TRACK: AUTONOMOUS EXPLORATION AND TASK-PROGRESS-CONDITIONED KNOWLEDGE UTILIZATION FOR MOBILE GUI AGENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Offline knowledge acquisition and online knowledge utilization pipeline provides application-specific interaction knowledge for long-horizon mobile GUI tasks. However, existing methods do not sufficiently align how knowledge is acquired and reused with the evolving information needs of long-horizon GUI decision-making, often resulting in incomplete guidance or knowledge that is mismatched to the current task progress. Hence, we propose **TRACK**, a GUI Agent framework that constructs a multi-granularity knowledge base through *Offline Exploration and Knowledge Synthesis*, and adaptively injects the knowledge required at each step through *Task-Progress-Conditioned Knowledge Utilization*. During the offline stage, **TRACK** performs traceable and verifiable autonomous branch exploration around the major functionalities of each application. It then synthesizes the resulting exploration branches into complementary task-path knowledge for what to do, element knowledge for where to act, and skill knowledge for how to execute, forming a multi-granularity knowledge base. During online execution, we use the knowledge base to locate the current task progress from historical executions and to route the knowledge applicable to each decision step. By coupling multi-granularity knowledge base with offline autonomous exploration and progress-conditioned online utilization, **TRACK** provides complementary and progress-aligned guidance throughout long-horizon interactions. Experiments on AndroidWorld and DroidTask show that **TRACK** achieves strong task completion performance with marginally higher execution efficiency on Qwen3.5-Plus, while its constructed knowledge base further improves the success rates of integrated agents by up to 11.63% and simultaneously reduces their execution steps. Our project page is available at https://sites.google.com/view/track-gui.

1 INTRODUCTION

GUI agents are increasingly expected to complete long-horizon tasks in real mobile applications, where successful execution require not only general visual-language reasoning (Liu et al., 2026c; Huang et al., 2026), but also application-specific knowledge of functional organization, state-dependent control behavior, and executable operations (Ye et al., 2025; Li et al., 2026a; Xie et al., 2026). Although such knowledge can be internalized through parameter adaptation, repeatedly adapting models to individual applications is costly and makes acquired experience difficult to inspect, update, or transfer. An alternative is to acquire application-specific knowledge offline and inject it during online inference, enabling experience to be independently accumulated and reused (Zhang et al., 2025). Accordingly, we study how to autonomously acquire such knowledge and effectively organize and utilize it for online GUI decision-making.

Existing studies encode application-specific interaction experience into various knowledge forms. The graph/path-oriented approaches model GUI states, elements, actions, and transitions to support structured navigation (Guan et al., 2025; Qin et al., 2026; Chai et al., 2026; Xu et al., 2026). The interface-oriented methods construct application documentation or functional descriptions through exploration (Zhang et al., 2025; Xie et al., 2025a). The trajectory-based methods preserve successful demonstrations or synthesize task supervision from explored behaviors (Sun et al., 2025; Liu et al., 2026b; LYU et al., 2026; Li et al., 2026b). Then, the skill-oriented methods abstract

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

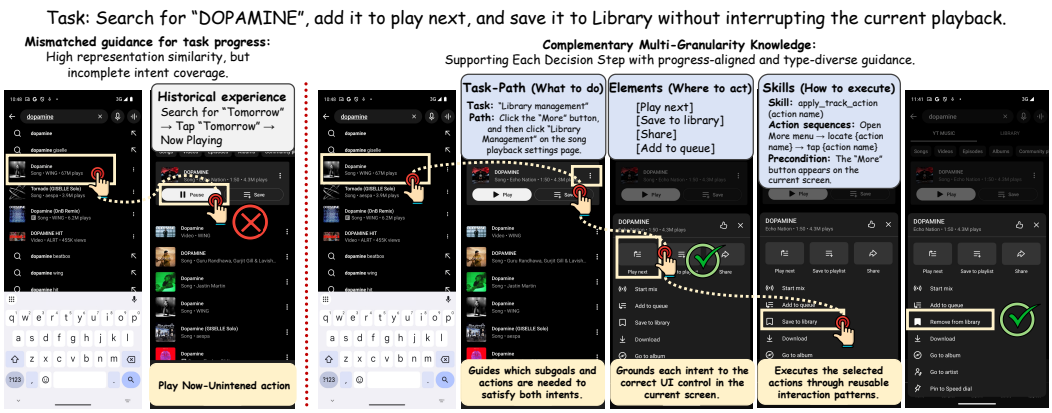


Figure 1: Motivation for aligning knowledge acquisition and utilization with step-level GUI decisions. Relevant experience alone may provide incomplete guidance or suggest an action inappropriate for the current task progress, such as playing "DOPAMINE" immediately instead of playing it next and saving it to the library. Our design acquires complementary task-path, element, and skill knowledge for what to do, where to act, and how to execute, and selectively utilizes it according to the current task progress at each interaction decision step.

recurring action sequences into reusable procedures (Liu et al., 2026a; Xie et al., 2025b). These representations provide useful but different aspects of interaction experience for downstream GUI decision-making. However, existing methods do not sufficiently account for the evolving information needs of long-horizon GUI decision-making when acquiring and reusing knowledge, resulting in incomplete guidance or knowledge that is not aligned with the task progress at each decision step.

To address this limitation, we rethink knowledge acquisition and utilization around the requirements of each decision step. For acquisition, we use functionality-guided autonomous exploration to reliably synthesize complementary multi-granularity knowledge base, reducing incomplete support for what to do, where to act, and how to execute. For utilization, we condition knowledge selection on task progress, preventing relevant but progress-inappropriate knowledge records from misleading the current decision. Figure 1 illustrates how the knowledge base guides sequential decision-making.

Specifically, we propose **TRACK**, consisting of offline exploration and knowledge synthesis, as well as task-progress-conditioned knowledge utilization stages. The offline stage constructs a multi-granularity knowledge base through autonomous app exploration. **TRACK** first extracts the major functionalities of an application from its APK file and uses them as exploration anchors to constrain the exploration scope. Starting from these anchors, it constructs an exploration tree via DFS and multi-agent collaboration. A state-aware backtracking mechanism restores usable states when exploration reaches undesired or terminal states, enabling reliable exploration across functional regions. **TRACK** then synthesizes each valid branch into complementary task-path, element, and skill knowledge. The online stage utilizes this knowledge base conditioned on the task progress. **TRACK** retrieves the relevant task-path, then aligns the current execution step with the path steps to localize task progress. Based on the localized progress, it selects the knowledge type required by the current decision, chooses applicable knowledge records, and injects them into the model context. Therefore, these two stages enable autonomous construction and effective utilization of the knowledge base.

Our contributions are summarized as follows:

- To address the incomplete decision support of existing methods and their limited alignment with each execution progress, we propose **TRACK**, which automatically acquires validated interaction knowledge and organizes it into a multi-granularity knowledge base of task-paths, elements, and skills, providing complementary guidance at each decision step.
- **TRACK** contains offline exploration and knowledge synthesis to autonomously explore application-specific functionalities then construct a multi-granularity knowledge base. In the online stage, it integrates task-progress-conditioned knowledge utilization to adaptively select and inject the knowledge required at each decision step.

- **TRACK** achieves state-of-the-art performance with Qwen3.5-Plus and remains highly competitive with GLM-5.3-Flash. Its multi-granularity knowledge base enables strong performance across diverse applications, and it can be integrated into other agents as a plug-and-play module, providing gains across a variety of settings.

2 Related Work

Knowledge Acquisition for GUI Agents. Existing GUI agents acquire task-specific and environment-specific knowledge mainly through two paradigms. Synthesis-based approaches interact with GUI environments and transform collected experience into reusable knowledge, including application documentation, transition-aware representations, synthesized task trajectories, structured graphs, and procedural skills (Zhang et al., 2025; Xie et al., 2025a; Sun et al., 2025; Liu et al., 2026b; LYU et al., 2026; Qin et al., 2026; Liu et al., 2026a). Other approaches acquire knowledge from demonstrations, external resources, or accumulated execution experience without relying on autonomous environment exploration (Ding et al., 2026; Zhao et al., 2025; Agashe et al., 2025). The resulting knowledge therefore depends on how broadly and reliably useful interaction evidence can be acquired. **TRACK** follows the synthesis-based paradigm but emphasizes functionality-guided autonomous exploration, systematically collecting validated interaction branches and synthesizing them into complementary task-path, element, and skill knowledge.

Retrieval-Augmented GUI Agents. Retrieval augmentation complements the parametric knowledge of GUI agents with task-relevant external or experiential evidence during task execution. Existing methods retrieve navigation paths from structured GUI graphs (Guan et al., 2025; Qin et al., 2026), successful trajectories or episodic memories (Li et al., 2026b; Zeng et al., 2026), procedural skills (Liu et al., 2026a), external Web guidance (Xu et al., 2025), or hierarchical planning and operational knowledge (Zhou et al., 2026). Despite differences in knowledge sources and retrieval mechanisms, these methods mainly focus on retrieving knowledge relevant to the current task or state, while such relevance does not necessarily indicate applicability to the current task progress. **TRACK** instead localizes task progress and selectively routes task-path, element, and skill knowledge compatible with the current decision from multi-granularity knowledge base.

3 Method

TRACK acquires complementary multi-granularity knowledge through autonomous app exploration and selectively uses it according to online task progress. The framework consists of: (i) offline exploration and knowledge synthesis (Section 3.1), (ii) task-progress-conditioned knowledge utilization (Section 3.2), and (iii) multi-granularity knowledge base, as illustrated in Figure 2.

We model a task in application A as a finite-horizon partially observable interaction with latent state s_t, observation $o_t = (I_t, U_t)$, and history $h_t = (o_0, a_0, \ldots, o_{t-1}, a_{t-1})$. Here I_t is the screenshot captured at step t, and U_t is the corresponding parsed UI hierarchy, comprising the accessibility/XML tree and the visible elements with their text or content descriptions, roles, bounds, and interaction attributes. The GUI action space is $\mathcal{A}_{\text{GUI}} = \{\texttt{tap}, \texttt{text}, \texttt{long_press}, \texttt{swipe}\}$ together with `complete`, and action a_t induces $s_{t+1} \sim P_A(\cdot \mid s_t, a_t)$. Offline exploration converts validated interaction branches into frozen task-path, element, and skill knowledge for subsequent online decision-making, while online execution augments the action policy with knowledge selected according to task description g, h_t, o_t and the localized task progress.

3.1 Offline Exploration and Knowledge Synthesis

The offline stage aims to obtain validated interaction branches from which complementary decision knowledge can be synthesized. We use APK-level functional entry points as anchors to constrain exploration to major application functions. Within each anchored region, DFS expands alternative interaction branches, while state-aware backtracking restores a previous validated state when a branch terminates or deviates so that unexplored alternatives can continue to be expanded. At each node, the Planner proposes the next GUI action under the current exploration objective, the Reflector

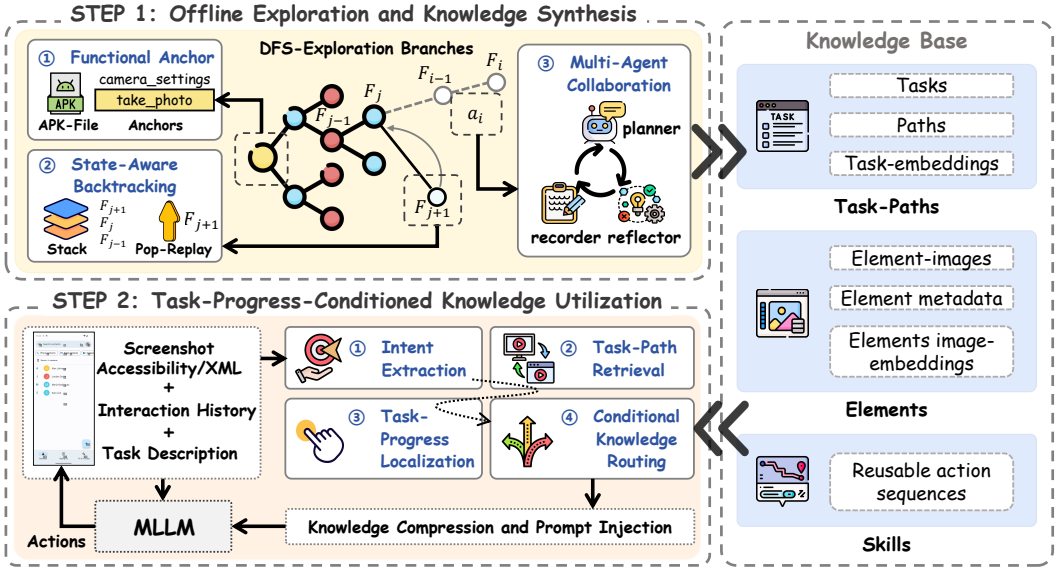


Figure 2: Overview of **TRACK**. It consists of offline exploration and knowledge synthesis, and task-progress-conditioned knowledge utilization. Offline, **TRACK** uses major app functions as exploration anchors, expands interaction branches with DFS and backtracking, and coordinates multiple agents to predict and validate actions for successive node expansion. The validated branches are then synthesized into three complementary knowledge types, namely task-paths, elements, and skills, and are further constructed into a multi-granularity knowledge base. Online, **TRACK** localizes task progress and selectively injects the knowledge records applicable to the current decision step.

evaluates the resulting transition, and the Recorder stores validated transitions and their UI evidence:

$$\begin{aligned} u_k &= \mathrm{pop}(\mathcal{Q}_A), \\ \mathcal{B}_k &= \text{DFS-Backtrack}_A(u_k, \pi_{\text{Planner}}, \pi_{\text{Reflector}}), \\ \mathcal{R}_k &= \pi_{\text{Recorder}}(\mathcal{B}_k) = \{(s_{k,j}, a_{k,j}, s_{k,j+1}, r_{k,j}, e_{k,j})\}_{j \in \mathcal{J}_k}. \end{aligned} \tag{1}$$

Here $\mathcal{Q}_A$ is the queue of uncovered functional anchors, u_k is the anchor selected for expansion, $\mathcal{B}_k$ is the DFS branch grown from that anchor, and $\mathcal{J}_k$ indexes its retained transitions. Each tuple in $\mathcal{R}_k$ comprises source state $s_{k,j}$, executed GUI action $a_{k,j}$, resulting state $s_{k,j+1}$, Reflector outcome $r_{k,j}$, and Recorder evidence $e_{k,j}$. Specifically, $r_{k,j} \in \{\texttt{successful}, \texttt{ineffective}, \texttt{unexpected}\}$ is a semantic transition label rather than an environment reward, and only `successful` transitions are retained in $\mathcal{R}_k$. The evidence $e_{k,j}$ contains the source and result UI observations, the grounded target-element attributes, and the functionality description that supports later synthesis. More detailed exploration of this process is provided in the Appendix A.1.

To ensure reliable DFS exploration under non-invertible Android navigation, we introduce a state-aware backtracking mechanism. Rather than treating `Back` as a deterministic inverse action, the controller matches the current UI against stored stack states and replays validated interactions when the intended parent state is not directly recovered. Each replayed transition is verified before exploration resumes. This combination of state matching and verified replay enables reliable sibling-branch exploration and prevents incorrect recovery states from contaminating subsequent knowledge construction. Detailed implementation is provided in the Appendix A.2.

Rather than preserving each branch as a single trajectory representation, **TRACK** jointly synthesizes three complementary knowledge types from the same validated provenance. These types capture complementary decision requirements by providing task-level guidance on what to do, element-level grounding on where to act, and skill-level procedures on how to execute:

$$(p_b, E_b, S_b) = \Phi(\mathcal{R}_b, q_b) = \Big(\text{RevSynth}(\mathcal{R}_b, q_b), \text{ElemExtract}(\mathcal{R}_b), \text{SkillExtract}(\mathcal{R}_b)\Big). \tag{2}$$

Here q_b is the outcome-oriented objective of branch b, and $p_b = (g_b, (d_{b,1}, \ldots, d_{b,L_b}))$ is the synthesized task-path: g_b is its task goal, $d_{b,i}$ is its i-th semantic step, and L_b is the path length. E_b

contains element records that associate validated operations with their supporting UI controls, and S_b contains reusable action sequences with preconditions and expected effects. Repeated records are normalized, confidence-filtered, merged, and deduplicated.

3.2 TASK-PROGRESS-CONDITIONED KNOWLEDGE UTILIZATION

The synthesized records are then stored in the multi-granularity knowledge base namespace $\nu(A)$ resolved for application A:

$$\mathcal{K}_{\nu(A)} = \big(\mathcal{K}^{\text{task-path}}_{\nu(A)}, \mathcal{K}^{\text{elem}}_{\nu(A)}, \mathcal{K}^{\text{skill}}_{\nu(A)}\big). \quad (3)$$

For all applications, the knowledge base namespace mapping rules described in Appendix A.4 are applied to map each application into our frozen knowledge base. For a single knowledge record of each app: The $\mathcal{K}^{\text{task-path}}_{\nu(A)}$ stores task goals, ordered semantic paths, and so on. The $\mathcal{K}^{\text{elem}}_{\nu(A)}$ stores UI attributes, functionality descriptions, visual evidence, and provenance, and so on. The $\mathcal{K}^{\text{skill}}_{\nu(A)}$ stores reusable procedures with preconditions, parameters, action sequences, expected effects, provenance, and so on.

Algorithm 1: Task-Progress-Conditioned Knowledge Utilization

Require: Task description g, initial observation o_0, **TRACK** knowledge base $\mathcal{K}_{\nu(A)}$
Ensure: Online action sequences $(a_0, a_1, \ldots)$

$h_0 \leftarrow \varnothing$
for $t = 0, 1, \ldots, H-1$ **do**
  $\mathcal{C}^{\text{path}}_t \leftarrow \text{Retrieve}_{\text{path}}(g, \mathcal{K}^{\text{task-path}}_{\nu(A)})$
  $\widehat{d}_{t+1} \leftarrow \text{ProgressLocalize}(\mathcal{C}^{\text{path}}_t, h_t, o_t)$
  $\mathcal{C}^{\text{elem}}_t \leftarrow \text{Retrieve}_{\text{elem}}(\widehat{d}_{t+1}, o_t)$
  $\mathcal{C}^{\text{skill}}_t \leftarrow \text{Retrieve}_{\text{skill}}(\widehat{d}_{t+1}, o_t)$
  $\mathcal{C}^{\text{use}}_t \leftarrow \text{Route}(\widehat{d}_{t+1}, \mathcal{C}^{\text{path}}_t, \mathcal{C}^{\text{elem}}_t, \mathcal{C}^{\text{skill}}_t)$
  $a_t \leftarrow \pi_{\text{Online}}(g, h_t, o_t, \widehat{d}_{t+1}, \mathcal{C}^{\text{use}}_t)$
  if $a_t =$ `complete` **then**
    return $(a_0, a_1, \ldots, a_t)$
  end if
  Execute a_t and observe o_{t+1}
  $h_{t+1} \leftarrow (h_t, o_t, a_t)$
end for
return $(a_0, a_1, \ldots)$

At online step t, **TRACK** first converts the task description into $z_g = \text{NormIntent}(g) = (\iota_g, \tau_g, \kappa_g)$, a normalized structured intent representation for downstream matching comprising an intent label ι_g, target entities τ_g, and explicit task constraints κ_g. It retrieves a ranked pool of task-path candidates from the application-resolved knowledge base using task-intent/task-goal similarity. Candidates below the retrieval threshold or from a mismatched application namespace are discarded. Thus, the online stage does not commit to the first nearest path: it retains multiple plausible candidates for state-aware comparison. For a candidate $p = (g_p, (d_1, \ldots, d_{L_p}))$, g_p is the stored task goal, d_i is its i-th semantic step, and L_p is its number of steps.

We jointly select a path and localize its progress. For each candidate, a valid alignment position is one whose preceding steps are supported as completed by the history and the current UI while the current position remains unresolved. The terminal position is considered only when the task-level goal predicate is verified. Each valid path-position pair is assigned a joint compatibility score.

$$\begin{aligned} J_t(p,\ell) &= \alpha r^{\text{ret}}_t(p) + \beta s^{\text{prefix}}_t(p,\ell) + \gamma s^{\text{ui}}_t(d_\ell, o_t) + \eta s^{\text{suffix}}_t(p,\ell) - \zeta c^{\text{conflict}}_t(p,\ell), \\ (\widehat{p}_t, \widehat{\ell}_t) &= \underset{p \in \mathcal{P}_t,\, \ell \in \mathcal{I}_t(p)}{\arg\max} \; J_t(p,\ell). \end{aligned} \quad (4)$$

Here $\mathcal{P}_t$ is the filtered candidate pool, $\mathcal{I}_t(p)$ is the set of valid alignment positions for candidate p, r^{ret}_t is the retrieval score, s^{prefix}_t measures agreement between the candidate's completed prefix and h_t, s^{ui}_t measures whether the current UI supports the next semantic step, s^{suffix}_t measures consistency of the remaining suffix with the current task constraints, and c^{conflict}_t penalizes contradictions with observed history, UI state, or verified task outcomes. The coefficients $\alpha, \beta, \gamma, \eta, \zeta$ are fixed nonnegative weights. This objective makes candidate-path ranking and progress localization one decision: a path with high goal similarity but an incompatible prefix or next-step UI is rejected in favor of a lower-ranked path with stronger state alignment.

The selected position yields the next unresolved semantic step on $\widehat{p}_t$. If the selected position is terminal, **TRACK** emits `complete` only after the task-level goal predicate is verified. If no candidate has a valid alignment or the best pair falls below the localization-confidence threshold during online execution, **TRACK** returns a re-retrieval/re-planning signal and does not inject a stale path. Implementation details and thresholds are provided in Appendix A.3.

3.2.1 CONDITIONAL KNOWLEDGE ROUTING

TRACK selects knowledge according to task progress and the current UI instead of fixing the online decision to one representation. Let $\mathcal{C}_t^m$ denote candidate records of type $m \in \mathcal{N} = \{\text{path}, \text{element}, \text{skill}\}$ and χ_m indicate whether type m is compatible with the current step:

$$\begin{aligned} \mathcal{M}_t &= \Big\{ m \in \mathcal{N} : \chi_m(g, h_t, o_t, \widehat{d}_{t+1}, \mathcal{C}_t^m) = 1 \Big\}, \\ \mathcal{C}_t^{\text{use}} &= \bigcup_{m \in \mathcal{M}_t} \text{Select}_m(\mathcal{C}_t^m \mid g, h_t, o_t, \widehat{d}_{t+1}). \end{aligned} \tag{5}$$

The task-path knowledge is selected when its semantic guidance agrees with the localized progress, the element knowledge is selected when the current step requires UI-specific grounding, and the skill knowledge is selected when reusable action sequences match the localized objective and their predefined conditions. Outdated or incompatible records are excluded, and **TRACK** proceeds without external knowledge when no knowledge type is applicable.

3.2.2 KNOWLEDGE-GUIDED ACTION DECISION

The online agent predicts the next GUI action using the task description, interaction history, current observation, localized semantic step, and selected knowledge:

$$a_t = \pi_{\text{Online}}\Big(\text{Prompt}[g, h_t, o_t, \widehat{d}_{t+1}, \mathcal{C}_t^{\text{use}}]\Big), \qquad a_t \in \mathcal{A}_{\text{GUI}} \cup \{\texttt{complete}\}. \tag{6}$$

After execution, the agent evaluates whether the transition advances the task description, then appends the resulting action and evidence to h_{t+1}. Online interaction updates only the execution history, while the application-specific knowledge base remains frozen. Algorithm 1 illustrates the process of online task-progress-conditioned knowledge utilization.

4 EXPERIMENTS

4.1 EXPERIMENTS SETTINGS

Benchmarks and models. We evaluate **TRACK** on task suites drawn from two open-source mobile GUI benchmarks: AndroidWorld (Rawles et al., 2025) and DroidTask (Wen et al., 2024). The resulting evaluation suite spans five functional categories, 14 applications, and 129 tasks, covering diverse everyday mobile interaction scenarios. To focus on application-specific interaction knowledge, we retain single-application tasks from the selected applications and exclude tasks that require cross-application interactions. All evaluated tasks follow the original benchmark definitions, initialization, reset procedures, and evaluators. We use Qwen3.5-Plus and GLM-5.3-Flash as the online decision models to evaluate the effectiveness and model generality of **TRACK**. Detailed application categories and task distributions are provided in Appendix A.6.

Baselines and variations. We organize baselines by the way they obtain and use interaction knowledge: (i) *Non Offline-Online methods*: Mobile-Agent-v2 (Wang et al., 2024) and AndroidWorld (Rawles et al., 2025), (ii) *Offline-Online methods*: AutoDroid (Wen et al., 2024), AppAgent (Zhang et al., 2025), KG-RAG (Guan et al., 2025), and GUI-explorer (Xie et al., 2025a), and (iii) *Online-Exploration method*: MobileUse (Li et al., 2026a). In knowledge-plugin ablation, our knowledge base TRACK-KB serves as a plug-and-play external retrieval module: it injects relevant knowledge into the model's context through simple similarity matching without replacing the host architecture. Thus, AndroidWorld (T3A) versus AndroidWorld+TRACK-KB isolates the effect of injecting the external knowledge on AndroidWorld. On DroidTask, we compare AutoDroid-No-KB, native AutoDroid, and AutoDroid+TRACK-KB under the same settings. In the knowledge usage ablation, we evaluate the impact of **TRACK**'s knowledge injection methods and knowledge types.

4.2 MAIN RESULTS

Effectiveness on AndroidWorld. As shown in Table 1, **TRACK** achieves the highest SR with both Qwen3.5-Plus and GLM-5.3-Flash, reaching 81.40% and 83.72%, respectively, while maintaining the second-lowest AS with both models. Compared with GUI-explorer, the strongest SR baseline,

Table 1: Success Rates (SR, %) and Average Steps (AS) of methods with two online decision models on AndroidWorld and DroidTask benchmarks for mobile GUI interaction tasks. Bold and underlined entries denote the best and second-best values within each column, respectively.

Methods	AndroidWorld				DroidTask			
	Qwen3.5-Plus		GLM-5.3-Flash		Qwen3.5-Plus		GLM-5.3-Flash	
	SR↑	AS↓	SR↑	AS↓	SR↑	AS↓	SR↑	AS↓
Non Offline-Online								
Mobile-Agent-v2 (Wang et al., 2024)	46.51	**8.35**	51.16	**8.21**	86.05	5.30	88.37	8.45
AndroidWorld (Rawles et al., 2025)	65.12	10.25	58.14	10.04	86.05	6.08	87.21	7.89
Offline-Online								
AutoDroid (Wen et al., 2024)	32.55	13.56	30.23	15.28	<u>97.67</u>	4.14	**98.84**	4.19
AppAgent (Zhang et al., 2025)	48.84	14.23	65.12	13.74	94.19	5.51	94.19	5.95
KG-RAG (Guan et al., 2025)	58.14	13.47	46.51	13.40	82.56	<u>4.12</u>	84.88	**4.05**
GUI-explorer (Xie et al., 2025a)	<u>79.07</u>	9.88	<u>76.74</u>	10.56	96.51	**3.99**	91.86	<u>4.10</u>
Online-Explore								
MobileUse (Li et al., 2026a)	67.44	12.91	69.77	14.33	93.02	6.72	94.19	6.55
TRACK (Ours)	**81.40**	<u>9.49</u>	**83.72**	<u>9.19</u>	**98.84**	9.44	<u>95.35</u>	8.00

Table 2: Results of TRACK-KB as a plug-and-play module on other agent frameworks.

Variations	Qwen3.5-Plus		GLM-5.3-Flash	
	SR↑	AS↓	SR↑	AS↓
AndroidWorld	65.12	10.25	58.14	10.04
+TRACK-KB	**69.77**	**9.97**	**69.77**	**9.17**
AutoDroid	97.67	**4.14**	**98.84**	**4.19**
-No-KB	95.35	4.70	96.51	3.83
+TRACK-KB	**98.84**	4.75	94.19	6.78

Table 3: Ablation results of knowledge utilization and knowledge types on AndroidWorld.

Variations	Qwen3.5-Plus		GLM-5.3-Flash	
	SR↑	AS↓	SR↑	AS↓
TRACK	**81.40**	9.49	**83.72**	9.19
TRACK-No-KB	79.07	**8.53**	65.12	**7.75**
TRACK-Fixed-KB	67.44	10.28	67.44	9.48
w/o Elements	76.74	8.64	79.07	9.47
w/o Skills	62.79	9.93	67.44	8.52
w/o Path-and-Localization	74.42	8.61	74.42	9.38

TRACK further improves task completion while requiring fewer interaction steps. These joint gains show that our method improves decision reliability without relying on longer execution trajectories. The consistent advantage across different online decision models further indicates that the effectiveness of **TRACK**'s knowledge acquisition and utilization pipeline is not tied to a particular LLM.

Cross-benchmark generalization. Table 1 also evaluates **TRACK** on DroidTask, where it achieves the highest SR with Qwen3.5-Plus (98.84%) and the second-best SR with GLM-5.3-Flash (95.35%). Several methods reach SRs above 90.00% on this benchmark. One factor is DroidTask's evaluation protocol, in which task completion depends on self-judgment by the agent. Consequently, the measured SR is influenced by how each agent determines when the task has been completed, leading to near-saturated results for several methods. Despite this saturation, **TRACK** remains competitive across both online decision models, although its higher AS indicates that its advantage on DroidTask is primarily reflected in reliable task completion rather than execution efficiency. Together with the AndroidWorld results, these findings show that acquired complementary knowledge and progress-conditioned utilization can consistently support task completion across different benchmarks and online decision models, while the efficiency gains are more pronounced on AndroidWorld.

4.3 Ablation Studies

We conduct ablations to examine two key questions: whether the TRACK-KB constructed by **TRACK** during the offline process improves task completion, and whether its online utilization design is necessary to exploit this knowledge effectively. Table 2 evaluates the quality of TRACK-KB by inserting it or replacing the knowledge base of existing agents. Table 3 further examines knowledge utilization strategy and the contribution of different knowledge granularities to performance.

Transferability of TRACK-KB. Table 2 shows that TRACK-KB provides decision support for other agents beyond the complete **TRACK** framework. After being integrated into the AndroidWorld framework, it consistently improves SR on both models, with a clear gain of 11.63% on GLM-5.3-Flash. Replacing AutoDroid's original knowledge base with TRACK-KB improves Qwen3.5-Plus SR. These results indicate that the multi-granularity knowledge base produced by our offline process transfers across frameworks and, in some cases, reduces AS. However, the decrease observed for AutoDroid with GLM-5.3-Flash also shows that high-quality knowledge alone does not guarantee better decisions. How the knowledge is matched to the current decision state remains important.

Knowledge utilization beyond knowledge availability. Table 3 shows that **TRACK**'s gains depend not only on access to TRACK-KB, but also on how this knowledge is used during decision-making. Removing the KB leads to a drop in SR, indicating that external knowledge provides important guidance. TRACK-Fixed-KB with a simple similarity-matching retrieval strategy performs below TRACK-No-KB on Qwen-3.5-Plus, suggesting that such retrieved knowledge can mislead decisions at certain steps. By localizing task progress and injecting knowledge according to each decision state, **TRACK** enables TRACK-KB to produce more targeted guidance.

Complementarity across knowledge granularities. Removing any knowledge component reduces SR on both models, with skill removal causing the largest drop. This indicates that the three knowledge types are complementary rather than merely additive, as incomplete integration can misguide decisions. Task-path knowledge determines what to do next, element knowledge identifies where to act, and skill knowledge specifies how to execute. Together, they support task completion. Overall, the ablations validate both framework stages: offline synthesis constructs complementary multi-granularity knowledge, while online utilization effectively provides step-specific guidance.

4.4 Knowledge-Hit Effectiveness Analysis

Multiple factors shape knowledge retrieval. As shown in Figure 3, task-path retrieval dominates, followed by skill and element knowledge on AndroidWorld, whereas DroidTask shows the same dominance but with element retrieval exceeding skill retrieval. Across both benchmarks, task-level retrieval hit rates remain stable across models, while step-level coverage varies more substantially, reflecting model-dependent knowledge selection. The relationship between task-level and step-level metrics also depends on knowledge granularity: task-paths often span multiple steps, whereas elements and skills are required only at specific decision points. Overall, progress-conditioned routing filters out retrieved knowledge misaligned with task progress, preventing unnecessary injection.

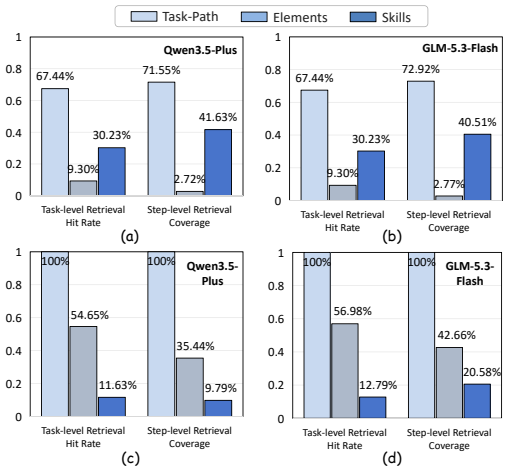


Figure 3: Knowledge retrieval analysis on AndroidWorld (a, b) and DroidTask (c, d) benchmarks. Task-level retrieval hit rate and step-level retrieval hit rate refer to the proportions of certain types of knowledge that are retrieved at the task level and at the step level, respectively.

Retrieval does not imply injection need. As shown in Tables 4 and 5, injected knowledge on AndroidWorld is dominated by skills, followed by task-paths and elements. Compared with the retrieval results in Figure 3, both THR and SIC decrease substantially after progress-conditioned routing, especially with task-path knowledge showing the largest reduction. This suggests that semantically relevant knowledge is not necessarily needed at the current step. In terms of RIC, element knowledge achieves the highest retention rate, although patterns vary across models. Overall, progress-conditioned routing filters out retrieved knowledge that is similar but misaligned with task progress, preventing unnecessary information from being injected into the model.

Table 4: Actual knowledge injections on AndroidWorld with Qwen3.5-Plus. THR denotes task-level knowledge hit rate, SIC denotes step-level knowledge injection coverage, and RIC denotes retrieval-to-injection conversion.

Knowledge	THR (%)	SIC (%)	RIC (%)
Task-Paths	4.65	2.30	3.22
Elements	4.65	0.42	26.09
Skills	23.26	34.31	16.48

Table 5: Actual knowledge injections on AndroidWorld with GLM-5.3-Flash. THR denotes task-level knowledge hit rate, SIC denotes step-level knowledge injection coverage, and RIC denotes retrieval-to-injection conversion.

Knowledge	THR (%)	SIC (%)	RIC (%)
Task-Paths	4.65	2.13	2.92
Elements	4.65	1.49	47.73
Skills	23.26	34.54	17.05

4.5 Failure Analysis

Failure taxonomy and reduction of decision-stage failures. We categorize failed tasks by the stage at which execution deviates from the objective, with error types assigned by an LLM judge. Completion verification failure occurs when execution terminates before all required subgoals are completed. Precondition mismatch occurs when an action is selected before its prerequisite state has been established. Progress failure means that the agent follows an incorrect semantic subgoal or deviates from the intended task path. Action execution failure covers errors in UI grounding, action execution, or subsequent recovery. As shown in Figure 4, **TRACK** reduces the proportions of the first three failure categories, while action execution failures remain unchanged. These reductions show that **TRACK** effectively improves task progression and state-dependent decision-making. Task-path knowledge supports progress and completion judgment, while progress-conditioned utilization, together with element and skill knowledge, provides state-compatible action guidance.

Analysis of remaining failures. The remaining failures consist of four progress failures and three action execution failures. The unchanged number of action execution failures suggests that correction based solely on the model's feedback from previous failures remains limited, and that knowledge utilization alone often cannot fully resolve these low-level errors in UI grounding and action execution. The remaining progress failures indicate that knowledge routing depends on accurate estimation of task progress. When the progress state is incorrectly identified, the agent may retrieve knowledge that is locally plausible but inconsistent with the actual task state. These observations suggest that future improvements should focus on more robust progress correction during online execution, state-divergence detection, and recovery-aware replanning.

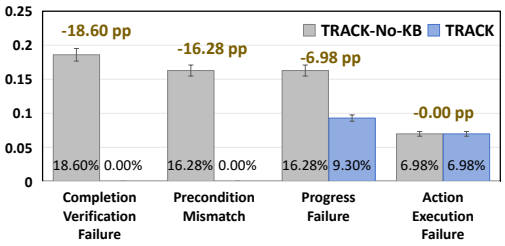


Figure 4: Task-level error statistics and distribution on AndroidWorld with GLM-5.3-Flash.

5 Conclusion

In this work, we propose **TRACK** for automatically acquiring and effectively utilizing application-specific knowledge in long-horizon GUI interactions. Its offline exploration is organized around major application functionalities and reliably collects validated interaction branches, which are synthesized into complementary multi-granularity knowledge base. During online, **TRACK** localizes task progress and selectively injects knowledge aligned to each execution step. Experiments show that **TRACK** achieves state-of-the-art performance with Qwen3.5-Plus and remains competitive with GLM-5.3-Flash. Its multi-granularity knowledge base can be injected into other agents as a plug-and-play component, improving task completion and reducing execution steps in several settings.

Limitation. **TRACK** freezes the constructed multi-granularity knowledge base during online execution, preventing it from continually absorbing newly encountered interaction experience. Addressing this limitation requires developing continually evolving knowledge bases, as well as extending knowledge acquisition and utilization from single-app settings to cross-app tasks.

AI USE STATEMENT

In this work, we used generative AI tools to improve the presentation of the manuscript, including revising sentence structure, logical flow, grammar, and overall readability. All AI-assisted suggestions were carefully reviewed and evaluated by the authors before being incorporated into the work. The authors also thoroughly manually reviewed the entire final manuscript to ensure the correctness of its technical content and claims. We ultimately take responsibility for the final content of this work, including all text and claims produced with the aid of generative AI.

ETHICS STATEMENT

This work studies an autonomous GUI agent that explores mobile applications and executes actions through the GUI. All experiments are conducted in emulator-based, controlled benchmark environments. The agents are not deployed on real devices or in real-world settings. Our experiments do not collect any private user data. We use benchmark-provided applications, tasks, and evaluation mechanisms solely for academic research purposes. Accordingly, the proposed method is intended to improve knowledge acquisition and decision-making for benign mobile automation rather than to bypass application permissions or security mechanisms.

REPRODUCIBILITY STATEMENT

We provide the information required to reproduce our method and experiments throughout the paper and the supplementary material. The methodology section describes the offline exploration and knowledge construction procedure, the representations of task-path, element, and skill knowledge, and the task-progress-conditioned knowledge utilization mechanism. The experimental section specifies the evaluation benchmarks, online models, baselines, evaluation metrics, and so on. The appendix provides additional implementation details, including the exploration and online execution pipelines, extended experimental settings, the app distribution of the selected benchmarks, parameter analysis, knowledge-hit effectiveness analysis, prompt templates, and so on. We evaluate on reproducible Android benchmark environments, including AndroidWorld and DroidTask, and use their provided task initialization and evaluation mechanisms whenever applicable. A fully anonymized code implementation is provided in the supplementary material.

REFERENCES

Saaket Agashe, Jiuzhou Han, Shuyu Gan, Jiachen Yang, Ang Li, and Xin Wang. Agent s: An open agentic framework that uses computers like a human. In *International Conference on Learning Representations*, volume 2025, pp. 22924–22946, 2025.

Yuxiang Chai, Han Xiao, Xinyu Fu, Jinpeng Chen, Rui Liu, and Hongsheng Li. Ui-kobe: Knowledge-oriented behavior exploration for lightweight graph-guided gui agents. *arXiv preprint arXiv:2605.29534*, 2026.

Zihan Ding, Longxu Dou, Qi Gao, Xiangwu Guo, Shengchao Hu, Zilong Huang, Zihang Jiang, Lei Ke, Mengcheng Lan, Weixian Lei, et al. Ui-mate: Advancing open-weight foundation gui agents with in-context demonstrations. *arXiv preprint arXiv:2608.15930*, 2026.

Ziyi Guan, Jason Chun Lok Li, Zhijian Hou, Pingping Zhang, Donglai Xu, Yuzhi Zhao, Mengyang Wu, Jinpeng Chen, Thanh-Toan Nguyen, Pengfei Xian, et al. Kg-rag: enhancing gui agent decision-making via knowledge graph-driven retrieval-augmented generation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 5396–5405, 2025.

Chi-Pin Huang, Yueh-Hua Wu, Min-Hung Chen, Frank Wang, and Fu-En Yang. Thinkact: Vision-language-action reasoning via reinforced visual latent planning. *Advances in Neural Information Processing Systems*, 38:82782–82802, 2026.

Ning Li, Xiangmou Qu, Jiamu Zhou, Muning Wen, Kounianhua Du, Xingyu Lou, Qiuying Peng, Jun Wang, and Weinan Zhang. Mobileuse: A hierarchical reflection-driven gui agent for autonomous mobile operation. *Advances in Neural Information Processing Systems*, 38:40361–40388, 2026a.

Runze Li, Yuwen Zhai, Bo Xu, LiWu Xu, Nian Shi, Wei Zhang, Ran Lin, and Liang Wang. Echotrail-gui: Building actionable memory for gui agents via critic-guided self-exploration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 9347–9356, 2026b.

Dawei Liu, Zongxia Li, Hongyang Du, Xiyang Wu, Shihang Gui, Yongbei Kuang, and Lichao Sun. Graph-of-skills: Dependency-aware structural retrieval for massive agent skills. *arXiv preprint arXiv:2604.05333*, 2026a.

Guangyi Liu, Pengxiang Zhao, Liang Liu, Zhiming Chen, Yuxiang Chai, Yaozhen Liang, Wenhao Wang, Siheng Chen, Zhengxi Lu, Shuai Ren, et al. Learnact: Few-shot mobile gui agent with a unified demonstration benchmark. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 29820–29843, 2026b.

Yuhang Liu, Pengxiang Li, Zishu Wei, Congkai Xie, Xueyu Hu, Xinchen Xu, Shengyu Zhang, Xiaotian Han, Hongxia Yang, and Fei Wu. Infiguiagent: A multimodal generalist gui agent with native reasoning and reflection. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1035–1051, 2026c.

Rui LYU, Juncheng Mo, Tianyi Chu, Chen Rao, Hongyi Jing, Jiajie Teng, Jiafu Chen, Shiqi Zhang, Liangzi Ding, Shuo Fang, et al. M ̈2-miner: Multi-agent enhanced mcts for mobile gui agent data mining. In *International Conference on Learning Representations*, volume 2026, pp. 5827–5855, 2026.

Zerui Qin, Sheng Yue, Xingyuan Hua, Yongjian Fu, and Ju Ren. Executable agentic memory for gui agent. *arXiv preprint arXiv:2605.12294*, 2026.

Chris Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, et al. Androidworld: A dynamic benchmarking environment for autonomous agents. In *International Conference on Learning Representations*, volume 2025, pp. 406–441, 2025.

Qiushi Sun, Kanzhi Cheng, Zichen Ding, Chuanyang Jin, Yian Wang, Fangzhi Xu, Zhenyu Wu, Chengyou Jia, Liheng Chen, Zhoumianze Liu, et al. Os-genesis: Automating gui agent trajectory construction via reverse task synthesis. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5555–5579, 2025.

Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. Mobile-agent-v2: Mobile device operation assistant with effective navigation via multi-agent collaboration. *Advances in Neural Information Processing Systems*, 37:2686–2710, 2024.

Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. Autodroid: Llm-powered task automation in android. In *Proceedings of the 30th annual international conference on Mobile computing and networking*, pp. 543–557, 2024.

Bin Xie, Rui Shao, Gongwei Chen, Kaiwen Zhou, Yinchuan Li, Jie Liu, Min Zhang, and Liqiang Nie. Gui-explorer: Autonomous exploration and mining of transition-aware knowledge for gui agent. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5650–5667, 2025a.

Yiping Xie, Song Chen, Jingxuan Xing, Wei Jiang, Zekun Zhu, Yingyao Wang, Pi Bu, Jun Song, Yuning Jiang, and Bo Zheng. Secagent: Efficient mobile gui agent with semantic context. *arXiv preprint arXiv:2603.08533*, 2026.

Yuquan Xie, Zaijing Li, Rui Shao, Gongwei Chen, Kaiwen Zhou, Yinchuan Li, Dongmei Jiang, and Liqiang Nie. Mirage-1: Augmenting and updating gui agent with hierarchical multimodal skills. *arXiv preprint arXiv:2506.10387*, 2025b.

Ran Xu, Kaixin Ma, Wenhao Yu, Hongming Zhang, Joyce C Ho, Carl Yang, and Dong Yu. Retrieval-augmented gui agents with generative guidelines. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 17877–17886, 2025.

Yifan Xu, Xiao Liu, Xinghan Liu, Jiaqi Fu, Jiayu Huang, Hanchen Zhang, Bohao Jing, Shudan Zhang, Yuting Wang, Yuxiao Dong, et al. Mobilerl: Online agentic reinforcement learning for mobile gui agents. In *International Conference on Learning Representations*, volume 2026, pp. 35282–35315, 2026.

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

Ziyun Zeng, Hang Hua, Bocheng Zou, Mu Cai, Rogerio Feris, and Jiebo Luo. Mementogui: Learning agentic multimodal memory control for long-horizon gui agents. *arXiv preprint arXiv:2605.18652*, 2026.

Chi Zhang, Zhao Yang, Jiaxuan Liu, Yanda Li, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. Appagent: Multimodal agents as smartphone users. In *Proceedings of the 2025 CHI conference on human factors in computing systems*, pp. 1–20, 2025.

Yuyang Zhao, Wentao Shi, Fuli Feng, and Xiangnan He. Appagent-pro: A proactive gui agent system for multidomain information integration and user assistance. In *Proceedings of the 34th ACM International Conference on Information and Knowledge Management*, pp. 6767–6771, 2025.

Yuxiang Zhou, Jichang Li, Yanhao Zhang, Haonan Lu, and Guanbin Li. Mobile-agent-rag: driving smart multi-agent coordination with contextual knowledge empowerment for long-horizon mobile automation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 29939–29947, 2026.

A APPENDIX

A.1 ANCHOR-GUIDED ROBUST BRANCH EXPLORATION

Acquiring functions and scheduling anchors. Before execution, **TRACK** loads an application-specific functional specification from an APK-level application inventory automatically extracted and structured for each app. The inventory is produced by the APK analysis pipeline from static artifacts (e.g., the manifest, resources, and DEX), without manual annotation intervention in APK processing or feature extraction. Any unresolved entry cues and functional coverage gaps are handled by the subsequent autonomous exploration process, which operates on this inventory rather than relying on additional human input. The specification represents a major app capability as a function $f \in \mathcal{F}_A$ and associates it with four elements: (i) an ordered `exploration_feature_queue` that prioritizes sub-features to explore within the function, (ii) an anchor record containing a root-entry cue and a coverage label, (iii) root-entry mappings that link the function to statically identifiable UI controls (e.g., by resource-id, text, or content description), which are resolved against the live UI hierarchy at runtime, and (iv) intent-feature groups used for coverage accounting.

Thus, in other words, an anchor is not a pre-defined benchmark task or a fixed interaction script: it is essentially a function-level exploration root, such as changing a camera setting or managing a conversation. At runtime, **TRACK** compares this inventory with the already saved task catalog and the anchors exhausted in the current session, and initializes

$$\mathcal{Q}_A = \big[f \in \mathcal{F}_A : \neg\,\mathrm{Covered}(f, \mathcal{K}_A) \;\wedge\; f \notin \mathcal{X}_A\big], \qquad q_k = \mathrm{Objective}(u_k), \quad u_k = \mathrm{pop}(\mathcal{Q}_A). \tag{7}$$

Here $\mathcal{F}_A$ is the ordered set of app functions, $\mathcal{K}_A$ is the current saved catalog, $\mathcal{X}_A$ contains session-exhausted anchors, $\mathrm{Covered}(\cdot)$ typically tests whether a validated record already represents a function, $\mathrm{Objective}(\cdot)$ generally derives a temporary outcome-oriented exploration objective from an anchor, u_k is the selected anchor, and q_k is its corresponding objective. The anchor and its root-entry cue supply the starting direction, whereas every concrete action remains grounded in the live screenshot and UI hierarchy. This coverage-aware scheduling prevents the controller from repeatedly rediscovering already fully represented functions while preserving the freedom to investigate alternative procedures within an uncovered function still.

DFS expansion with validated branch control. For anchor u_k, the controller initializes a root DFS node with the launch-state observation and objective q_k. At the current node, it considers only untried, current-UI-grounded action candidates compatible with the anchor, the current state, the accumulated history, and previously observed interaction signatures. The Planner selects one executable action. After execution, the resulting observation is checked before the branch can be extended. Once a transition is verified, the corresponding child node is created, and DFS continues along that branch, exploring the sub-branch depth-first before moving to sibling nodes. If the attempted action is ineffective, redundant, leaves the intended app context, or leads to a terminal state, the resulting transition is not admitted as new evidence. The controller then marks that alternative as attempted and backtracks to a verified ancestor before trying another sibling from there. The restoration itself is deliberately separated from this high-level control flow. Appendix A.2 specifies the state matching, replay, and bounded recovery used to implement it.

Planner-Reflector-Recorder collaboration. These agents are connected through a propose-verify-record loop. Given (q_k, o_t, h_t) and the remaining unexplored interactive element candidates in the UI, the planner incorporates the reflector's reflective suggestions to avoid repeating failed choices, and then proposes a new GUI action a_t. After performing actions in the environment and generating o_{t+1}, reflector evaluates observed transfers against anchored targets, checks whether they are effective, consistent with the current branch, and whether they can be completed, and returns accepted/rejected results along with corrective guidance. Only accepted transfers are passed to recorder, which updates its source state, action, target state, semantic UI evidence, and results-oriented trajectory. The updated history of the recorder serves as the planner's next context, while rejected transfers input the reflector's guidance into the prediction of the next action after recovery. Thus, the exploration tree grows only through verified edges, while each completed branch retains the evidence needed for subsequent task-path, element, and skill records for reverse synthesis.

Overall, **TRACK** combines function-level coverage scheduling with state-backtracking DFS and closed-loop planner-reflector-recorder collaboration to support reliable knowledge acquisition. First, function-level scheduling allocates exploration to application functionalities that have not yet been covered rather than repeatedly visiting represented regions. Second, DFS expands alternatives from verified parent states, while state-aware backtracking restores usable contexts for continued branch exploration. Third, only validated state transitions are retained as evidence for subsequent knowledge synthesis. As a result, the generated branches provide broader functional coverage and more reliable interaction evidence for constructing reusable task-path, element, and skill knowledge.

A.2 State-Aware Backtracking Mechanism

DFS can expand alternative GUI interaction branches only if the device can be returned to a previously validated parent state. This requirement is nontrivial on Android: a system `Back` event may dismiss a transient surface, traverse multiple navigation levels, leave the target application, or land on a screen whose live UI differs from the state expected by the exploration controller. Consequently, **TRACK** does not regard `Back` as a deterministic inverse action. Instead, it implements state-aware backtracking as a recovery procedure that verifies the live UI against the exploration stack and replays already validated interactions when necessary. The mechanism is part of the offline knowledge-construction pipeline and it is used to preserve branch recoverability and record reliability, rather than being an independently ablated online component.

Recoverable exploration stack. Each DFS node is typically stored as $F_i = (\alpha_i, s_i, h_i, q_i)$, where α_i is the interaction that reached node i from its parent, s_i is the captured GUI state, h_i is the interaction history, and q_i is the temporary exploration objective. In the implementation, a state usually includes the screenshot, UI hierarchy XML, extracted elements, and a state signature. When a branch is abandoned, its entering interaction is marked as explored in the parent state and blocked where appropriate, and the leaf node is then removed and backtracking targets the new stack top F_j. Thus, a subsequent DFS expansion generally starts from a known parent context instead of the residual UI left by an unsuccessful or completed child branch.

State verification and local replay. Let $\tilde{s}$ denote the current state captured after a recovery action and $\mathcal{S} = \{s_0, \ldots, s_j\}$ the states explicitly retained in the current stack. $D_{\mathrm{xml}}(\tilde{s}, s_i)$ is the normalized XML-tree structural difference typically returned by `compare_xml`: the number of unmatched child nodes divided by the average number of nodes in the two UI hierarchies. **TRACK** first gives priority to frames with the same state signature and an acceptable XML difference, it then falls back to the closest acceptable stack state if none exists. Define

$$
\begin{aligned}
\mathcal{I}_{\mathrm{sig}} &= \{i : \mathrm{sig}(\tilde{s}) = \mathrm{sig}(s_i),\ D_{\mathrm{xml}}(\tilde{s}, s_i) \le \delta_{\mathrm{xml}}\}, \\
\mathcal{I}_{\mathrm{xml}} &= \{i : D_{\mathrm{xml}}(\tilde{s}, s_i) \le \delta_{\mathrm{xml}}\}, \\
i^\star = \mathrm{Match}(\tilde{s}, \mathcal{S}) &= \begin{cases} \arg\min\limits_{i \in \mathcal{I}_{\mathrm{sig}}} D_{\mathrm{xml}}(\tilde{s}, s_i), & \mathcal{I}_{\mathrm{sig}} \neq \varnothing, \\ \arg\min\limits_{i \in \mathcal{I}_{\mathrm{xml}}} D_{\mathrm{xml}}(\tilde{s}, s_i), & \mathcal{I}_{\mathrm{sig}} = \varnothing,\ \mathcal{I}_{\mathrm{xml}} \neq \varnothing, \\ -1, & \mathcal{I}_{\mathrm{xml}} = \varnothing. \end{cases}
\end{aligned} \tag{8}
$$

Here $\mathrm{sig}(s)$ is a deterministic state signature constructed from normalized visible-element role-layout-zone pairs and text/content-description tokens, excluding ephemeral element IDs. $\delta_{\mathrm{xml}} = 0.2$ is the fixed maximum accepted normalized XML difference. $\mathcal{I}_{\mathrm{sig}}$ and $\mathcal{I}_{\mathrm{xml}}$ are the signature-constrained and XML-only candidate sets, respectively. The $i^\star$ is the matched stack index, and the -1 denotes that the current UI cannot be safely aligned with any stored frame. The implementation breaks equal XML-distance matches in favor of the deeper stack state. On a successful match, the newly captured UI evidence is merged into the stored state, allowing later comparisons to use refreshed observations without discarding prior branch memory.

Before issuing `Back`, the controller performs this matching test. If the device is already aligned with F_j, no navigation action is taken. Otherwise, it issues a bounded number of `Back` events and captures the resulting UI after each transition. A match at $i^\star = j$ completes recovery. If $0 \le i^\star < j$, the device has returned too far, then **TRACK** restores the desired parent state by replaying the recorded interactions $\alpha_{i^\star+1}, \ldots, \alpha_j$. Replay is accepted only when every post-action

live state agrees with its expected stack state:

$$\mathrm{ReplayOK}(i,j) = \mathbb{I}\left[\bigwedge_{\rho=i+1}^{j} D_{\mathrm{xml}}\big(\tilde{s}_\rho, s_\rho\big) \le \delta_{\mathrm{xml}}\right]. \tag{9}$$

In Eq. 9, ρ typically indexes replayed stack frames, $\tilde{s}_\rho$ is the state usually observed after replaying α_ρ, s_ρ is the stored expected state, and $\mathbb{I}[\cdot]$ equals one only when all replayed transitions then pass verification. This per-step check explicitly prevents a superficially plausible replay from directly silently propagating an incorrect UI state into later exploration.

Escalating recovery and safety controls. If repeated `Back` events yield no stack match, if the foreground package is the launcher or outside the allowed target-app packages, or if local replay fails, the runtime then escalates once to Level-2 recovery. This routine force-stops and relaunches the target application, captures its fresh state, aligns it to a matching stack frame when possible, and otherwise replays the recorded stack path from the root frame. If replay stops at frame $\rho < j$, frames above ρ are truncated because their reachability is no longer verified. The procedure is further bounded by a maximum number of action attempts, a maximum number of back presses, transition waits before observation, and a single Level-2 escalation per recovery call. The same stack and branch progress are persisted in exploration checkpoints, and resume-time stack synchronization applies the same match-replay logic after an interruption.

This design provides three practical benefits. First, it enables systematic sibling-branch expansion despite non-invertible Android navigation, improving functional coverage within an anchor. Second, it isolates branches by ensuring that a new action is proposed from a verified state rather than an unknown residual screen, which protects the provenance of transitions later synthesized into task-path, element, and skill records. Third, it limits recovery cost: inexpensive local matching and replay are attempted before a relaunch, while explicit bounds and stack truncation prevent unbounded recovery loops or continued exploration from unvalidated states.

A.3 Candidate-Path Retrieval and Localization Details

The online retriever first forms a ranked candidate pool rather than selecting a single most similar task-path record. For candidate $p = (g_p, (d_1, \ldots, d_{L_p}))$, the retrieval score is computed from the cosine similarity between the normalized user intent and stored task goal:

$$\mathcal{P}_t = \mathrm{TopK}_{p \in \mathcal{K}^{\mathrm{task-path}}_{\nu(A)}} \left[r^{\mathrm{ret}}_t(p)\right], \qquad r^{\mathrm{ret}}_t(p) = \mathrm{sim}(\mathrm{emb}(z_g), \mathrm{emb}(g_p)). \tag{10}$$

Here K is the retrieval budget. The implementation filters candidates below the configured retrieval threshold and candidates whose application namespace does not match the active application.

For each retained candidate p, the valid alignment positions are

$$\mathcal{I}_t(p) = \left\{ \ell \in \{1, \ldots, L_p + 1\} : \begin{array}{l} \mu(d_i, h_t, o_t) = 1 \text{ for all } i < \ell, \\ \mu(d_\ell, h_t, o_t) = 0 \text{ if } \ell \le L_p \end{array} \right\}. \tag{11}$$

Here $\mu(d_i, h_t, o_t)$ indicates whether semantic step d_i is supported as completed by the history or current UI. The terminal position $\ell = L_p + 1$ is admitted only when the task-level goal predicate is verified. For the joint score, d_{L_p+1} denotes this terminal predicate. The selected semantic step is

$$\widehat{d}_{t+1} = \begin{cases} d_{\widehat{\ell}_t}, & \widehat{\ell}_t \le L_{\widehat{p}_t}, \\ \texttt{complete}, & \widehat{\ell}_t = L_{\widehat{p}_t} + 1 \text{ and the goal predicate is verified.} \end{cases} \tag{12}$$

If the candidate pool ultimately has no valid alignment, or if the best joint score is below the localization-confidence threshold, the implementation requests re-retrieval/re-planning and accordingly suppresses the corresponding path from knowledge injection.

A.4 Extended Experimental Settings

This section records implementation details that are omitted from the main text for readability. Unless otherwise specified, all settings were fixed before evaluation and the same frozen application-specific knowledge base was used throughout the corresponding comparison. No task goal, task instance, or online trajectory was written back to the knowledge base during evaluation.

Execution environment. The GUI environment was executed on a local Windows workstation using a Pixel 7 Android emulator exposed as ADB device. AndroidWorld and DroidTask retained their native reset, initialization, and evaluator procedures.

Offline collection and knowledge granularity. Offline exploration was performed once for each knowledge-base namespace using its associated app configuration (APK/package, initial state, and evaluator). A validated branch was converted into three complementary record types. Task-path records contain `task`, ordered `path`, and `is_task_completed`, and this flag is not redundant in the frozen files, which include both `true` and `false` values. Element records always contain `screenshot_path` and `functionality_description`, and records from richer sources additionally retain UI grounding, source-task, outcome-count, branch/anchor, or alignment fields when available. Skill records consistently contain a stable ID/name, description, parameters, precondition, ordered action sequence, scope, source, quality, and creation/update timestamps, with optional coverage and alignment fields. Repeated skill discoveries are merged using the record signature over name, parameters, precondition, and action sequence rather than appended as duplicate skills. Most evaluated applications map to an individual application-aligned namespace. For a small number of applications, multiple app variants share the same application identity and are therefore assigned to a single shared namespace, rather than being split into separate namespaces. This assignment is determined solely by application identity and is fixed before evaluation. It is not a separate experimental condition and does not introduce any cross-version training or transfer procedure. Offline exploration is performed once per namespace using its associated app configuration, including the APK or package, initial state, and evaluator, and the resulting validated records are stored under that namespace following the same record types and deduplication rules described above. The shared namespace is built and frozen in the same way as any individual namespace, and during online evaluation it is only retrieved from, not rewritten. Crucially, the assignment of applications to either an individual namespace or a shared namespace is identical across all compared knowledge conditions, such as TRACK-Fixed-KB and TRACK-KB. Therefore, the only factor that varies between conditions is the knowledge condition or online decision model itself, not the namespace granularity. During online execution, the runner links each application to its pre-assigned namespace and does not rewrite this multi-granularity knowledge base, but only leverages it. This design ensures that the knowledge base remains uncontaminated.

Knowledge files and vector persistence. The logical JSON schema is as follows. The skill file is formatted as {``skills'': `[record_1,...]`}. Task-path metadata is stored as a dictionary indexed by the stable record ID, where each value contains `task`, `path`, and `is_task_completed`. Element metadata is stored in a similar manner in `elements.meta.json`, with each value containing `screenshot_path` and `functionality_description`, and optionally `kb_aligned`.

Task-path and element vectors are persisted in the corresponding FAISS files `tasks.index` and `elements.index`, and the JSON metadata additionally stores the string-to-int64 ID map, reverse ID map, vector dimension, and metric. Both stores use `IndexIDMap2(IndexFlatIP)` with L2-normalized vectors, so the inner product is cosine similarity. The indexed vectors are not duplicated in the JSON metadata. Element screenshots remain external files referenced by metadata. Skills are not represented by a separate FAISS index in the reported implementation: they are loaded from the application-specific JSON file and ranked by lexical overlap plus intent-aware preference rules.

For task-path retrieval, task descriptions are embedded using the `Qwen3-VL-Embedding-2B` checkpoint. This checkpoint is used for both online embedding retrieval and offline index building, and the processed vectors are dimensioned at 2048.

Retrieval, injected and other thresholds. The default retrieval count is `top-k=8` for task-paths, `top-k=6` for elements, and `top-k=5` for skills. Task-path matches with cosine score at least 0.38 satisfy the **task-path strong threshold**, and scores in $[0.12, 0.38)$ satisfy the **task-path soft threshold** when the intent-alignment checks pass. Element retrieval uses a cosine element-score threshold of 0.52, and skill retrieval uses a lexical skill-score threshold of 0.08. The FAISS store searches up to $5\times$ the requested top-k (capped by the index size), filters by the threshold, and returns at most top-k records. The planner is capped at 10 steps. Conditional injection uses element-localization trigger 0.35 and minimum skill-trigger score 0.18, with at most 4 task-path steps, 2 skills, and 3

elements injected into decision context. These thresholds are used for retrieval and routing control, not for evaluating task execution results, as task success is determined by the evaluator.

The online conditions retrieve the same application-level files and use the same task list and evaluator, differing only in terms of the knowledge conditions or online decision models specified for the experiment settings. The TRACK-No-KB condition disables external knowledge, while the TRACK-Fixed-KB injects TRACK-KB via simple similarity matching, without modifying the original agent's workflow, action parser, action space, or executor, etc.

A.5 Application Selection and Task Distribution

Application selection. To obtain a functionally diverse evaluation benchmark, we first organize common mobile application scenarios into five functional categories: *Time & Scheduling*, *Communication*, *File Management*, *Imaging & Creativity*, and *Utilities & Others*. We then select applications from each benchmark according to these categories, prioritizing broad functional coverage rather than including multiple applications with highly similar functionality. Subject to the applications available in each benchmark, this selection results in 8 applications from AndroidWorld and 7 applications from DroidTask.

This application-level selection also considers the cost of application-specific knowledge acquisition. **TRACK** performs offline autonomous exploration and knowledge synthesis separately for each selected application, including interaction exploration, transition validation, and obtaining the corresponding multi-granularity knowledge base. Therefore, substantially increasing the number of applications would also proportionally increase the offline exploration and knowledge-acquisition cost. We consequently use a compact set of applications that covers diverse functional scenarios while keeping application-specific exploration tractable.

Importantly, the application selection is based on functional coverage and exploration feasibility rather than downstream task performance. After the applications are selected, we retain the eligible single-application tasks associated with them and exclude only tasks that require cross-application interaction, consistent with our focus on application-specific interaction knowledge.

Category coverage. The selected applications cover complementary types of mobile interaction. Time and scheduling applications involve temporal configuration and structured event manipulation. Communication applications require contact-oriented and message-oriented interactions. File-management applications involve hierarchical navigation and file operations. Imaging and creativity applications contain camera, drawing, or related visual operations. The utilities category covers additional application-specific functionality not captured by the preceding groups. This category-based organization is intended to increase functional diversity across the evaluation benchmark rather than enforce an equal number of applications or tasks in each category.

The resulting application and task distributions for AndroidWorld and DroidTask are reported in Appendix A.6. Because the availability and task distributions of application types differ between the two benchmarks, the numbers of applications and tasks are not explicitly balanced across categories. Instead, our selection emphasizes coverage of distinct interaction patterns while preserving the original tasks associated with the selected applications.

A.6 App distribution of selected benchmarks

Based on the rationale for our application category selection described above, we categorize the evaluated applications into five groups. Following this taxonomy, the distribution of application categories across the two benchmarks is summarized in Table 6. Since our evaluation focuses exclusively on single-application tasks, we exclude tasks that require interactions across multiple applications.

The selected AndroidWorld contains tasks from 8 applications: Simple Calendar Pro (17), Clock (3), Contacts (2), Simple SMS Messenger (6), Files (2), Camera (2), Simple Draw Pro (1), and Broccoli (10), resulting in a total of 43 tasks. The DroidTask test set covers 7 applications: Simple Calendar (17 tasks), Simple Clock (12), Simple Contacts Pro (10), Simple SMS Messenger (14), Simple File Manager Pro (13), Simple Camera (6), and Simple Notes Pro (14), for a total of 86 tasks.

Table 6: Application categories and apps used in AndroidWorld and DroidTask.

Category	AndroidWorld Apps	DroidTask Apps
Time & Scheduling	Simple Calendar Pro, Clock	Simple Calendar, Simple Clock
Communication	Contacts, Simple SMS Messenger	Simple Contacts Pro, Simple SMS Messenger
File Management	Files	Simple File Manager Pro
Imaging & Creativity	Camera, Simple Draw Pro	Simple Camera
Utilities	Broccoli	Simple Notes Pro

A.7 THE KNOWLEDGE BASE OF **TRACK**

Table 7 summarizes the application-specific knowledge base of **TRACK**. It contains 11 knowledge base namespaces spanning the AndroidWorld and DroidTask. Camera/Simple Camera, Contacts/Simple Contacts Pro, and Clock/Simple Clock each share one namespace across the two benchmarks, and every other row denotes a distinct application-specific namespace. First, the 128 anchors quantify functional coverage: an anchor corresponds to a major app function and provides an exploration starting point, rather than denoting a benchmark task or a fixed interaction script. Thus, the anchor count measures the breadth of functions that the offline explorer is asked to investigate, while the resulting records measure the reusable knowledge produced from that exploration.

The record counts expose the intended multi-granularity design. Across the 11 namespaces, the knowledge base contains 165 task-path records, 507 element records, and 162 skill records (834 records in total). Task-path records preserve ordered, semantically high-level execution steps and therefore describe what to do. Element records capture the UI entities and grounding cues needed to determine where to act. Skill records reusable action sequences from exploration branches that can be reused by the online-stage task, thereby characterizing how to execute an action. Different apps naturally exhibit different knowledge distributions. For instance, element records are numerous when an app exposes many state-dependent controls (e.g., Clock and Contacts), whereas skill records are relatively dense in apps with reusable operations (e.g., Files). As a result, the table summarizes each namespace's exploration coverage and multi-granularity knowledge structure, spanning the aforementioned five predefined app categories.

Table 7: The multi-granularity application-specific knowledge base of **TRACK**. Each row denotes one knowledge-base namespace. † marks a namespace shared by the AndroidWorld and DroidTask counterparts, and unmarked rows denote distinct application-specific namespaces.

KB Namespaces	Anchors	Records (Task-paths / Elements / Skills)
Files	11	Task-paths = 12 / Elements = 21 / Skills = 48
Broccoli	12	Task-paths = 19 / Elements = 19 / Skills = 12
Simple Draw Pro	13	Task-paths = 13 / Elements = 55 / Skills = 2
Simple Calendar	19	Task-paths = 19 / Elements = 73 / Skills = 19
Simple Notes Pro	8	Task-paths = 8 / Elements = 23 / Skills = 3
Simple Calendar Pro	8	Task-paths = 6 / Elements = 21 / Skills = 6
Simple SMS Messenger	9	Task-paths = 14 / Elements = 22 / Skills = 14
Clock / Simple Clock†	6	Task-paths = 23 / Elements = 143 / Skills = 13
Simple File Manager Pro	14	Task-paths = 18 / Elements = 2 / Skills = 3
Camera / Simple Camera†	10	Task-paths = 15 / Elements = 25 / Skills = 6
Contacts / Simple Contacts Pro†	18	Task-paths = 18 / Elements = 103 / Skills = 36
Totals	**128**	**Task-paths = 165 / Elements = 507 / Skills = 162**

After construction, the namespace-level knowledge base is frozen and reused without online write-back. The three † namespaces in Table 7 are shared across their AndroidWorld and DroidTask counterparts, while every unmarked namespace typically remains exclusive to its listed application. The TRACK-Fixed-KB and **TRACK** conditions then access the same records within the resolved namespace, allowing improvements from the online knowledge utilization mechanism to be separated from improvements due merely to supplying external knowledge. Consequently, the online

results explicitly evaluate utilization of the same frozen namespace-level knowledge base rather than a separately tuned repository for each experimental condition.

A.8 Per-Application Success Rates on AndroidWorld

Figure 5 shows SR for AndroidWorld used in the main experiments. Qwen3.5-Plus is employed as the online decision-making. For each app, we execute its test tasks and calculate the SR within that app. The methods follow the same taxonomy as Section 4.2: Non Offline-Online (Mobile-Agent-v2, AndroidWorld), Offline-Online (AutoDroid, AppAgent, KG-RAG, and GUI-explorer), Online Explore (MobileUse), and **TRACK** (Ours).

Results. **TRACK** achieves the highest or tied-highest success rates on 6 of the 8 evaluated applications: Camera (100.00%), Clock (100.00%), Contacts (100.00%), Files (100.00%), Broccoli (80.00%), and Simple Calendar Pro (76.47%). On the two applications with relatively larger numbers of tasks, Broccoli and Simple Calendar Pro, our method matches the best-performing method, successfully completing 8/10 and 13/17 tasks, respectively. Overall, it demonstrates stronger per-application performance than the compared baselines across most applications.

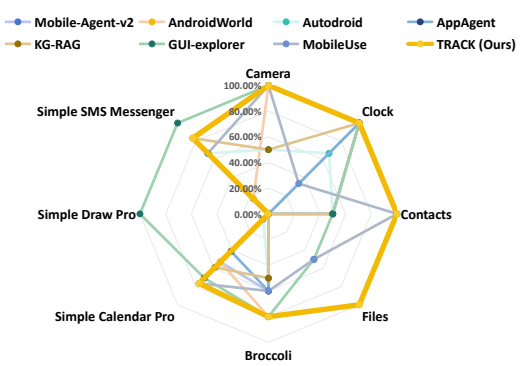


Figure 5: Per-application success rates on the AndroidWorld in the main experiments, with Qwen3.5-Plus as the online model.

The performance gains, however, are not uniform across applications. On Simple SMS Messenger, **TRACK** achieves 83.33% (5/6), matching KG-RAG but remaining below GUI-Explorer, which achieves 100.00%. **TRACK** also fails on the single Simple Draw Pro task, whereas GUI-Explorer succeeds. Since Simple Draw Pro contains only one evaluation task, this result does not indicate that **TRACK** is entirely ineffective on this application.

A.9 Parameter Analysis

Tables 8, 9, and 10 analyze how retrieval budgets, injection budgets, and routing thresholds affect decision outcomes, examining the trade-off between retaining sufficient decision evidence and suppressing redundant or mismatched context.

Balanced retrieval and injection budgets are critical. Across both online decision models, the default moderate retrieval and injection budgets achieve the highest success rates on AndroidWorld, while either reducing or expanding them degrades performance. Expanding the retrieval pool to the largest tested setting reduces SR to 60.47% and 62.79% for Qwen3.5-Plus and GLM-5.3-Flash, respectively, while larger injection budgets do not improve the performance of both models simultaneously over sparse injection. These trends show that exposing the agent to more knowledge does not necessarily improve decision-making. Small budgets may omit useful task-path, element, or skill records, whereas large retrieval pools and injected contexts introduce redundant or progress-misaligned guidance. Thus, effective knowledge utilization requires a balanced amount of candidate evidence and injected context.

Selective routing requires jointly calibrated gates. Both alternative threshold configurations underperform the default setting across the two models. The lower-gate setting causes a particularly large degradation on Qwen3.5-Plus, while uniformly increasing the thresholds recovers only part of the performance. This suggests that routing quality depends on balancing candidate retention and selective injection rather than applying uniformly permissive or restrictive filtering. Overall, reliable knowledge utilization requires retaining sufficient candidates for progress localization while selectively injecting concise, state-relevant knowledge and filtering stale or low-value context.

Table 8: Effect of retrieval budget on AndroidWorld success rates (%).

Retrieval Records	Qwen3.5-Plus	GLM-5.3-Flash
Task / Elements / Skills top-k = 4 / 3 / 3	55.81	74.42
Task / Elements / Skills top-k = 12 / 8 / 9	60.47	62.79
TRACK (Ours)	**81.40**	**83.72**

Table 9: Effect of injection budget on AndroidWorld success rates (%).

Injected Records	Qwen3.5-Plus	GLM-5.3-Flash
Path steps / Skills / Elements = 2 / 1 / 1	74.42	76.74
Path steps / Skills / Elements = 6 / 3 / 5	69.77	72.09
TRACK (Ours)	**81.40**	**83.72**

Table 10: Effect of routing thresholds on AndroidWorld success rates (%).

Thresholds	Qwen3.5-Plus	GLM-5.3-Flash
Task strong / soft = 0.30 / 0.08, Element = 0.42, skill = 0.04, element-localization / skill trigger = 0.25 / 0.12	48.84	58.14
Task strong / soft = 0.46 / 0.18, Element = 0.62, skill = 0.12, element-localization / skill trigger = 0.45 / 0.24	62.79	74.42
TRACK (Ours)	**81.40**	**83.72**

A.10 Knowledge-Hit Effectiveness Analysis of DroidTask

Tables 11 and 12 reveal a routing profile that differs from AndroidWorld. Task-path knowledge is retrieved and injected for every DroidTask task and each decision step on both models (THR/SIC/RIC=100%), so no task-path candidate is discarded in these runs. Element records are also routed with high conversion. Their RIC is 83.64% with Qwen3.5-Plus and 84.55% with GLM-5.3-Flash, although their step-level injection coverage remains limited to 35.44% and 42.66%, respectively. In contrast, only about half of retrieved skill records are injected (RIC: 50.29% and 50.16%). Thus, conditional routing preserves the always-routed task-path context, retains most element candidates once their UI-grounding condition is met, and still filters a substantial share of locally retrieved skills before they enter the decision context.

Remove only records with unaligned progress. On DroidTask, task-paths knowledge dominates the injection, followed by elements knowledge, and finally skills knowledge. Compared with Figure 3, we find that the tasks and steps for which knowledge type was retrieved still receive injections of the corresponding knowledge types after routing, albeit with a decline in the RIC metric, indicating that some records were removed. Different models have only a minor effect on injection, except for step-level skills knowledge. Therefore, this indicates that the decisions made during the retrieval stage regarding which tasks and steps should leverage knowledge are preserved after routing, with only some knowledge records that are unaligned with the current execution progress being removed.

A.11 Mean Candidates per Selected Decision Step

To complement the binary retrieval-hit statistics in Section 4.4, we measure the density of retrieved records at each selected decision step. For knowledge type m, let $\mathcal{C}_t^m$ denote the candidate records returned at step t, and let T denote the number of LLM decision steps. We compute

$$\mathrm{MCPS}_m = \frac{\sum_{t=1}^{T} |\mathcal{C}_t^m|}{T}. \tag{13}$$

Table 11: Actual knowledge injections on DroidTask with Qwen3.5-Plus. THR denotes task-level knowledge hit rate, SIC denotes step-level knowledge injection coverage, and RIC denotes retrieval-to-injection conversion.

Knowledge Type	Qwen3.5-Plus		
	THR (%)	SIC (%)	RIC (%)
Task-Paths	100	100	100
Elements	54.65	35.44	83.64
Skills	11.63	9.79	50.29

Table 12: Actual knowledge injections on DroidTask with GLM-5.3-Flash. THR denotes task-level knowledge hit rate, SIC denotes step-level knowledge injection coverage, and RIC denotes retrieval-to-injection conversion.

Knowledge Type	GLM-5.3-Flash		
	THR (%)	SIC (%)	RIC (%)
Task-Paths	100	100	100
Elements	56.98	42.66	84.55
Skills	12.79	20.58	50.16

Table 13: Retrieved candidate density of steps on AndroidWorld. The results report the mean number of retrieved records per decision step, multiplied by 100 for compact presentation. The parenthesized values give the retrieved-record count divided by the number of selected decision steps.

Knowledge Type	Candidates Per Step (%)	
	Qwen3.5-Plus	GLM-5.3-Flash
Task-Paths	71.55% (342/478)	72.92% (342/469)
Elements	4.81% (23/478)	9.38% (44/469)
Skills	208.16% (995/478)	202.56% (950/469)

Table 14: Retrieved candidate density of steps on DroidTask. The results report the mean number of retrieved records per decision step, multiplied by 100 for compact presentation. The parenthesized values give the retrieved-record count divided by the number of selected decision steps.

Knowledge Type	Candidates Per Step (%)	
	Qwen3.5-Plus	GLM-5.3-Flash
Task-Paths	100% (1185/1185)	100% (1001/1001)
Elements	142.87% (1693/1185)	135.16% (1353/1001)
Skills	58.40% (692/1185)	123.08% (1232/1001)

This is a retrieval-side metric: it counts the number of candidate records returned by the retriever before online routing.

A.12 Analysis of Retrieved Candidate Density

Skills form the densest retrieval pool. Across both online models, the retrieval-density ordering is consistent: skills form the largest candidate pool, followed by task-paths, while element candidates remain sparse. Specifically, Qwen3.5-Plus and GLM-5.3-Flash retrieve 2.0816 and 2.0256 skill records per selected decision step, 0.7155 and 0.7292 task-path records, and 0.0481 and 0.0938 element records, respectively. This pattern reflects the distinct roles of the three knowledge types. Skills encode reusable local procedures and can thus match multiple action contexts. In contrast, task-paths capture higher-level structured task progression, while element knowledge is applicable only under specific UI-grounding conditions. The resulting retrieval pool is thus heterogeneous and strongly skill-heavy. This also motivates conditional knowledge routing: candidate abundance should be treated as retrieval evidence rather than a direct indication of decision need, since unconditional concatenation would introduce redundant or weakly relevant context.

The knowledge candidate densities are stable across models. Table 13 shows that Qwen3.5-Plus and GLM-5.3-Flash exhibit similar task-path and skill densities, despite differing in the number of decision steps (478 vs. 469). Both models follow the same overall trend: skills dominate the candidate set, task-paths show moderate density, and elements are retrieved only occasionally. The largest relative difference lies in element knowledge, which rises from 4.81% to 9.38% per step record. However, this corresponds to only 23 vs. 44 retrieval records, indicating that the difference reflects a higher retrieval proportion rather than a larger total number.

DroidTask has a denser, element-centered candidate pool. Table 14 exhibits a different retrieval profile from AndroidWorld. Element knowledge forms the densest pool, with 1.4287 and 1.3516 records per step for Qwen3.5-Plus and GLM-5.3-Flash, respectively. Skills contribute 0.5840 and 1.2308 records per step, respectively. Hence, candidate retrieval on DroidTask is not skill-heavy:

multiple element candidates are commonly available for decision steps, whereas task-path retrieval supplies a single persistent global record.

Task-path and element densities are stable, while skill retrieval is model-sensitive. The two DroidTask runs agree closely on the always-present task-path record and on element density (1.4287 vs. 1.3516 records per step), even though they contain different numbers of selected steps (1185 vs. 1001). Skill density varies more substantially, from 0.5840 for Qwen3.5-Plus (692 records) to 1.2308 for GLM-5.3-Flash (1232 records). Because GLM-5.3-Flash produces more skill records despite having fewer selected steps, this gap reflects differences in the trajectory-level retrieval opportunities generated by the models rather than only the denominator. Together with the approximately 50% skill retrieval-to-injection conversion in Tables 11 and 12, these results show why routing remains necessary: the system should distinguish the reusable local procedures that are needed at the current step from the additional candidates returned along a trajectory.

A.13 Retrieved and Injected Knowledge Footprint

Measurement scope. As shown in Table 15, the retrieved and injected knowledge footprint measure the cumulative UTF-8 byte lengths of the serialized knowledge records observed along the execution trajectories. Retrieved storage footprint counts all candidate records returned before routing, whereas injected storage footprint counts the knowledge blocks actually inserted into the decision prompts. These values therefore represent trajectory-level knowledge payloads rather than the on-disk size of the knowledge base: repeated retrievals or injections of the same record are counted again, and index files and other runtime metadata are excluded.

Results analysis. The retrieval footprint is dominated by skills for both online models. Skills account for 85.75% (949.85 vs. 1107.73 kB) for Qwen3.5-Plus and 84.71% (900.32 vs. 1062.79 kB) for GLM-5.3-Flash, while task-paths contribute only 13.46% and 13.68%, respectively, and elements contribute less than 2% in either run. This composition agrees with the candidate-density results in Table 13. Skill candidates are the most numerous because local reusable procedures can match many intermediate UI states, whereas task-paths are more structured and element records require a specific grounding opportunity. Thus, the storage burden of an unrouted retrieval context would be concentrated in skills.

Table 15: Cumulative storage footprint of retrieved and injected knowledge on AndroidWorld. Values are in decimal kB (1 kB = 1000 bytes). The retrieved values count serialized candidate payloads across retained decision steps, while injected values count the knowledge blocks actually written to the saved prompts.

Knowledge Type	Qwen3.5-Plus		GLM-5.3-Flash	
	Retrieved	Injected	Retrieved	Injected
Task-Paths	149.11	5.46	145.34	4.98
Elements	8.77	1.77	17.14	6.34
Skills	949.85	145.89	900.32	141.41
Totals	**1107.73**	**153.12**	**1062.80**	**152.73**

The injected footprint is much smaller than the retrieved footprint. Routing exposes only 13.82% (153.12 vs. 1107.73) and 14.37% (152.73 vs. 1062.80) of the respective retrieved payloads for the two models. This reduction is most pronounced for task-paths, whose injected footprints are only 5.46 kB and 4.98 kB, corresponding to roughly 3.66% and 3.43% of their retrieved task-path payloads. This is consistent with the knowledge-hit analysis in the main text: task-path retrieval is relatively frequent, but the router injects it only when global procedural guidance remains useful after progress localization.

Different models exhibit different element selection outcomes. Qwen3.5-Plus and GLM-5.3-Flash inject 20.18% (1.77 vs. 8.77) and 36.99% (6.34 vs. 17.14) of their retrieved payloads into the model context, respectively. This difference is consistent with the higher retrieval-to-injection conversion observed for GLM-5.3-Flash in the main experiments, indicating that a larger proportion of element knowledge records is injected through routing in this model's trajectories.

A.14 DEMOS

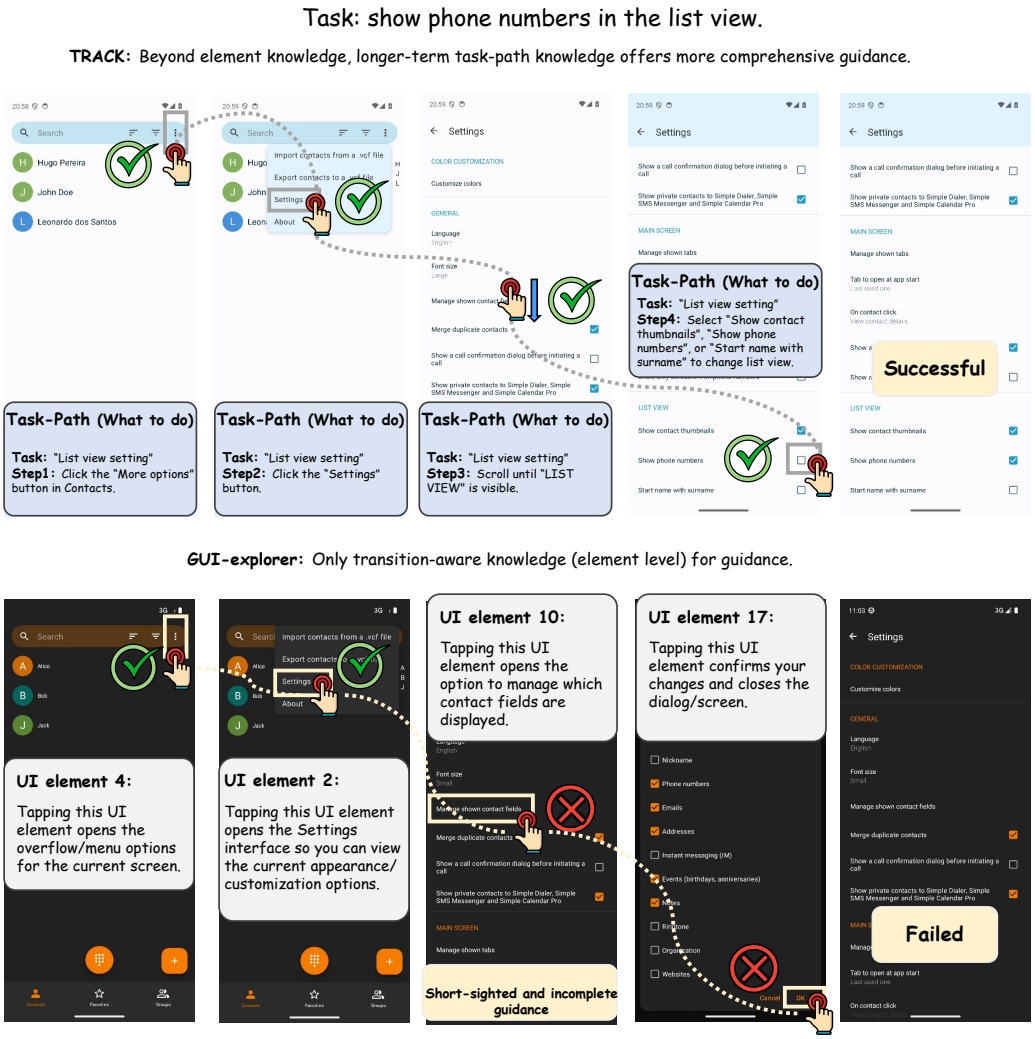


Figure 6: Demonstration of task-path knowledge guiding key steps through the scroll to LIST VIEW path in an online procedure. In the list view setting task, **TRACK** navigates to the settings page and scrolls until the LIST VIEW option becomes visible, then interacts with it. The task-path knowledge directs these key steps, resulting in successful task completion. By contrast, GUI-explorer fails the task, as it lacks such knowledge and consequently clicks an incorrect button.

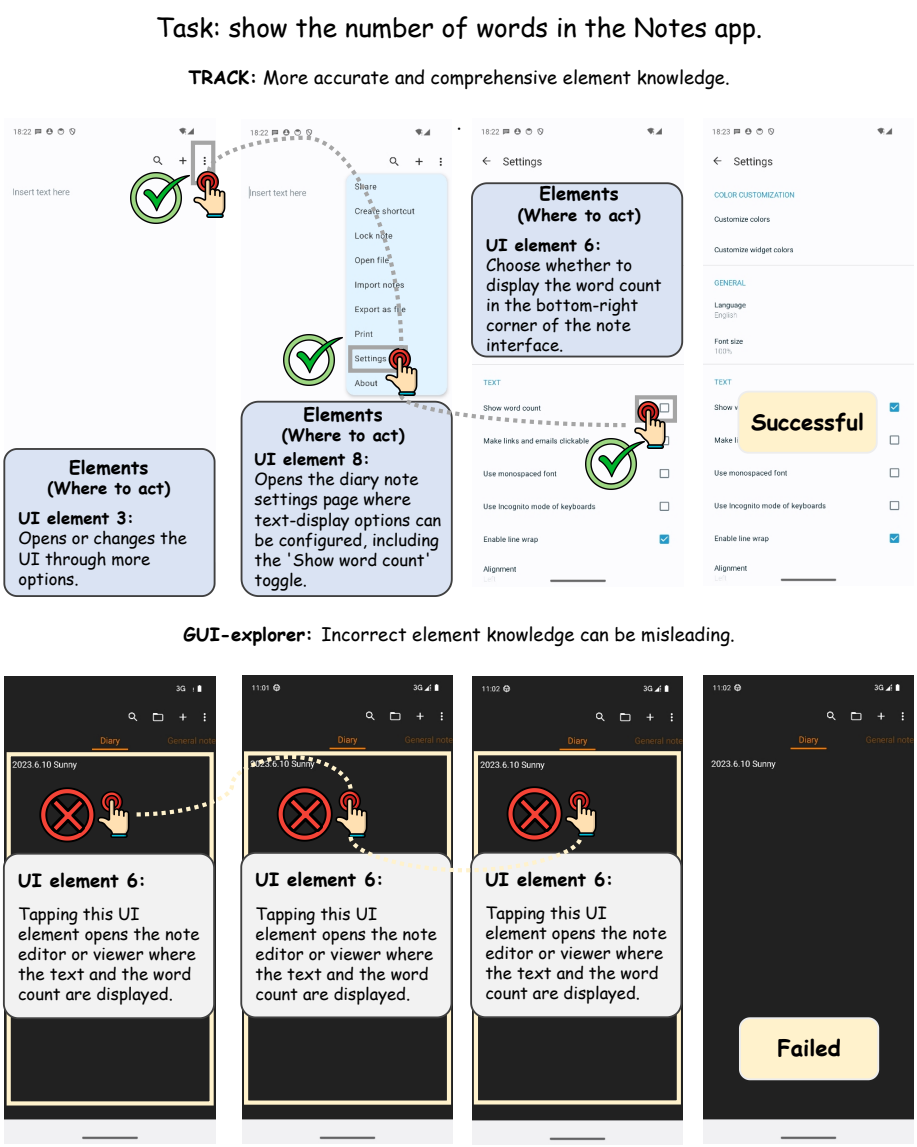


Figure 7: Demonstration of element knowledge guiding key steps. In the task of showing the number of words in the notes app, **TRACK** provides detailed and accurate knowledge of the interactive elements, namely option, settings, and show word count, thereby enabling successful task completion. In contrast, GUI-explorer, having been provided with incorrect element knowledge, repeatedly clicks on the editing page and consequently fails to complete the task.

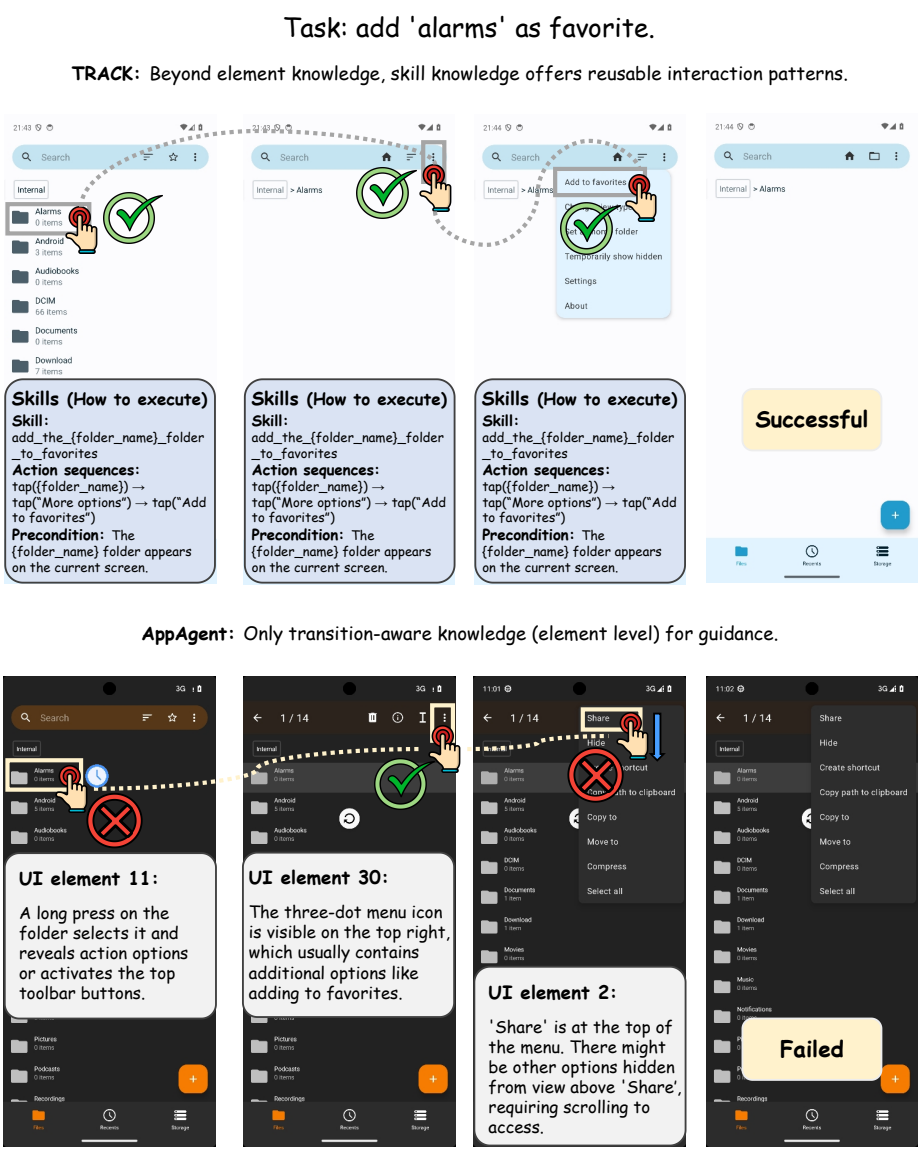


Figure 8: Demonstration of skill knowledge guiding key steps. In the add 'alarms' as favorite task, **TRACK** provides the add_the_{folder_name}_folder_to_favorites skill and completes the task step-by-step and reliably according to the usable interaction pattern of this skill. In contrast, AppAgent only provides transition-aware knowledge, which offers clearly insufficient guidance and thus leads to erroneous steps such as long press and click share.

A.15 PROMPTS

For reproducibility, this section presents offline and online prompts. The Appendix A.15.1, Appendix A.15.2 and Appendix A.15.3 are the LLM prompts used for offline exploration. The online planner prompt is provided in Appendix A.15.4.

A.15.1 OFFLINE PLANNER PROMPT

```
You are the Planner agent of a mobile application exploration system. Select one
     interaction that discovers or advances a new, user-meaningful task rather than merely
     navigating the interface.

## Input schema
- available_elements: {available_elements}
- attempted_actions: {explored_interactions_current_screen}
- effective_history: {history_path}
- current_objective: {current_task_objective}
- reflector_feedback: {previous_suggestion}
- similar_tasks and saved_catalog: {similar_tasks_from_memory}, {saved_tasks_catalog}

## Core rules
1. Infer one coherent, outcome-focused user task from the trajectory, objective, and
     visible UI values.
2. Choose an action only if it advances that task or reveals a genuinely new user intent,
     prefer a visible state change or user outcome over navigation-only progress.
3. Use attempted actions, similar tasks, the saved catalog, and reflector feedback to avoid
      duplicate intents and failed paths.
4. Keep the objective imperative and specific (verb + object + outcome), do not describe
     taps, screens, or flow mechanics as the objective.
5. Do not perform destructive actions unless the current task requires them. Choose `
     complete` only when the outcome is achieved or no worthwhile new task remains.

## Abstract action schema
- `tap`: {target_id}
- `text`: {target_id, text}
- `long_press`: {target_id}
- `swipe`: {direction: up|down|left|right, optional target_id, optional distance}
- `complete`: {reason}
`target_id` must refer to a control in the current live UI, an environment adapter may
     convert it to coordinates or a native candidate action.

## Output schema
Return one JSON function call and no other text:
{
  "function": "tap|text|long_press|swipe|complete",
  "parameters": {
    "target_id": "<live UI id when required>",
    "text": "<required for text>",
    "direction": "<required for swipe>",
    "reasoning": "Inferred user task, why this action advances it, and why it avoids
     duplicates or failed paths."
  }
}
```

A.15.2 OFFLINE REFLECTOR PROMPT

```
You are the Reflector agent of a mobile application exploration system. Evaluate whether
     the latest interaction meaningfully advances a coherent, user-oriented task, not
     merely whether the screen changed.

## Input schema
- effective_history: {history}
- planner_reasoning: {planner_reasoning}
- executed_action: {function, target_id, action_args}
- target_element: {interaction_UI_element}
- ui_before and ui_after: {before_UI_elements}, {after_UI_elements}
- previous_suggestion: {previous_suggestion}

## Evaluation rules
1. Infer the branch's user task from the history, planner reasoning, and visible values.
2. Compare the before and after states and verify that the action took effect.
3. Mark `successful` only for meaningful same-intent progress, mark `ineffective` for a no-
     op, wrong control, or navigation without user-value progress, and mark `unexpected`
     for errors, blockers, unrelated content, or intent drift.
4. Do not confuse opening a page, list, or intermediate screen with task completion.
5. Give concrete task-oriented guidance: continue after success, use an alternative route
     after ineffectiveness, and backtrack after unexpected behavior. Do not recommend the
     same element--action signature again.

## Output schema
Return only one JSON object:
{
  "evaluation_result": "successful|ineffective|unexpected",
  "reasoning": "Task inference, decisive before/after evidence, and semantic consistency.",
  "functionality_description": "User-facing affordance of the target element.",
  "suggestion": "Concrete next-step guidance for the same task."
}
```

A.15.3 OFFLINE RECORDER PROMPT

```
You are the Recorder agent of a mobile application exploration system. After each verified
     successful interaction, update the trajectory and refine the branch into a benchmark-
     style user-task instruction.

## Input schema
- previous_objective: {previous_task_objective}
- effective_history: {previous_exploration_history}
- verified_action: {function, target_id, action_args}
- resulting_state: {current_state_description}

## Core rules
1. Record only verified successful interactions and preserve chronological order.
2. Record the source screen, action, visible label or role, and resulting state. Never use
     internal IDs.
3. Infer the user's outcome from the trajectory and resulting UI, not from navigation
     mechanics.
4. Write one direct imperative objective (verb + object + outcome), including visible
     parameters when available. A page or list opening is intermediate progress, not the
     final task.
5. Keep one coherent intent per branch. Retain the state suffix only on the newest
     trajectory entry.

## Output schema
Return only one JSON object:
{
  "updated_exploration_history": "Full updated trajectory",
  "updated_task_objective": "One direct, outcome-focused benchmark task instruction"
}
```

A.15.4 ONLINE PLANNER PROMPT

```
You are the online Planner for a mobile GUI task. Select exactly one grounded action that
     advances the user goal.

## Input schema
- user_goal: {user_goal}
- effective_history: {history_path}
- available_elements: {available_elements}
- compressed_knowledge_injection: {compressed_knowledge}
- execution_feedback: {last_feedback}

## Decision rules
1. Follow the state-aligned remaining plan and its next unresolved semantic subgoal when it
      is compatible with the live UI. If it is incompatible, re-plan from the goal, history
     , and live UI rather than following stale knowledge.
2. Use the compressed knowledge injection as advisory evidence only. Ground every targeted
     action to a current UI candidate. Never reuse historical or retrieved identifiers.
3. Select exactly one action per turn. Skip already satisfied steps and do not repeat an
     action that produced no progress on an unchanged screen.
4. Use a skill only when its preconditions hold in the live UI. Completion is allowed only
     when the current state or verifier confirms the full goal.

## Output schema
Return one structured action and no other text:
{
  "action_type": "<one action in the current action space | complete>",
  "target_id": "<current UI candidate id, omit only when no target is needed>",
  "action_args": {
    "text": "<required for text input>",
    "direction": "<required for scroll/swipe>"
  },
  "reasoning": "<next semantic subgoal, live grounding evidence, and why the action
     advances the goal>"
}
```