Unlocking Smarter Device Control: Foresighted Planning with a World Model-Driven Code Execution Approach

Xiaoran Yin[1]*, Xu Luo[1]*, Hao Wu[1], Lianli Gao[1], Jingkuan Song[2]†
[1]University of Electronic Science and Technology of China, [2]Tongji University

Abstract

The automatic control of mobile devices is essential for efficiently performing complex tasks that involve multiple sequential steps. However, these tasks pose significant challenges due to the limited environmental information available at each step, primarily through visual observations. As a result, current approaches, which typically rely on reactive policies, focus solely on immediate observations and often lead to suboptimal decision-making. To address this problem, we propose **Foresighted Planning with World Model-Driven Code Execution (FPWC)**, a framework that prioritizes natural language understanding and structured reasoning to enhance the agent's global understanding of the environment by developing a task-oriented, refinable *world model* at the outset of the task. Foresighted actions are subsequently generated through iterative planning within this world model, executed in the form of executable code. Extensive experiments conducted in simulated environments and on real mobile devices demonstrate that our method outperforms previous approaches, particularly achieving a 44.4% relative improvement in task success rate compared to the state-of-the-art in the simulated environment.

1 Introduction

Mobile devices have become central to modern life, facilitating a wide range of activities such as browsing the web, reading news, communicating via email, online chatting, ticket booking, and shopping. As reliance on these devices intensifies, the development of autonomous agents that can seamlessly replicate human interactions to perform such tasks is increasingly crucial. These agents must autonomously interpret textual and visual information from views (i.e., screenshots) and language instructions, navigating the complexity of device interfaces to execute precise actions.

Recent advancements in large language models (LLMs) and vision language models (VLMs) (Zhu et al., 2023) provide a promising foundation for developing such agents. When prompted effectively (Qin et al., 2024), these models can generate structured, executable actions given task descriptions and observations. Existing works like AppAgent (Zhang et al., 2023) and Mobile-Agent (Wang et al., 2024a) have leveraged VLMs to create device-control agents.

However, most existing methods encounter difficulties in making reliable decisions, resulting in a low task success rate. This challenge primarily arises from the complexity of executing tasks on mobile devices, which require a sequence of distinct steps transitioning between views. Each step provides only limited contextual information about the underlying digital environment. Consequently, to accurately predict globally optimal actions, agents must first develop a comprehensive understanding of the environment. Furthermore, they need to implement a forward-looking policy based on this understanding to anticipate the long-term consequences of their actions. In contrast, most prior approaches operate in a ReAct-like action loop (Yao et al., 2022), heavily relying on information from the current observation and a coarse-grained historical context, which leads to myopic decision-making.

To overcome these limitations, we propose Foresighted Planning with World Model-Driven Code Execution (FPWC), a novel framework that enables agents to construct a global understanding of the environment they operate in, referred to as the world model (Ha and Schmidhuber, 2018b) (See Figure 1). A world model is essentially a simulation of the environment, such that planning can be done directly within the model, and it enables the agent to predict future states and outcomes without direct

*Equal contributions.
†Corresponding Author.

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 5643–5666
November 4-9, 2025

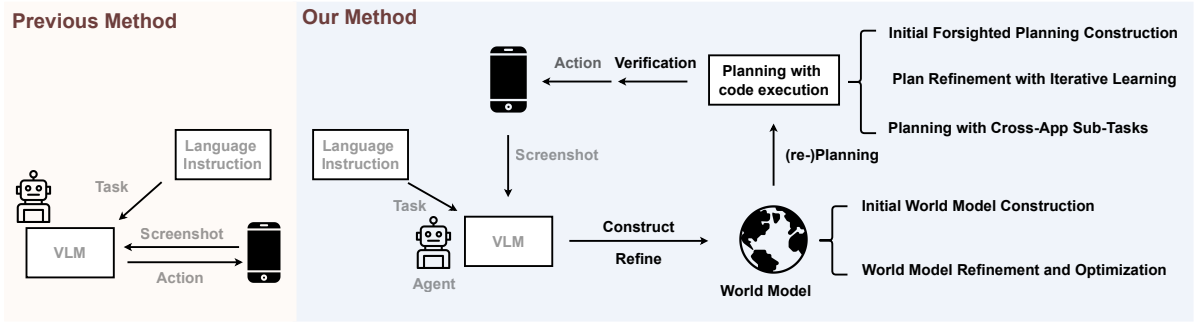

Figure 1: **Comparison of our proposed framework, Foresighted Planning with World Model-Driven Code Execution (FPWC), with existing approaches.** Unlike prior methods that predominantly rely on reactive actions based solely on immediate observations, our framework leverages a task-specific world model to achieve a more comprehensive understanding of the environment. This world model facilitates iterative planning and the generation of executable code tailored to task completion. Furthermore, both the world model and the planning process are designed to be refined dynamically in an online manner.

interaction with the real environment (Hafner et al., 2019a). We represent the world model as a text-based directed graph, where descriptions of views are the nodes and transitions between views are edges. The agent initializes this model given task description, using pre-trained knowledge in VLMs to infer precise, task-relevant contents written as structured texts. The agent then makes explicit plans by specifying a rule-based policy written as executable code, specifying how an agent should traverse the world model in order to complete the task. As tasks progress, the agent dynamically refines the world model and the planning process in a closed loop, ensuring adaptability and online learning.

To constrain the complexity of the constructed world model, we strictly limit it to describing a single mobile application that is most pertinent to the task at hand. To adapt our method for tasks necessitating interaction between multiple applications, we incorporate a recursive function within the framework. This function, when called in a plan, generates a sub-agent responsible for executing sub-tasks in other applications that are essential for the current task. The sub-agent mirrors the main agent, possessing its own world models and plans for the sub-task delegated by the main agent, where natural language instructions guide both high-level planning and low-level execution.

In summary, this paper makes the following contributions:

- We introduce a novel language-centric task-oriented world model for device-control agents that enhances the agent's global understanding of the environment by representing it as a text-based directed graph, where nodes are linguistic descriptions of views and edges are transitions between them.
- We develop a foresighted planning mechanism that allows the agent to generate executable code for task completion by specifying a rule-based policy within the world model, ensuring adaptability and continuous learning through dynamic refinement.
- Extensive experiments in both simulated environments and on real mobile devices demonstrate that our approach significantly outperforms existing methods, achieving a 44.4% relative improvement in task success rate in the simulated environment. These empirical results further yield critical insights into the design and effectiveness of world model-based planning algorithms for mobile device control tasks.

2 Methodology

To help the device-control agent make informed and foresighted decisions at each step, we endow it with the capability to generate grounded, executable plans based on a self-constructed and dynamically updated world model – a structured, text-based representation of the app. The following sections provide a detailed overview of our framework FPWC. See Figure 1 for an overview.

2.1 Preliminaries

Device-control agents. We formalize mobile device control as a partially observable Markov decision process (POMDP) with discrete time step $t \in [1, T]$. At each step, given device system state

s_t, an observation o_t (e.g., screenshots) is generated via an observation function $o_t \sim p(o_t|s_t)$. For a task goal g specified by text, an agent performs an action a_t based on history observations and actions via a *goal-conditioned* policy $a_t \sim \pi(a_t|o_{\leq t}, a_{<t}, g)$, resulting in the next state s_{t+1} via a transition function $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$. A trajectory induced by a policy π is defined as $\tau = \{s_t, a_t\}_{t=1}^{T} \sim p_\pi(\tau)$. The goal is to implement a policy π that maximizes the expected success rate $\mathrm{E}_{\tau \sim p_\pi(\tau)}[\Pr(f(\tau) = 1)]$ with the success detector $f(\tau)$.

World models. Formally, a world model is a simulation of the underlying transition function $p(s_{t+1}|s_t, a_t)$ (and optionally the observation function $p(o_t|s_t)$) of the POMDP (Ha and Schmidhuber, 2018b). This simulation is achieved through an approximation $p_\theta(s_{t+1}|s_t, a_t)$, which is parameterized by θ. Utilizing a learned world model, *planning* can be effectively achieved by optimizing a sequence of actions $\{a_i\}_{i=t}^{T}$ that maximizes the task success rate within the simulated environment constructed by the model: $\max_{\{a_i\}_{i=t}^{T}} \mathrm{E}_{s_{t+1} \sim p_\theta(s_{t+1}|s_t, a_t)}[\Pr(f(\tau) = 1)]$. Crucially, the planning process is done without interacting with the actual environment.

2.2 Initial World Model Construction

A significant challenge in device-control agents is the limited information about the environment that can be obtained from observations. To facilitate a comprehensive understanding of the environment and thereby develop a more anticipatory policy, we construct language-centric world models for device-control agents.

In constructing a world model $p_\theta(s_{t+1}|s_t, a_t)$ as an approximation of the underlying transition function, it is essential to incorporate knowledge of both (1) all linguistically defined states within the environment and (2) all possible transitions between these states. Unlike real-world environments which possess an infinite number of states with continuous changes, the states of mobile devices are finite, and their transitions are discrete. For instance, the settings application on a mobile device typically consists of only a limited number of pages. Therefore, the world model of mobile devices can be effectively represented as a graph $G = (V, K)$, where the set of vertices V corresponds to textual descriptions of states (i.e., pages) and the set of edges K represent the possible transitions between these states.

In what form should we model the vertices and edges within the graph? Given our objective to furnish the agent with a high-level comprehension of the environment, a detailed graph model is unnecessary. Consequently, we employ natural language descriptions to represent states rather than predicting raw images, and we use high-level action labels, such as "tap the Wi-Fi button," to denote edges. This approach is inspired by the latent states utilized in Dreamer (Hafner et al., 2019a). An illustrative example is provided below:

```
# Settings app navigation graph
Vertices:
  Name: "Main page of the Settings app"
  Description: The main page of the Settings App that can be
used to navigate to different settings of the phone.

Edges:
  Edge: E("Main page of the Settings app", "swipe down") ->
"Main page of the Settings app" # Show more settings in the
bottom
```

Each vertex is characterized by a concise name label that encapsulates the state, accompanied by a comprehensive description pertinent to the task. Each edge is defined by a function E, which maps a given vertex and an action to a resulting vertex. Note that we do not explicitly model the exact probability distribution of $p_\theta(s_{t+1}|s_t, a_t)$. Instead, we enumerate all potential states that may arise from a given state s_t and action a_t. The associated uncertainties are addressed through the use of logical statements within code-based planning, as detailed in Section 2.3. This approach is complemented by our functional and structured textual representation of world models, which facilitates seamless integration.

To construct the world model, one potential approach involves manually creating a graph for each specific task. However, this method is labor-intensive and lacks the ability to generalize to novel tasks. Instead, we propose utilizing LLMs to enable the agent to approximate the graph. Specifically, we prompt LLMs to construct the graph given the task instruction before its initial interaction with the environment:

$$G = \mathrm{LLM}(g), \tag{1}$$

where g represents the task goal represented in text form. Given that LLMs have been pretrained on extensive text corpora, we anticipate that they possess the requisite knowledge, although it must be explicitly structured, akin to the chain-of-thought methodology employed in Large Language Models (LLMs) (Wei et al., 2022). Furthermore, we provide an in-context example from a different task

to assist the agent in constructing an initial graph that encompasses the essential elements necessary for task completion (see the prompt template in the supplementary material for details). Although this initial model may not be comprehensive, it serves as a foundational framework for subsequent refinement, as elaborated in Section 2.4.

2.3 Initial Foresighted Planning Construction

The primary advantage of employing world models lies in their capacity to enable an agent to develop a comprehensive understanding of its environment, thereby facilitating grounded and precise planning. While a straightforward approach to planning involves determining a sequence of actions (Song et al., 2023; Hafner et al., 2019b), certain tasks encompass uncertainties or iterative processes that cannot be effectively addressed through a purely sequential method. For instance, in the task of enabling Wi-Fi, the agent must first verify whether Wi-Fi is currently disabled. If it is, the agent should iteratively test various Wi-Fi networks to identify an available connection. To manage such uncertainties, we utilize programming languages, such as Python, which possess the capability to handle logical statements, as the interface for planning. Python's Turing-completeness, structured syntax, and compatibility with vision-language models (VLMs) pretrained on code (Li et al., 2023; Roziere et al., 2023) render it an optimal choice. Given the world model and visual observations, we prompt the VLM to generate a Python function to address the task. An illustrative example of enabling Wi-Fi is provided below:

```
def plan():
  E("Main page of the Settings app", "tap Wi-Fi button")
  if not isTRUE("Wi-Fi button on"):
     E("Wi-Fi (WLAN) settings", "tap the WLAN switch")
     ...
  return "Task completed"
```

In our approach, we redefine the edges within the world model, denoted by the callable function E, as abstract actions within the planning process. During the execution of these functions E, the abstract actions (e.g., "tap Wi-Fi button") are translated by VLMs into commands executable by mobile devices. Given the limited visual grounding capabilities of VLMs (Zheng et al., 2024), we incorporate a verification step into the translation process. Specifically, when selecting an element on the views for interaction, we "zoom in" by cropping the element and querying the VLMs to confirm whether the element matches their intended target. If the element is not as expected, the VLMs will select an alternative. The construction of the plan is facilitated through meticulously prompted VLMs, with inputs comprising the task instruction, the graph of the world model, and the initial visual observation:

$$\text{plan} = \text{VLM}(g, G, o_0). \tag{2}$$

To address uncertainty, we introduce an additional function, `isTRUE`, which dynamically verifies specific conditions given current visual observation. When the graph lacks the necessary information to solve a task, the VLM is capable of synthesizing new vertices and edges, which are marked with an `imagined` parameter set to True. Furthermore, an in-context example is included in the prompt to assist the agent in formulating appropriate plans (see the prompt template in the supplementary material for details). To ensure robustness, the agent executes the code within a controlled sandbox environment, where it is dynamically compiled and executed. In the rare event of execution failure, the agent captures the error and recursively refines the plan based on the current context. Our methodology effectively combines the adaptability of world models with the precision of programming, enabling the agent to address both simple and complex tasks effectively.

2.4 World Model and Plan Refinement with Iterative Learning

An intelligent agent should progressively learn about the environment (Zhao et al., 2024) and revise its knowledge when encountering contradictory observations. In our device-control agent, the initial graph and plan constructed before task execution are only rough estimates and may contain errors. To address this, we enable the agent to refine the world model and plan dynamically based on new observations (see the prompt template in the supplementary material for details):

$$G_{\text{new}}, \text{plan}_{\text{new}} = \text{VLM}(g, G, \text{plan}, o_t, a_{t-1}). \tag{3}$$

The action a_{t-1} denotes the most recent action executed by the agent, facilitating the contextualization of the refinement process. During the refinement process of the world model, the agent is permitted to modify the graph structure by adding, removing, or replacing vertices and edges in response to conflicting observations. Regarding the planning process, the agent generates a new, complete plan from the current step onward whenever necessary.

Algorithm 1 Foresighted Planning with World Model-Driven Code Execution

Require: Language instruction g, Initial observation o_0, Timestep T

procedure INITIALIZATION
  $G = LLM(g)$ ▷ Initialize world model
  $P = VLM(g, G, o_0)$ ▷ Initialize plan
  $c = CodeParser.get\_next\_line(P)$
  $t = 0$ ▷ Code in plan,refresh step
end procedure
FPWC Loop:
while c is not $\varnothing$ and $t < T$ **do**
  if $E(\cdot)$ in c **then**
    $A_{wrong} = \varnothing$ ▷ Wrong actions set
    $verify = False$ ▷ Action Verification
    while $verify$ is $False$ **do**
      $a_t = VLM(E(\cdot), A_{wrong})$
      $o_{crop} = Crop(o_t, a_t)$ ▷ Zoom in
      $verify = VLM(g, o_{crop}, E, a_t)$
      $A_{wrong} = A_{wrong} \cup \{a_t\}$
    end while
    $o_{t+1} = Execute(o_t, a_t)$
      ▷ Refine world model and plan
    $G, P = VLM(g, G, P, o_{t+1}, a_t)$
  else if $other\_app\_agent(\cdot)$ in c **then**
    $New\_Agent(subtask, new\_App\_name)$
  else
    $Execute(c)$ ▷ Execute normal code
  end if
  $c = CodeParser.get\_next\_line(P)$
  $t = t + 1$
end while

Upon completion of the task, the agent has the option to save the refined graph for future use. This capability allows the agent to leverage prior knowledge, thereby enhancing efficiency over time. The process of self-refinement ensures that the agent continuously adapts, rectifying errors and deepening its understanding with each subsequent task. Further exploration of this mechanism for continual learning on multiple tasks is reserved for future research.

2.5 Planning with Cross-App Tasks

Mobile devices often require the use of multiple applications to complete certain tasks. For instance, following a user on YouTube may involve first enabling Wi-Fi through the "Settings" app if it is turned off, and then returning to the YouTube app to complete the task. Integrating the world models of multiple apps into a single prompt is both challenging and computationally expensive. To address this, our approach focuses on constructing world models for individual apps. To manage cross-app tasks, we introduce a hierarchical agent framework, where a "parent agent" can dynamically create a "child agent" when a task extends beyond the scope of the current app. This is achieved by incorporating a function, `other_App_agent(AppName, Subtask)`, into the parent agent's plan. When invoked, this function spawns a new computational process to instantiate the child agent, which possesses the same capabilities as the parent agent. The child agent independently constructs its own world model and generates a plan to achieve the specified subtask. During this time, the parent agent's process is temporarily suspended until the child agent completes its task. Once the child agent finishes, control is returned to the parent agent, which resumes its original task. This recursive agent creation mechanism can be invoked multiple times within a single task, as needed. Our full algorithm is summarized in Algorithm 1.

3 Experiment

Through our experiments, we aim to address three pivotal questions that align with the objectives outlined in our study: (1) How does our approach compare to state-of-the-art device-control methods in terms of metrics such as success rate and efficiency across benchmark tasks? (2) Can our method effectively tackle tasks that previous approaches fail to solve due to the demand for foresight planning? (3) How well does our agent generalize to tasks on real-world mobile devices, including cross-application scenarios? In order to gain deeper insights, we further evaluate (a) the extent to which individual components contribute to the overall performance of our agent, and (b) the trade-offs between computational efficiency (e.g., latency, resource usage) and task execution performance.

3.1 Experiment Setup

To answer the questions above, we conducted experiments primarily utilizing **MobileAgentBench**, a standardized framework designed for evaluating device-control agents. This benchmark comprises a curated set of 100 tasks spanning 10 open-source applications, including, but not limited to, *Calen-*

dar, *Contacts*, and *Recorder*. By enabling an automated evaluation pipeline, MobileAgentBench ensures reproducibility and mitigates the biases inherent in manual human intervention, thereby producing more reliable and consistent evaluation results.

While MobileAgentBench focuses predominantly on simulation tasks confined to individual applications, we developed an additional **real-world benchmark** to complement it. This supplementary benchmark encompasses 103 tasks across 9 applications, designed to reflect real-world complexities such as multi-step workflows and physical device interactions (see Appendix table 7 for representative examples). By extending the evaluation framework to include physical devices and cross-application tasks, this benchmark provides a broader perspective on the generalizability and adaptability of our proposed approach.

Metrics. To comprehensively evaluate the proposed method, a diverse set of evaluation metrics was employed, carefully tailored to the specific characteristics of the **MobileAgentBench** tasks and the **real-world** benchmarks. For the **MobileAgentBench** tasks, we adopted the six metrics originally introduced in the corresponding paper:

- **Success Rate (SR)**: $\mathrm{SR} = \frac{N_{\text{success}}}{M_{\text{tasks}}}$, measuring the proportion of tasks successfully completed.
- **Step-wise Efficiency (SE)**: $\mathrm{SE} = \frac{S_{\text{actual}}}{S_{\text{optimal}}}$, assessing unnecessary/redundant actions (excludes failed tasks).
- **Latency**: Average seconds between consecutive user actions, indicating waiting time.
- **Tokens**: Counts input and output tokens processed by the VLM.
- **False Negative (FN)**: $\mathrm{FN} = \frac{N_{\text{early}}}{N_{\text{failure}}}$, rate of tasks erroneously terminated prematurely.
- **False Positive (FP)**: $\mathrm{FP} = \frac{N_{\text{late}}}{M_{\text{success}}}$, rate of tasks incorrectly continued after completion.

In our real-world evaluations, we concentrate exclusively on the most critical metrics, to assess the adaptability and robustness of agents under varying conditions:

- **Success Rate (SR):** Proportion of tasks successfully completed, following MobileAgentBench's definition. Evaluates task completion under real-world constraints.
- **Completion Rate (CR)**: $\mathrm{CR} = \frac{S_{\text{correct}}}{S_{\text{total}}}$, measuring progress through correctly executed steps versus total required steps. Provides nuanced assessment of partial advancement toward goals.

3.2 Implementation Details

To empirically evaluate the performance of our device-control agent, we compare with five methods in MobileAgentBench. Additionally, we compared our approach with two representative methods in real-world scenarios. The details of the experiments are provided below (see Appendix for more details).

- **Baseline methods.** The evaluation encompasses the following baselines in MobileAgentBench: AndroidArena (Xing et al., 2024), AutoDroid (Wen et al., 2023), AppAgent (Zhang et al., 2023) (exploration mode disabled), CogAgent (Hong et al., 2024) and MobileAgent (Wang et al., 2024a).
- **Benchmarking Environment.** Both simulated and real experiments are conducted on a Google Pixel 3a emulator running Android 14 operating system, while AutoDroid is tested using Android 9 due to compatibility issues with more recent Android versions.
- **VLM Standardization.** To ensure fair comparison, all agents utilized GPT-4V as the underlying VLM.
- **Action Space.** Each Agent is allowed to perform five primary actions to simulate device control: `Tap`, `Long_press`, `Swipe`, `Text`, and `Back`. See (Zhang et al., 2023) for details.

3.3 MobileAgentBench Results

To answer the question of how our approach compares to state-of-the-art (SOTA) device-control methods in terms of success rates and efficiency (Question 1), we evaluated our agent using the MobileAgentBench benchmark. As shown in Table 1, our agent, integrating a world model, planner, and action verifier, achieves the highest success rate of 39%, surpassing AutoDroid (27%), MobileAgent (26%), and others, thereby demonstrating the efficacy in handling single-app tasks.

In terms of step-wise efficiency, our agent achieves a value of 1.15. Although this is not the lowest among all evaluated methods, it effectively represents the trade-off between efficiency and achieving the highest success rate (SR). This balance underscores the necessity of additional steps to ensure task completion in complex scenarios. Notably, our approach demonstrates a commendable equilibrium between the false negative rate

Agent	SR	SE	Latency (s)	Tokens	FN Rate	FP Rate
AndroidArena*	0.22	**1.13**	18.61	750.47	**0.09**	0.33
AutoDroid*	0.27	3.10	**4.85**	963.48	0.93	**0.01**
AppAgent	0.12	2.13	12.54	891.43	0.74	0.46
CogAgent	0.08	2.42	6.76	**579.84**	1.00	0.04
MobileAgent	0.26	1.33	15.01	1236.88	0.19	0.31
FPWC	**0.39**	1.15	26.13	2120.45	0.15	0.29

Table 1: **Comparison between FPWC and existing methods.** *denotes the agent includes an additional offline exploration phase.

Agent	SR	CR	Avg. Steps
AppAgent	8.7%	21.7%	14.5
MobileAgent	19.4%	45.9%	13.2
FPWC	**33.0%**	**62.8%**	**11.9**

Table 2: **Comparison between FPWC and baseline methods on real-world device.** In the absence of standardized evaluation for real-world tasks, the experimental results are assessed by three human experts.

(0.15) and the false positive rate (0.29), indicating a reliable avoidance of premature task termination and a controlled prevention of over-executing actions. In contrast, other methods exhibit either a high false negative rate, such as AutoDroid and CogAgent, or a high false positive rate, as seen in AndroidArena. Our method successfully maintains balanced termination dynamics.

To further explore the trade-offs between computational efficiency and task execution performance (Sub-question (b)), we analyzed the tokens and latency. Our agent processed a higher number of tokens (2120.45) compared to baselines such as CogAgent (579.84) and AutoDroid (963.48). This increased token usage enables dynamic decision-making during task execution, differentiating it from models like AutoDroid, which rely on pre-computed offline strategies and achieve the lowest latency of 4.85 seconds. Despite our method's higher latency (26.13 seconds), it offers greater robustness and adaptability in single-app environments.

3.4 Real-World Results

e generalization capabilities of our agent in real-world mobile device tasks, including cross-application scenarios (Question 3), we conducted a comparative evaluation against two state-of-the-art baselines: AppAgent (Zhang et al., 2023) and MobileAgent (Wang et al., 2024a). These baselines were selected due to their strong alignment with real-world usage patterns, particularly their lack of an offline preparation phase, which is critical in mobile device control tasks where immediacy and adaptability are essential.

As shown in Table 2, our agent achieves a success rate (SR) of 33.0% and a completion rate (CR) of 62.8%, compared to AppAgent with an SR of 8.7% and a CR of 21.7%, while MobileAgent achieved an SR of 19.4% and a CR of 45.9%. Furthermore, our agent also demonstrates higher efficiency by completing tasks using the fewest average steps (11.9) compared to AppAgent (14.5) and MobileAgent (13.2).

These results underscore the superior capability of our agent in handling diverse real-world scenarios. By exhibiting higher task success rates, better completion ratios, and optimized step efficiency, our approach sets a new baseline for practical device-control applications. This represents a substantial stride toward more robust and efficient solutions for real-world mobile environments. See Appendix and demo in SM for visualization results for, e.g., cross-app tasks.

3.5 Ablation Study

To investigate the role of individual components in the overall performance of our agent (Sub-question (a)) and examine its ability to solve tasks requiring foresight planning (Question 2), we performed an ablation study, with results summarized in Table 3. Each component is integral to the overall performance of the proposed method. Specifically, the inclusion of the **world model** significantly enhances decision-making capabilities, demonstrating the pivotal role of environmental context in grounding the agent's actions. Similarly, the **planning** module contributes to a moderate yet essential improvement, showcasing the utility of anticipatory decision-making for sequential task execution. The integration of the **self-refinement** mechanism boosts the success rate from 0.30 to 0.35, underscoring the importance of dynamic plan updates and incremental knowledge adjustment. A marginal improvement is observed with the incorporation of **self-verification**, which mitigated inaccuracies in visual analysis conducted by GPT-4V. This module ensures task-relevant information is prioritized, thereby delivering more robust and reliable task outcomes.

As expected, resource consumption grows with system complexity due to more frequent model calls and context-rich prompts—consistent with trends observed in LLM-based agents like Auto-

Module Combination	Task Comprehension	Long-Horizon Reasoning	Adaptive Control	SR	Tokens
plain	-	-	-	0.12	893.58
①	+5 (5)	+7 (7)	+0 (0)	0.24	1477.01
①+②	+1 (6)	+5 (12)	+0 (0)	0.30	1754.18
①+②+③	+1 (7)	+0 (12)	+4 (4)	0.35	1979.34
①+②+③+④	+0 (7)	+0 (12)	+4 (8)	0.39	2120.45

Table 3: **Ablation Study on each component of FPWC.** (①:World Model,②:Planning,③:Self-Refinement,④:Self-Verification)

GPT (Gravitas, 2023) and MetaGPT (Hong et al., 2023), where gains come at computational cost.

To better understand FPWC's advantage over prior methods, we analyze cases where AppAgent fails but FPWC succeeds. We identify three critical factors: **Task Comprehension**, **Long-Horizon Reasoning**, and **Adaptive Control**.

Task Comprehension refers to the agent's ability to accurately interpret the semantic intent behind language instructions and map them to the correct UI and functionalities. Traditional reactive agents often depend heavily on immediate visual cues and coarse task goal, which leads to brittle, shallow interpretations of instructions. In contrast, FPWC constructs a task-specific **world model** (+5) that offers a structured and abstract representation of the environment. As a result, the agent can perform deeper language-to-function grounding—mapping high-level language goals to appropriate UI regions or app modules—even when those are not visible in the current screen. Thus, the world model acts as a semantic scaffold that guides comprehension beyond surface-level patterns.

Long-Horizon Reasoning entails the formulation of coherent multi-step strategies that span multiple views and latent system states. FPWC achieves this by tightly integrating a predictive **world model** (+7) with a structured, **code-based planning** (+5) mechanism. The world model encodes transitions and dependencies between UI states as a directed graph, enabling the agent to simulate and reason about future states before acting. Planning is then conducted by generating executable code that operates over this graph, specifying clear, rule-based policies to reach the task goal. Unlike reactive agents—where high-level reasoning ("thought") is often loosely coupled with concrete actions—FPWC's use of code ensures that intentions are explicitly and consistently translated into behavior. This eliminates ambiguity, promotes logical consistency, and enforces syntactic structure, resulting in long-range plans that are not only effective but also interpretable and verifiable.

Adaptive Control refers to the agent's ability to dynamically adjust its behavior in response to unexpected feedback or execution failures. For example, in the task *"Open About View"*, FPWC initially follows an incorrect path, but upon realizing the mismatch between expected and actual UI, it revises its plan and re-attempts with an alternative strategy. This behavior highlights the role of **Self-Refinement** (+4), which enables the agent to update its world model and regenerate plans in real time based on new observations. Similarly, in *"Filter media in the gallery and only show images and videos"*, the agent must ensure RAW image formats are excluded. FPWC succeeds by validating UI selections through targeted queries, showcasing the contribution of **Self-Verification** (+4) in preventing incorrect interactions by confirming element semantics before action execution. Together, these mechanisms promote resilience and precision in complex, state-sensitive tasks.

Our findings highlight that the **world model** is the key driver of FPWC's success, enabling accurate instruction grounding and long-horizon reasoning. This underscores that building a structured internal understanding of the environment is essential for robust device-control agent performance.

4 Conclusion

In this work, we proposed a novel framework for intelligent device control that integrates graph-based world models to enhance task planning and decision-making. By reformulating planning as a code-generation process, our approach enables the iterative refinement of executable Python scripts through a VLM, achieving adaptability and lifelong learning capabilities. To address challenges in spatial localization and fine-grained scene understanding, we introduced a zoom-in self-verification mechanism that improves accuracy in complex environments. Experimental results demonstrated significant performance improvements over exist-

ing methods in autonomous device control tasks, validating the effectiveness of our framework.

5 Limitations

Despite the promising results, our framework faces several key limitations rooted in the current capabilities of VLMs. First, the precision of interface element recognition remains suboptimal, particularly for small or visually ambiguous components, which can lead to errors in generated code. Second, the VLM's ability to predict long-term outcomes or generate highly specific functions is constrained by its training data and contextual reasoning capacity. These shortcomings may degrade performance in scenarios requiring foresighted planning or precise low-level control. Additionally, the computational cost of token-intensive code generation and verification processes limits real-time applicability. Future work should focus on optimizing model efficiency, enhancing multimodal reasoning, and integrating domain-specific knowledge to overcome these barriers.

6 Ethical Considerations

The deployment of autonomous device control systems raises critical ethical concerns that require careful attention. First, the reliance on visual perception and user interface interaction necessitates robust privacy safeguards to prevent unauthorized access to sensitive environments or personal data. Second, potential safety risks arise from misinterpretations of visual inputs or incorrect code execution, which could lead to unintended physical consequences. We emphasize the importance of implementing rigorous validation mechanisms and fail-safe protocols to ensure system reliability. Furthermore, the generalization capability of VLMs introduces biases inherited from their training data, which must be actively mitigated through diverse dataset curation and fairness-aware design. Finally, the accessibility of such frameworks to non-expert users underscores the need for transparent documentation and ethical guidelines to prevent misuse in harmful applications.

References

Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, and 1 others. 2022. Flamingo: a visual language model for few-shot learning. *NIPS*, 35.

Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Xi Chen, Krzysztof Choromanski, Tianli Ding, Danny Driess, Avinava Dubey, Chelsea Finn, and 1 others. 2023. Rt-2: Vision-language-action models transfer web knowledge to robotic control. *arXiv preprint arXiv:2307.15818*.

Significant Gravitas. 2023. Autogpt. https://github.com/Significant-Gravitas/Auto-GPT. Accessed: 2023-10-01.

Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. 2023. Leveraging pretrained large language models to construct and utilize world models for model-based task planning. *NIPS*, 36:79081–79094.

David Ha and Jürgen Schmidhuber. 2018a. Recurrent world models facilitate policy evolution. *NIPS*, 31.

David Ha and Jürgen Schmidhuber. 2018b. World models. *arXiv preprint arXiv:1803.10122*.

Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. 2019a. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*.

Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. 2019b. Learning latent dynamics for planning from pixels. In *ICML*.

Danijar Hafner, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba. 2020. Mastering atari with discrete world models. *arXiv preprint arXiv:2010.02193*.

Haoran He, Chenjia Bai, Ling Pan, Weinan Zhang, Bin Zhao, and Xuelong Li. 2024. Large-scale actionless video pre-training via discrete diffusion for efficient policy learning. *arXiv preprint arXiv:2402.14407*.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, and 1 others. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, and 1 others. 2024. Cogagent: A visual language model for gui agents. In *CVPR*, pages 14281–14290.

Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. 2019. When to trust your model: Model-based policy optimization. *NIPS*, 32.

Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, and 1 others. 2019. Model-based reinforcement learning for atari. *arXiv preprint arXiv:1903.00374*.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Chong Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan Salakhutdinov, and Daniel Fried. 2024. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. *arXiv preprint arXiv:2401.13649*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. 2023. Code as policies: Language model programs for embodied control. In *ICRA*, pages 9493–9500. IEEE.

Hao Liu, Wilson Yan, Matei Zaharia, and Pieter Abbeel. 2024. World model on million-length video and language with ringattention. *arXiv e-prints*, pages arXiv–2402.

Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V Le, Barret Zoph, Jason Wei, and 1 others. 2023. The flan collection: Designing data and methods for effective instruction tuning. In *ICML*, pages 22631–22648. PMLR.

Kolby Nottingham, Prithviraj Ammanabrolu, Alane Suhr, Yejin Choi, Hannaneh Hajishirzi, Sameer Singh, and Roy Fox. 2023. Do embodied agents dream of pixelated sheep: Embodied decision making using language guided world modelling. In *ICML*, pages 26311–26325. PMLR.

Yiran Qin, Enshen Zhou, Qichang Liu, Zhenfei Yin, Lu Sheng, Ruimao Zhang, Yu Qiao, and Jing Shao. 2024. Mp5: A multi-modal open-ended embodied system in minecraft via active perception. In *CVPR*, pages 16307–16316. IEEE.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, and 1 others. 2020. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609.

Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. 2023. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *ICCV*, pages 2998–3009.

Austin Stone, Ted Xiao, Yao Lu, Keerthana Gopalakrishnan, Kuang-Huei Lee, Quan Vuong, Paul Wohlhart, Sean Kirmani, Brianna Zitkovich, Fei Xia, and 1 others. 2023. Open-world object manipulation using pre-trained vision-language models. *arXiv preprint arXiv:2303.00905*.

Richard S Sutton. 1991. Dyna, an integrated architecture for learning, planning, and reacting. *ACM Sigart Bulletin*, 2(4):160–163.

Hao Tang, Darren Key, and Kevin Ellis. 2024. Worldcoder, a model-based llm agent: Building world models by writing code and interacting with the environment. *arXiv preprint arXiv:2402.12275*.

Karthik Valmeekam, Matthew Marquez, Sarath Sreedharan, and Subbarao Kambhampati. 2023. On the planning abilities of large language models-a critical investigation. *NIPS*, 36:75993–76005.

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

Wenhai Wang, Jiangwei Xie, ChuanYang Hu, Haoming Zou, Jianan Fan, Wenwen Tong, Yang Wen, Silei Wu, Hanming Deng, Zhiqi Li, and 1 others. 2023. Drivemlm: Aligning multi-modal large language models with behavioral planning states for autonomous driving. *arXiv preprint arXiv:2312.09245*.

Zihao Wang, Shaofei Cai, Anji Liu, Yonggang Jin, Jinbing Hou, Bowei Zhang, Haowei Lin, Zhaofeng He, Zilong Zheng, Yaodong Yang, and 1 others. 2024b. Jarvis-1: Open-world multi-task agents with memory-augmented multimodal language models. *TPAMI*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *NIPS*, 35:24824–24837.

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

Junlin Xie, Zhihong Chen, Ruifei Zhang, Xiang Wan, and Guanbin Li. 2024. Large multimodal agents: A survey. *arXiv preprint arXiv:2402.15116*.

Mingzhe Xing, Rongkai Zhang, Hui Xue, Qi Chen, Fan Yang, and Zhen Xiao. 2024. Understanding the weakness of large language model agents within a complex android environment. In *KDD*, pages 6061–6072.

Jianwei Yang, Hao Zhang, Feng Li, Xueyan Zou, Chunyuan Li, and Jianfeng Gao. 2023. Set-of-mark prompting unleashes extraordinary visual grounding in gpt-4v. *arXiv preprint arXiv:2310.11441*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. *NIPS*, 36.

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*.

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

Jiazhao Zhang, Kunyu Wang, Rongtao Xu, Gengze Zhou, Yicong Hong, Xiaomeng Fang, Qi Wu, Zhizheng Zhang, and He Wang. 2024. Navid: Video-based vlm plans the next step for vision-and-language navigation. *arXiv preprint arXiv:2402.15852*.

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

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. 2024. Expel: Llm agents are experiential learners. In *AAAI*, volume 38, pages 19632–19642.

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*.

Deyao Zhu, Jun Chen, Xiaoqian Shen, Xiang Li, and Mohamed Elhoseiny. 2023. Minigpt-4: Enhancing vision-language understanding with advanced large language models. *arXiv preprint arXiv:2304.10592*.

A Related Work

(Multimodal) Language Agents. Large language models (LLMs) with instruction following capabilities (Longpre et al., 2023) have enabled the development of language agents (Hong et al., 2023) that can reason (Yao et al., 2024), plan (Valmeekam et al., 2023), and interact with the real or digital world (Liang et al., 2023) using natural language instructions. While text serves as a powerful medium for reasoning and communication, many real-world tasks require multimodal input, particularly vision, which cannot always be effectively translated into text. To address this, vision language models (VLMs) (Alayrac et al., 2022) have integrated vision into LLMs, giving language agents "eyes". These multimodal agents (Xie et al., 2024) can handle tasks requiring fine-grained operations, such as navigation (Zhang et al., 2024), mobile manipulation (Brohan et al., 2023; Stone et al., 2023), games (Wang et al., 2024b), autonomous driving (Wang et al., 2023), computer operations (Hong et al., 2024; Koh et al., 2024), and more.

Device-control agents. Research on device-control agents is relatively new. Auto-UI (Zhang and Zhang, 2023) uses an VLM trained to directly output actions given views and action histories. AppAgent (Zhang et al., 2023) and Mobile-Agent (Wang et al., 2024a) leverage more advanced off-the-shelf VLMs, such as GPT-4v, using a ReAct-style action loop for decision-making. AppAgent combines views and XML files as inputs and includes an optional self-exploration phase to better understand UI elements. Mobile-Agent, on the other hand, relies solely on views but incorporates tools like OCR, Grounding DINO, and CLIP for improved localization.

While these agents can handle basic tasks, they struggle with issues like shortsightedness, dead loops, and poor performance in cross-App tasks. Our method overcomes these challenges by incorporating a text-based world model that supports action verification, forward-looking planning, and state prediction.

World Models. In the context of reinforcement learning, world models (Ha and Schmidhuber, 2018a; Kaiser et al., 2019) refer to model-based RL methods (Sutton, 1991) that learn a model to predict the transition dynamics and rewards in the environment, which plays the same role as the environment simulator. As the raw observations are difficult to predict, world models often predict future states in latent representations of an RNN (Hafner et al., 2019a, 2020). A learned world model can help for planning (Schrittwieser et al., 2020) or policy learning by imagination of rollouts (Hafner et al., 2019a; Janner et al., 2019). Some recent works (He et al., 2024; Liu et al., 2024) learn world models that directly predict raw image observations by pretraining on a vast number of videos. More recently, there are some works (Guan et al., 2023; Tang et al., 2024) utilizing LLMs to construct world models. DECKARD (Nottingham et al., 2023) builds a text world model of producible items in Minecraft in the form of Python dictionaries, demonstrating how linguistic abstractions improve planning. (Guan et al., 2023) For example, some studies use LLMs to construct world models in planning domain definition language (PDDL) and Python code, respectively. Compared with these methods, our choice of building world models as a graph using textual interface descriptions and executable code is more suitable for device-control

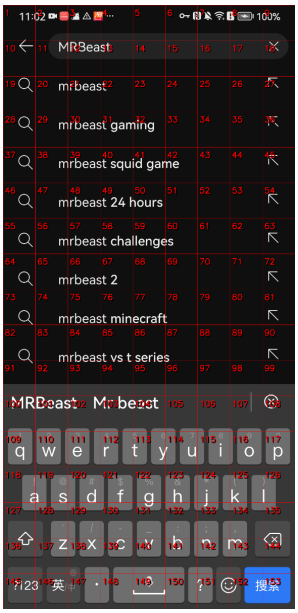

Figure 2: **Grid Labeling for UI Interaction Verification.** When UI element detection via XML files is unreliable or errors persist, the screen is divided into uniformly sized rectangular grids based on resolution. Each grid cell is assigned a unique identifier, allowing the agent to validate and execute actions accurately by selecting specific regions instead of misidentified elements.

agents.

B Experiment Details

For each views captured from the device, the corresponding XML file is extracted, containing metadata about interactive user interface (UI) elements, such as their types and bounding box coordinates. However, discrepancies may arise where the XML file fails to accurately identify or capture some UI elements. To address this, constraints are introduced: the minimum distance between any two elements must exceed a predefined threshold, and the bounding box of any individual element must not occupy more than a quarter of the screen. Subsequently, the UI elements on the screen are assigned numerical labels in accordance with the following set-of-mark prompting (Yang et al., 2023).

In case where the XML file omits certain UI elements or the agent repeatedly selects incorrect elements, an alternative approach is employed. The screen is divided into equally sized rectangular grids based on the screen resolution, and each rectangle is assigned a unique number, as Figure 2 shows.

We use five basic actions:

- `Tap(number : int)`: this action triggers the tapping operation on the center of the region corresponding to the UI element/grid.
- `Long_press(number : int)`: this action triggers the long press operation on the center of the region corresponding to the UI element/grid.
- `Swipe(number : int, direction : str, dist : str)` for XML files and `Swipe(start_number : int, end_number : int)` for grid: this action triggers the swiping operation. For XML files, the agent should specify 8 directions and three levels of distances. For grids, the agent should specify the start grid and the end grid.
- `Text(text : str)`: this action directly implements typing of texts.
- `BACK()`: this action is performed on the system level that can make the phone return to the previous level of the current page, usually used to exit an App.

The agent is permitted to terminate a task upon successfully executing the entire plan. Additionally, it may conclude the task prematurely, using a designated `FINISH` action, if it determines that the objective has already been achieved. For the construction of the world model, we constrain the agent to include only the specified five actions above in the edge, ensuring that the generated plan remains both structured and manageable.

Once a plan is formulated, it is executed as a Python program. The function 'E' is implemented through a sequence of steps, including action selection, validation of the chosen action, and necessary refinement to the graph and plan. If revisions to the plan are required, the current program terminates, and a new execution is initiated. For implementing the `other_app_agent` function, a new process launches the Python script with different apps and task configurations, using the `os.system` package. The `is_true` function is realized by querying the VLM regarding specific questions related to the given image.

C Graph Complexity Analysis

To gain clearer insights into the size and complexity of our initial world models, we performed a detailed analysis, the results of which are presented in the Table 4.

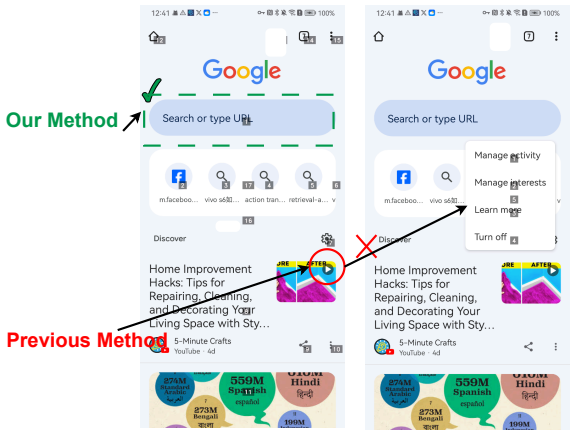

Figure 3: Example highlighting the benefits of employing zoom-in verification prior to executing actions. In this scenario, previous method incorrectly identifies and selects an unintended UI element (Element 7, representing the settings), leading to difficulties in reversing the action. By contrast, our method mitigates such errors by conducting close verification of the elements before proceeding, thereby ensuring accurate action execution.

D Self-Refinement Analysis

To more directly evaluate the impact of the self-refinement mechanism, we performed an analysis using our experiment logs. To clarify how the metrics in our table are calculated, we define the following counts for each app:

- $N_{\text{use\_refine}}$: the number of tasks where our agent triggered the refinement mechanism.
- $N_{\text{fail\_path}}$: the number of tasks that the baseline agent failed and on which our agent used refinement.
- N_{saved}: the number of these "failing path" tasks that our agent successfully recovered and completed.

Based on these, the two metrics are calculated as:

1. **Refinement Trigger Rate** = $\frac{N_{\text{use\_refine}}}{10} \times 100\%$
2. **Recovery Rate (on Failing Tasks)** = $\frac{N_{\text{saved}}}{N_{\text{fail\_path}}}$

This analysis, as summarized in Table 5 yields two key insights:

1. **How often?** The refinement mechanism was triggered in approximately 16% of tasks, primarily in complex scenarios.
2. **How well?** When triggered on tasks that the baseline failed, it achieved a recovery rate of approximately 38% (5 successes out of 13 attempts). This moderate but critical success rate is consistent with our framework's overall performance and accounts for the 5% absolute gain in success rate.

Table 4: Statistics of Initial World Models

Task APP	Vertex Count	Edge Count
FileManager	6.5 ± 1.45	16.3 ± 2.35
Calculator	9.8 ± 2.12	29.4 ± 3.02
Calendar	7.5 ± 1.95	21.0 ± 2.68
Contacts	6.2 ± 1.52	15.5 ± 2.19
Gallery	8.9 ± 1.79	24.9 ± 2.86
Recorder	9.5 ± 2.19	28.5 ± 2.97
MusicPlayer	7.2 ± 1.70	20.2 ± 2.61
Launcher	5.8 ± 1.22	14.5 ± 2.02
Notebook	6.1 ± 1.58	15.3 ± 2.28
Messager	8.2 ± 1.87	23.0 ± 2.76
YouTube	15.5 ± 2.61	37.8 ± 3.48
X	14.2 ± 2.55	34.5 ± 3.39

E Per-Step Cost Example

In this appendix, we provide a comprehensive analysis of latency and token consumption to offer a clearer understanding of the framework's efficiency and scalability.

- **Macro-Level Cost:** At a high level, token consumption serves as a strong proxy for latency. This is because a higher token count for a VLM call directly leads to longer **inference time**, which is the predominant component of overall latency. Our detailed macro-level token analysis can be found in Table 3.
- **Micro-Level Example:** For a more intuitive, step-by-step analysis, we annotate Figure 4 with the estimated cost for each round. The breakdown for the "Subscribe to MrBeast" task is as follows:

Table 6 summarizes this integrated analysis of token and latency estimates, highlighting that complex reasoning steps (planning/refinement) incur the highest costs, whereas verification and execution are relatively economical. This provides a clear view of the framework's cost structure.

F Real-World Benchmark

In Table 7, we show representative examples of the real-world benchmark.

Table 5: Refinement Analysis Results

Task APP	Refinement Trigger Rate	Recovery Rate (on Failing Tasks)
FileManager	20%	0/2
Calculator	20%	2/2
Calendar	20%	1/1
Contacts	10%	0/0
Gallery	20%	0/1
Recorder	20%	1/2
MusicPlayer	20%	0/2
Launcher	0%	0/0
Notebook	10%	0/1
Messager	20%	1/2

Table 6: Per-Round Token and Latency Breakdown for the "Subscribe to MrBeast" Task

Round 1	Round 2	Round 3	Round 4	Round 5	Round 6	Round 7	Round 8	Round 9	Round 10	Round 11	Round 12
1387.12	1592.45	158.67	896.33	315.78	322.10	165.23	298.99	305.21	147.01	311.05	151.89
10.21s	12.55s	1.82s	7.88s	3.45s	3.51s	1.93s	3.24s	3.37s	1.56s	3.41s	1.62s

G Additional Illustrative Examples

We provide an illustrative example in Figure 3 to highlight the benefits of integrating an action verification mechanism prior to the execution of an agent's actions. To further demonstrate the robustness of our approach, Figure 4 presents a complete example in which the agent successfully performs a complex task: Subscribe to "MrBeast" with Wi-Fi off — a condition unknown to the agent at the outset. Unlike AppAgent and MobileAgent, both of which fail to accomplish this task, our agent navigates the challenge successfully. A video demonstration of this scenario is included in the supplementary material. Please refer to it if interested.

These examples effectively showcase the feasibility of accomplishing complex tasks under diverse and uncertain conditions. Combined with our results, they highlight the agent's ability to construct a task-oriented world model, anticipate potential challenges, and adapt its actions accordingly. The success in these demonstrative cases underpins the broader applicability of our method and establishes its advantages over previous approaches. Moreover, the detailed documentation and video evidence supplement the findings, offering a transparent basis for validation and replication by the research community. By solving tasks where existing methods falter, our approach emphasizes the practical utility of foresighted policies and supports its contribution to advancing the field of automated device control.

H Prompt Template

This section provides the prompt templates used in our framework.

Application	Task description
Word	1. Create a new Word document, write “Testing123”. Name the document as “Project1”. 2. Create a new Word document, write the followings in three lines: (1) Review (2) Edit (3) Submit. Do not include the numbers in the text.
Chrome	1. Open Chrome, navigate to History and clear all browsing history. 2. Go to the WIKIPEDIA page for “Artificial Intelligence”, save it as a PDF, and download it.
Gmail	1. Send an email to project@example.com with the subject “AI” and the body text: ”Hello world!” 2. Send an email to project@example.com with the subject ”CS” and the body text: ”Computer Science.” Attach the first image from the photo album.
Maps	1. In Maps, search for “Eiffel Tower”, get driving directions, and start navigation. 2. In Maps ,search for “Central Park” and save it to Favorite list.
TaoBao	1. In TaoBao, add the first item in the cart to the order and proceed to checkout. 2. Search for “sunglasses” in TaoBao and filter results to show items priced between 150 and 200.
TikTok	1. Change the user’s display name to “John_Doe” in TikTok. 2. In TikTok, follow the creator of the current video being played.
WeChat	1. Create a new Post in Moments with the caption “Hello World!” and attach the first photo from the album. 2. Send a text message saying “How are you?” to File Transfer in WeChat.
X	1. Follow the user “@Geoffrey Hinton” in X. 2. In X, search for the topic “Artificial Intelligence” and like the first tweet in the results.
YouTube	1. Search for “AI” in YouTube and play the first video. 2. Open the YouTube settings and change the language to “Spanish.”
Multi-App	1. Open X, search for the hashtag “#MachineLearning” (with Wi-Fi turned off). 2. Share a PDF version of the Word document titled ”Project1” to File Transfer via WeChat.

Table 7: **Representative tasks in our real-world benchmark.**

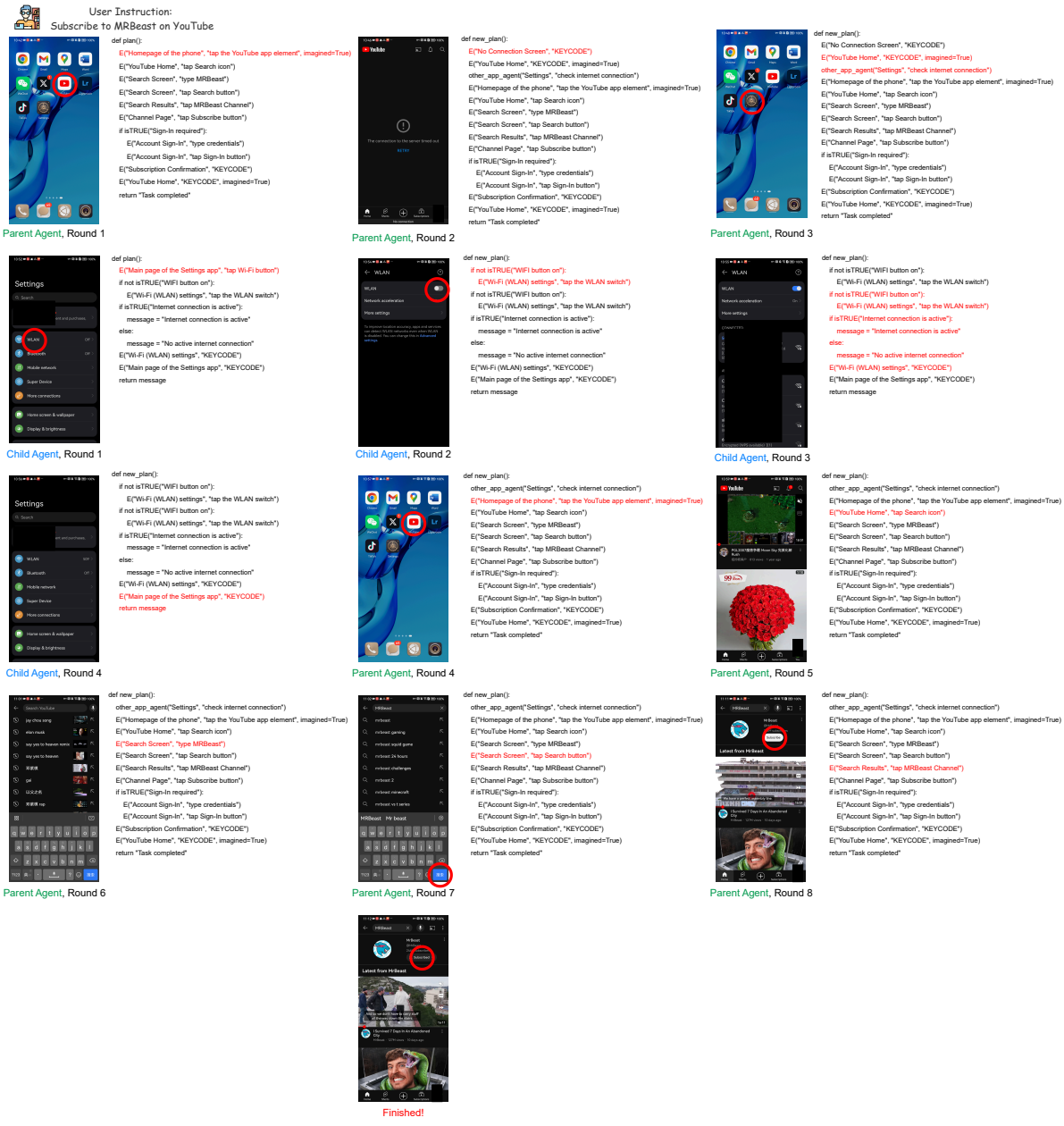

Figure 4: Illustrative example demonstrating the functionality of the proposed method. The task is to subscribe to "MrBeast" on YouTube, with the added challenge that the Wi-Fi is initially disabled—a condition unknown to the agent beforehand. The agent is capable of making long-term decisions while dynamically revising its plan when necessary. Additionally, it can generate a secondary agent to address tasks that require interaction beyond its current application.

generate_initial_world_model

smallYou are a graph expert who is trained to generate a graph of a given App on the smartphone. A task to do is to <task_description>. The vertices of the graph is each screen of the App, and the edges are the possible transitions between screens, which are triggered by tapping, swiping or long pressing on UI elements of the screen. The root of the graph is the initial screen of the App, and the graph should cover as more possible screens and transitions of the App as possible.

Output format:
Vertices:
Name: <Vertex> Description: <Description> can-self-act: <True or False> Edges:
Edge: E(<Vertex>, action) -> <Vertex> # <Description>
"can-self-act" means that the vertex (screen) can direct to itself, which means there exists actions that can be performed on the screen that does not change the vertex, but change the state of the screen. For example, swipe the main page of the Settings App can show some other choices of settings, but still stay at the main page. There are only 5 action types: tap, swipe, long_press, type, BACK (BACK is used to get back to the last level of page, e.g., from the mainpage of an App to the homepage of the phone). Remember that the action type should be put in the first place, for example, "tap the i-th Wi-Fi network", "swipe down/left/right/up", "BACK", "long_press the button", "type the password", etc.

The following shows an example of a partial graph (not complete) of the Settings App:

Vertices:
Name: "Main page of the Settings app" Description: The main page of the Settings App that can

be used to navigate to different settings of the phone. can-self-act: True

Name: “Wi-Fi (WLAN) settings” Description: The Wi-Fi settings page that can be used to connect to a Wi-Fi network. can-self-act: True

Name: “Page connecting to i-th Wi-Fi (WLAN)” Description: The page that shows all things needed to connect to i-th Wi-Fi network. can-self-act: True

Name: “Choose Privacy setting of i-th Wi-Fi” Description: Use randomized MAC or device MAC. can-self-act: False

Name: “Choose Proxy setting of i-th Wi-Fi” Description: Proxy setting of None, Manual or Auto. can-self-act: False

Name: “Choose IP settings of i-th Wi-Fi” Description: IP settings of Dynamic or Static. can-self-act: False

Name: “WiFi Connecting” Description: The page showing that phone is still trying to connect to the specific WIFI. can-self-act: False

Name: “WiFi password incorrect” Description: The page showing that the password of WIFI is incorrect. can-self-act: False

Edges:
Edge: E(“Main page of the Settings app“, “BACK“) ->“Homepage of the phone” #Back to the phone homepage

Edge: E(“Main page of the Settings app“, “swipe down“) ->“Main page of the Settings app” #Show more settings in the bottom

Edge: E(“Main page of the Settings app“, “swipe up“) ->“Main page of the Settings app” #Show more settings on the top

Edge: E(“Main page of the Settings app“, “tap Wi-Fi button“) ->“Wi-Fi (WLAN) settings” #Open the Wi-Fi setting page

Edge: E(“Wi-Fi (WLAN) settings“, “tap the WLAN button“) ->“Wi-Fi (WLAN) settings” #Open Wi-Fi

Edge: E(“Wi-Fi (WLAN) settings“, “tap the WLAN button“) ->“Wi-Fi (WLAN) settings” #Close Wi-Fi

Edge: E(“Wi-Fi (WLAN) settings“, “tap the i-th Wi-Fi network“) ->“Page connecting to i-th Wi-Fi (WLAN)” #Open the page to connect to i-th Wi-Fi

Edge: E(“Page connecting to i-th Wi-Fi (WLAN)“, “type password“) ->“Page connecting to i-th Wi-Fi (WLAN)” #Type the password of the specific Wi-Fi

Edge: E(“Page connecting to i-th Wi-Fi (WLAN)“, “tap the privacy setting“>->“Choose

Privacy setting of i-th Wi-Fi" #Open the page to choose privacy setting of the specific Wi-Fi

Edge: E("Choose Privacy setting of i-th Wi-Fi", "tap 'Use randomized MAC' option") ->"Page connecting to i-th Wi-Fi (WLAN)" #Use randomized MAC

Edge: E("Choose Privacy setting of i-th Wi-Fi", "tap 'Use device MAC' option") ->"Page connecting to i-th Wi-Fi (WLAN)" #Use device MAC

Edge: E("Choose Privacy setting of i-th Wi-Fi", "tap 'CANCEL' button") ->"Page connecting to i-th Wi-Fi (WLAN)" #Cancel the privacy setting

Edge: E("Page connecting to i-th Wi-Fi (WLAN)", "tap the proxy setting">->"Choose Proxy setting of i-th Wi-Fi" #Open the page to choose proxy setting of the specific Wi-Fi

Edge: E("Choose Proxy setting of i-th Wi-Fi", "tap 'None' option") ->"Page connecting to i-th Wi-Fi (WLAN)" #Choose "None" proxy setting

Edge: E("Choose Proxy setting of i-th Wi-Fi", "tap 'Manual' option") ->"Page connecting to i-th Wi-Fi (WLAN)" #Choose "Manual" proxy setting

Edge: E("Choose Proxy setting of i-th Wi-Fi", "tap 'Auto' option") ->"Page connecting to i-th Wi-Fi (WLAN)" #Choose "Auto" proxy setting

Edge: E("Choose Proxy setting of i-th Wi-Fi", "tap 'CANCEL' button") ->"Page connecting to i-th Wi-Fi (WLAN)" #Cancel the proxy setting

Edge: E("Page connecting to i-th Wi-Fi (WLAN)", "tap the IP setting") ->"Choose IP settings of i-th Wi-Fi" #Open the page to choose IP setting of the specific Wi-Fi

Edge: E("Choose IP settings of i-th Wi-Fi", "tap 'Dynamic' option") ->"Page connecting to i-th Wi-Fi (WLAN)" #Choose "Dynamic" IP setting

Edge: E("Choose IP settings of i-th Wi-Fi", "tap 'Static' option") ->"Page connecting to i-th Wi-Fi (WLAN)" #Choose "Static" IP setting

Edge: E("Choose IP settings of i-th Wi-Fi", "tap 'CANCEL' button") ->"Page connecting to i-th Wi-Fi (WLAN)" #Cancel the IP setting

Edge: E("Wi-Fi (WLAN) settings", "tap 'CANCEL' button") ->"Main page of the Settings app" #Return to the main page

Edge: E("Page connecting to i-th Wi-Fi (WLAN)", "tap 'CONNECT' button") ->"WiFi Connecting" #The phone is still trying to connet the specific WIFI

Edge: E("Page connecting to i-th Wi-Fi (WLAN)", "tap 'CONNECT' button") ->"WiFi password incorrect" #Incorrect passward for the specific WIFI

Edge: E("Page connecting to i-th Wi-Fi (WLAN)", "tap 'CONNECT' button") ->"Wi-Fi (WLAN) settings" #The phone has successfully connected to the specific WIFI

Remember only give explanations when the action is self-act or has multiple possible results. The App name is <App_name>, generate a graph of the App based on your understanding of the APP. Do not give anything else except the graph in the output. You should include necessary vertices and edges for completing the task

generate_initial_plan

You are an intelligent agent trained to complete tasks on a smartphone. You will generate a Python-like executable plan based on the task description, app graph, and the intial screenshot provided. Your plan will help navigate through the app screens and complete the task step by step. Your output will be used directly as executable code, so follow the required format strictly.

To help you reason systematically, follow the steps outlined below:
—
Context:
1. **Input Information**:
- **Task Description**: The task you need to complete is to <task_description>.
- **Graph**:
- **Vertices**: <Vertices>
- **Edges**: <Edges>
- **Initial Screen**: The smartphone's initial screen is provided as a screenshot Use it as the starting point to create the plan.
2. **Graph Details**:
- **Vertices** respresent the app's screens.
- **Edges** define the possible transitions between these screens.Each edge is represented as E(vertex,action),where:
- vertex: The source screen name from which the transition occurs.
- action: The action that leads to the destination screen.
- If you need to imagine new vertices or edges, set the parameter imagined=True in the corresponding function.
3. **Available Functions**:
- **E(vertex,action)**: Execute an action to transition to another screen.
- **isTRUE(statement)**: Check if a given statement is true on the current screen.
- **wait()**: Pause execution until the screen changes.
- **other_app_function(app_name,sub_task)**: Execute high-level tasks in other apps.
4. **Action Types**:
There are 5 valid action types:
- 'tap': Taps a specific UI element on the screen.
- 'swipe': Performs a swipe gesture on the screen.
- 'long_press": Long presses a specific UI element on the screen.
- 'type': Types a given input into a text field. - 'KEYCODE':
- Definition: Simulate hardware key actions like "Back" or "Home".
- **Use Cases**:
1. Return to the previous screen.
2. Exit an app and return to the phone's main screen.
—
Your Task:
1. **Generate a Plan**:
- Your plan must start with the current screen and proceed step by step to complete the task.
- If the current screen is not in the graph, imagine a "current vertex" and include 'imagined=True'

in any corresponding functions.
- If the task involves actions in another app, you must use the 'other_app_function' to complete those actions.Specify the 'app_name' and 'sub_task' explicitly.
2. **Required Format**:
- **Current Vertex**: Identify the current screen(vertex) based on the provided information. If the screen is imagined, state it explicitly.
- **Plan**: A Python function named 'new_plan' that implements the steps to complete the task.
—
Example Output:
Current vertex:Homepage of the phone
Plan:
def new_plan():
# tap the Settings app element
E("Homepage of the phone", "tap the Settings app element", imagined = True)
E("Main page of the Settings app", "tap 'Wi-Fi' button")
# if the Wi-Fi button is off, turn it on
if not isTRUE("WIFI button on"):
E("Wi-Fi (WLAN) settings", "tap the WLAN button")

# iterate through all Wi-Fi networks on the screen
i = 1
while True:
# if the i-th Wi-Fi network is out of screen, swipe down to show more Wi-Fi networks
if isTRUE(f"the {i}-th Wi-Fi network on the screen is out of screen"):
E("Wi-Fi (WLAN) settings", "swipe down")
if isTrue("All Wi-Fi options are the same as before", compare_screen = True):
return "No Wi-Fi with password 57889999 found"
i = 1
E("Wi-Fi (WLAN) settings", f"tap the {i}-th Wi-Fi network on the screen")
E("Page connecting to i-th Wi-Fi", "type password 57889999")
E("Page connecting to i-th Wi-Fi", "tap 'CONNECT' button")
if isTRUE("Wi-Fi connected"):
# if the Wi-Fi is connected, break the loop
break
elif isTRUE("Wi-Fi still connecting"):
# if the Wi-Fi is still connecting, wait for the screen to change
wait()
else:
# if the password is incorrect, tap the 'CANCEL' button and continue to the next Wi-Fi network
E("Page connecting to i-th Wi-Fi", "tap 'CANCEL' button")
i += 1
return "Task completed"

Important Notes:
1. **Strict Output Rules**:
- Your output must strictly follow the required format. Do **NOT** include any description, explanations, or text outside the required format. - The plan must be a valid Python-like function with no syntax errors.
- Your output **must start** with 'Current vertex: <current_vertex>'.
- Always begin the function with 'def new_plan():'.

2. **Error Handling**:
- If the graph is missing necessary information, imagine additional vertices or edges and include 'imagined=True' in the corresponding functions.
- If the task cannot be completed, return an appropriate error message in the plan.
3. **Plan Constraints**:
- Always start from the current screen and ensure transitions follow the graph.
- When navigatiing between apps, return to the home screen of the phone using 'KEYCODE' before switching to another app.
- Use 'other_app_function' explicitly for actions involving other apps.

refinement

You are an intelligent agent trained to complete tasks on a smartphone. Your role is to analyze the given app graph, the planned code, and two screenshots (before and after an action) to determine whether the plan or the graph needs revision.

To help you reason systematically, follow the steps outlined below:
—
Context:
1. **Task**: The overall task you need to complete is to <task_description>.
2. **Graph Information**: The app you are working on, <app>, is represented as a directed graph:
- **Vertices**: The screens of the app.
<Vertices>
- **Edges**: The possible transition functions between screens.
<Edges>
3. **Plan Code**:
A Python-like plan has been created based on the graph to complete the task. The plan is as follows:
<plan>
The plan may include imagined vertices or edges with the parameter 'imagined=True'.
4. **Current Information**:
- The previous screenshot (before the action) corresponds to the vertex: <previous_vertex>.
- The last action taken corresponds to the function: E(<previous_vertex>, <action>).
- The detailed action performed is: <detailed_action>(the numeric tag of the UI element).
5. **Additional Information**:
- Thoughts: <thought>
- Summary: <summary>
- Action History: <act_history>
- Repeated Elements: <repeated_doc>
6. **Constraints and Notes**:
- The app graph may need revision based on your observation of the screenshots and the action taken.
- **Action Validity**:
- If the action does not result in a change in the vertex(screen), you must identify it as an ineffective action.
- If the detailed action was wrong (e.g., incorrect numeric tag), you must remind the user to choose a different numeric tag.
- **Plan Revision**:
- If the task is completed but the plan is not, revise the plan to directly return the result.
- If adding **other_app_function**, ensure it is included in the **revised plan**.

- Always exit to the home screen of the phone before calling **other_app_function**.
—
Your Task:
1. Observe the current screenshot (second image).
2. Determine the current vertex based on the screenshot.
3. If necessary, revise the vertices, edges, and planned code.
4. Assess whether the action taken was successful and expected.
5. Provide reminders if needed to correct the action or address issues in the plan.
—
Output Format:
Your response must strictly follow the format below:

Observation of the current screenshot: <Describe what you observe in the image>
Thoughts: <Explain your reasoning about the graph, plan, and the observed results, and why any revisions are necessary>

Removed vertices:
<Name: ”<vertex_name>” Description: <description>can-self-act: <True/False>... (if no changes, leave blank but keep ”Removed vertices: ” intact)>
Added vertices:
<Name: ”<vertex_name>” Description: <description>can-self-act: <True/False>... (if no changes, leave blank but keep ”Added vertices: ” intact)>
Removed edges:
<Edge: E(”<start_vertex>”, ”<action>”)->”<end_vertex>” #<comment>... (if no changes, leave blank but keep ”Removed edges: ” intact)>
Added edges:
<Edge: E(”<start _vertex>”, ”<action>”)->”<end_vertex>” #<comment>... (if no changes, leave blank but keep ”Added edges: ” intact)>
Current vertex: <current_vertex>
Added functions for other apps:
<other_app_function(”<app_name>”, ”<sub_task>”) ... (if no changes, leave blank but keep ”Added functions for other apps: ” intact)>
Successful and expected action: <True/False>
Ineffective: <True/False>
Revised plan: <def new_plan(): # Revised code here... ... (if no changes, leave blank but keep ”Revised plan: ” intact)>
Remind: <Write necessary reminders here; if none, leave blank but keep ”Remind: ” intact>
Impact of the action on the element on the task:
<Describe the effect of the action on the task; this must explicitly connect the transition between the two images and its impact on the task>
—

Example Output:
Removed vertices:
Name: ”Main page of the Settings app” Desciption: Ohhhhhh. can-self-act: True ...
Added vertices:
Name: ”Main page of the Settings app” Desciption: The main page of the Settings App that can be used to navigate to other settings. can-self-act: True ...
The change of edges should be the form like:

Removed edges:
Edge: E(”Main page of the Settings app”, ”KEYCODE”)->”Main page of Taobao” #Open Taobao ...
Added edges:
Edge: E(”Main page of the Settings app”, ”KEYCODE”) ->”Homepage of the phone” #Back to the phone homepage ...
—
Important Notes:
1. **Removed/Added Vertices and Edges**:
- If any vertices or edges are revised, the removed ones **must exactly match the original graph**, including the comments after ”#”.
- Added vertices and edges must have clear descriptions and comments.
2. **Current Vertex**:
- If the two screenshots differ in content, the vertex should change to reflect the current screenshot.
3. **Action Assessment**:
- If the action is ineffective, mark it as True under ”Ineffective”.
- If the action was not successful or expected, mark it as False under ”Successful and expected action”.
4. **Revised Plan**:
- If the plan is revised, ensure it begins at the current vertex and includes transitions as required.
- Any add ”other_app_function” **must** be included in the revised plan.
5. **Strict Output Format**: All sections must be present, even if left blank.