2.90%. Similarly, Qwen2.5-7B-**ft** and GLM-4-9B-**ft** show marked increases, reaching SRs of 26.09% and 23.19%, respectively. In SoM mode, fine-tuning leads to significant improvements as well. For instance, Qwen2-VL-7B-Instruct-**ft** achieves an SR of 18.84%, a substantial rise from its baseline SR of 3.62%. Other models, such as Qwen2-VL-2B-**ft** and LLaMA3.2-11B-Vision-**ft**, also exhibit notable improvements, with SRs increasing to 14.49% and 11.59%, respectively. These results show that instruction-tuned open-source models achieve performance levels approaching or surpassing some closed-source models, such as GPT-4o and Claude-3-Opus, highlighting significant gains in operational rationality and efficiency.

Analysis of Agent Frameworks. We assess ReAct and SeeAct frameworks with GPT-4o and Gemini-1.5-Pro in XML and SoM modes. Table 3 shows that ReAct significantly improves performance only in the XML mode. SeeAct does not enhance performance consistently due to the model's reasoning limitations with multimodal input. ReAct and SeeAct frameworks increase token usage, which harms efficiency. As shown in Table 4, XML+ReAct settings produce an average of 67.89 tokens, while models post-instruction tuning average only 4.96 tokens.

Table 3: The impact of the ReAct and SeeAct frameworks on SR results. Notably, model performance is significantly improved in XML+ReAct mode. Full results of this table are shown in Appendix F.3

Mode	Model	SR
XML	GPT-4o	25.36
	Gemini-1.5-Pro	18.84
XML+ReAct	GPT-4o	33.33
	Gemini-1.5-Pro	31.16
XML+SeeAct	GPT-4o	24.64
	Gemini-1.5-Pro	21.01
SoM	GPT-4o	31.16
	Gemini-1.5-Pro	16.67
SoM+ReAct	GPT-4o	31.88
	Gemini-1.5-Pro	15.94
SoM+SeeAct	GPT-4o	30.43
	Gemini-1.5-Pro	21.01

Table 4: Average generation tokens of different modes. We used the LLaMA3 tokenizer for calculation. FT represents instruction tuning models.

Mode	FT	XML/SoM	ReAct	SeeAct
#Avg. Gen. Tokens	4.96	23.56	67.89	129.12

6 Conclusion

In this work, we introduced AndroidLab, a framework tackling challenges in training and evaluating Android agents. AndroidLab provides a reproducible environment, unified action spaces, and a benchmark of 138 tasks across nine apps. We defined a method for using the UI tree and device state to identify sub-goals, enabling our metrics to support task completion via any paths and ensuring fair and consistent comparisons. Based on AndroidLab, we constructed the Android Instruction dataset, using it to fine-tune six open-source models, increasing LLM success rates by 5x and LMMs by nearly 9x. AndroidLab offers a reproducible benchmark, open datasets, and tools, advancing research in efficient and privacy-preserving mobile agents.

Limitations

Limited Expandability of Evaluation Tasks. All evaluation tasks in our study are predefined and hardcoded. This means that if new evaluation tasks need to be added in the future, they must be individually and manually integrated, which is a time-consuming and not easily scalable process.

Fixed Wait Time for Actions. In the action space, the model waits for a fixed period after selecting each action to allow the device to respond. However, this fixed waiting time does not account for the variability of response times across Android devices. Such variability can be attributed to several factors, including the device model, age, and user-specific configurations. Consequently, it is challenging to establish a universally applicable wait time for responses.

Lack of Cross-Platform Capability. It is important to note that our evaluation framework is limited to the Android operating system and cannot be used to evaluate models operating on other systems, such as iOS or other device platforms. This limitation renders our framework applicable solely to a single platform. Although some tools (e.g., XCUITest, WebDriverAgent) can transform iOS operations and page information into an XML-like format, we have observed that, since these tools are third-party software, the page information obtained through this transformation process is not entirely consistent with the results directly retrieved from Android devices. This discrepancy fails to meet the requirement for fairness, and the UI tree structures are also not completely aligned. Therefore, we do not plan to extend ANDROIDLAB to other platforms.

Potential Risks

Risk Avoidance in Benchmark Design. In the design of our benchmark, we have avoided potentially risky operations such as payments and sending messages. Additionally, our benchmark is tested on virtual machines without an internet connection, further preventing the actual execution of these operations. However, in real-world scenarios where agents are used, special attention should be paid to the correctness of such operations when the user provides these kinds of tasks. We plan to add sensitive operation protection in future systems, meaning these operations require explicit user consent before execution.

Ensuring XML Quality for Apps. The XML quality of certain apps might be poor, possibly loading too much or too little content. In actual deployment, it is essential to carefully inspect the XML quality of each app to ensure accurate usage.

Privacy Issues and Solutions. One major ethical concern in applying Android agents involves privacy issues. The evaluation process of models trained with user data could potentially lead to the leakage of private information. To mitigate this, we propose the Android Instruction Dataset, which is annotated by humans and ensures the removal of sensitive private information. This dataset allows models to achieve performance close to proprietary models without compromising user privacy.

Existing agent technologies often require extensive device information to function correctly, which involves transmitting private data to servers hosting these models. Our framework provides an alternative solution by enabling open-sourced models to achieve competitive performance and allowing for the private deployment of models. This eliminates the need to send data to external servers, enhancing user information security. Future work will focus on advancing on-device model training to further address privacy concerns comprehensively.

Preventing Misuse in Sensitive Applications. Another concern is the potential misuse of Android agents in sensitive applications, such as web scraping, targeted advertising, and monetary transactions. The Android Instruction Dataset we provide is generated from predefined seeds, excluding dangerous actions to minimize misuse.

References

Anthropic. 2023. Introducing claude.

Hao Bai, Yifei Zhou, Mert Cemri, Jiayi Pan, Alane Suhr, Sergey Levine, and Aviral Kumar. 2024. Digirl: Training in-the-wild device-control agents with autonomous reinforcement learning. arXiv preprint arXiv:2406.11896.

Jinze Bai, Shuai Bai, et al. 2023. Qwen technical report.

Andrea Burns, Deniz Arsan, Sanjna Agrawal, Ranjitha Kumar, Kate Saenko, and Bryan A Plummer. 2021. Mobile app tasks with iterative feedback (motif): Addressing task feasibility in interactive visual environments. arXiv preprint arXiv:2104.08560.

Alice Coucke, Alaa Saade, Adrien Ball, Théodore Bluche, Alexandre Caulier, David Leroy, Clément Doumouro, Thibault Gisselbrecht, Francesco Caltagirone, Thibaut Lavril, Maël Primet, and Joseph

Dureau. 2018. Snips voice platform: an embedded spoken language understanding system for private-by-design voice interfaces.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. 2023. Mind2web: Towards a generalist agent for the web. arXiv preprint arXiv:2306.06070.

Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Diego Rojas, Guanyu Feng, Hanlin Zhao, Hanyu Lai, et al. 2024. Chatglm: A family of large language models from glm-130b to glm-4 all tools. arXiv preprint arXiv:2406.12793.

Izzeddin Gur, Hiroki Furuta, Austin Huang, Mustafa Safdari, Yutaka Matsuo, Douglas Eck, and Aleksandra Faust. 2023. A real-world webagent with planning, long context understanding, and program synthesis. arXiv preprint arXiv:2307.12856.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, and Jie Tang. 2023. Cogagent: A visual language model for gui agents.

Raghav Kapoor, Yash Parag Butala, Melisa Russak, Jing Yu Koh, Kiran Kamble, Waseem Alshikh, and Ruslan Salakhutdinov. 2024. Omniact: A dataset and benchmark for enabling multimodal generalist autonomous agents for desktop and web. arXiv preprint arXiv:2402.17553.

Hanyu Lai, Xiao Liu, Iat Long Iong, Shuntian Yao, Yuxuan Chen, Pengbo Shen, Hao Yu, Hanchen Zhang, Xiaohan Zhang, Yuxiao Dong, et al. 2024. Autowebglm: Bootstrap and reinforce a large language model-based web navigating agent. arXiv preprint arXiv:2404.03648.

Juyong Lee, Taywon Min, Minyong An, Changyeon Kim, and Kimin Lee. 2024. Benchmarking mobile device control agents across diverse configurations.

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

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

Xiao Liu, Hanyu Lai, Hao Yu, Yifan Xu, Aohan Zeng, Zhengxiao Du, Peng Zhang, Yuxiao Dong, and Jie Tang. 2023. Webglm: Towards an efficient web-enhanced question answering system with human preferences. arXiv preprint arXiv:2306.07906.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. 2021. Webgpt: Browser-assisted question-answering with human feedback. arXiv preprint arXiv:2112.09332.

OpenAI. 2023. Gpt-4 technical report.

Jiayi Pan, Yichi Zhang, Nicholas Tomlin, Yifei Zhou, Sergey Levine, and Alane Suhr. 2024. Autonomous evaluation and refinement of digital agents.

Divyanshu Rai, Sumbul Siddiqui, Mahesh Pawar, and Sachin Goyal. 2019. Robotic process automation: the virtual workforce. International Journal on Future Revolution in Computer Science & Communication Engineering, 5(2):28–32.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy Lillicrap, and Oriana Riva. 2024. Androidworld: A dynamic benchmarking environment for autonomous agents.

Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. 2023. Android in the wild: A large-scale dataset for android device control. arXiv preprint arXiv:2307.10088.

Mário Romao, Joao Costa, and Carlos J Costa. 2019. Robotic process automation: A case study in the banking industry. In 2019 14th Iberian Conference on information systems and technologies (CISTI), pages 1–6. IEEE.

Liangtai Sun, Xingyu Chen, Lu Chen, Tianle Dai, Zichen Zhu, and Kai Yu. 2022. Meta-gui: Towards multi-modal conversational agents on mobile gui.

Gemini Team, Machel Reid, Nikolay Savinov, and Denis Teplyashin et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971.

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

Sagar Gubbi Venkatesh, Partha Talukdar, and Srini Narayanan. 2023. Ugif: Ui grounded instruction following.

Bryan Wang, Gang Li, and Yang Li. 2023. Enabling conversational interaction with mobile ui using large language models.

Peng Wang, Shuai Bai, Sinan Tan, Shijie Wang, Zhihao Fan, Jinze Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, et al. 2024. Qwen2-vl: Enhancing vision-language model's perception of the world at any resolution. arXiv preprint arXiv:2409.12191.

Mingzhe Xing, Rongkai Zhang, Hui Xue, Qi Chen, Fan Yang, and Zhen Xiao. 2024. Understanding the weakness of large language model agents within a complex android environment. arXiv preprint arXiv:2402.06596.

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

Jianwei Yang, Hao Zhang, Feng Li, Xueyan Zou, Chunyuan Li, and Jianfeng Gao. 2023a. Set-of-mark prompting unleashes extraordinary visual grounding in gpt-4v.

Zhao Yang, Jiaxuan Liu, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2023b. Appagent: Multimodal agents as smartphone users. arXiv preprint arXiv:2312.13771.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. arXiv preprint arXiv:2210.03629.

Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, et al. 2022. Glm-130b: An open bilingual pre-trained model. arXiv preprint arXiv:2210.02414.

Zhuosheng Zhan and Aston Zhang. 2023. You only look at screens: Multimodal chain-of-action agents. arXiv preprint arXiv:2309.11436.

Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. 2024. Gpt-4v (ision) is a generalist web agent, if grounded. arXiv preprint arXiv:2401.01614.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, et al. 2023. Webarena: A realistic web environment for building autonomous agents. arXiv preprint arXiv:2307.13854.

A Details of Tasks

In our experiment, we use various apps to conduct various tests (succinctly presented in Table 5). The following mobile apps are chosen:

- **Bluecoins**: A personal finance management app used for tracking expenses and income.
- **Calendar**: A calendar app helps in organizing schedules and setting reminders.
- **Cantook**: An e-book reader for storing, managing, and reading e-books.
- **Clock**: A clock app for displaying the time, setting alarms, and using a stopwatch.
- **Contacts**: A contact management app for storing and organizing contact information.
- **Maps.me**: An offline map app for navigation and exploring locations.
- **PiMusic**: A music player app for organizing and playing locally stored music files.
- **Settings**: A settings app for configuring device settings and preferences.
- **Zoom**: A video conferencing app for hosting and joining online meetings.

The selection of these apps goes through multiple iterations to ensure their suitability for our evaluation purposes. A key criterion for the final selection is that each app functions independently, without requiring an internet connection or user account login. This ensures that the evaluations can be consistently replicated under the same conditions, eliminating external dependencies and reducing the risk of privacy breaches. As a result, this approach maintains the reliability and reproducibility of our results.

B Detail of Operation Modes

B.1 XML Mode

As shown in Figure 5, in this mode, we prompt models with task description, interaction history, and current compressed XML information. The models are supposed to output an action in function-call format. The actions are applied on coordinates shown in XML.

B.2 SoM Mode

As shown in Figure 6, in this mode, we prompt models with task description, interaction history, and current screenshot with set of marks(Yang et al., 2023a). The models are also supposed to output an action in function-call format. Different from XML mode, the actions are performed on specified elements via marked indices.

B.3 ReAct Modes

We follow (Yao et al., 2022) for ReAct prompting. In this mode, we perform both text-only and multi-modal testing. As shown in Figure 7 and Figure 8, the text-only and multi-modal prompts are based on Section B.1 and Section B.2 respectively. We

Table 5: List of Android Eval apps used along with corresponding example task, sub-goals, and the number of tasks.

APP	Example Task	Sub-Goals	# tasks
Bluecoins	Record an income of 8000 CNY in the books, and mark it as "salary".	· type: income · cash: 8000 CNY · note: salary	15
Calendar	Edit the event with title "work", change the time to be 7:00 PM.	· title: work · state: editing · date: today · time: 7 PM	14
Cantook	Mark Hamlet as read.	· book: Hamlet · state: 100% read	12
Clock	I need set an 10:30PM clock every weekend, and label it as "Watch Football Games".	· time: 10:30PM · frequency: every weekend · label: Watch Football Games	27
Contacts	Add a contacts whose name is Xu, set the working phone number to be 12345678, and mobile phone number to be 87654321.	· name: Xu · working phone number: 12345678 · mobile phone number: 87654321	15
Maps.me	Check the driving distance and time between Bus stop of 2700 Coast Avenue and Bus Stop Route 51.	· driving distance: 7.0km · driving time: 8 min	15
PiMusic	Sort Pink Floyd's songs by duration time in descending order.	· page: ARTISTS · artist: Pink Floyd · order: descending by duration	12
Setting	Show battery percentage in status bar.	· battery percentage: displayed	23
Zoom	I need to join meeting 1234567890 without audio and video.	· meeting ID: 1234567890 · audio: off · video: off	5

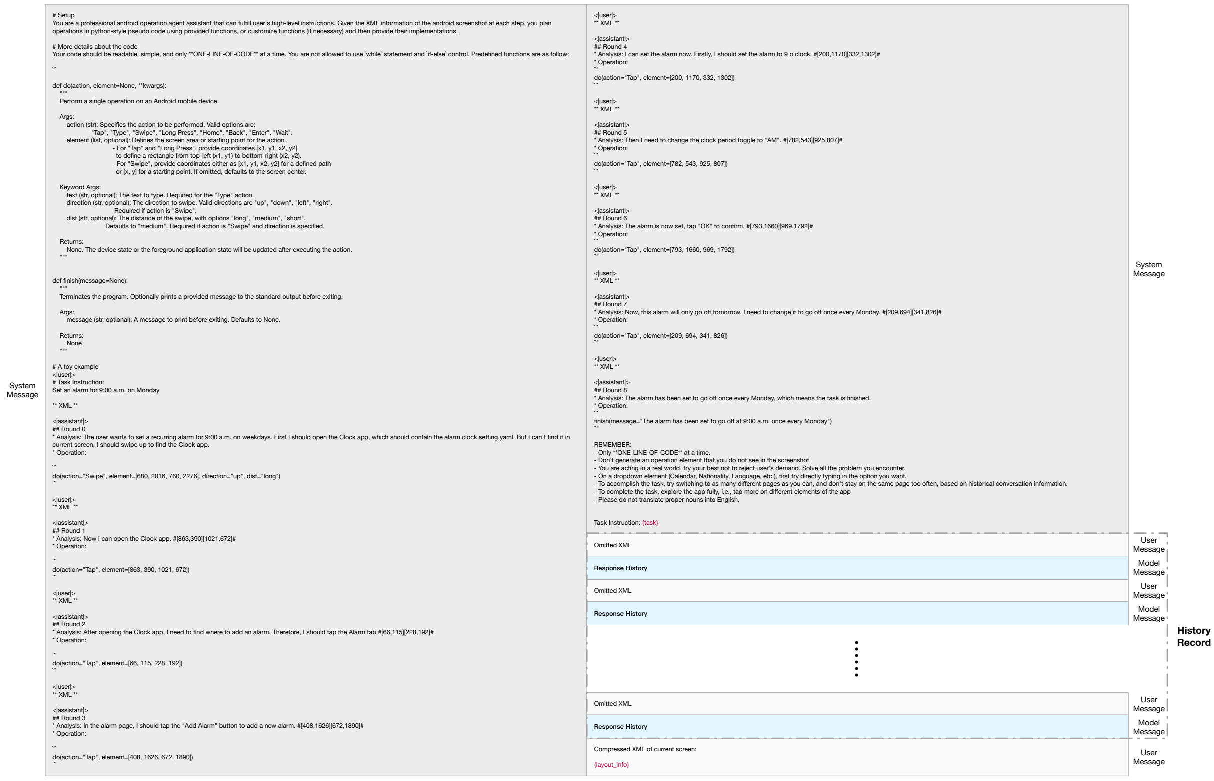


Figure 5: Prompts of XML Mode for Text-only Testing

add prompts that allow models to think step by step before outputting actions.

B.4 SeeAct Modes

We follow (Zheng et al., 2024) for SeeAct prompting. The raw prompts of SeeAct are designed for web browsers. To adopt that in android environments, we make some modifications, and the final prompts are shown in Figure 9 for multi-modal testing and Figure 10 for text-only testing.

For multi-modal and text-only testing, the information of mobile phones is given by screenshots and compressed XML respectively. The models are expected to generate a detailed description of the action, its corresponding element, and parameters in round 1, and the expected function-call format in round 2.

System Message

You are an agent that is trained to complete certain tasks on a smartphone. You will be
given a screenshot of a smartphone app. The interactive UI elements on the screenshot are labeled with numeric tags
starting from 1.

You can call the following functions to interact with those labeled elements to control the smartphone:

1.tap(index: int)

Taps the UI element labeled with the given number.
Example: tap(5)

2.text(input_str: str)

Inserts the given text into an input field.
Example: text("Hello, world!")
Since we use ADB keyboard, if ADB keyboard ON is displayed on the bottom of the screen, you can use this function.
If you think that the keyboard is displayed after your previous operation, you can try to use this function to input text.

3.long_press(index: int)

Long presses the UI element labeled with the given number.
Example: long_press(5)

4. swipe(index: int, direction: str, dist: str)

Swipes the UI element in the specified direction and distance. "direction" is a string that
represents one of the four directions: up, down, left, right. "dist" determines the distance of the swipe and can be one
of the three options: short, medium, long.
Example: swipe(21, "up", "medium")

5. back()

Simulates a back button press on the smartphone.

6. home()

Simulates a home button press on the smartphone.

7. wait(interval: int)

Pauses the execution for the given number of seconds. Default is 5 second.

8. finish(message: str)

Ends the task and provides the final output. You can return the final output of the task as a string.
Example: finish("Task completed")

Now, given the following labeled screenshot, you need to think and call the function needed to proceed with the task.
Your output should include only action part in the given format:

Action: <The function call with the correct parameters to proceed with the task. If you believe the task is completed or
there is nothing to be done, you should use finish function. You cannot output anything else except a function call
in this field.>

Whenever you think the task is finished, you should use finish function to avoid extra operations.

If you found yourself in a loop or the task is not proceeding as expected, you might consider changing your operation and try other methods.
If you operate same action 5 times, the program will automatically stop.
If tap operation is not working, you can try long press operation.

You can only take one action at a time, so please directly call the function.

Task Instruction: {task}

History Record

User Message: Omitted Screenshot

Model Message: **Response History**

User Message: Omitted Screenshot

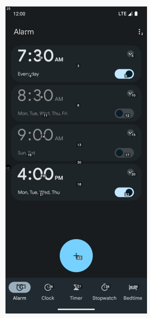
Model Message: **Response History**

⋮

User Message: Omitted Screenshot

Model Message: **Response History**

User Message



Screenshot with set of marks

Figure 6: Prompts of SoM Mode for Multi-modal Testing

System
Message

~~~~
# Setup
You are a professional android operation agent assistant that can fulfill user's high-level instructions. Given the XML information of the android screenshot at each step, you plan operations in python-style pseudo code using provided functions, or customize functions (if necessary) and then provide their implementations.

# More details about the code
Your code should be readable, simple, and only **ONE-LINE-OF-CODE** at a time. You are not allowed to use `while` statement and `if-else` control. Predefined functions are as follow:

```

def do(action, element=None, **kwargs):
    """
    Perform a single operation on an Android mobile device.

    Args:
        action (str): Specifies the action to be performed. Valid options are:
                  "Tap", "Type", "Swipe", "Long Press", "Home", "Back", "Enter", "Wait".
        element (list, optional): Defines the screen area or starting point for the action.
                          - For "Tap" and "Long Press", provide coordinates [x1, y1, x2, y2]
                            to define a rectangle from top-left (x1, y1) to bottom-right (x2, y2).
                          - For "Swipe", provide coordinates either as [x1, y1, x2, y2] for a defined path
                            or [x, y] for a starting point. If omitted, defaults to the screen center.

    Keyword Args:
        text (str, optional): The text to type. Required for the "Type" action.
        direction (str, optional): The direction to swipe. Valid directions are "up", "down", "left", "right".
                         Required if action is "Swipe".
        dist (str, optional): The distance of the swipe, with options "long", "medium", "short".
                    Defaults to "medium". Required if action is "Swipe" and direction is specified.

    Returns:
        None. The device state or the foreground application state will be updated after executing the action.
    """


def finish(message=None):
    """
    Terminates the program. Optionally prints a provided message to the standard output before exiting.

    Args:
        message (str, optional): A message to print before exiting. Defaults to None.

    Returns:
        None
    """

```

Now, given the following XML information, you need to think and call the function needed to proceed with the task.
Your output should include Obs, Thought and Act in the given format:

Obs
Retrieve the result of executing the instruction from the external environment. This is equivalent to obtaining the result of the current step' s behavior, preparing for the next step.
Note: In order to reduce the number of function calls, the Obs step executes at the beginning of the next turn. So if current step is not the first step, you should observe the result of the previous step in the current step.

Thought
Reasoning and textual display of the process. What do I want to do, and what are the prerequisites to achieve this.

Action
Generate the instruction to interact with the android environment.

Here is an one-shot example:

Obs: The user wants to set an alarm for 9:00 a.m. on weekdays. The XML shows the clock app is open.
Thought: After opening the Clock app, I need to find where to add an alarm. Therefore, I should tap the Alarm tab #[66,115][228,192]#
Action:
```
do(action="Tap", element=[66,115,228,192])
```

REMEMBER:
- Only Obs, Thought and **ONE-LINE-OF-CODE** at a time.
- Don't generate an operation element that you do not see in the screenshot.
- You are acting in a real world, try your best not to reject user's demand. Solve all the problem you encounter.
- On a dropdown element (Calendar, Nationality, Language, etc.), first try directly typing in the option you want.
- To accomplish the task, try switching to as many different pages as you can, and don't stay on the same page too often, based on historical conversation information.
- To complete the task, explore the app fully, i.e., tap more on different elements of the app
- Please do not translate proper nouns into English.

Task Instruction: {task}
~~~~

System
Message

History
Record

Omitted XML	User Message
Response History	Model Message
Omitted XML	User Message
Response History	Model Message
⋮	
Omitted XML	User Message
Response History	Model Message
Compressed XML of current screen: {layout_info}	User Message

Figure 7: Prompts of XML Mode for ReAct Testing.

System
Message

~~~~
You are an agent that is trained to complete certain tasks on a smartphone. You will be
given a screenshot of a smartphone app. The interactive UI elements on the screenshot are labeled with numeric tags
starting from 1.

You can call the following functions to interact with those labeled elements to control the smartphone:

1.tap(index: int)

Taps the UI element labeled with the given number.
Example: tap(5)

2.text(input_str: str)

Inserts the given text into an input field.
Example: text("Hello, world!")
Since we use ADB keyboard, if ADB keyboard ON is displayed on the bottom of the screen, you can use this function.
If you think that the keyboard is displayed after your previous operation, you can try to use this function to input text.

3.long_press(index: int)

Long presses the UI element labeled with the given number.
Example: long_press(5)

4. swipe(index: int, direction: str, dist: str)

Swipes the UI element in the specified direction and distance. "direction" is a string that
represents one of the four directions: up, down, left, right. "dist" determines the distance of the swipe and can be one
of the three options: short, medium, long.
Example: swipe(21, "up", "medium")

5. back()

Simulates a back button press on the smartphone.

6. home()

Simulates a home button press on the smartphone.

7. wait(interval: int)

Pauses the execution for the given number of seconds. Default is 5 second.

8. finish(message: str)

Ends the task and provides the final output. You can return the final output of the task as a string.
Example: finish("Task completed")

Now, given the following labeled screenshot, you need to think and call the function needed to proceed with the task.
Your output should include Obs, Thought and Act in the given format:

Obs
Retrieve the result of executing the instruction from the external environment. This is equivalent to obtaining the result of the current step' s behavior, preparing for the next step.

Note: In order to reduce the number of function calls, the Obs step executes at the beginning of the next turn.
So if current step is not the first step, you should observe the result of the previous step in the current step.

Thought
Reasoning and textual display of the process. What do I want to do, and what are the prerequisites to achieve this.

Action
Generate the instruction to interact with the android environment.

Here is an one-shot example:

Obs: The user wants to set an alarm for 9:00 a.m. on weekdays. The screenshot shows the clock app is open.
Thought: I need to open the clock app labeled with 5 and find the first alarm listed .
Action:
```
tap(5)
```


Whenever you think the task is finished, you should use finish function to avoid extra operations.

If you found yourself in a loop or the task is not proceeding as expected, you might consider changing your operation and try other methods.
If you operate same action 5 times, the program will automatically stop.
If tap operation is not working, you can try long press operation.

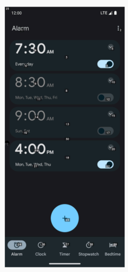
You can only take one action at a time, so please directly call the function.

Task Instruction: {task}
~~~~

System
Message

History
Record

User Message	Omitted Screenshot
Model Message	**Response History**
User Message	Omitted Screenshot
Model Message	**Response History**
	⋮
User Message	Omitted Screenshot
Model Message	**Response History**



Screenshot with set of marks

User
Message

Figure 8: Prompts of SoM Mode for ReAct Testing.

System Message

You are assisting humans doing smartphone navigation tasks step by step. At each stage, you can see the smartphone by a screenshot and know the previous actions before the current step decided by yourself that have been executed for this task through recorded history. You need to decide on the first following action to take.

Here are the descriptions of all allowed actions: "Tap", "Type", "Swipe", "Long Press", "Home", "Back", "Enter", "Wait".

User Message

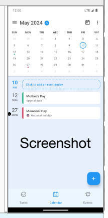
You are asked to complete the following task: {task}

Previous Actions:

{previous_actions}

The screenshot below shows the smartphone you see. Think step by step before outlining the next action step at the current stage. Clearly outline which element in the smartphone users will operate with as the first next target element, its detailed location, and the corresponding operation.

To be successful, it is important to follow the following rules:
1. You should only issue a valid action given the current observation.
2. You should only issue one action at a time.
3. Terminate when you deem the task complete.



Round 1 Model Generation

Action Generation

User Message

(Reiteration)
First, reiterate your next target element, its detailed location, and the corresponding operation.

(Final Answer)
Below is a multi-choice question, where the choices are elements in the smartphone. From the screenshot, find out where and what each one is on the smartphone, taking into account both their text content and path details. Then, determine whether one matches your target element if your action involves an element. Choose the best matching one.

{option_prompt}

Conclude your answer using the format below. Ensure your answer is strictly adhering to the format provided below.

Predefined functions are as follow:

````
```
def do(action, element=None, **kwargs):
    """
    Perform a single operation on an Android mobile device.

    Args:
        action (str): Specifies the action to be performed. Valid options are:
                      "Tap", "Type", "Swipe", "Long Press", "Home", "Back", "Enter", "Wait".
        element (list, optional): Defines the screen area or starting point for the action.
                                  - For "Tap" and "Long Press", provide coordinates [x1, y1, x2, y2]
                                    to define a rectangle from top-left (x1, y1) to bottom-right (x2, y2).
                                  - For "Swipe", provide coordinates either as [x1, y1, x2, y2] for a defined path
                                    or [x, y] for a starting point. If omitted, defaults to the screen center.

    Keyword Args:
        text (str, optional): The text to type. Required for the "Type" action.
        direction (str, optional): The direction to swipe. Valid directions are "up", "down", "left", "right".
                                   Required if action is "Swipe".
        dist (str, optional): The distance of the swipe, with options "long", "medium", "short".
                              Defaults to "medium". Required if action is "Swipe" and direction is specified.
    Returns:
        None. The device state or the foreground application state will be updated after executing the action.
    """
    ...


def finish(message=None):
    """
    Terminates the program. Optionally prints a provided message to the standard output before exiting.

    Args:
        message (str, optional): A message to print before exiting. Defaults to None.

    Returns:
        None
    """
    ...
```
````

Your code should be readable, simple, and only **ONE-LINE-OF-CODE** at a time. You are not allowed to use `while` statement and `if-else` control. Please do not leave any explanation in your answers of the final standardized format part, and this final part should be clear and certain.

Example if you want to swipe up from an element located at [680,2016][760,2276] with a long distance:
```
do(action="Swipe", element=[680, 2016, 760, 2276], direction="up", dist="long")
```

Example if you deem the task complete and want to finish with a message:
```
finish(message="The alarm on 9:00 AM weekday has been set")
```

Round 2 Model Generation

Action Grounding

Figure 9: Prompts of SoM Mode for SeeAct Testing.

Round 1

System Message

You are assisting humans doing smartphone navigation tasks step by step. At each stage, you can see the smartphone by compressed layout information and know the previous actions before the current step decided by yourself that have been executed for this task through recorded history. You need to decide on the first following action to take.

Here are the descriptions of all allowed actions: "Tap", "Type", "Swipe", "Long Press", "Home", "Back", "Enter", "Wait".

User Message

You are asked to complete the following task: {task}

Previous Actions:

{previous_actions}

The compressed layout information below shows the smartphone you see.

{layout_info}

Think step by step before outlining the next action step at the current stage. Clearly outline which element in the smartphone users will operate with as the first next target element, its detailed location, and the corresponding operation.

To be successful, it is important to follow the following rules:
1. You should only issue a valid action given the current observation.
2. You should only issue one action at a time.
3. Terminate when you deem the task complete.

Model Generation

Action Generation

User Message

(Reiteration)
First, reiterate your next target element, its detailed location, and the corresponding operation.

(Final Answer)
Below is a multi-choice question, where the choices are elements in the smartphone. From compressed layout information, find out where and what each one is on the smartphone, taking into account both their text content and path details. Then, determine whether one matches your target element if your action involves an element. Choose the best matching one.

{option_prompt}

Conclude your answer using the format below. Ensure your answer is strictly adhering to the format provided below.

Predefined functions are as follow:

```
def do(action, element=None, **kwargs):
    """
    Perform a single operation on an Android mobile device.

    Args:
        action (str): Specifies the action to be performed. Valid options are:
                      "Tap", "Type", "Swipe", "Long Press", "Home", "Back", "Enter", "Wait".
        element (list, optional): Defines the screen area or starting point for the action.
                                  - For "Tap" and "Long Press", provide coordinates [x1, y1, x2, y2]
                                    to define a rectangle from top-left (x1, y1) to bottom-right (x2, y2).
                                  - For "Swipe", provide coordinates either as [x1, y1, x2, y2] for a defined path
                                    or [x, y] for a starting point. If omitted, defaults to the screen center.

    Keyword Args:
        text (str, optional): The text to type. Required for the "Type" action.
        direction (str, optional): The direction to swipe. Valid directions are "up", "down", "left", "right".
                                   Required if action is "Swipe".
        dist (str, optional): The distance of the swipe, with options "long", "medium", "short".
                              Defaults to "medium". Required if action is "Swipe" and direction is specified.

    Returns:
        None. The device state or the foreground application state will be updated after executing the action.
    """
    ...


def finish(message=None):
    """
    Terminates the program. Optionally prints a provided message to the standard output before exiting.

    Args:
        message (str, optional): A message to print before exiting. Defaults to None.

    Returns:
        None
    """
    ...
```

Your code should be readable, simple, and only **ONE-LINE-OF-CODE** at a time. You are not allowed to use `while` statement and `if-else` control. Please do not leave any explanation in your answers of the final standardized format part, and this final part should be clear and certain.

Example if you want to swipe up from an element located at [680,2016][760,2276] with a long distance:
```
do(action="Swipe", element=[680, 2016, 760, 2276], direction="up", dist="long")
```

Example if you deem the task complete and want to finish with a message:
```
finish(message="The alarm on 9:00 AM weekday has been set")
```

Round 2

Model Generation

Action Grounding

Figure 10: Prompts of XML Mode for SeeAct Testing.

C Details of XML Compression Algorithm

Currently, the inputs effectively handled by mainstream Large Language Models (LLMs) are generally within 8k tokens to 16k tokens. Beyond this length, the model's performance significantly declines. However, the raw XML obtained through methods provided by Google Android often requires tens of thousands of tokens after being converted into tokens after conversion. Therefore, it is necessary to simplify the XML information before feeding it to the model. Some existing XML simplification algorithms still retain a lot of structural information and descriptive representations from the original XML. In many complex pages, the simplified XML obtained is still far more than 16k tokens in length.

Since the original XML is used to define the layout and elements of the user interface, it includes all the components on a page. Thus, the original XML contains many nodes that exist only for structural and layout purposes. These nodes do not provide useful page information, which is the main reason for the excessive length of the original XML. Additionally, a page often contains more nodes than are displayed on the screen, such as in scrollable pages. Thus, the original XML will also include many off-screen nodes.

First, we determine whether to retain off-screen nodes, controlled by an input parameter remain_nodes (retain nodes when remain_nodes=True). For instance, when it is necessary to summarize the entire page's information, we can retain off-screen nodes to save operations (like scrolling the screen to see the full text) and directly obtain the complete page's text information. If the requirement is related to operation simulation, such as simulating clicking elements or scrolling, we can choose to delete off-screen nodes to prevent interference with the model. Specifically, in the original XML, the bounds property of all on-screen nodes must be within [0,0][Window_Height, Window_Width] and must be contained by their parent node. Therefore, we only need to determine whether the current node's bounds are contained by its parent node to identify all the nodes within the screen range.

The original XML also contains many nodes that exist only for structural and layout purposes, which do not include useful page information. Thus, we will delete these redundant nodes. We will judge whether a node is redundant based on its attributes. If a node has at least one of the following attributes as True: "checkable", "checked", "clickable", "focusable", "scrollable", "long-clickable", "password", "selected", or if the text or content-desc is not empty, we consider this node functional. Nodes that do not meet this criterion are redundant, and we will delete all such nodes.

The descriptions of each attribute in the original XML are overly redundant and consume many tokens. Finally, we will simplify these attribute descriptions. For the functional attributes "checkable", "checked", "clickable", "focusable", "scrollable", "long-clickable", "password", "selected", since most cases will be False, we will only display attributes with True values. The "index", "resource-id", and "package" attributes do not help the model understand the page, so we will delete them directly. The "class" attribute, to some extent, indicates the main function of a node, so we will retain its last part (the class is always in x.x.x.x format, with varying dot counts, and we will keep only the part after the last dot, e.g., retaining FrameLayout from android.widget.FrameLayout). The "text" and "content-desc" attributes represent the node's text information, and we will merge and display them separately. The "bounds" attribute indicates the node's position on the page and is one of the most critical attributes, so we will display it separately.

Ultimately, for the following node:

<node index="0" text="XXX" resource-id="" class="android.view.View" package="com.autonavi.minimap" content-desc="" checkable="false" checked="false" clickable="false" enabled="true" focusable="false" focused="false" scrollable="false" long-clickable="false" password="false" selected="false" bounds="[290,844][346,885]" />

We will simplify it to:

[n42] View;;; XXX; [290,844][346,885]

In summary, by reducing nodes to remove redundant and off-screen nodes and simplifying the node attribute descriptions, we will rewrite the XML into a new, more concise format to obtain a more streamlined XML.

D Details of Metrics

D.1 Success Rate (SR)

For Operation Tasks, we probe task completion via unique Android emulator states. For Query Tasks, advanced LLMs verify if the model's predicted results match the standard answers, avoiding errors from direct string comparisons, achieving an accuracy rate of over 98% (Cf. Appendix F.5). The Success Rate is calculated as the average task completion rate across all tasks: $SR = \sum_{i=1}^{N} S_i/N$, where N is the total number of tasks, and S_i is a binary value indicating whether task i is successfully completed. We provide an example in Fig 4.

D.2 Sub-Goal Success Rate (Sub-SR)

Tasks are decomposed into sub-goals, and completion is assessed sequentially. This finer metric rewards models with stronger understanding and operational capabilities. It is common for models to only achieve partial goals, as shown in Fig 14. This approach allows us to distinguish the model's capabilities at a finer granularity. Sub-Goal Success Rate is calculated by averaging the success rate of sub-goals within a task, followed by averaging across all tasks: $SubSR = \sum_{i=1}^{N} \left(\sum_{j=1}^{M_i} G_{ij}/M_i\right)/N$, where M_i is the number of sub-goals in task i, and G_{ij} is a binary value indicating whether sub-goal j in task i is completed.

D.3 Reversed Redundancy Ratio (RRR)

As shown in previous work (Xing et al., 2024), redundancy is measured by comparing the length of the model operation path (L) with a human operator's path length ($\hat{L}$). We calculate RRR by averaging the redundancy score across tasks: $RRR = \left(\sum_{i=1}^{N'} \hat{L}_i/L_i\right)/N'$, where N' is the number of tasks with $SR > 5\%$, L_i is the length of the model's operation path for task i, and $\hat{L}_i$ is the length of the human benchmark path.

D.4 Reasonable Operation Ratio (ROR)

This metric evaluates the proportion of operations after which the screen changed. Unchanged screens indicate the operation was ineffective and thus deemed unreasonable. ROR is calculated by averaging the reasonable operation ratios across tasks: $ROR = \left(\sum_{i=1}^{N} O_{r,i}/O_{t,i}\right)/N$, where $O_{r,i}$ is the number of operations that resulted in a screen change for task i, and $O_{t,i}$ is the total number of operations performed in task i.

One possible misconception is that ROR is true as long as the model performs an operation. However, we observed multiple situations that can cause ROR to be false.

1. Tap Operations: Some positions might be marked as clickable in the XML interface, but clicking them does nothing. For instance, many text elements are marked as clickable, but their function only displays information rather than triggers navigation. While this might be due to errors from the software developers, the agent needs to learn through SFT which buttons need to be clicked to perform tasks accurately.

2. Type Operations: Typing is only effective if it's done in an activated input field, usually following a prior action that selects that field.

3. Swipe Operations: Swiping in the incorrect location or direction will not affect the mobile device.

E Case Study

In the case shown in Fig 11, an agent with GPT-4o as the base model was asked to find the reason behind a specific expenditure at a specific date via the Bluecoins app. It correctly navigated to the right date, opened the expense section, extracted the required information, and returned it to the user without unnecessary actions. This resulted in a high RRR of 1.25 and a reasonable operation ratio of 1.0, reflecting efficient and successful task completion.

As shown in Fig 12, the agent with GPT-4o as the base model was given the task of changing the home time zone to Tokyo in the clock. Initially, it added a new clock for Tokyo, which was irrelevant. Then it navigated to the settings, correctly updated the home time zone, and completed the task. Although the task was successful, the metric penalized redundant initial steps, assigning an RRR of 0.5.

In the example shown in Fig 13, the GLM4 agent operating in SoM mode successfully navigated from my location to University South by searching for the destination and displaying the route. However, it unnecessarily clicked on the WiFi button, which was redundant. Therefore, the task was deemed successful, but the RRR score dropped to 0.875 due to the additional action.

The GPT-4o agent in XML mode, as shown in Fig 14, was tasked with joining a meeting without audio and video. It successfully entered the meeting ID but struggled with the audio and video settings, ultimately failing to turn off the video. Two of three sub-goals, including entering the meeting ID, not connecting to audio, and turning

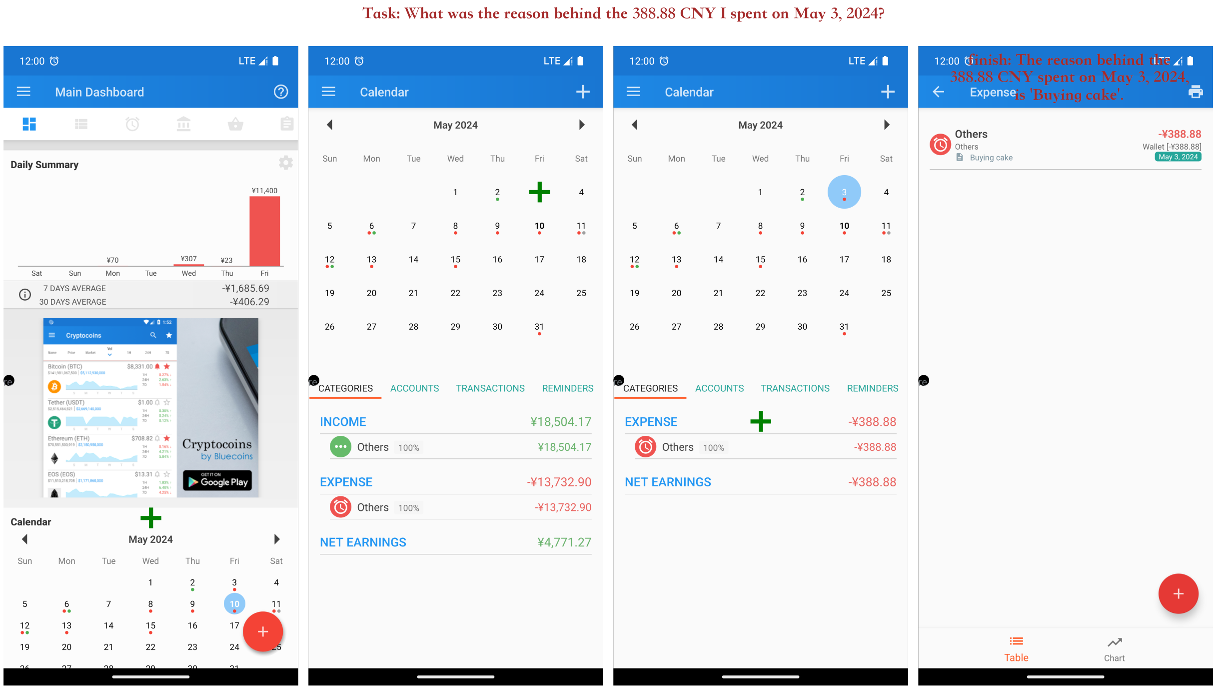


Figure 11: User Study: Successful Task of GPT-4o agent with no Redundant Operation under XML Mode

off the video, succeeded. The task was considered unsuccessful overall due to the failure to turn off the video.

In the case shown in Fig 15, a GPT-4o agent was tasked with adding a contact and setting a phone number but failed to click the input field before typing, leaving both sub-goals incomplete. The task was deemed unsuccessful.

The Llama3 agent in SoM mode, as shown in Fig 16, was tasked with setting the alarm volume to the max but failed to navigate to the correct column. In addition, it repeatedly scrolled up, completely missing the goal. As a result, the task was deemed unsuccessful and the agent was penalized with a low reasonable operation ratio, scoring 0.8.

F Additional Results

F.1 Detailed results across different APPs

Table 6 shows the number of tasks correctly completed by various models across different apps without employing the ReAct and SeeAct frameworks. This table shows that GPT-4o and GPT-4-1106-Preview perform relatively well, completing 78 and 79 tasks, respectively. In the XML mode, GPT-4-1106-Preview stands out as the top performer, with 43 tasks completed. Comparatively, in the SoM mode, GPT-4o excels, achieving a significantly higher number of tasks than the other models. Most models exhibit high success rates in tasks like "Contacts" and "Setting". Overall, GPT-4o and GPT-4-1106-Preview outperform the other models significantly in both XML and SoM modes, while Gemini-1.5-Pro shows a reasonable number of task completions across various apps.

Table 7 shows the performance improvements observed after implementing the ReAct and SeeAct frameworks on different models across various apps. Notably, GPT-4o shows significant enhancement, with the number of completed tasks increasing from 35 to 46 in XML+ReAct mode and 43 to 44 in SoM+ReAct mode. Gemini-1.5-Pro also benefits, rising from 26 to 43 tasks. The improvements are evident in specific apps like "Bluecoins", especially in high-complexity, multi-step tasks. GPT-4o leads in performance across all frameworks, showing how ReAct and SeeAct improve the model.

F.2 Detailed results across different multimodal training modes

We compare different multimodal training modes in Table 8. Under the same training data and base model settings, BBOX mode removes specified sets-of-masks from the screen. It is worth mentioning that datasets like AITW only provide click positions rather than bounding boxes (BBOX) and

Task: Change home time zone to Tokyo in clock

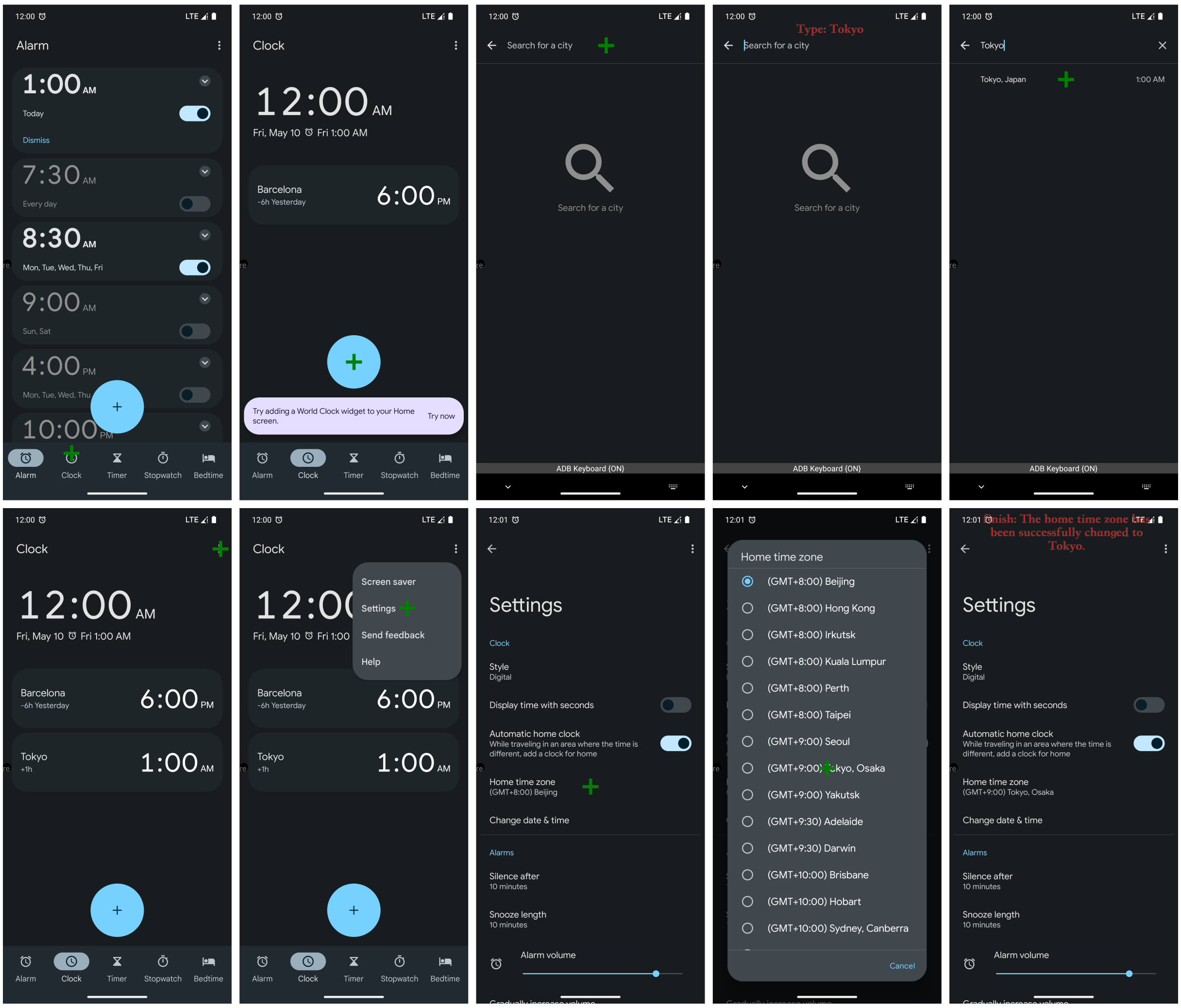


Figure 12: User Study: Successful Task of GPT-4o agent with Redundant Operation under XML Mode

Task: Navigate from my location to University South

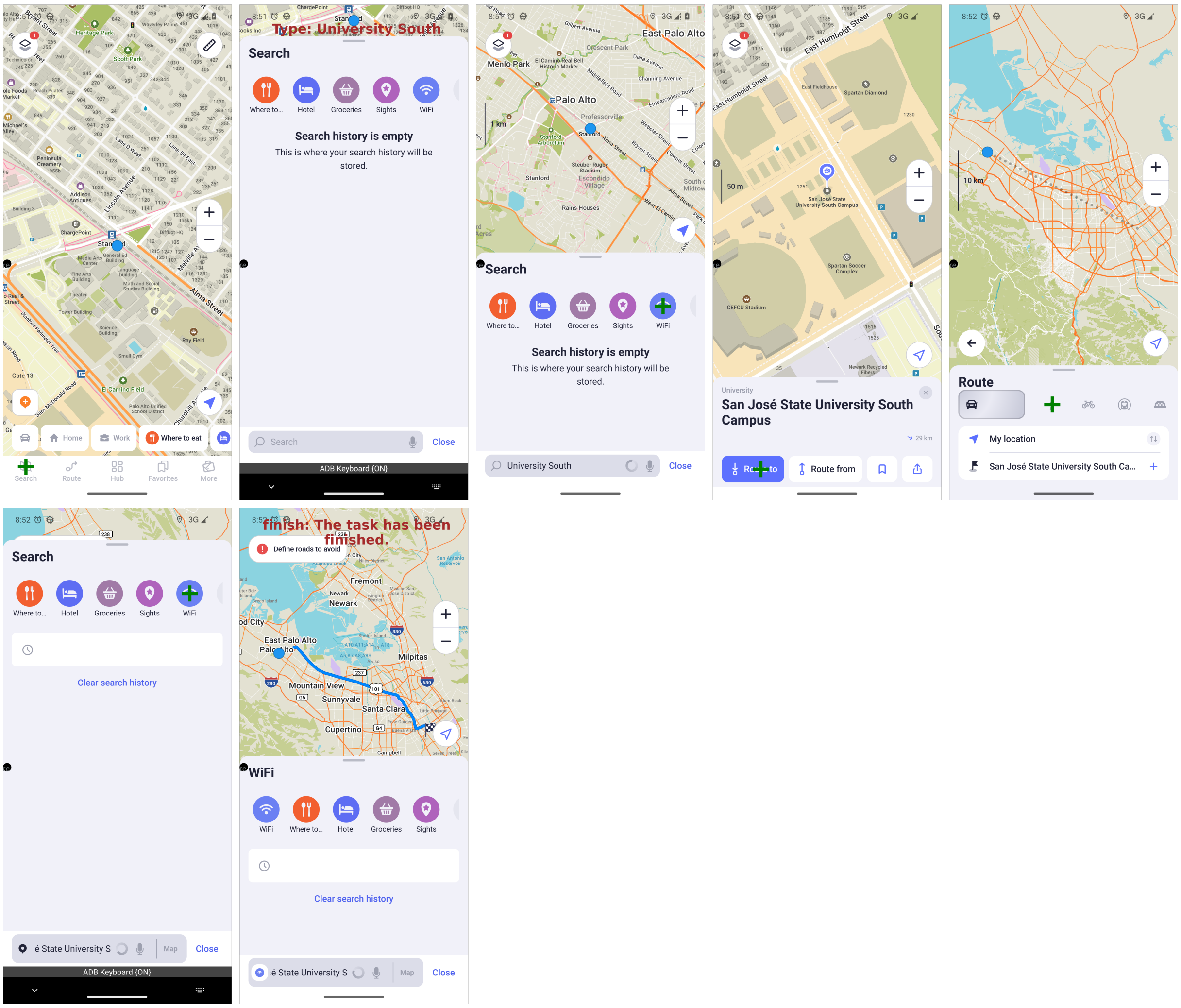


Figure 13: User Study: Successful Task of GLM4 agent with Redundant Operation under SoM Mode

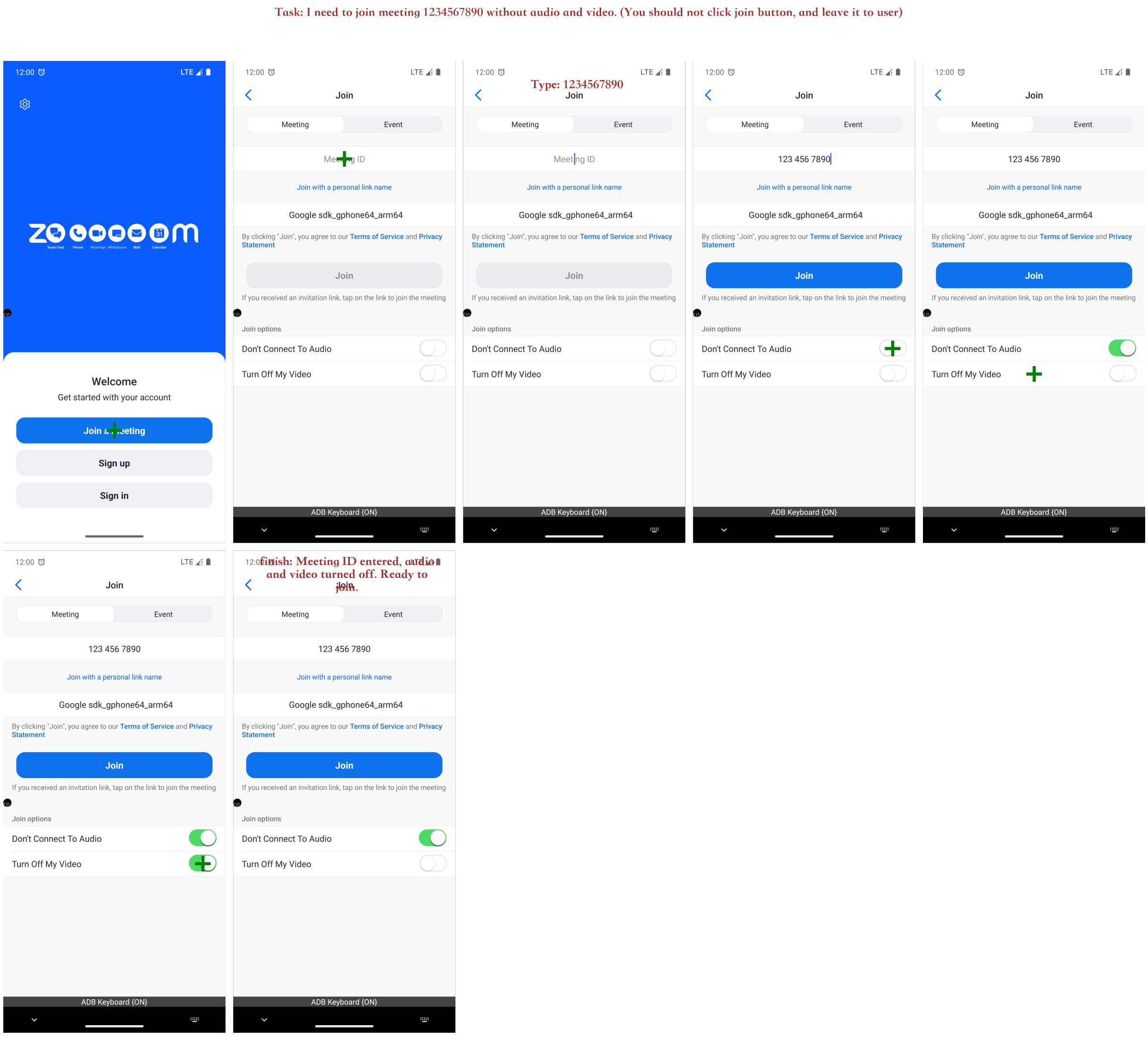


Figure 14: User Study: Unsuccessful Task of GPT-4o agent under XML Mode

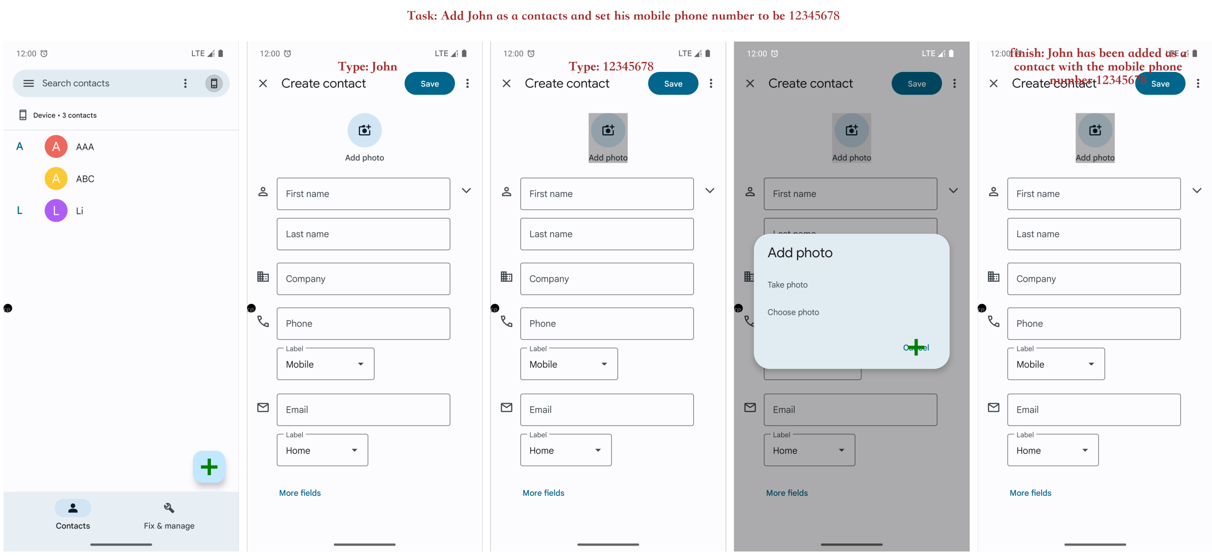


Figure 15: User Study: Unsuccessful Task of GPT-4o agent under XML Mode

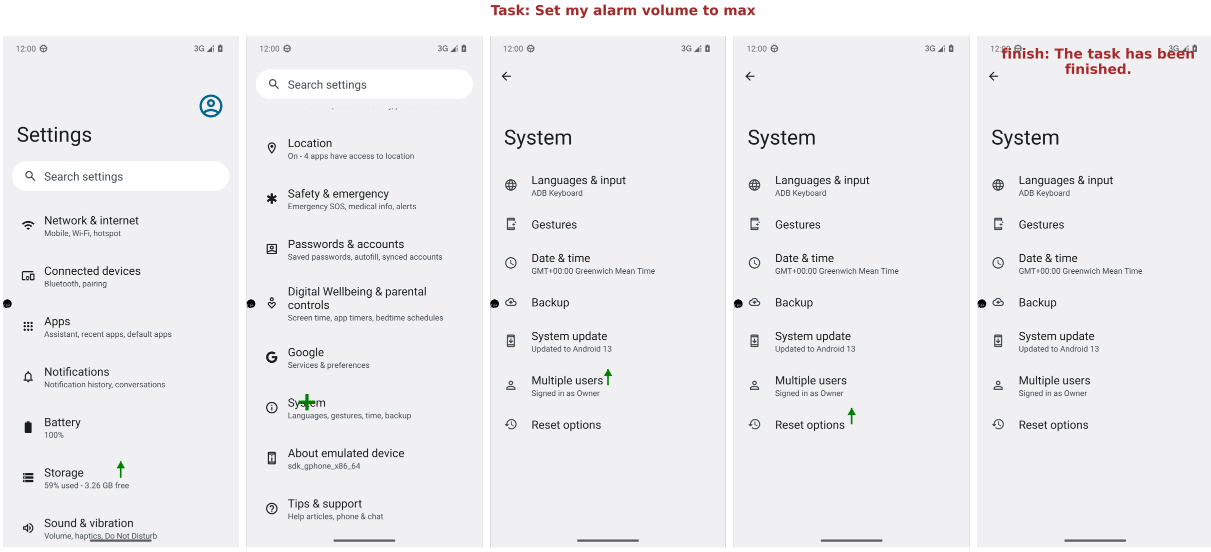


Figure 16: User Study: Unsuccessful Task of Llama3 agent under SoM Mode

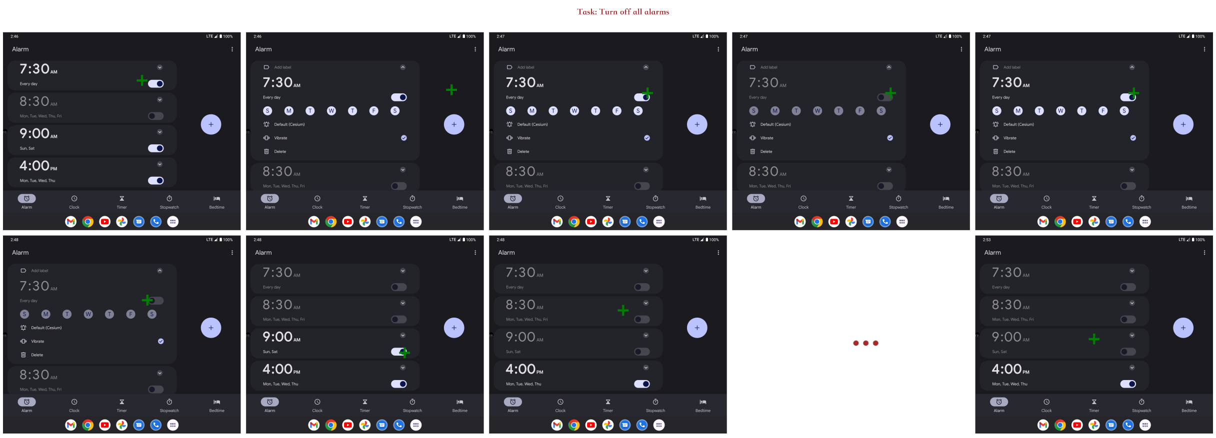


Figure 17: User Study: Unsuccessful Task under XML Mode

Task: Turn my phone to Dark theme

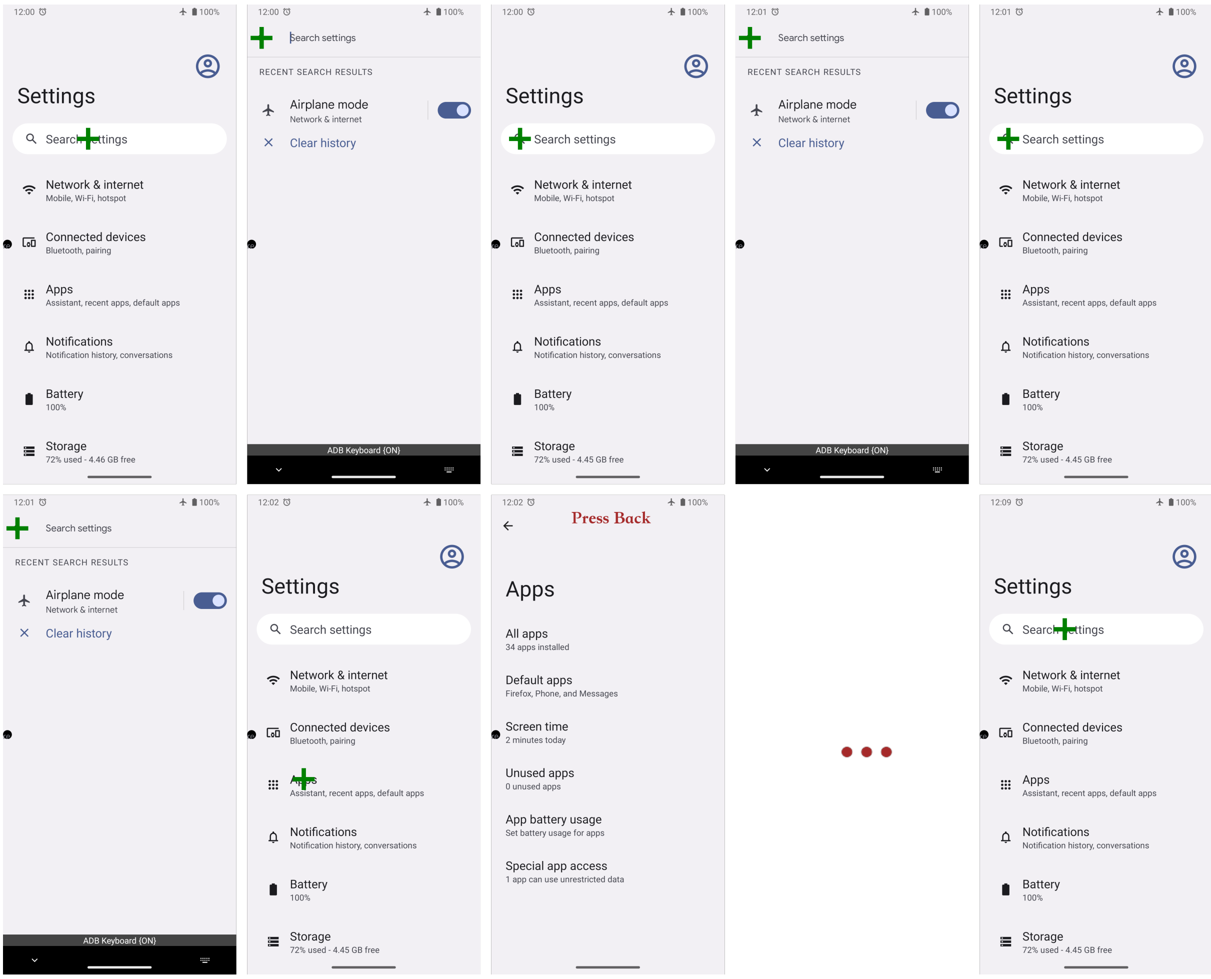


Figure 18: User Study: Unsuccessful Task under XML Mode

Table 6: The number of tasks completed by all models across all apps in different modes.

Mode	Model	Bluecoins 15	Calendar 14	Cantook 12	Clock 27	Contacts 15	Maps.me 15	PiMusic 12	Setting 23	Zoom 5	Total 138
XML	GPT-4o	1	0	3	8	5	5	2	10	1	35
	GPT-4-1106-Preview	1	4	6	4	6	6	4	9	3	43
	Gemini-1.5-Pro	1	1	3	6	3	4	3	4	1	26
	Gemini-1.0	0	1	1	4	2	0	1	2	1	12
	GLM4-PLUS	2	0	4	9	6	3	2	10	2	38
	LLaMA3.1-8B-Instruct	0	0	0	2	0	0	0	1	0	3
	Qwen2.5-7B-Instruct	0	1	0	2	0	1	1	1	2	8
	GLM4-9B-Chat	0	1	0	2	1	1	0	3	2	10
	LLaMA3.1-8B-**ft**	3	5	6	8	6	5	0	4	1	38
	Qwen2.5-7B-**ft**	3	4	5	6	6	3	1	7	1	36
	GLM4-9B-**ft**	2	4	5	5	8	1	0	7	0	32
SoM	GPT-4o	1	1	5	7	8	2	2	13	4	43
	GPT-4-Vision-Preview	1	1	5	8	6	2	2	8	3	36
	Gemini-1.5-Pro	0	0	5	2	5	0	1	7	3	23
	Gemini-1.0	0	0	2	3	3	0	1	5	1	15
	Claude-3-Opus	1	0	1	2	4	0	3	7	0	18
	Claude-3.5-Sonnet	4	2	4	9	7	0	3	10	1	40
	LLaMA3.2-11B-Vision-Instruct	0	0	0	1	0	0	0	1	0	2
	Qwen2-VL-2B-Instruct	0	0	0	0	0	0	0	0	0	0
	Qwen2-VL-7B-Instruct	0	0	0	2	1	0	0	1	1	5
	LLaMA3.2-11B-Vision-**ft**	0	2	2	3	1	5	0	3	0	16
	Qwen2-VL-2B-Instruct-**ft**	1	4	1	3	2	3	0	5	1	20
	Qwen2-VL-2B-Instruct-**ft**	0	0	1	7	7	6	0	4	1	26

Table 7: The improvement in model performance after employing the ReAct and SeeAct frameworks, is reflected in the increased number of successfully completed tasks across various apps.

Mode	Model	Bluecoins 15	Calender 14	Cantook 12	Clock 27	Contacts 15	Maps.me 15	PiMusic 12	Settings 23	Zoom 5	Total 138
XML	GPT-4o	1	0	3	8	5	5	2	10	1	35
	Gemini-1.5-Pro	1	1	3	6	3	4	3	4	1	26
XML+ReAct	GPT-4o	2	0	4	12	7	6	2	11	2	46
	Gemini-1.5-Pro	4	0	4	6	6	6	3	11	3	43
XML+SeeAct	GPT-4o	1	2	4	8	5	3	2	7	2	34
	Gemini-1.5-Pro	1	0	6	6	5	0	2	8	1	29
SoM	GPT-4o	1	1	5	7	8	2	2	13	4	43
	Gemini-1.5-Pro	0	0	5	2	5	0	1	7	3	23
SoM+ReAct	GPT-4o	3	1	5	7	7	3	0	15	3	44
	Gemini-1.5-Pro	1	1	3	2	4	1	2	7	1	22
SoM+SeeAct	GPT-4o	6	1	4	11	6	0	2	9	3	42
	Gemini-1.5-Pro	1	0	6	6	5	0	2	8	1	29

do not offer a way to reconstruct the click-box from XML. Therefore, data from AITW and similar datasets are more challenging to learn from.

F.3 Detailed results of SeeAct and ReAct methods

We have provided detailed results on the impact of the SeeAct and ReAct frameworks on model performance in Fig 9, including all four metrics.

F.4 Influence of Windows Size.

As shown in Figure 19, experiments with three Android VMs of varying sizes in SoM mode show optimal agent performance on screens matching commonly used smartphones (e.g., Pixel 7 Pro, Pixel 8 Pro). Performance drops on smaller (Pixel 3a) and larger screens (Pixel Fold).

Most Android phones share screen sizes similar to the Pixel 7 Pro or Pixel 8 Pro, which may make such data prevalent in proprietary multimodal training for closed-source models. As a result, these models might struggle with devices like the Pixel Fold, whose screen resembles a tablet. For example, as is shown in Fig 17, a GPT-4o agent effectively turned off alarms on Pixel 7 Pro and Pixel 8 Pro but failed to locate all alarm buttons on the Pixel Fold, despite their visibility on the screen.

Performance issues also occur on smaller devices like the Pixel 3a, despite its slight deviation from typical phone sizes. For instance, as is shown

Table 8: Different multi-modal modes of instruction tuning. We use the same set of training data but only add a set-of-mask index on SoM mode. Note that AITW dataset even could not provide accurate bbox, but only point. We use CogVLM2 as base model.

Operation Mode	SR	Sub-SR	RRR	ROR
BBOX	5.79	6.03	47.95	55.05
SoM	11.59	16.06	57.37	85.58

Table 9: The impact of the ReAct and SeeAct frameworks. Notably, model performance is significantly improved in XML+ReAct mode.

Mode	**Model**	**SR**	**Sub-SR**	**RRR**	**ROR**
XML	GPT-4o	25.36	30.56	107.45	86.56
	Gemini-1.5-Pro	18.84	22.40	57.72	83.99
XML+ReAct	GPT-4o	33.33	38.22	97.93	90.74
	Gemini-1.5-Pro	31.16	34.54	92.08	90.31
XML+SeeAct	GPT-4o	24.64	27.31	93.78	79.62
	Gemini-1.5-Pro	21.01	25.53	75.97	89.06
SoM	GPT-4o	31.16	35.02	87.32	85.36
	Gemini-1.5-Pro	16.67	18.48	105.95	91.52
SoM+ReAct	GPT-4o	31.88	39.19	104.69	89.80
	Gemini-1.5-Pro	15.94	21.38	109.81	84.16
SoM+SeeAct	GPT-4o	30.43	36.24	97.45	88.56
	Gemini-1.5-Pro	21.01	25.53	75.97	89.06

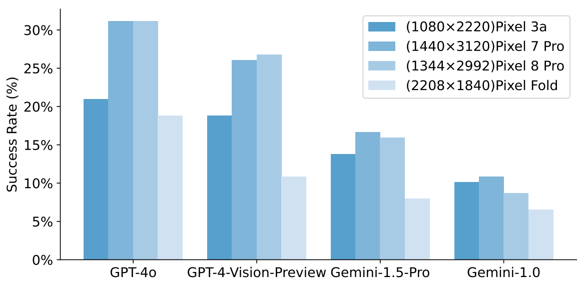


Figure 19: The performance of five models across four different device types is presented. Among these, the Pixel 3a is a smaller-sized phone, the Pixel 7 Pro and Pixel 8 Pro are of sizes comparable to commonly used phones, and the Pixel Fold is akin to a tablet.

in Fig 18, on Pixel 7 Pro and Pixel 8 Pro, the "Dark Theme" setting is accessible immediately, while on Pixel 3a, it requires swiping or searching. Evaluation setting like GPT-4o in SoM mode, which relies on visible information, struggled there, failing this task on Pixel 3a but succeeding on larger devices.

F.5 Evaluation Accuracy of Query Tasks

To check the accuracy of the query task evaluation using the LLM-judge method, we randomly sampled 50 examples for manual review. We asked the annotators to determine whether the task was completed based on the screenshots, operations in the completion record, and the finish information. Then, we compared their judgments with our automated method. Among these sampled query tasks, 49 were accurately evaluated by the LLM-judge method, resulting in an accuracy rate of 98%. One judgment was somewhat controversial. The task was " Could you tell me how much I spent on May 10, 2024?" The correct answer should have been "11400CNY," but the finish message only provided the price without including the unit. The LLM-judge considered this response incorrect, although this judgment is debatable.

Here is our LLM-judge prompt:

You need to judge the model answer as True or False based on the Standard Answer we provided. You should return either [True] or [False].

Question: {question}

Model Answer: {model_answer}

Standard Answer: {standard_answer}

F.6 Out-of-domain Evaluation

In this work, we are committed to providing an in-domain training and test set. However, our data collection method can easily be extended to nearly all apps. Unfortunately, for most commonly used apps, we cannot conduct directly reproducible tests. Nevertheless, we chose the AITW web shopping subset provided by Digirl (Bai et al., 2024) as our out-of-domain (OOD) test set to evaluate our model's generalization ability. This test selected 96 tasks from the AITW web shopping subset as the test set, which were executed interactively in the emulator and evaluated using the advanced model to determine whether they were correctly executed.

We compared all offline methods, and our method achieved higher test results post-SFT with llama-3.1-8b than all previous methods, second only to the online training method proposed by Digirl, as shown in FIg 10.

In future work, we will further explore the possibility of extending our existing methods to a broader domain. This includes collecting data from more apps and adopting exploration-based reinforcement learning methods, among other strategies.

G Introduction of Android Debug Bridge(ADB) usage in ANDROIDLAB

Android Debug Bridge (ADB) is a powerful and widely used command-line tool that serves as a communication bridge between Android devices and host machines. ADB is part of the Android Software Development Kit (SDK) and is crucial in enabling developers and researchers to interact with Android devices for debugging, automation, and data collection. By providing a unified interface, ADB allows users to execute commands, transfer files, manage apps, and retrieve system information, making it an essential tool in Android development and testing.

One of ADB's primary strengths lies in its versatility. It supports various operations, such as installing and uninstalling applications, reading system logs, capturing screenshots, and automating user interactions. ADB is compatible with physical devices, emulators, and virtual machines, which makes it a flexible solution for various experimental and development scenarios. Furthermore, its integration with shell commands gives users granular control over device functionality, including accessing low-level system settings and processes.

ADB is widely utilized in Android-related research. For example, AndroidEnv (Toyama et al., 2021), a simulation environment for reinforcement learning, uses ADB for tasks such as app launching, querying activities, resetting episodes, and handling task extras, serving as a foundation for works like AITW (Rawles et al., 2023). AppAgent (Yang et al., 2023b) employs ADB to define action spaces, leveraging multimodal methods with GPT-4v for Android device control. AndroidArena (Xing et al., 2024) addresses challenges in Android evaluation, using ADB for action operations and XML information retrieval in its benchmark implementation. AITW (Rawles et al., 2023) utilizes ADB to execute tasks in creating a dataset of over 5 million Android screenshots. Similarly, Digirl (Bai et al., 2024) applies offline reinforcement learning for Android performance enhancement, employing ADB for screen data retrieval and device interaction.

G.1 How Our Work Utilizes ADB

G.1.1 Data Collection

ADB is utilized to extract XML-based user interface information and capture screenshots from Android devices. These capabilities enable systematic analysis of UI layouts and visual feedback, providing a foundation for evaluating app performance and user interaction flows.

G.1.2 Device Control

ADB commands allow precise control of Android devices, facilitating tasks such as launching applications, simulating user interactions (e.g., clicks, swipes, and text input), and managing input events. These functionalities are critical for ensuring reproducibility in experimental workflows, as they eliminate human variability and automate complex interaction sequences.

G.1.3 Performance Overhead

To address potential performance overhead caused by frequent ADB command executions, we incorporate delays ranging from 3 to 5 seconds between commands. Additionally, we provide adequate initialization time for each device or virtual instance to ensure a stable environment. Empirical observations from our experiments confirm that these measures mitigate significant performance delays attributable to ADB, preserving the accuracy and reliability of our results.

G.1.4 Communication Stability

To improve communication stability, we standardize the use of Android Virtual Devices (AVDs) as docker in experimental platform. This approach eliminates common issues such as USB disconnections or unstable network connections, ensuring a consistent and reliable testing environment.

Table 10: AITW Web Shopping Test Accuracy for Different Models

Model	Method	AITW Web Shopping Test Accuracy (%)
GPT-4V	Set of Mark	8.3
Gemini 1.5 Pro	Set of Mark	11.5
CogAgent	Supervised Training	38.5
AutoUI	Supervised Training	17.7
Digirl	Filtered Supervised Training	45.8
AndroidLab	Supervised Training (llama-3.1-8b)	48.5

G.1.5 Limitations

While ADB offers extensive control over Android devices, it has several limitations. For instance, ADB cannot simulate sensor data such as accelerometer readings, biometric inputs like fingerprints, or hardware-specific features such as NFC communication. These constraints highlight the need for alternative methods or tools to complement ADB in specialized scenarios. Despite these limitations, ADB remains an invaluable tool for automating and standardizing Android research and testing workflows.

H AI Assistants In Writing

During the writing of this paper, we used AI to correct grammatical errors and unreasonable descriptions.

I Details of Android Instruction Dataset

I.1 Overview of Data Construction

1. **Task Derivation and Expansion**: We used academic datasets (Rawles et al., 2023; Coucke et al., 2018) and manually wrote instructions to seed task generation. Language models were employed to create additional tasks, which were reviewed and added to the dataset, ensuring realistic and executable instructions.

2. **Self-Exploration Reward Model Construction**: First, we utilized advanced Large Language Models (LLMs) and Large Multimodal Models (LMMs) to automate the construction of trajectories. Using the instructions we had generated, we tasked these models to autonomously complete tasks in AVD, with both humans and models annotating whether the tasks were successfully completed. We improved upon the method described in (Pan et al., 2024), exploring and determining an approach to build a reward model using combined images as input information (Cf. Appendix I.2). This reward model achieved an accuracy rate of 87.64

3. **Manual Annotation**: This process involved four steps:

(1) **Instruction Check**, where annotators evaluated the feasibility of the given tasks;

(2) **Preliminary Familiarization**, allowing them to explore the app interface before performing tasks;

(3) **Task Execution**, in which the annotators executed and documented each task step;

(4) **Cross-Verification**, where a second annotator reviewed based on direct observation of the operation sequence, and the reward model scored the task trace to ensure its accuracy. If either of the two checks fails, we will ask the annotator to re-annotate.

I.2 Details of Reward

In order to develop a reward model, a subset of tasks was selected from the training data. The model, which had undergone preliminary supervised fine-tuning, was tasked with performing multiple rounds of sampling on these tasks. Subsequently, the sampled trajectories were reviewed by GPT-4, which evaluated their correctness and provided a rationale for its decisions. These evaluations formed the training data for our reward model. We constructed 3000 samples for training and 300 samples for evaluation.

When determining the criteria by which the reward model should evaluate the trajectories, three methods were devised:

1. Using the compressed XML of the final step.
2. Using a screenshot of the final step.
3. Combining screenshots from all steps in the trajectory into a single large image.

In Table 11, we compare the accuracy on the test set (relative to human annotations) achieved using different methods. The results show that the Combined Image method achieves the best reward model accuracy.

Table 11: The accuracy of different reward model construction methods on the human-annotated test set.

base model	Final XML	Final Image	Combined Image
llama3.2-11b-vision	/	72.87	69.77
qwen2vl-7b-inst	/	81.40	87.64
llama3.1-8b-inst	77.62	/	/

We use the following template as the reward model's instruction:

You are an expert in evaluating the performance of an Android navigation agent. The agent is designed to help a human user navigate the device to complete a task. Given the user's instructions and all screenshots of the agent executing the task, your goal is to decide whether the agent has successfully completed the task or not.

All screenshots of the task are stitched together in the image. You must go through all the screenshots one by one.

CAREFUL! You need to pay more attention to the image than the agent's finish message because the agent might hallucinate!

IMPORTANT Format your response into two lines as shown below:

Thoughts: <your thoughts and reasoning process>" Status: "YES" or "NO"

User Instruction: {instruction}

Action History: {last_actions}

Bot response to the user: {response if response else "N/A"}.

I.3 Annotation Tool

We designed an annotation tool to record operation trajectories and page information (XML) more accurately and efficiently.

Acquisition of Page Information: Android Debug Bridge (ADB) is currently the most widely used tool for obtaining page information (Yang et al., 2023b; Rawles et al., 2024). ADB is a versatile command-line utility that retrieves the XML data of the current page. However, when dealing with a diverse range of mobile applications, ADB sometimes fails to acquire the XML for particular pages. Specifically, ADB waits for all UI components on the page to become idle before retrieving component information. ADB stops the XML acquisition if this process exceeds a predefined time limit. This issue is particularly evident on mobile pages with dynamic components, such as playback bars and animations in audio players, where continuously active elements prevent ADB from obtaining the XML. To address this, we reimplemented the XML acquisition functionality using the Android Accessibility Service, allowing annotators to determine the appropriate timing for retrieving page XML.

Recording Operation Trajectories: We mainly need to record three types of user actions: clicks, swipes, and text input. For click actions and swipe actions, annotators complete the actions directly on the phone, while we use ADB commands to capture screen events. We determine whether the action was a click or swipe based on the press, release positions, and duration of these events. We utilize the ADB keyboard for text input to complete the entire input in a single operation, minimizing the number of annotations required. Before each action, the user must first use the annotation tool to record the current page information, ensuring that the recorded page data matches the context observed during human interaction.

I.4 Details of Human Annotation

In the process of constructing our data, we utilize crowdsourced annotations. To ensure that the privacy information of the annotators is not disclosed, we adopt the following measures:

1. Before the annotation begins, we explicitly inform the annotators that the annotated data will be used to fine-tune models, and part of the data will be open-sourced. Annotators who disagree may opt out of the annotation process.

2. During the annotation process, all annotated data are first stored locally by the annotators. If an annotator believes that specific data involves privacy disclosure, they may choose not to use it or skip the task.

3. After the annotation is completed, we mask and replace sensitive information such as usernames and chat logs before using the data for training. Additionally, such data will not be open-sourced.

All annotators sign formal contracts and are compensated according to reasonable standards.

I.5 Instructions Given To Annotators

We provide the instructions given to the annotators below. Note that our targets are expanded by hand-written instructions or academic datasets with available licenses.

Task Overview

For each labeling task, a target task will be given, such as: *Navigate to XXX using Amap (Gaode Map).*

The annotator must complete the task using their phone and follow the labeling process described below to ensure it is accurately executed and recorded.

To perform this annotation task, you must install ADB (Android Device Bridge) on your computer to control the phone and install the corresponding APK. Since the task involves collecting low-level information, we will require the phone to enable multiple permissions. Still, we guarantee that the information will not be transmitted in real-time during collection. The transmitted information includes the operation details, screenshots before and after each operation, and the corresponding XML files (only containing information from the current page). You can review and decide whether to keep the annotation data. If the annotation process involves screenshots or other information that you do not want to be used for training, you can:

1. Skip the screenshot or specify that parts of the screenshot be hidden.
2. Skip the entire target task.
3. Skip all tasks involving the currently annotated app.

Your data will not be used for purposes other than training the model.

After completing the annotation, you must upload all the tasks you were responsible for in one go. We have designed a plugin to store all the content in a unified folder.

A complete annotation consists of multiple operations is called a sequence (trace). Each single-step operation is recorded once, and the definition of a single-step operation is detailed in the annotation documentation.

Please follow the steps below for plugin usage to install the annotation plugin.

Plugin Usage Instructions

Installing ADB and Connecting Phone to Computer

For your Android phone, you need to perform the following settings:

1. Connect the phone to the computer via a USB cable.
2. Ensure that the **Developer Options** and **USB Debugging Mode** are enabled on the Android phone:
   - Go to *Settings - Developer Options - Android Debugging*. Check the box for *Allow USB debugging*. If unavailable, go to *Settings - System Updates - Developer Options - USB Debugging*.
   - If you can't find the developer options, go to *Settings - About Phone* and tap the *Build Number* seven times.
   - If these methods don't work, search for how to enable developer options and USB debugging specific to your phone model.
   - If you still encounter issues, seek help in the group chat.
3. Reconnect the phone to the computer, and on the phone, click *Allow file transfer/USB debugging/higher permissions*. Also, allow the connection on the computer (if prompted).
4. After entering Developer Mode, turn off the following animations under *Developer Options* to increase the success rate of retrieving XML information via ADB commands:
   - Window Animation Scale.
   - Transition Animation Scale.
   - Animator Duration Scale.

Follow the steps above until the following result is displayed using the command *adb devices*:

```
adb devices
List of devices attached
1a0d5d59 device
```

The number before *device* is randomly generated. You should see only one device. If there is more than one, try disconnecting other devices or closing virtual machines.

Installing ADB Keyboard

Download the ADB Keyboard APK.

Run: `adb install <APK full path>`

Enable permissions on the phone and agree to the installation.

Once the installation is complete, set ADB Keyboard as the default input method in the phone settings. You can try the following two lines of code:

```
ime enable com.android.adbkeyboard/.AdbIME
ime set com.android.adbkeyboard/.AdbIME
```

If successful, when you open any text box, you'll see the message *ADB Keyboard ON* at the bottom of the screen. If unsuccessful, manually change the input method in the settings.

Running Test Script

1. Open the command line, run *adb devices*, and ensure correct output.

2. Run the following commands in adb shell:

   ```
   input keyevent KEYCODE_BACK
   input keyevent KEYCODE_HOME
   input keyevent KEYCODE_ENTER
   ```

   If there's no error or response, it's fine. If you see *Command execution failed*, ensure you're using the correct method sequence, not *Press xxx* commands like *adb shell input keyevent KEYCODE_A*.

3. Open any text input field and run the following commands in adb shell:

   ```
   input keyevent KEYCODE_A
   ```

   The setup succeeds if the letter "a" appears on the screen.

Annotation Plugin Usage Instructions

You can perform the following operations on the phone. After completing any one of these operations, do not proceed until the command line shows *Operation completed*. If the phone has not responded yet (such as loading a new page), wait until the page is fully loaded before clicking the next *Begin*.

1. **Click or Swipe**: Perform this directly on the phone. Click slowly, holding for 0.2 to 0.5 seconds.

2. **Text Input**: If the ADB Keyboard was successfully installed, you can input text. Before entering text, click on the text box in the previous step and ensure that the *ADB Keyboard ON* symbol appears at the bottom of the screen. Click the *Type* button on the GUI interface, enter the desired text in the computer's input box (Chinese/English), then click *OK*. You will observe the input on the phone, and the command line will display *Simulating typing xxx*.

3. **Press xxx**: Three preset buttons are defined: *Press Home* (Home key), *Press Back* (Back key), and *Press Enter* (keyboard Enter key). The command line will show *Simulating press xxx*.

4. **Finish Task**: If you believe the task is complete, click the *Finish* button on the GUI. If the task requires an answer, fill in the response in the popup text box. If not, click *OK*.

After finishing a task, you can close the command line and GUI windows. If there are no issues with the annotation, you can return to Step 2 to start the next annotation. Otherwise, follow these steps:

1. The command line will output the *Save Path*, which contains all saved information for the annotation. You may delete the folder if you believe an error occurred or sensitive information was recorded.

2. Each task has a prefix consisting of the first 32 characters of the task name. Ensure that the final submission includes one and only one instance of each non-skipped task.

3. If certain operations were recorded incorrectly without affecting the phone's state, you may delete those steps. The step sequence is stored in *Save Path/traces/trace.jsonl*. Record the steps you need to delete.

4. If a screen contains sensitive information that can be removed while still being used for training, record the steps and describe the sensitive information in detail.

Summary of Key Points

1. Always use *adb devices* before starting the annotation to ensure a successful connection.

Table 12: Actions Counts

Action	Count
Tap	58383
Type	13533
finish	10586
Swipe	6600
Launch	5220
Back	52

2. Reopen the app_for_xxx/dist/label(.exe) for each annotation instruction.

3. The storage path must not contain Chinese characters.

4. Click *Begin* before each operation and wait for the message *Begin your operation...* to appear before proceeding. If you proceed without waiting, the operation will be invalid. If the state cannot be recovered, you must restart the task. Make sure to click *Begin* before finishing as well.

5. After each operation is completed, wait until the corresponding success message appears in the command line and you see the output *Operation completed* before clicking *Begin* for the next action. Failure to follow these two key rules may result in invalid data. It's better to proceed slowly and carefully than rush and make mistakes.

I.6 Detailed Statistics of Android Instruct dataset

We provide statistics of the Android Instruct dataset in Fig 20.

I.7 Actions

Android Instruction dataset includes a wide variety of user actions, with the frequency of each type of action carefully recorded. These actions are summarized in Table 12.

These statistics show the diverse nature of user interactions we captured in our data. They provide essential insights for understanding and modeling user behaviors in detail.

I.8 Apps

Table 13 presents the number of traces and the average trace length for each app in Android Instruction dataset. This detailed breakdown provides valuable insights into how users interact with different apps, which is important for improving model performance.

These statistics show the volume and complexity of interaction data across various apps. This information is critical for helping models understand how users interact with these apps.

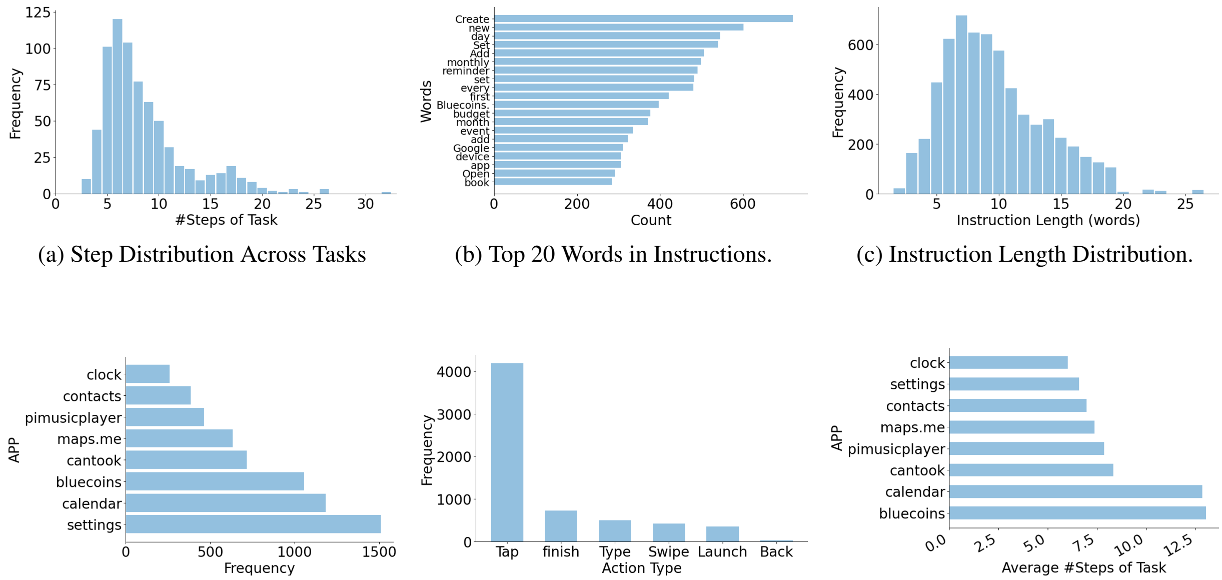
J Discussion about ANDROIDLAB's different from Web Agents

Android agents differ from general web agents, such as those developed within frameworks like WebArena, in several key aspects. These distinctions arise from differences in their environments, action spaces, and reproducibility challenges.

First, the environments in which these agents operate are inherently different. Android agents primarily rely on XML-based information to interact with mobile applications, reflecting the structural characteristics of mobile interfaces. In contrast, web agents depend predominantly on HTML/DOM data and often incorporate screen screenshots as part of their observation space, leveraging the structured nature of web environments.

Second, the action space of Android agents is specifically tailored to mobile interactions. These actions include tapping, swiping, typing, and pressing hardware buttons such as Home and Back, all miming typical user behavior on mobile devices. On the other hand, web agents interact with web elements through actions like clicking, keypressing, and navigating URLs, with their interactions rooted in the manipulation of DOM trees and other web-based structures.

Finally, reproducibility poses unique challenges for each type of agent. For Android agents, dynamic environments and network dependencies often complicate reproducibility. To address these issues, we employ preloaded virtual devices and offline setups, ensuring consistent experimental conditions. In the case of web agents, frameworks like WebArena mitigate reproducibility challenges by using self-deployed websites, thereby reducing reliance on external and potentially inconsistent web environments.



(a) Step Distribution Across Tasks

(b) Top 20 Words in Instructions.

(c) Instruction Length Distribution.

(d) APP Distribution.

(e) Actions Distribution.

(f) Average Task Length per App

Figure 20: Statistics for Android Instruct dataset. We collect 726 traces and 6208 steps across Apps in ANDROIDLAB benchmark.

Table 13: Top 10 apps ranked by trace count, along with their Average Trace Length.

App	Trace Count	Average Trace Length
chrome	3698	9.50
twitter	1388	7.61
google maps	633	7.85
gmail	399	9.37
quora	334	8.57
booking.com	334	12.43
settings	295	6.81
temu	293	8.69
tasks	252	7.32