FROM TAPPING TO HOPPING: AUGMENTING MOBILE GUI AGENTS WITH APP-NATIVE DEEPLINKS

Anonymous authors
Paper under double-blind review

ABSTRACT

Mobile GUI agents complete tasks using GUI actions like taps and swipes. These actions are broadly applicable across applications, but reaching a task-relevant screen can require many intermediate navigation steps, lengthening trajectories and creating more opportunities for error. Fortunately, many Android applications expose deeplinks—external entry points to specific in-app screens—that provide a higher-level navigation interface. A single deeplink call can replace a sequence of screen-by-screen GUI actions. We therefore introduce hybrid interaction, using deeplinks for direct navigation and GUI actions for other on-screen operations and fallback. To enable this, we discover candidate deeplinks through static analysis, validate them on real devices, and describe their observed landing screens. This process creates a verified and grounded deeplink catalog that pairs each working deeplink with a description of its landing screen. Using this catalog, we introduce **GUI-Hopper**, a deeplink-augmented GUI agent that learns when to use a deeplink and when to rely on GUI actions. On MobileWorld, GUI-Hopper achieves higher task success with fewer steps than its GUI-only counterpart; its gains persist even without deeplinks at inference. On MobileWorld-Real, it also improves task success in commercial applications on real devices, further demonstrating the benefits of hybrid interaction.

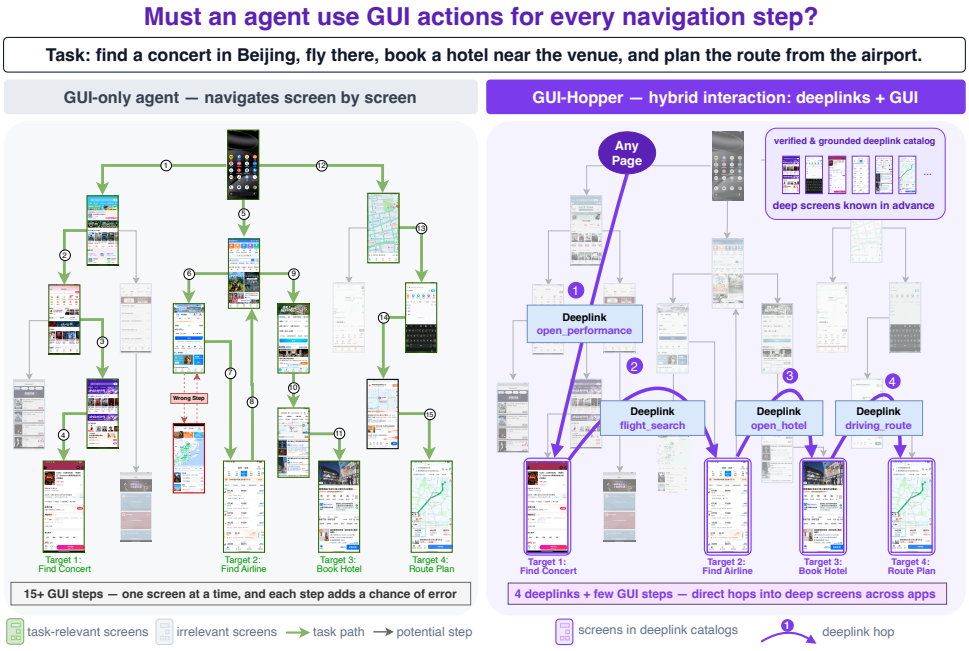

Figure 1: **From GUI-only to hybrid interaction.** For a long-horizon mobile task spanning multiple applications, a GUI-only agent navigates screen by screen, requiring 15+ GUI steps (left). GUI-Hopper uses verified deeplinks to reach task-relevant screens directly and GUI actions for the remaining interaction (right), shortening the trajectory and reducing opportunities for error.

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

1 INTRODUCTION

Mobile GUI agents have advanced rapidly with multimodal large language models (MLLMs) (Zhang et al., 2023; Qin et al., 2025; Xu et al., 2026). Most operate exclusively through GUI actions: the agent observes the current screen and acts using taps, swipes, etc. This interaction method is native to mobile interfaces, broadly applicable, requires no application-specific API, and supports fine-grained control. Yet relying on GUI actions means that every navigation path must be completed screen by screen. To reach a screen several GUI steps away, a GUI agent must interact with every intermediate screen before the relevant controls become available. Each intermediate screen lengthens the trajectory and introduces another opportunity for error. This raises a practical question: must an agent use GUI actions for every navigation step?

Deeplinks offer an alternative way to navigate. Android applications expose them as external entry points to specific in-app screens (Google, 2026a). As illustrated by the flight-search example in Figure 1, a single deeplink call can supply the origin, destination, and travel dates and open the corresponding results screen directly. Beyond reducing navigation steps, a deeplink catalog gives the agent a structured, application-level view of directly reachable destinations and the parameters they accept—information not visible in the current screenshot. Deeplinks and GUI actions therefore play complementary roles: deeplinks provide higher-level navigation, whereas GUI actions support fine-grained on-screen operations and fallback. We call their joint use within a task *hybrid interaction*.

Before an agent can use deeplinks, it must know which ones work and where they lead. We begin by applying existing static-analysis techniques to extract candidate deeplinks (Zhao et al., 2026). However, these candidates are not yet suitable for use as agent actions: some are stale or redundant, some require parameter values that are not specified in their declarations, and their behavior may depend on runtime context. Moreover, a deeplink URI alone rarely conveys what the deeplink does or where it leads. To resolve this ambiguity, we develop an automated pipeline that searches for valid parameter values when needed and directly executes each candidate on real devices. The pipeline discards candidates that fail or do not lead to a meaningful screen and pairs each retained deeplink with a language description of its observed landing screen. Together, these steps transform raw candidates into a *verified and grounded deeplink catalog*.

A reliable catalog alone is not enough: an agent must still determine whether a given deeplink will help with the current task. We therefore introduce **GUI-Hopper**, a deeplink-augmented GUI agent trained to use a deeplink when it is useful and otherwise rely on GUI control. To teach this decision, negative augmentation exposes GUI-Hopper to cases in which the matching deeplink is withheld, only irrelevant deeplinks are provided, or an invocation fails at runtime. We then optimize GUI-Hopper through online reinforcement learning to encourage correct deeplink use and efficient task completion. At inference, the system retrieves deeplinks whose landing-screen descriptions match the task, and GUI-Hopper decides whether to invoke one. After each invocation, the system checks whether the expected screen has been reached. If no suitable deeplink is available or verification fails, GUI-Hopper continues the task through ordinary GUI interaction.

We evaluate whether this combination improves task completion and interaction efficiency. On MobileWorld (Kong et al., 2026), GUI-Hopper completes more tasks with fewer interaction steps than a matched GUI-only agent, and the advantage holds across the three model scales we train. GUI-Hopper continues to outperform the GUI-only agent when evaluated without deeplinks, showing that its gains persist beyond direct deeplink use. GUI-Hopper also performs well on real devices in MobileWorld-Real (Zhou et al., 2026). Together, these results point to a paradigm shift: from GUI-only mobile agents to agents that combine GUI control with deeplinks when they are available.

The contributions of this paper are threefold. First, we introduce GUI-Hopper, which combines deeplinks for higher-level navigation with GUI actions for fine-grained on-screen operations and fallback. This allows the agent to reach task-relevant screens directly when suitable deeplinks are available while retaining GUI control throughout the task. To the best of our knowledge, we are the first to systematically integrate deeplinks into mobile GUI agents across catalog construction, model training, and inference-time execution. Second, we develop an automatic deeplink mining and grounding pipeline that discovers candidate deeplinks, tests them on real devices, and pairs each verified deeplink with a description of its observed landing screen. Third, we train GUI-Hopper to select deeplinks and return to GUI control when needed, using negative augmentation and reinforcement learning with rewards for task success, deeplink use, and interaction efficiency. Experiments

show that GUI-Hopper improves task success and interaction efficiency, retains its gains when evaluated without deeplinks, and performs well on real devices.

2 Related Work

Mobile GUI Agents. Existing work on mobile GUI agents can be broadly divided into two categories. Framework-based approaches orchestrate off-the-shelf multimodal models through prompting, ranging from single-agent exploration to multi-agent collaboration with memory (Zhang et al., 2023; Wang et al., 2024; 2025b). More recent approaches instead train native GUI models end to end, scaling unified architectures across platforms through data flywheels and online reinforcement learning (Qin et al., 2025; Wang et al., 2025a; Zhou et al., 2025; Gao et al., 2026; Xu et al., 2026; Yan et al., 2025; Cao et al., 2026). Recent benchmarks emphasize longer tasks, cross-application workflows, and real-device interaction (Kong et al., 2026; Sui et al., 2026; Zhou et al., 2026), making trajectory efficiency important alongside task success (Jang et al., 2026; Xu et al., 2025). Yet despite these advances, agents in both categories still perform every navigation step through the GUI, screen by screen—the inefficiency targeted by our work.

Tool-Augmented Mobile Agents. Tool-augmented mobile agents complement GUI interaction with tools that execute reusable action sequences or directly invoke functions within applications. Academic approaches generally construct tools on the agent side, either by distilling reusable skills from agent trajectories (Jiang et al., 2025; Wang et al., 2025b) or by building tool libraries from documentation and synthetic traces; several studies further train models to invoke these tools (Yang et al., 2025; Hu et al., 2026; Xie et al., 2025). These agent-constructed tools can be brittle: trajectory-derived skills may be tied to the UI versions on which their source trajectories were collected, while synthesized tool libraries may not behave reliably in real applications. Industrial systems, by contrast, rely on platform interfaces such as Apple App Intents and Google App Actions that application developers integrate directly (Apple Inc., 2024; Google, 2024). These interfaces are dependable once integrated, but their coverage is limited by developer adoption. Deeplinks offer a third path that avoids much of this trade-off. Because they are app-native, deeplinks can be reused without requiring developers to implement a new agent-facing API. When mined systematically, they also expose an application's directly addressable pages. Recent work provides agents with manually curated deeplinks at inference time (Zhao et al., 2026; Lychee Team, 2026). We go further by automatically mining and validating deeplinks on real devices, training the agent to choose between deeplinks and GUI actions, and supporting inference with retrieval and landing-screen verification. To our knowledge, no prior work addresses this full pipeline.

Deeplinks in the Android Ecosystem. Android applications expose deeplinks as external entry points to directly launch specific in-app screens (Google, 2026a). Research on deeplinks has examined their generation, use in software testing, and security. On the generation side, uLink and Aladdin generate deeplinks for existing applications, motivated by the observation that many applications expose few developer-defined links (Azim et al., 2016; Ma et al., 2018). In software testing, deeplinks improve the coverage of automated exploration by providing direct access to deeply nested pages (Hu et al., 2024). Security studies have measured hijacking risks at scale and shown that the verification machinery can itself be bypassed (Liu et al., 2017; Tang et al., 2020). Recent analyses further report that production OEM assistants already invoke deeplinks to bypass screen-by-screen UI navigation (Wu et al., 2025). Collectively, this literature establishes deeplinks as a mature mechanism in Android. It does not, however, study how a trained mobile agent can learn to combine deeplinks with GUI actions.

3 Method

GUI-Hopper is a mobile agent trained to use deeplink actions alongside GUI control. GUI-Hopper is built on three key components (Figure 2): a mining pipeline that turns app-provided deeplinks into a verified and grounded catalog (Section 3.2), a hybrid-action training strategy that teaches the model when to invoke a deeplink and when to continue with GUI actions (Section 3.3), and an agent harness for verified deeplink use on real phones (Section 3.4).

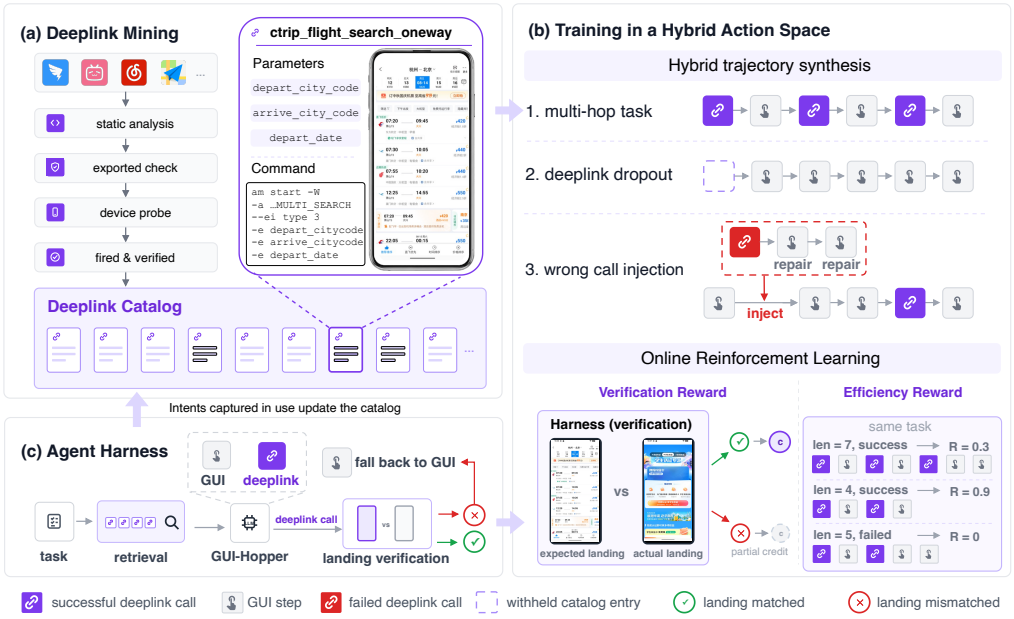

Figure 2: **Overview of our approach.** (a) A mining pipeline verifies deeplinks on real devices and builds a grounded catalog. (b) Hybrid trajectory synthesis and online reinforcement learning train GUI-Hopper in the hybrid action space. (c) At inference, the harness retrieves task-specific deeplink candidates and checks whether each call reaches its expected Activity, with GUI fallback on failure.

3.1 Preliminaries

Deeplinks in Android. Android applications organize user-facing screens into *Activities*. An Activity is launched through an *Intent*, a system message that identifies the target component and carries invocation parameters. In `AndroidManifest.xml`, an application can associate URI patterns with Activities that external callers may open. Android resolves an Intent carrying the URI to an Activity that can handle it and launches that Activity; we refer to this URI-based entry point as a deeplink (Google, 2026b). Because a raw URI rarely reveals its landing screen, we expose each deeplink to the model as a tool with a name, a landing-screen description, and parameter definitions. Section 3.2 describes how these fields are constructed. In the formulation below, a deeplink action invokes one such catalog entry with concrete parameters.

Task formulation. We model mobile interaction as a partially observable Markov decision process (Kaelbling et al., 1998) with environment states $s_t \in \mathcal{S}$, observations $o_t \in \mathcal{O}$, actions $a_t \in \mathcal{A}$, and transition dynamics $\mathcal{T}(s_{t+1} \mid s_t, a_t)$. The action space is hybrid: $\mathcal{A} = \mathcal{A}_{\text{GUI}} \cup \mathcal{A}_{\text{DL}}$. Here, $\mathcal{A}_{\text{GUI}}$ contains atomic screen operations such as tapping, swiping, and typing. Each deeplink action is a pair (d, x), where d is a catalog entry supplied for the current task and x assigns concrete values to its parameters. At step t, the agent receives an observation $o_t = (v_t, f_t)$, where v_t is the current screenshot and f_t is the harness's feedback after the preceding action. GUI actions produce no structured feedback. For a deeplink action, f_t reports whether the invocation succeeded and whether the observed landing Activity matches the expected Activity recorded in the catalog. The policy chooses $a_t = \pi_\theta(\mathcal{I}, o_t, \mathcal{H}_t)$ from the task instruction $\mathcal{I}$, the current observation o_t, and the recent interaction history $\mathcal{H}_t = (o_{t-h}, a_{t-h}, \ldots, o_{t-1}, a_{t-1})$. We optimize the policy parameters to maximize the expected trajectory reward $J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[\mathcal{R}(\tau)]$, with $\mathcal{R}$ defined in Section 3.3.

3.2 Deeplink Mining and Description

Android applications already expose deeplink entry points, but their declarations alone do not make these entries usable as agent actions. Building the catalog requires two steps: verifying which candidates work on a device and representing each verified entry with a landing-screen description and parameter definitions. Static analysis provides initial candidates but cannot determine whether

a candidate works or where it lands. This limitation is especially important for entries that route to different screens depending on parameter values not specified in their declarations. We therefore test candidates on real devices and use the observed results to verify and describe each retained entry.

Mining pipeline. The pipeline (Figure 2(a)) begins by extracting candidate entries from manifest declarations and shortcut definitions (Google, 2026b;c) through static analysis of the application package. We first filter out candidates whose target components are not exported and therefore cannot be invoked externally. For the remaining candidates, we use a read-only package-manager query to check whether they resolve to installed components, without launching the application or invoking a model. Finally, we test each resolvable candidate on a real device and retain only those that open a meaningful screen. For parameterized entries, we test candidate values and retain a parameter only if changing its value visibly changes the resulting screen. After verification, we consolidate duplicate entries and exclude those pointing to transient targets to form the final catalog. Across the 26 commercial applications selected for MobileWorld-Real evaluation, the resulting catalog contains 1,251 verified entries.

Describing deeplinks for agent use. For each verified deeplink, we create a catalog entry describing where it leads and how to invoke it. Each entry specifies a tool name, a description of the observed landing screen, and parameter definitions with formats and example values. Since a screen often supports multiple functions, we describe the landing screen rather than a single function. We also specify what remains to be done after landing, making clear that reaching the screen does not necessarily mean that the task is complete. Together, these details help a mobile agent select a task-relevant deeplink and supply appropriate parameter values. To enable the harness to execute the call and verify its landing, each entry also stores an invocation template and the expected landing Activity. The agent harness fills this template with the selected parameter values to construct the call, then compares the observed Activity with the expected one. Appendix B presents the complete catalog entry format and prompt format, while Appendix A provides implementation details and statistics.

3.3 Training in a Hybrid Action Space

Training in a hybrid action space requires more than learning the syntax of a deeplink call. The model must decide whether to invoke a deeplink or continue with GUI actions as the available deeplink set varies, and it must recover when a call fails. We first use hybrid trajectory synthesis and negative augmentation to teach selection and recovery. We then apply online reinforcement learning to refine these behaviors.

Hybrid trajectory synthesis. We synthesize training trajectories from tasks constructed around the mined catalog (Figure 2(b)). They cover both tasks that chain multiple deeplink calls and tasks that require GUI actions after a deeplink call, demonstrating how the two action types can be combined. Because an unsuitable deeplink may take the agent directly to an unrelated screen, our negative augmentation targets both prevention and recovery. To prevent incorrect calls, we introduce *deeplink dropout*: we sample successful deeplink trajectories and collect GUI-only trajectories for the same tasks, supplying only task-irrelevant deeplinks in the prompt. These trajectories demonstrate how to fall back on GUI actions when retrieval provides no useful deeplinks. To train recovery, we apply *wrong-call injection* during trajectory generation. We deliberately invoke an unsuitable deeplink and retain the subsequent sequence in which the agent recognizes that the landing screen is irrelevant to the task, recovers from the error, and resumes the task. Together, these trajectories teach the agent to use suitable deeplinks, rely on GUI actions when none are suitable, and recover from incorrect calls.

Online reinforcement learning. The synthesized trajectories teach the model the basic decision pattern, while online interaction provides direct feedback on whether its deeplink choices reach the expected destination and contribute to successful, efficient task completion. We therefore further optimize GUI-Hopper with online reinforcement learning using both call-level and trajectory-level feedback. Each catalog entry records its verified landing Activity. After execution, the feedback f_t reports whether the invocation succeeded and whether the observed Activity matches the expected one (Section 3.1), providing a call-level verification signal. At the trajectory level, $s(\tau)$ records task success, while $|\tau|$ counts the number of interaction steps and provides the basis for the efficiency

term. We derive the trajectory reward from these quantities:

$$\mathcal{R}(\tau) = s(\tau)\,\big(1 + \alpha\, c(\tau) + \beta\, e(\tau)\big), \tag{1}$$

where $s(\tau) \in \{0, 1\}$ indicates whether the task is completed. The call score $c(\tau)$ evaluates each task-relevant deeplink, assigning full credit when it is invoked and reaches its expected landing, partial credit when it is invoked but fails the landing check, and zero when it is not invoked. The efficiency score $e(\tau) = \max(0,\, 1 - |\tau|/S_{\mathrm{ref}})$ is zero for trajectories at or above the reference length S_{ref} and increases linearly as trajectories become shorter. Because $s(\tau)$ gates the entire reward, a failed trajectory receives zero reward regardless of its deeplink calls or length. Among successful trajectories, the reward increases with the call score and with reductions in length below S_{ref}. We compute advantages with the group relative policy optimization (GRPO) estimator (Shao et al., 2024). All trajectories within a GRPO group receive the same deeplink candidate set, so their relative advantages are computed under the same action space. Candidate sets vary across groups to expose the model to different deeplink availability without mixing conditions within a group.

3.4 Agent Harness

The mining pipeline in Section 3.2 produces a verified and grounded deeplink catalog, while the training strategy in Section 3.3 teaches the model to choose between deeplink and GUI actions. To use the catalog and trained model together on real phones, the agent harness retrieves task-specific deeplinks, executes and verifies deeplinks with GUI fallback, and captures parameter values observed during use (Figure 2(c)).

Task-level retrieval. Exposing the full catalog at every step would consume substantial context and present the model with a large choice set. We therefore construct a task-specific candidate set before execution. The model decomposes the task into ordered navigation goals and identifies the application associated with each goal. For each goal, the harness restricts the candidate pool to that application and ranks the entries using semantic and lexical matching. It then merges the per-goal rankings and returns at most 10 deeplinks. The agent uses this compact candidate set throughout the task instead of the full catalog.

Landing verification and fallback. A deeplink is invoked through a system command, whose return status alone does not establish whether the intended screen was reached. With incorrect parameter values, the application may open an unexpected screen without reporting an error. Our harness therefore checks whether the call executed successfully and whether the observed landing Activity matches the expected one recorded in the catalog. It returns both results as the feedback f_t defined in Section 3.1. If either check fails, the next observation makes the failure explicit, allowing the agent to recover through GUI actions rather than continue as though the deeplink had succeeded.

Deeplink parameter capture during use. Deeplink parameters such as dates, locations, and item identifiers allow later calls to reuse information from a user's interaction history. During ordinary app use, the harness observes the Intents issued by the application, extracts reusable parameter values, and associates them with the corresponding catalog entries. The agent can then invoke these entries with values observed from that user, for example to reopen the page for a previously booked hotel or retrieve a past order by its identifier.

Together, the three components implement the deeplink-specific parts of the formulation in Section 3.1: retrieval constructs a task-specific candidate set from $\mathcal{A}_{\mathrm{DL}}$, verification produces f_t, and parameter capture adds observed values x for later calls (d, x). Additional implementation details are provided in Appendix D.

4 Experiments

4.1 Setup

Benchmarks and metrics. We evaluate GUI-Hopper on MobileWorld (Kong et al., 2026) and MobileWorld-Real (Zhou et al., 2026). MobileWorld provides a simulated Android environment with 13 built-in applications, where we evaluate all 117 tasks. We evaluate each MobileWorld task

Table 1: Experiment results on MobileWorld. GUI-Hopper improves success and efficiency over GUI-only training across three model scales. Its success-rate gains persist under GUI-only evaluation.

Model	Eval. with deeplink	SR (%) ↑	Avg. length (steps) ↓	
			Success-only	All
General-purpose VLMs				
Claude Opus 4.6 (Anthropic, 2026)	×	44.4	18.2	28.5
Gemini 3.1 Pro (Google DeepMind, 2026)	×	58.1	21.1	25.6
GPT-5.6-Sol (OpenAI, 2026)	×	70.1	21.9	26.5
Seed-2.0-Pro (ByteDance Seed, 2026a)	×	63.3	21.3	26.9
Specialized GUI Models				
GUI-Owl-1.5-4B-Instruct (Xu et al., 2026)	×	25.1	16.1	28.4
GELab-Zero-4B (Yan et al., 2025)	×	16.0	14.2	29.9
PhoneBuddy-4B (Tang et al., 2026)	×	23.1	12.1	27.8
ForgeOwl-8B (Liu et al., 2026)	×	41.3	16.8	28.2
AutoGLM-Phone-9B (Liu et al., 2024)	×	15.0	18.6	35.0
UI-TARS-1.5-7B (ByteDance Seed, 2025)	×	8.9	13.1	28.4
MAI-UI-8B (Zhou et al., 2025)	×	27.6	17.8	36.8
UI-Venus-1.5-30B-A3B (Gao et al., 2026)	×	19.9	12.3	31.0
UI-Venus-2-9B (Venus Team et al., 2026)	×	68.1	19.9	24.7
Qwen3.5-based models				
Qwen3.5-4B	×	15.7	12.4	32.9
+ GUI training	×	52.7	19.2	26.6
+ Hybrid training (ours)	×	53.0	18.9	23.5
+ Hybrid training (ours)	✓	**60.1**	17.5	20.9
Qwen3.5-9B	×	20.8	16.5	30.2
+ GUI training	×	54.4	19.4	25.7
+ Hybrid training (ours)	×	56.7	19.1	23.1
+ Hybrid training (ours)	✓	**63.5**	17.5	21.8
Qwen3.5-35B-A3B	×	17.7	13.0	27.5
+ GUI training	×	57.8	19.2	25.2
+ Hybrid training (ours)	×	61.8	19.0	24.3
+ Hybrid training (ours)	✓	**64.1**	16.7	21.0

three times per setting and compute success rate over all 351 task executions. For MobileWorld-Real, we select 26 frequently used applications from the full benchmark and apply the proposed automated pipeline to mine their deeplinks. We then select 119 tasks involving these applications from the full benchmark for evaluation. For both benchmarks, we report task success rate and the average number of interaction steps for successful trajectories and for all trajectories.

Training settings. For MobileWorld, we use Qwen3.5 (Qwen Team, 2026a) models at three scales: 4B, 9B, and 35B-A3B. At each scale, we compare the base model, an agent trained with GUI actions alone, and GUI-Hopper. The two trained agents use paired trajectories from the same tasks and otherwise follow identical training configurations; they differ only in whether deeplink actions and their associated feedback are available. We perform supervised fine-tuning for six epochs with a peak learning rate of 3×10^{-6}. After a warmup over the first 5% of training steps, the learning rate follows a cosine schedule and decays to 1×10^{-7}. We use a global batch size of 64, a microbatch size of 1, and a context length of 28,672 tokens. Online reinforcement learning uses a group size of 8, a rollout temperature of 1.0, and a constant learning rate of 1×10^{-6}, with $\alpha = 0.15$, $\beta = 0.10$, and $S_{\text{ref}} = 25$. Appendix C provides the complete training configuration.

Construction of deeplink catalogs. We construct the deeplink catalogs using the mining, verification, and grounding pipeline described in Section 3.2. The pipeline yields 34 entries for MobileWorld's built-in applications and 1,251 entries for the 26 selected MobileWorld-Real applications. At inference, we retrieve up to 10 candidate deeplinks from the catalog based on the current task and provide them to GUI-Hopper. In the main comparisons, the base model and GUI-only agent receive no deeplink candidates and operate solely in $\mathcal{A}_{\text{GUI}}$.

Table 2: Experiment results on MobileWorld-Real, a real-device mobile GUI agent benchmark. GUI-Hopper improves success and efficiency over GUI-only training on physical devices.

Model	Eval. with deeplink	SR (%) ↑	Avg. length (steps) ↓	
			Success-only	All
General-purpose VLMs				
Claude Opus 4.6 (Anthropic, 2026)	×	75.0	21.8	25.9
Gemini 3.1 Pro (Google DeepMind, 2026)	×	96.9	14.9	15.6
GPT-5.6-Sol (OpenAI, 2026)	×	96.6	21.2	21.7
Kimi K2.6 (Moonshot AI, 2026)	×	68.9	19.8	28.6
Qwen3.7-Plus (Qwen Team, 2026b)	×	74.7	20.0	27.1
Seed-2.1-Pro (ByteDance Seed, 2026b)	×	89.6	20.2	22.8
Specialized GUI Models				
GELab-Zero-4B (Yan et al., 2025)	×	34.1	20.5	35.3
GUI-Owl-1.5-32B-Instruct (Xu et al., 2026)	×	10.0	23.7	40.4
UI-Venus-1.5-30B-A3B (Gao et al., 2026)	×	43.8	11.3	31.9
UI-Venus-2-9B (Venus Team et al., 2026)	×	84.2	17.7	22.4
Qwen-UI-Agent-27B (Zhou et al., 2026)	×	92.0	16.5	18.1
Qwen3.5-based models				
Qwen3.5-4B	×	21.1	13.7	33.4
+ GUI training	×	70.2	17.6	26.5
+ Hybrid training (ours)	✓	**84.0**	16.7	20.4
Qwen3.5-9B	×	18.9	14.4	30.6
+ GUI training	×	77.1	17.4	24.2
+ Hybrid training (ours)	✓	**86.6**	16.6	20.4
Qwen3.5-35B-A3B	×	39.5	12.8	32.1
+ GUI training	×	79.2	17.3	23.7
+ Hybrid training (ours)	✓	**89.1**	15.7	18.2

4.2 Main Results

4.2.1 MobileWorld

Table 1 compares the base model, the GUI-only agent, and GUI-Hopper across three model scales. Both trained agents substantially improve over their corresponding base models. Across all three model scales, GUI-Hopper with access to deeplinks consistently achieves higher success rates than the GUI-only agent using GUI actions alone. It reduces average steps across all trajectories by 15.26–21.30%, with shorter successful trajectories at every scale. Verified deeplinks can take the agent directly to task-relevant screens, bypassing lengthy GUI navigation and reducing opportunities for navigation errors in long-horizon tasks.

Even when evaluated with GUI actions alone, GUI-Hopper's hybrid-SFT checkpoints achieve higher success rates than the GUI-only agent across all three model scales. GUI-Hopper also completes tasks in fewer GUI steps than the GUI-only agent. Together, these results show that the hybrid-trained model retains effective GUI ability: its gains are not confined to executions that can use direct navigation shortcuts. Appendix E.1 provides a visualization of the successful-trajectory length distributions at 9B.

One possible explanation is that hybrid-action training encourages the model to break tasks into intermediate goals and identify the screens needed to achieve them. This pattern, learned with hybrid interactions, may also generalize to mobile use with solely GUI actions, helping the agent avoid unnecessary navigation. An illustrative GUI-only case is provided in the appendix (Figure 6).

4.2.2 MobileWorld-Real

We deploy the complete pipeline on MobileWorld-Real to verify whether hybrid interaction generalizes to commercial applications running on physical devices. As summarized in Table 2, GUI-Hopper consistently outperforms the GUI-only agent in both task completion and trajectory efficiency, confirming that the efficiency advantages of deeplink-guided navigation effectively translate from simulation to complex real-world mobile environments.

4.3 Ablations

4.3.1 Contribution of Training Stages

We examine the contribution of the training stages (Table 3). Both the hybrid-SFT model and its online-RL refinement use Deeplink-Ten at evaluation. We include the base model evaluated with GUI actions as a reference.

Table 3: Training-stage ablation on Qwen3.5-4B. Online RL further improves success and efficiency after hybrid SFT.

Stage	SR (%) ↑	Length ↓
Base (GUI)	15.7	32.9
+ Hybrid SFT	57.8	22.5
+ Online RL	**60.1**	**20.9**

Trained variants use hybrid evaluation. Length: average steps over all trajectories.

Online reinforcement learning increases success rate from 57.83% to 60.11%, while reducing the average number of steps across all trajectories from 22.52 to 20.91. With the deeplink supply unchanged, these gains suggest that online feedback helps the agent make better use of the hybrid action space learned during supervised training.

4.3.2 Effect of Deeplink Supply

Although deeplinks can bypass lengthy GUI navigation, agents must select task-relevant deeplinks to actually realize this efficiency. Table 4 compares the effectiveness and robustness of Qwen3.7-plus and GUI-Hopper under different deeplink supply conditions.

Table 4: Deeplink supply on MobileWorld. Task-matched deeplinks improve general VLM performance even without hybrid training. Our trained GUI-Hopper gains more from deeplinks and retains these gains when irrelevant candidates are included.

Supply	Qwen3.7-plus		GUI-Hopper (9B)	
	SR (%) ↑	Length ↓	SR (%) ↑	Length ↓
NA	61.5	25.6	56.7	23.1
GT	62.4	21.4	63.8	21.7
Ten	59.5	21.4	63.5	21.8
Full	53.0	24.1	62.8	21.7

NA: no deeplinks; GT: task-matched; Ten: retrieval top 10; Full: full catalog. Length: average steps over all trajectories.

For Qwen3.7-plus, task-matched deeplinks (GT) primarily boost navigation efficiency—reducing trajectory length by 16.25%—even without hybrid-action training. However, expanding the catalog to Ten and Full steadily degrades its success rate (falling to 52.99%), indicating that broader candidate sets introduce decision noise that the base model cannot filter out.

In contrast, GUI-Hopper exhibits remarkable robustness to catalog expansion, maintaining nearly identical success rates ($\sim$63%) and step counts ($\sim$21.7) across settings of 'GT', 'Ten', and 'Full'. Notably, Ten trails the oracle GT setting by merely 0.29 percentage points. This confirms that hybrid-action training enables the policy to reliably discern and invoke task-relevant actions from noisy pools, supporting a practical division of labor where retrieval prunes the search space while the policy executes precise selection.

5 Conclusion

In this work, we challenge the assumption that mobile agents must navigate screen-by-screen. We introduce GUI-Hopper, a framework that integrates app-native deeplinks as a high-level navigation channel alongside fine-grained GUI control. To enable this, we develop an automated pipeline that mines deeplinks, validates them on real devices, and grounds them in descriptions of their landing screens. Through hybrid trajectory synthesis and online reinforcement learning, the model learns a robust policy to invoke deeplinks when optimal and seamlessly fall back to GUI actions when necessary, on both synthetic environments and real mobile phones. Our evaluations on MobileWorld and MobileWorld-Real demonstrate that GUI-Hopper significantly improves task success and reduces trajectory lengths across all tested model scales. Notably, these gains persist even when deeplinks are withheld at inference, underscoring enhanced underlying task-planning capabilities. By bridging the gap between app-native programmatic interfaces and fine-grained visual control, this work establishes a new foundation for the next generation of mobile GUI agents.

Weaknesses & Limitations. Without system privileges, deeplink coverage is limited to screens that applications expose for external access. Furthermore, our current evaluation is restricted to the Android ecosystem. Future work will explore system-level integrations and dynamically enrich the deeplink catalog using user-specific parameters captured during live interaction.

AI USE STATEMENT

In this work, we used generative AI tools for drafting and polishing the text of the paper and for assisting with the implementation of experiment and analysis code. We have not used generative AI tools for generating the research ideas or the experimental design, and the remaining required-disclosure categories are not applicable to this work. Additionally, generative AI models serve as components of the method itself: as described in Section 3.2, they generate and filter the natural-language descriptions in the deeplink catalog. We have reviewed all AI-assisted work: AI-drafted text was revised by the authors, and AI-assisted code was verified and tested for correctness. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

The agent interacts with applications only through their open interfaces: GUI operations available to any user, and deeplink entry points that applications expose for external invocation by design. Our experiments did not use real users' personal information. The user-specific examples in Appendix D illustrate potential applications of the implemented parameter-capture mechanism.

REPRODUCIBILITY STATEMENT

The mining pipeline is described in Section 3.2, with implementation details, prompts, and statistics in Appendix A. Appendix B presents the deeplink schema and the prompt format seen by the model. Training settings are provided in Section 4.1 and Appendix C, and the harness implementation is detailed in Appendix D. The evaluation protocol—benchmarks, task counts, supply conditions, and metrics—is specified in Section 4.1. We will release the trained models and the deeplink catalog for MobileWorld.

REFERENCES

Anthropic. Introducing Claude Opus 4.6. https://www.anthropic.com/news/claude-opus-4-6, 2026.

Apple Inc. App intents. https://developer.apple.com/documentation/appintents, 2024.

Tanzirul Azim, Oriana Riva, and Suman Nath. uLink: Enabling user-defined deep linking to app content. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2016.

ByteDance Seed. UI-TARS-1.5. https://seed-tars.com/1.5/, 2025.

ByteDance Seed. Seed 2.0 official launch. https://seed.bytedance.com/en/blog/seed-2-0-official-launch, 2026a.

ByteDance Seed. Seed2.1 officially released: Advancing AI productivity. https://seed.bytedance.com/en/blog/seed2-1-officially-released-advancing-ai-productivity, 2026b.

Wanxia Cao, Chengzhen Duan, Pei Fu, Pengzhi Gao, Niu Lian, Fazhan Liu, Hui Liu, Heng Qu, Qinzhuo Wu, Zhehao Yu, et al. Xiaomi-gui-0 technical report. *arXiv preprint arXiv:2606.31410*, 2026.

Changlong Gao, Zhangxuan Gu, Yulin Liu, Xinyu Qiu, Shuheng Shen, Yue Wen, Tianyu Xia, Zhenyu Xu, Zhengwen Zeng, Beitong Zhou, Xingran Zhou, Weizhi Chen, Sunhao Dai, Jingya Dou, Yichen Gong, Yuan Guo, Zhenlin Guo, Feng Li, Qian Li, Jinzhen Lin, Yuqi Zhou, Linchao Zhu, Liang Chen, Zhenyu Guo, Changhua Meng, and Weiqiang Wang. UI-Venus-1.5 Technical Report. *arXiv preprint arXiv:2602.09082*, 2026. URL https://arxiv.org/abs/2602.09082.

Google. App actions overview. https://developer.android.com/develop/devices/assistant/overview, 2024.

Google. About deep links. https://developer.android.com/training/app-links, 2026a.

Google. Intents and intent filters. https://developer.android.com/guide/components/intents-filters, 2026b.

Google. Create shortcuts. https://developer.android.com/develop/ui/compose/system/shortcuts/creating-shortcuts, 2026c.

Google DeepMind. Gemini 3.1 Pro model card. https://deepmind.google/models/model-cards/gemini-3-1-pro/, 2026.

Han Hu, Han Wang, Ruiqi Dong, Xiao Chen, and Chunyang Chen. Enhancing GUI exploration coverage of Android apps with deep link-integrated monkey. *arXiv preprint arXiv:2404.19307*, 2024. URL https://arxiv.org/abs/2404.19307.

Xuhao Hu, Xi Zhang, Haiyang Xu, Kyle Qiao, Jingyi Yang, Xuanjing Huang, Jing Shao, Ming Yan, and Jieping Ye. ToolCUA: Towards optimal GUI-tool path orchestration for computer use agents. *arXiv preprint arXiv:2605.12481*, 2026. URL https://arxiv.org/abs/2605.12481.

Lawrence Keunho Jang, Jing Yu Koh, Daniel Fried, and Ruslan Salakhutdinov. Odysseys: Benchmarking web agents on realistic long horizon tasks. *arXiv preprint arXiv:2604.24964*, 2026.

Wenjia Jiang, Yangyang Zhuang, Chenxi Song, Xu Yang, Joey Tianyi Zhou, and Chi Zhang. AppAgentX: Evolving GUI agents as proficient smartphone users. *arXiv preprint arXiv:2503.02268*, 2025. URL https://arxiv.org/abs/2503.02268.

Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1–2):99–134, 1998. doi: 10.1016/S0004-3702(98)00023-X.

Quyu Kong, Xu Zhang, Zhenyu Yang, Nolan Gao, Chen Liu, Panrong Tong, Chenglin Cai, Hanzhang Zhou, Jianan Zhang, Liangyu Chen, et al. Mobileworld: Benchmarking autonomous mobile agents in agent-user interactive and mcp-augmented environments. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6142–6167, 2026.

Fang Liu, Chun Wang, Andres Pico, Danfeng Yao, and Gang Wang. Measuring the insecurity of mobile deep links of Android. In *USENIX Security Symposium*, 2017.

Guangyi Liu, Pengxiang Zhao, Gao Wu, Yiwen Yin, Mading Li, Liang Liu, Congxiao Liu, Zhang Qi, Mengyan Wang, Liang Guo, Jiangning Zhang, and Yong Liu. MobileForge: Annotation-free adaptation for mobile GUI agents with hierarchical feedback-guided policy optimization. *arXiv preprint arXiv:2606.19930*, 2026. URL https://arxiv.org/abs/2606.19930.

Xiao Liu, Bo Qin, Dongzhu Liang, Guang Dong, Hanyu Lai, Hanchen Zhang, Hanlin Zhao, Iat Long Iong, Jiadai Sun, Jiaqi Wang, et al. AutoGLM: Autonomous foundation agents for GUIs. *arXiv preprint arXiv:2411.00820*, 2024. URL https://arxiv.org/abs/2411.00820.

Lychee Team. KnowAct-GUIClaw: Know deeply, act perfectly, personal GUI assistant with self-evolving memory and skill. *arXiv preprint arXiv:2607.12625*, 2026.

Yun Ma, Ziniu Hu, Yunxin Liu, Tao Xie, and Xuanzhe Liu. Aladdin: Automating release of deep-link APIs on Android. In *Proceedings of the Web Conference (WWW)*, 2018.

Moonshot AI. Kimi K2.6: Advancing open-source coding. https://www.kimi.ai/blog/kimi-k2-6, 2026.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition. https://openai.com/index/gpt-5-6/, 2026.

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

Qwen Team. Qwen3.5: Towards native multimodal agents. https://qwen.ai/blog?id=qwen3.5, 2026a.

Qwen Team. Qwen3.7-Plus: Multimodal agent intelligence. https://qwen.ai/blog?id=qwen3.7-plus, 2026b.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024. URL https://arxiv.org/abs/2402.03300.

Yifan Sui, Xin Huang, Hongbing Li, Fang Xu, Jiahe Lv, Haolong Yan, Yeqing Shen, Litao Liu, Zhimin Fan, Ziyang Meng, et al. Androiddaily: A verifiable benchmark for mobile gui agents on real-world closed-source applications. *arXiv preprint arXiv:2605.27761*, 2026.

Yutian Tang, Yulei Sui, Haoyu Wang, Xiapu Luo, Hao Zhou, and Zhou Xu. All your app links are belong to us: Understanding the threats of instant apps based attacks. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

Zhengyang Tang, Xin Lai, Pengyuan Lyu, Xinyuan Wang, Tianyi Bai, Chenxin Li, Yiduo Guo, Huawen Shen, Yuxuan Liu, Junyi Li, Zhengyao Fang, Yang Ding, Yi Zhang, Weinong Wang, Xingran Zhou, Liang Wu, Fei Tang, Sunqi Fan, Shangpin Peng, Zheng Ruan, Anran Zhang, Benyou Wang, Ji-Rong Wen, Rui Yan, Chengquan Zhang, and Han Hu. PhoneBuddy: Training open models for agentic phone use. *arXiv preprint arXiv:2606.23049*, 2026. URL https://arxiv.org/abs/2606.23049.

Venus Team, Zhuohan Cai, Haoxing Chen, Jiaxuan Chen, Weizhi Chen, Changlong Gao, Zhangxuan Gu, Yuan Guo, Yusong Hu, Jianrong Jiang, Jianguo Li, Runze Li, Jinzhen Lin, Zhenyu Ma, Changhua Meng, Han Peng, Xinyu Qiu, Shuheng Shen, Zhongyi Shui, Weiqiang Wang, Ming Wen, Zhuoer Xu, Hang Yan, Kaiwen Yang, Ruilin Yao, Nanjun Yu, Zhengwen Zeng, Lianrui Zhang, Yunzhu Zhang, Zhe Zhao, and Beitong Zhou. UI-Venus-2 technical report. *arXiv preprint arXiv:2609.00028*, 2026. URL https://arxiv.org/abs/2609.00028.

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

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

Zhenhailong Wang et al. Mobile-Agent-E: Self-evolving mobile assistant for complex tasks. *arXiv preprint arXiv:2501.11733*, 2025b.

Liangxuan Wu et al. From assistants to adversaries: Exploring the security risks of mobile LLM agents. *arXiv preprint arXiv:2505.12981*, 2025.

Weikai Xie, Li Zhang, Shihe Wang, Rongjie Yi, and Mengwei Xu. DroidCall: A dataset for LLM-powered Android intent invocation. In *Findings of the Association for Computational Linguistics: EMNLP*, 2025.

Haiyang Xu, Xi Zhang, Haowei Liu, Junyang Wang, Zhaozai Zhu, Shengjie Zhou, Xuhao Hu, Feiyu Gao, Junjie Cao, Zihua Wang, Zhiyuan Chen, Jitong Liao, Qi Zheng, Jiahui Zeng, Ze Xu, Shuai Bai, Junyang Lin, Jingren Zhou, and Ming Yan. Mobile-agent-v3.5: Multi-platform fundamental gui agents. *arXiv preprint arXiv:2602.16855*, 2026.

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

Haolong Yan, Jia Wang, Xin Huang, Yeqing Shen, Ziyang Meng, Zhimin Fan, Kaijun Tan, Jin Gao, Lieyu Shi, Mi Yang, et al. Step-gui technical report. *arXiv preprint arXiv:2512.15431*, 2025.

Yuhao Yang, Zhen Yang, Zi-Yi Dou, Anh Nguyen, Keen You, Omar Attia, Andrew Szot, Michael Feng, Ram Ramrakhya, Alexander Toshev, et al. Ultracua: A foundation model for computer use agents with hybrid action. *arXiv preprint arXiv:2510.17790*, 2025.

Chi Zhang et al. AppAgent: Multimodal agents as smartphone users. *arXiv preprint arXiv:2312.13771*, 2023.

Pengxiang Zhao et al. MAS-Bench: A unified benchmark for shortcut-augmented hybrid mobile GUI agents. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2026. arXiv:2509.06477.

Hanzhang Zhou, Xu Zhang, Panrong Tong, Jianan Zhang, Liangyu Chen, Quyu Kong, Chenglin Cai, Chen Liu, Yue Wang, Jingren Zhou, and Steven Hoi. MAI-UI Technical Report: Real-World Centric Foundation GUI Agents. *arXiv preprint arXiv:2512.22047*, 2025. URL https://arxiv.org/abs/2512.22047.

Hanzhang Zhou, Panrong Tong, Xu Zhang, Quyu Kong, Chenglin Cai, Tianyu Xia, Gongjie Zhang, Jianan Zhang, Long Li, Long Chen, Lei Wang, Gaole Dai, Pengxiang Li, Liangyu Chen, Yue Wang, and Steven Hoi. Qwen-UI-Agent technical report: Toward next-generation real-world centric foundation GUI agents. *arXiv preprint arXiv:2607.28227*, 2026. URL https://arxiv.org/abs/2607.28227.

A MINING PIPELINE: IMPLEMENTATION, PROMPTS, AND CATALOG STATISTICS

This appendix supplements Section 3.2 with implementation details, final catalog statistics, and the prompts used in the mining pipeline.

Static analysis. Candidate entries are extracted with the Shortcut-Anything tool released alongside MAS-Bench (Zhao et al., 2026), which parses manifest declarations and shortcut definitions from the application package. Entries that differ only in how parameters are represented in the URI are consolidated by URI structure. Static extraction alone does not establish whether an entry works, whether it opens a meaningful screen, or which parameter values it accepts. The remainder of the pipeline answers these questions on real devices.

Exportability filtering. Android only allows external invocation of components that the application explicitly exports. We therefore filter out candidates whose target components are not exported.

Resolvability screening. For each remaining candidate, we use a read-only package-manager query to check whether an installed component can handle it, without launching the application or invoking a model.

On-device verification. We test the resolvable candidates on real devices and retain only those that open a meaningful screen. For parameterized entries, we test candidate values and retain a parameter only if changing its value visibly changes the landing screen.

Deduplication. Different verified entries can lead to the same screen—the search screen of one map application was reachable through 11 entries spanning 5 domain variants and 3 URI schemes. We consolidate duplicates using criteria ordered by evidential strength: repeated calls on the device, screenshot and text comparisons, and a final command-level check in which byte-identical invocation commands take precedence over the other evidence. Uncertain cases are left unmerged to avoid merging distinct entries. We also exclude entries that open transient content instances rather than stable functional screens.

Catalog statistics. The final catalog contains 1,251 agent-callable entries across 26 applications, including 246 entries with input parameters. Table 5 reports the per-application distribution.

Table 5: Per-application counts in the verified deeplink catalog for MobileWorld-Real. The catalog contains 1,251 callable entries across 26 applications.

Application	Entries	Application	Entries
Bilibili	330	Huaxiaozhu	15
Xiaohongshu	177	Baidu Maps	11
Weibo	148	Meituan Dache	11
NetEase Cloud Music	128	NetEase News	9
Douyin	91	Tencent News	5
iQIYI	75	Baidu	4
Piaoniu	60	Kuaishou	3
AMap	44	Tomato Novel	3
Zhihu	43	Tencent Video	2
Taopiaopiao	35	Baidu Tieba	1
Didi	20	Toutiao Lite	1
Youdao	17	Dongchedi	1
Autohome	16	Doubao	1
Total			**1,251**

Prompts. A vision-language model assists with on-device verification and deduplication. During verification, the prompt guides the model to propose parameter values, invoke candidates, assess their landing screens, and write catalog descriptions in the format specified in Appendix B. It also

instructs the model to identify tracking- or session-bound parameters that should be excluded from the released catalog. This prompt is released together with the catalog. For deduplication, the two prompts below ask the model to group equivalent entries from the same application and URI base. The second also uses landing-screen screenshots as evidence. The calling program then selects which entry to retain under fixed rules.

Deeplink deduplication prompt

```
You are deduplicating a batch of Android deeplink tools. They come from the
same app and the same URI base, but they may be entirely different features
(sharing one routing shell), or one feature inflated into several entries
by different parameter values.

Criteria (follow strictly; do not improvise):
- SAME feature = lands on the same page, and the parameters are semantically
  interchangeable (only their values differ).
- DIFFERENT features = different landing pages, or parameters that denote
  different business objects.
- Same page but resting on a different tab / sub-page -> count as different
  features, and note "same page, different tab" in the reason.
- When unsure, put the entry in its own group and mark uncertain=true.
  Do not guess: a wrong merge costs far more than a missed one.

Output strict JSON, with no explanatory text:
{"groups":[{"members":["<id>"],"reason":"...","uncertain":false}]}
Use the short ids from the input verbatim (like c01); do not rewrite or
expand them. Every input id must appear in exactly one group -- none
missing, none repeated.
Do not output a "keep" field -- which entry to keep is decided by the
caller under fixed rules, not by you.
```

Deeplink deduplication prompt with screenshot evidence

```
Below are several deeplink candidates from the same app and the same URI
base. For each candidate you are given: id, tool name, parameter names,
textual description, and the screenshot of the landing page it actually
opened on a real device.

Judge whether they are the same feature, and split them into groups.

Criteria (follow strictly):
- SAME feature = lands on the same page, parameters only differ in value
  (e.g., different products on the same product-detail page).
- DIFFERENT features = different landing pages, or parameters that denote
  different business objects.
- Same page but resting on a different tab / sub-page -> different
  features; note it in the reason.
- A screenshot showing an error page / blank page / login page / loading
  state cannot serve as evidence of sameness -- two entries both failing
  to open does not make them the same feature. Put such an entry in its
  own group and mark uncertain=true.
- Trust the screenshot. When the screenshot conflicts with the textual
  description, follow the screenshot and state the conflict in the reason.
- When unsure, put the entry in its own group and mark uncertain=true.
  Do not guess: a wrong merge costs far more than a missed one.

Output strict JSON, with no explanatory text:
{"groups":[{"members":["<id>"],"reason":"...","uncertain":false}]}
Use the short ids from the input verbatim (like c01). Every input id must
appear in exactly one group -- none missing, none repeated.
```

B Deeplink Catalog Format and Agent Prompt

This appendix details the catalog entry format introduced in Section 3.2 and shows how deeplink candidates are presented to the model.

Schema. Each catalog entry contains model-facing and harness-facing fields. The model-facing fields hold the name, the description, and the parameter definitions. The tool description names the landing screen and ends with an `After firing:` clause describing the expected outcome and any GUI actions still required. Each parameter definition specifies its meaning, format constraints, an example value, and mistakes to avoid. The harness-facing fields, hidden from the model, store a

parameterized Android activity-manager command template and the expected landing Activity. The harness uses these fields to execute the call and compare the observed Activity with the expected one (Section 3.4). Figure 3 illustrates an AMap driving-route entry and its observed landing screen.

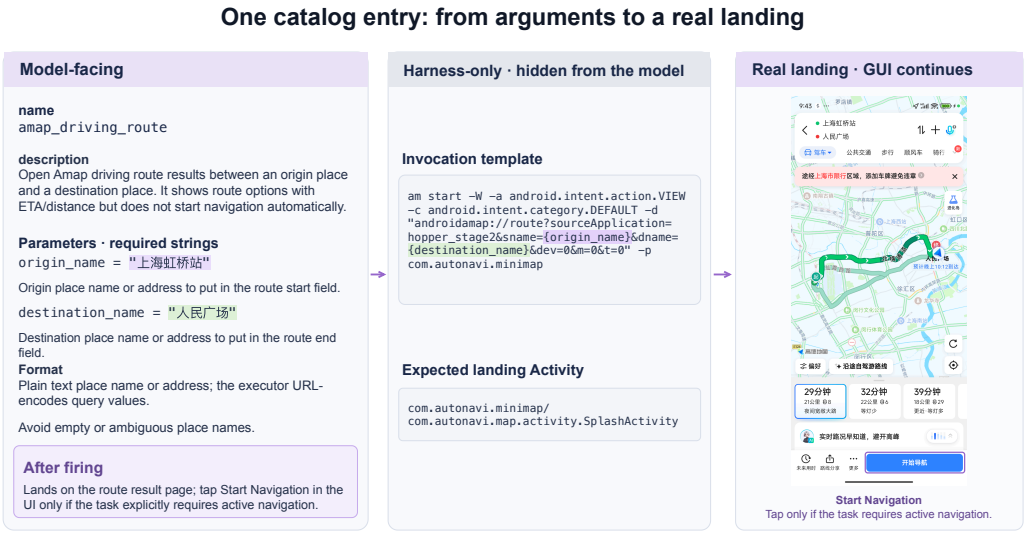

Figure 3: An AMap driving-route entry connecting model-facing descriptions with harness-side execution and landing verification. The deeplink opens route results for the specified origin and destination. Starting navigation remains a GUI action when required by the task.

Prompt format. The model-facing fields of the selected deeplink entries are included in the system prompt. The listing below presents the system prompt for one evaluation request, with the application list omitted and the 10 candidate deeplinks retained. The prompt defines the agent role, output format, and GUI action space before presenting the deeplink candidates as function-calling schemas. A final usage guideline instructs the agent to verify task completion through the GUI rather than rely on a successful deeplink call alone.

```
You are a GUI agent. You are given a task and your action history, with screenshots. You
    need to perform the next action to complete the task.

## Output Format
For each function call, return the thinking process in <think> </think> tags, and a json
    object with function name and arguments within <tool_call></tool_call> XML tags:
```
<think>
...
</think>
<tool_call>
{"name": "mobile_use", "arguments": <args-json-object>}
</tool_call>
```

## Action Space
{"action_type": "click", "coordinate": [x, y]}
{"action_type": "double_click", "coordinate": [x, y]}
{"action_type": "long_press", "coordinate": [x, y]}
{"action_type": "type", "text": ""}
{"action_type": "open", "text": "app_name"} # app_name must be chosen from the available
    apps listed in the Note section.
{"action_type": "drag", "start_coordinate": [x1, y1], "end_coordinate": [x2, y2]}
{"action_type": "system_button", "button": "button_name"} # Options: back, home, menu,
    enter
{"action_type": "wait", "waiting_time": "xxx"} # waiting_time denotes the wait time in
    seconds.
{"action_type": "ask_user", "text": "xxx"} # you can ask user for more information to
    complete the task.
{"action_type": "terminate", "answer": "xxx", "status": "success or failed"}

## Deeplink Tools
You are also provided with Deeplink tools, you can use them to complete the task.
```

```
{"type": "function", "function": {"name": "clock_open_timer", "description": "[Clock]
    Open Clock to the Timer screen. After firing: set duration and start timer.",
    "parameters": {"type": "object", "properties": {}, "required": []}}}
{"type": "function", "function": {"name": "calendar_open_view_or_event", "description":
    "[Calendar] Open Fossify Calendar to a date in day, month, year, agenda, or week
    view. After firing: calendar opened to requested date/view.", "parameters": {"type":
    "object", "properties": {"day_code": {"type": "string", "description": "Target date
    to open in the calendar. Format: YYYYMMDD. Example: 20261224. If omitted/empty:
    Omitted or empty uses device GMT current date. Avoid: Do not use dashed dates;
    invalid formats may crash."}, "view_to_open": {"type": "string", "description":
    "View selector: 1 month, 2 year, 3 agenda, 4 week. Format: Integer 1, 2, 3, or 4.
    Example: 1. If omitted/empty: Omit for day view; empty string is invalid. Avoid: Do
    not use empty or outside 1-4."}}, "required": []}}}
{"type": "function", "function": {"name": "clock_set_timer", "description": "[Clock]
    Create and start a countdown timer in Clock. After firing: timer starts; UI opens
    unless skip_ui=true.", "parameters": {"type": "object", "properties":
    {"length_seconds": {"type": "integer", "description": "Countdown duration in
    seconds. Format: Positive integer seconds. Example: 90. If omitted/empty: Required;
    no default."}, "message": {"type": "string", "description": "Timer label. Format:
    Plain text label. Example: Tea break. If omitted/empty: Omitted or empty uses
    default duration label."}, "skip_ui": {"type": "boolean", "description": "Whether to
    avoid foregrounding Clock. Format: Boolean true or false. Example: False. If
    omitted/empty: Omitted opens Clock; empty string is invalid. Avoid: Do not pass an
    empty boolean value."}}, "required": ["length_seconds"]}}}
{"type": "function", "function": {"name": "camera_open_photo_capture", "description":
    "[Camera] Open Android camera photo-capture UI for taking a still image. After
    firing: press shutter to capture.", "parameters": {"type": "object", "properties":
    {"output_uri": {"type": "string", "description": "Destination file URI for captured
    image. Format: file:///sdcard/.../*.jpg. Example:
    file:///sdcard/DCIM/Camera/reenrich_photo_capture.jpg. If omitted/empty: Optional;
    omit for app-managed destination. Empty is not useful. Avoid: Do not use empty or
    non-file destinations."}}, "required": []}}}
{"type": "function", "function": {"name": "mail_compose", "description": "[Mail] Open
    email compose with recipient and optional subject or body. After firing: tap Send to
    send email.", "parameters": {"type": "object", "properties": {"to": {"type":
    "string", "description": "Recipient email address. Format: RFC-style email address.
    Example: greg@example.com. If omitted/empty: Required; empty opens compose without
    recipient. Avoid: Do not use display names only."}, "subject": {"type": "string",
    "description": "Email subject line. Format: Plain text; URL-encode reserved
    characters. Example: ColdSubjectOnly. If omitted/empty: Omitted or empty leaves
    subject blank. Avoid: Do not combine with body when body must prefill."}, "body":
    {"type": "string", "description": "Email body text. Format: Plain text; URL-encode
    reserved characters. Example: ColdBodyOnly. If omitted/empty: Omitted or empty
    leaves body blank. Avoid: Subject parameter present causes body to be ignored."}},
    "required": ["to"]}}}
{"type": "function", "function": {"name": "contacts_insert", "description": "[Contacts]
    Open new-contact editor with contact fields prefilled; saving remains manual. After
    firing: tap Save to create the contact.", "parameters": {"type": "object",
    "properties": {"name": {"type": "string", "description": "Contact display name;
    editor splits full names. Format: Human name, preferably given and family name.
    Example: Alex Chen. If omitted/empty: Omitted or empty leaves name fields blank.
    Avoid: Do not expect full name to remain one field."}, "phone": {"type": "string",
    "description": "Contact phone number. Format: E.164 or local dialable number.
    Example: +14155550123. If omitted/empty: Omitted or empty leaves phone blank. Avoid:
    Phone alone does not infer a name."}, "company": {"type": "string", "description":
    "Contact organization name. Format: Plain organization name. Example: Northwind
    Labs. If omitted/empty: Omitted or empty leaves company blank."}, "email": {"type":
    "string", "description": "Contact email address. Format: Valid email address.
    Example: alex.chen@example.com. If omitted/empty: Omitted or empty leaves email
    blank."}, "job_title": {"type": "string", "description": "Role title under
    organization. Format: Plain job title. Example: Product Manager. If omitted/empty:
    Omitted or empty leaves title blank."}, "postal": {"type": "string", "description":
    "Contact postal address. Format: Single-line postal address. Example: 1 Market St,
    San Francisco, CA 94105. If omitted/empty: Omitted or empty leaves address blank."},
    "notes": {"type": "string", "description": "Contact note text. Format: Short
    free-text note. Example: Met at launch planning. If omitted/empty: Omitted or empty
    leaves notes blank."}}, "required": []}}}
{"type": "function", "function": {"name": "contacts_open", "description": "[Contacts]
    Open an existing contact's detail page by exact display name. After firing: contact
    detail page opens.", "parameters": {"type": "object", "properties": {"name":
    {"type": "string", "description": "Exact display name of an existing contact.
    Format: Exact Contacts display_name text. Example: Emily Johnson. If omitted/empty:
    Required; empty or unresolved names do not open a contact. Avoid: Do not use
    nicknames, partial names, or unknown contacts."}}, "required": ["name"]}}}
{"type": "function", "function": {"name": "calendar_insert_event", "description":
    "[Calendar] Open Calendar's new-event form with title, location, description, time,
    and all-day fields prefilled. After firing: review and tap Save to create the
    event.", "parameters": {"type": "object", "properties": {"title": {"type": "string",
    "description": "Event title. Format: Plain text. Example: DeepLink GMT Planning. If
    omitted/empty: Omitted or empty leaves title blank."}, "beginTime": {"type":
    "string", "description": "Event start time. Format: Epoch milliseconds; interpret
```

```
    date strings as GMT before firing. Example: 1760799600000. If omitted/empty: Omitted
    defaults to current time; empty long fails. Avoid: Do not local-time shift or pass
    empty strings."}, "endTime": {"type": "string", "description": "Event end time.
    Format: Epoch milliseconds; interpret date strings as GMT before firing. Example:
    1760803200000. If omitted/empty: Omitted defaults to current time; empty long fails.
    Avoid: Provide with beginTime for future events."}, "eventLocation": {"type":
    "string", "description": "Event location text. Format: Place name or address.
    Example: Conference Room 3. If omitted/empty: Omitted or empty leaves location
    blank."}, "description": {"type": "string", "description": "Event notes. Format:
    Plain text. Example: Discuss integration checkpoints. If omitted/empty: Omitted or
    empty leaves description blank."}, "allDay": {"type": "string", "description":
    "Whether the event is all-day. Format: true or false. Example: false. If
    omitted/empty: Omitted behaves as false; empty boolean fails. Avoid: Do not pass
    empty or non-boolean values."}}, "required": []}}}
{"type": "function", "function": {"name": "clock_set_alarm", "description": "[Clock]
    Create an alarm with hour, optional minute, and optional label. After firing: alarm
    created and Clock alarm list opens.", "parameters": {"type": "object", "properties":
    {"hour": {"type": "string", "description": "Alarm hour in 24-hour time. Format:
    Integer 0-23. Example: 20. If omitted/empty: Required; no default. Avoid: Do not use
    12-hour AM/PM text."}, "minutes": {"type": "string", "description": "Alarm minute.
    Format: Integer 0-59. Example: 20. If omitted/empty: Omitted defaults to 0. Avoid:
    Do not include colon-formatted time."}, "message": {"type": "string", "description":
    "Alarm label. Format: Short plain text label. Example: ReenrichAlarm. If
    omitted/empty: Omitted or empty creates an unlabeled alarm. Avoid: Avoid unquoted
    spaces in shell commands."}}, "required": ["hour"]}}}
{"type": "function", "function": {"name": "chrome_open_url", "description": "[Chrome]
    Open a complete URL in Chrome for manual reading or follow-up. After firing: page
    opened for reading.", "parameters": {"type": "object", "properties": {"url":
    {"type": "string", "description": "Complete web URL to open. Format: Absolute URL
    with scheme, usually https://. Example:
    https://example.com/?deeplink_stage2=chrome_open_url. If omitted/empty: Required;
    empty fails, omitted opens app chooser. Avoid: Do not omit scheme or pass an empty
    URL."}}, "required": ["url"]}}}

If you want to use Deeplink tools, you must output as the following format:
```
<think>
...
</think>
<tool_call>
{"name": <function-name>, "arguments": <args-json-object>}
</tool_call>
```
## Note
- Available Apps: `[... 47 application names, elided ...]`.
- You should use the `open` action to open the app as possible as you can, because it is
    the fast way to open the app.
- You must follow the Action Space strictly, and return the correct json object within
    <think> </think> and <tool_call></tool_call> XML tags.

# Deeplink shortcuts
Besides the GUI/MCP tools, the following functions are deeplink shortcuts that jump
    straight to a screen or pre-fill a form, saving navigation/typing.
- A deeplink only OPENS / PRE-FILLS. It NEVER sends, saves, or posts; finish with GUI
    actions (e.g. tap Send / Save).
- CRITICAL: a deeplink's success ONLY means the screen/form is ready, NOT that the task
    is done. Do NOT terminate or answer right after a deeplink; first finish the submit
    (tap Send / Save / Post / Confirm) and verify it on screen. Each tool description
    says whether a final submit is still needed.
- Its result is a JSON string {"success": bool, "error": ...}. Pre-filled content may be
    longer than the visible area; trust it (or scroll to verify) instead of re-typing.
- Use a shortcut only when it matches the current sub-goal; otherwise keep using the GUI
    tool.
```

C Training Configuration

This appendix completes the training configuration of Section 4.1. Supervised fine-tuning additionally uses a warmup ratio of 0.05, bf16 precision, and no sequence packing. At each model scale, the GUI-only and hybrid models use the same SFT configuration, differing only in training trajectories.

Online reinforcement learning initializes the policy from the SFT checkpoint. Each iteration samples 117 tasks with eight trajectories per task. We limit each trajectory to 50 interaction steps and the context to 20,480 tokens. The resulting interaction steps are used for policy updates with a global batch size of 512. The policy objective uses asymmetric clipping with $\epsilon_{\text{low}} = 0.2$ and $\epsilon_{\text{high}} = 0.28$,

without KL regularization. When computing $c(\tau)$, a task-relevant deeplink receives partial credit of 0.3 if it is invoked during the trajectory but none of its invocations reaches the expected Activity.

D AGENT HARNESS IMPLEMENTATION

Execution and feedback. The model generates a deeplink call in the format shown in Appendix B. The harness fills the catalog entry's activity-manager command template with the supplied parameter values and executes the resulting command on the device to dispatch an Android Intent. Successful dispatch does not guarantee arrival at the expected screen: Android routes the Intent through the application's entry logic, which may briefly bring a splash or routing Activity to the foreground before the destination appears. The harness therefore verifies the landing by checking whether the Activity specified in the catalog entry reaches the foreground. If the target application is in the foreground but the expected Activity has not appeared, the harness waits up to 10 seconds; if the foreground package is not the target application, it stops waiting within 4 seconds. The harness then returns the outcome—whether the dispatch succeeded, which Activity was actually reached, and whether it matches the expected one. This feedback, denoted by f_t in Section 3.1, appears beside the screenshot in the next observation as a text block prefixed with `Tool call result:`. If dispatch fails or the observed Activity does not match the expected one, the agent can use this feedback to continue through GUI actions.

Task-level retrieval. Retrieval runs once per task, before execution starts. A language model splits the task into navigation goals in execution order and names the application each goal belongs to. For each goal, the candidate pool is restricted to tools from the corresponding application. We use reciprocal rank fusion to combine embedding-based and BM25 rankings. The lexical scores affect the ranking but do not filter out candidates. We first select the highest-ranked candidate not already chosen from each goal's list. We then fill the remaining slots according to the fused ranking, returning up to 10 entries. To assess retrieval coverage, we manually annotated deeplink requirements for 100 tasks. Our retrieval method achieves a COMP@10 of 98.3%, indicating high coverage of task-relevant deeplinks with a candidate set of at most 10 entries.

Capture in use. The harness captures parameter values from Intents observed during application use. These Intents can specify an action, a target component, a data URI with query parameters, and key–value extras. For a mined entry, observation provides values used in the user's own interactions—for example, an order identifier, a saved address, or a travel date—that probing alone may not recover. The user-specific examples above illustrate potential deployment scenarios. Following the filtering criteria used in mining (Appendix A), the harness excludes parameters used for tracking or tied to a particular session. It associates retained values with the corresponding catalog entries so that later calls can reuse them.

E FURTHER ANALYSIS

E.1 GUI ABILITY WITHOUT DEEPLINKS

Hybrid interaction should expand an agent's capabilities without making it dependent on deeplink availability. Under GUI-only evaluation, GUI-Hopper achieves higher success rates than the GUI-only trained model at all three scales (Table 1). Figure 4 further compares their successful-trajectory length distributions at 9B, showing comparable execution efficiency with a slightly lower mean length for GUI-Hopper. Together, these results suggest that learning to use deeplinks complements GUI ability: the model's advantage is not confined to executions that invoke direct navigation shortcuts.

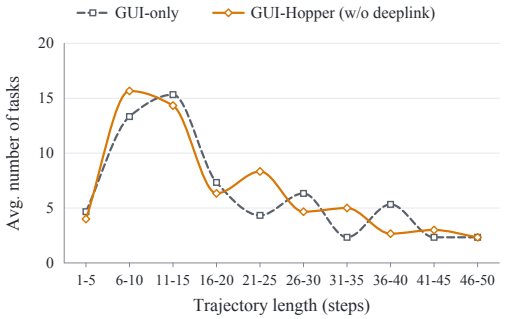

Figure 4: Successful-trajectory length distributions under GUI-only evaluation on MobileWorld.

One possible explanation is that hybrid-action training encourages the model to break tasks into intermediate goals and identify the screens needed to achieve them. This planning pattern may also guide execution when only GUI actions are available. The 35B-A3B case in Figure 6 illustrates this distinction. GUI-Hopper returns to the resume earlier to obtain the recipient's email address, whereas the baseline searches Mail and Contacts before returning to the document. Finding the address takes two steps for GUI-Hopper versus 21 for the baseline, while both use nine steps to compose and send the email. The savings therefore come from choosing a more direct path to the required information, rather than using fewer actions to operate the email form.

E.2 Efficiency across Task Structures

Table 6: Qwen3.5-9B performance by task structure on MobileWorld. Success gains are larger for single-app tasks, while trajectory reductions are larger for multi-app tasks.

Task group	Model / Evaluation	SR (%) ↑	Avg. length (steps) ↓	
			Success-only	All
Single-app	GUI-only / GUI	53.3	16.8	23.7
	GUI-Hopper / Hybrid	**68.5**	16.5	21.0
Multi-app	GUI-only / GUI	55.4	21.7	27.4
	GUI-Hopper / Hybrid	**59.1**	**18.5**	**22.4**

Single-app: 55 tasks; multi-app: 62 tasks; three runs per task. Success-only lengths use each condition's own successful trajectories.

Cross-application tasks require the agent to reach relevant screens in several apps, creating repeated opportunities for deeplinks to bypass GUI navigation. Table 6 compares the Qwen3.5-9B GUI-only model and GUI-Hopper on single-app and multi-app tasks. GUI-Hopper improves success in both groups while reducing average trajectory length over all runs by 18.4% on multi-app tasks and 11.2% on single-app tasks.

The 4B example in Figure 5 illustrates this cross-application use. GUI-Hopper invokes deeplinks to open Downloads and prefill an SMS, interleaving them with GUI actions to read the resume and send the message. This illustrates how app-native entry points can be combined with GUI actions to shorten multiple stages of a cross-application workflow.

E.3 Task Execution Time

Deeplinks can save time by bypassing repeated model decisions and device interactions during GUI navigation. To assess this practical benefit, we compare the GUI-only trained model using GUI actions with GUI-Hopper using both GUI and deeplink actions. Tables 7 and 8 report mean durations over successful and unsuccessful trajectories. For MobileWorld-Real, each comparison uses the same tasks at a given model scale and counts at most the first 50 steps of each trajectory, matching the main evaluation budget.

Table 7: Recorded task durations on MobileWorld. GUI-Hopper has lower recorded times at all three scales, although the two models use different timing methods.

Model size	GUI-only (s) ↓	GUI-Hopper (s) ↓
4B	346.5	181.7
9B	395.8	283.4
35B-A3B	493.9	212.9

On MobileWorld-Real, GUI-Hopper reduces mean execution time by 2.9–11.0% across the three model scales. Its efficiency advantage therefore extends from fewer actions to less elapsed time under the evaluated configurations. MobileWorld records also show lower durations for GUI-Hopper, although their different timing methods preclude a controlled speedup estimate.

Table 8: Execution time on MobileWorld-Real. GUI-Hopper reduces mean elapsed time by 2.9–11.0% relative to the GUI-only trained model.

Model size	GUI-only (s) ↓	GUI-Hopper (s) ↓
4B	424.8	378.0
9B	273.0	265.2
35B-A3B	271.2	245.4

F Case Studies

Trajectory compression with deeplinks. Figure 5 compares two successful trajectories from the 4B models on a MobileWorld task: finding a candidate's resume and sending an interview invitation by SMS. The GUI-only baseline takes 26 steps, including contact search, repeated scrolling, and message entry. GUI-Hopper completes the task in five steps. It uses `files_open_downloads` to open the Downloads directory, then opens the resume through the GUI to read the recipient's phone number. It passes the number and message to `sms_compose` and sends the prefilled message with a GUI click. The example illustrates how deeplinks shorten navigation and text entry while GUI actions complete the remaining operations.

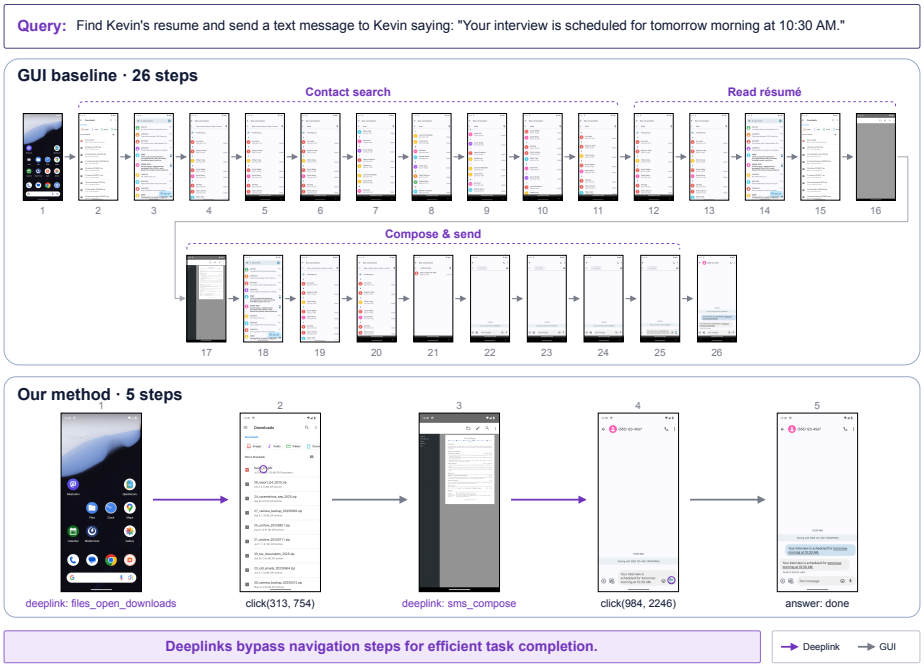

Figure 5: Complete successful trajectories for an interview-invitation task on MobileWorld. GUI-Hopper takes five steps versus 26 for the GUI-only baseline. Direct navigation and SMS prefilling shorten the workflow, while GUI actions remain necessary to read the resume and send the message. Screenshots show pre-action observations, and step counts include the final answer.

Task planning without deeplinks. Figure 6 examines a different source of trajectory compression: choosing where to obtain the information needed for the next operation. Both 35B-A3B models complete the interview-email task without deeplinks, but the GUI-only baseline takes 32 steps compared with 14 for GUI-Hopper. After locating the resume, the baseline searches Mail and Contacts before returning to read it. GUI-Hopper also briefly opens Mail, but returns to the resume earlier to obtain the recipient's email address. In the illustrated stage breakdown, finding the address takes 21 steps for the baseline and two for GUI-Hopper, while composing and sending the email

takes nine steps for each. The main difference is therefore how the models obtain the required information, rather than how they operate the email form. This case illustrates how identifying a useful intermediate goal can reduce unnecessary exploration even when execution relies on GUI actions.

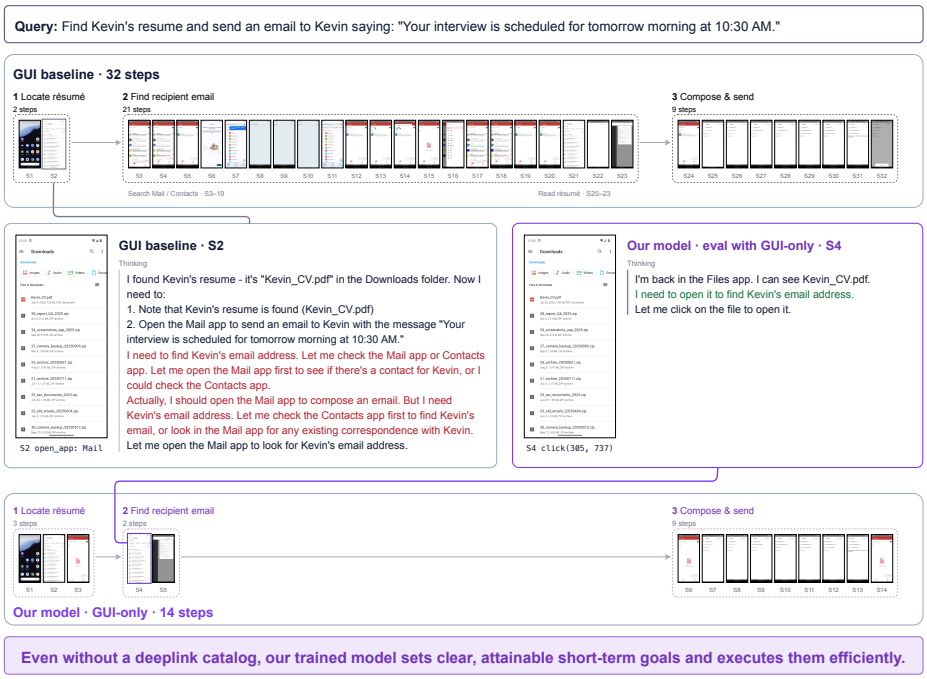

Figure 6: GUI-only trajectories and model reasoning for an interview-email task on MobileWorld. Both 35B-A3B models succeed without deeplinks. GUI-Hopper obtains the recipient's address from the resume earlier, completing the task in 14 steps compared with 32 for the GUI-only baseline. Screenshots show observations before each action, and step counts include the final answer.

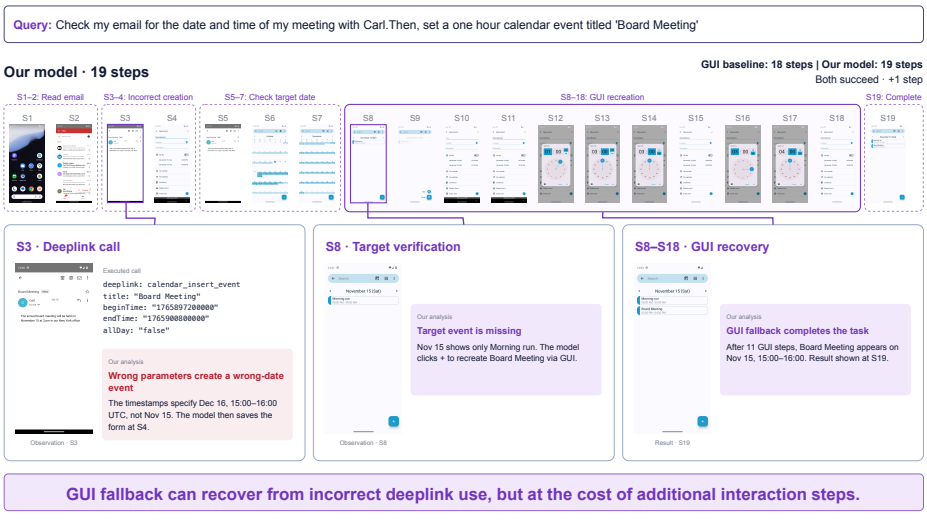

Figure 7: GUI recovery after incorrect deeplink use on a MobileWorld calendar task. The call reaches the expected Activity but prefills the wrong date, showing why the model must also check task-relevant screen contents. GUI-Hopper later recreates the event through GUI actions, taking 19 steps versus 18 for the GUI-only baseline. Removal of the earlier wrong-date event is not shown.

GUI recovery after incorrect deeplink use. Figure 7 illustrates an error that landing verification alone cannot detect. Asked to schedule a meeting on November 15, the 9B GUI-Hopper supplies timestamps for December 16 to `calendar_insert_event`. The call passes landing verification because it reaches the expected Calendar Activity. However, the model overlooks the incorrect date shown in the form and saves the event. When it checks November 15, it notices that the meeting is missing and recreates it through GUI actions. The recreation takes 11 steps within a 19-step trajectory, compared with 18 steps for the successful GUI-only baseline. Reaching the expected screen therefore does not establish that the supplied parameters satisfy the task: the model must also check the screen's task-relevant contents. Here, GUI recovery achieves the requested outcome at an interaction cost, although the trajectory does not show removal of the earlier wrong-date event.