GUI-SKE: STRUCTURED KNOWLEDGE EVOLUTION FOR NON-PARAMETRIC GUI AGENT ADAPTATION

Anonymous authors
Paper under double-blind review

ABSTRACT

GUI agents powered by multimodal large language models show promise for mobile task automation, yet can struggle in unseen applications where navigation structures and element-level interaction behaviors differ from prior experience. Parametric adaptation incurs additional optimization costs, while acquiring environment-specific knowledge and refining it using execution feedback remain challenges for non-parametric approaches. We introduce **GUI-SKE**, a non-parametric **S**tructured **K**nowledge **E**volution framework that adapts GUI agents by constructing and refining external knowledge while keeping model parameters fixed. GUI-SKE performs two-level exploration to acquire GUI-specific interaction evidence at different granularities and organizes it into linked functional, transition, and operational knowledge components. It derives pseudo-tasks with verification requirements from this evidence and uses rollout feedback as weak supervision for knowledge refinement. Group-wise trajectory comparison combines task-success and component-specific process feedback to guide targeted edits to the corresponding knowledge entries. Experiments on AndroidWorld, SPA-Bench, and our proposed ProMobile-Bench show that GUI-SKE achieves higher task success rates than the baselines included in our comparisons.

1 INTRODUCTION

GUI agents translate high-level task instructions into sequences of interface actions, enabling task automation across diverse software applications (Zhang et al., 2025; Lai et al., 2025; Qin et al., 2025; Wang et al., 2025). However, GUI interaction patterns vary across applications, and in mobile settings agents may encounter diverse apps where even familiar task goals can require unfamiliar navigation procedures and element-level interactions (Rawles et al., 2024; Chen et al., 2024; Yang et al., 2026; Zhang et al., 2024). For example, in an expense-entry task, agents may select the wrong category without knowing to scroll horizontally to reveal the target, as illustrated in Appendix A. This motivates adaptation methods that can acquire and organize the interaction knowledge needed to complete tasks in unfamiliar environments.

Existing approaches to GUI environment adaptation broadly follow two directions: parametric and non-parametric adaptation. Parametric adaptation methods (Bai et al., 2024; Shi et al., 2025; Du et al., 2026) update model parameters through fine-tuning or reinforcement learning, incurring additional computational costs for parameter optimization. In contrast, non-parametric adaptation methods keep model parameters fixed and use external knowledge to guide execution (Xie et al., 2025; Li et al., 2024). Such knowledge can be organized at different granularities, ranging from trajectory-level experience reuse (Zheng et al., 2023) to transition-aware interaction knowledge derived from exploratory trajectories (Xie et al., 2025). Knowledge constructed from finite observed interactions may still provide incomplete coverage of the target environment. Beyond initial knowledge acquisition, recent methods further use task outcomes or trajectory evaluations to select and refine reusable experience (Mi et al., 2026; Li et al., 2025). However, task outcomes provide limited guidance on what knowledge should be revised, while trajectory-level evidence still needs to be associated with specific knowledge content to support targeted refinement.

To address these challenges, we introduce GUI-SKE, a non-parametric framework for GUI agent adaptation through structured knowledge construction and refinement. It acquires interaction evi-

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.

dence at different granularities, organizes it into structured external knowledge, and further refines this knowledge using exploration-derived weak supervision.

Concretely, GUI-SKE consists of two adaptation stages: structured knowledge construction and knowledge update. For construction, we perform two-level exploration to collect GUI-specific interaction evidence at both page and element levels: global exploration captures observed page structures and transitions, while local exploration probes element-level actions and their effects. We organize this evidence into functional, transition, and operational knowledge, describing application functions, observed navigation paths, and page-specific interaction behavior, respectively, and providing explicit units for retrieval and refinement. For knowledge update, we generate pseudo-tasks with corresponding verification criteria from the explored evidence. Following the group-wise comparison principle of Training-Free GRPO (Cai et al., 2025), we compare multiple trajectories generated for each pseudo-task. Each trajectory is evaluated using a task-success reward and component-wise process rewards that assess task–function consistency, transition consistency, and element–action consistency. The resulting comparisons produce component-wise textual feedback that relates execution evidence to specific knowledge entries and guides edits to the corresponding entries.

We evaluate GUI-SKE on AndroidWorld (Rawles et al., 2024) and SPA-Bench (Chen et al., 2024) to assess adaptation in mobile GUI environments. To further evaluate adaptation on fine-grained editing operations and multi-step professional workflows, we introduce ProMobile-Bench, covering image, audio, and video applications. Across all three benchmarks, GUI-SKE achieves higher task success rates than the baselines included in our comparisons. Our main contributions are:

- We propose GUI-SKE, a non-parametric GUI agent adaptation framework that organizes interaction evidence from global and local exploration into linked functional, transition, and operational knowledge, supporting task-relevant retrieval and targeted revision.
- We develop a knowledge refinement mechanism using exploration-derived pseudo-tasks and verification criteria as weak supervision. Group-wise trajectory comparison combines task-success and component-specific process feedback to guide structured edits to the relevant knowledge entries.
- We introduce ProMobile-Bench, a benchmark for evaluating GUI agent adaptation in professional editing workflows across image, audio, and video applications.

2 Related Work

GUI Agent Adaptation in Unseen Environments. Adapting GUI agents to unseen applications may require application-specific interaction knowledge when task execution depends on interface-specific navigation and element-level interactions. Parametric approaches incorporate interaction experience through model updates (Bai et al., 2024; Du et al., 2026; Fan et al., 2025), with GUI-Bee (Fan et al., 2025) combining autonomous exploration and continual model updates. Non-parametric approaches retain acquired knowledge externally while keeping model parameters fixed. GUI-explorer (Xie et al., 2025) extracts transition-aware interaction knowledge from exploratory trajectories, while the non-parametric component of GTTA (Chen et al., 2026) constructs an in-context model of environment dynamics through exploration. Compared with these transition- and dynamics-oriented approaches, GUI-SKE additionally incorporates element-level interaction evidence and organizes the acquired knowledge for subsequent refinement.

Experience and Memory for Agent Adaptation. External memory allows agents to reuse interaction experience during subsequent execution. Synapse (Zheng et al., 2023) retrieves trajectories as in-context exemplars, while ReasoningBank (Ouyang et al., 2026) distills reusable reasoning strategies from successful and failed executions. HyMEM (Zhu et al., 2026) couples symbolic memory nodes and trajectory embeddings within an evolving memory graph, while SE-GA (Jin et al., 2026) combines inference-time memory retrieval with policy training on accumulated experience. Training-Free GRPO (Cai et al., 2025) refines experiential knowledge through group-relative rollout comparison without parameter updates. GUI-SKE focuses on environment-specific GUI knowledge at explicit component and entry granularity, combining group-relative comparison with component-specific feedback to support localized refinement.

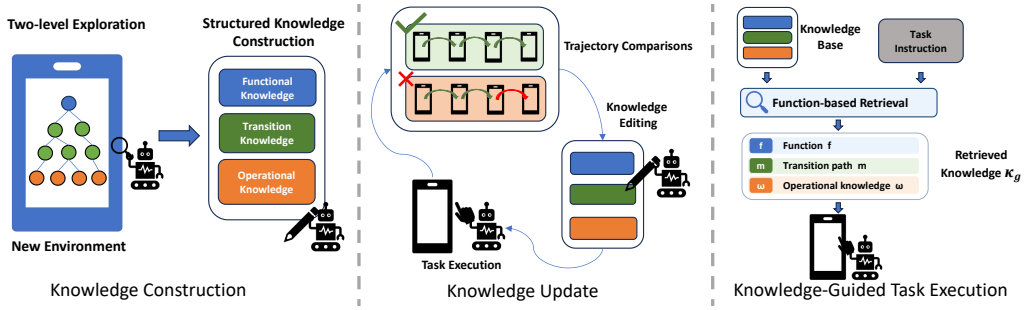

Figure 1: Overview of GUI-SKE. Two-level exploration constructs linked functional, transition, and operational knowledge, which is subsequently refined through group-wise trajectory comparison and structured editing. During task execution, function-based retrieval selects a fine-grained function and its linked transition path and operational knowledge once at task start to condition the frozen policy.

Environment-Grounded Task Synthesis. Environment-grounded task synthesis derives supervision from observed interactions. OS-Genesis (Sun et al., 2025) retrospectively generates tasks from exploratory interactions and constructs trajectories for policy training, while AutoPlay (Ramrakhya et al., 2026) synthesizes executable and verifiable tasks from exploration trajectories for demonstration collection and reinforcement learning. In these methods, generated tasks are used for data collection or policy optimization, whereas GUI-SKE uses pseudo-tasks as weak supervision for refining external knowledge without parameter updates.

3 METHOD

GUI-SKE adapts GUI agents to unseen applications by constructing and refining external knowledge while keeping model parameters fixed. Figure 1 summarizes structured knowledge construction, knowledge update, and subsequent knowledge-guided execution. Adaptation is performed once per application, after which the resulting knowledge is reused for subsequent tasks.

3.1 PRELIMINARIES

We consider task-conditioned GUI interaction in an unseen application. Let $\mathcal{S}$, $\mathcal{O}$, $\mathcal{A}$, and $\mathcal{G}$ denote the latent state space, observation space, atomic GUI action space, and task instruction space, respectively. The environment dynamics are described by an unknown transition kernel $\mathcal{T}$.

At time step t, the agent receives an observation $o_t = (I_t, U_t) \in \mathcal{O}$, where I_t is the GUI screenshot and U_t contains structured UI information extracted from the accessibility tree. The agent also has access to a static component-level functional prior $\mathcal{I}_{\text{act}}$, extracted from manifest metadata in our Android implementation. This prior provides coarse functional cues but does not specify runtime state transitions or element-level interaction rules.

Given a task instruction $g \in \mathcal{G}$ and external knowledge base $\mathcal{K}$, the agent selects an action according to

$$a_t \sim \pi_{\theta,\mathcal{K}}(\cdot \mid o_t, g), \tag{1}$$

where $\pi_{\theta,\mathcal{K}}$ denotes the multimodal policy with parameters θ, conditioned on $\mathcal{K}$. Dependence on the interaction history is omitted for brevity. The environment transitions from $s_t \in \mathcal{S}$ to $s_{t+1} \sim \mathcal{T}(\cdot \mid s_t, a_t)$ and returns the next observation. Latent states are used for formalization; in implementation, previously visited interfaces are revisited by replaying recorded action sequences.

During adaptation, the agent constructs and refines application-specific knowledge $\mathcal{K}$ from environment interactions and the static functional prior, while θ remains fixed. This process does not use evaluation-task instructions, reference trajectories, or benchmark-provided success evaluators. The resulting knowledge is reused across subsequent tasks within the same application without further updates.

3.2 Environment Exploration

To acquire environment-specific interaction evidence, GUI-SKE performs two-level exploration. Global exploration discovers page-level structures and navigation paths, while local exploration probes element-level actions and their effects on the terminal pages reached during global exploration. The overall exploration procedure is provided in Appendix B.1.

3.2.1 Global Exploration

Global exploration uses breadth-first search (BFS) over page states to explore the application's page structure. Following SceneDroid (Zhang et al., 2023), we distinguish page states by component types, resource identifiers, and UI hierarchy, ignoring text changes that leave the structure unchanged. Starting from the initial state s_0, a frozen multimodal model filters candidate navigation elements using the current observation and activity prior $\mathcal{I}_{\text{act}}$, following GUI-Explorer (Xie et al., 2025), yielding U_{explore}. For each $u \in U_{\text{explore}}$, the agent revisits the current page by replaying its recorded navigation path and executes the corresponding navigation action. Only previously unencountered successor states are added to the BFS queue. A reached state is designated as a terminal state s_E when the maximum depth is reached or $U_{\text{explore}} = \emptyset$. Each trajectory from s_0 to s_E is recorded as a global path:

$$p_i = \big((s_0, a_0), (s_1, a_1), \ldots, (s_{E_i-1}, a_{E_i-1}), s_{E_i}\big). \tag{2}$$

For subsequent knowledge construction, each state in the path is represented to the model by its corresponding observation $o_t = (I_t, U_t)$. The collection of these paths forms the global exploration set $\mathcal{P}_{\text{global}}$, recording the observed page structure and navigation paths. Under the same exploration budget, BFS discovers 29.5% more unique states than DFS in our comparison, motivating its use as the global exploration strategy; see Appendix E.7.

3.2.2 Local Exploration

Local exploration collects element-level action–effect evidence on the terminal pages identified during global exploration. Restricting probing to these pages controls interaction cost but may omit operational evidence on intermediate pages. For each terminal state s_E, the agent obtains its observation $o_E = (I_E, U_E)$ and selects a subset of elements $U_{\text{local}} \subseteq U_E$ for interaction probing. For each element $u \in U_{\text{local}}$, the agent applies candidate atomic actions $a \in \mathcal{A}$. Before each probe, the agent revisits the terminal page by replaying its recorded navigation path and obtains the current pre-action observation o_{pre}. After executing the instantiated action, it obtains the post-action observation o_{post}. A frozen multimodal model Φ_θ compares the pre- and post-action observations to detect interface changes and describe the inferred effect:

$$(\delta, e) = \Phi_\theta(o_{\text{pre}}, u, a, o_{\text{post}}), \tag{3}$$

where $\delta \in \{0, 1\}$ indicates whether an observable interface change is detected, and e describes the inferred effect. Each interaction j is represented as

$$\xi_j = (o_{\text{pre},j}, u_j, a_j, o_{\text{post},j}, \delta_j, e_j). \tag{4}$$

For each terminal page, local exploration collects a set of interaction samples $k_i = \{\xi_j\}_{j=1}^{N_i}$, where N_i is the number of probed interactions. Aggregating these page-level sample sets yields $\mathcal{P}_{\text{local}} = \{k_i\}_{i=1}^{M}$, which contains element-level interaction evidence collected across M terminal pages.

3.3 Structured Knowledge Base Construction

We organize the global paths $\mathcal{P}_{\text{global}}$ and local interaction evidence $\mathcal{P}_{\text{local}}$ into linked functional, transition, and operational knowledge components. Functional categories group fine-grained functions, each of which indexes its associated navigation path and page-specific operational evidence. This organization supports function-based retrieval and provides explicit units for component-wise feedback and knowledge editing. An example of retrieved knowledge is provided in Appendix A, and the complete construction procedure is summarized in Appendix B.2.

3.3.1 TRANSITION KNOWLEDGE CONSTRUCTION

Transition knowledge $\mathcal{K}_T$ represents observed navigation paths derived from $\mathcal{P}_{\text{global}}$. For each path p_i, the frozen multimodal model produces a fine-grained function descriptor f_i and a textual path representation m_i:

$$(f_i, m_i) = \Phi_\theta(p_i). \tag{5}$$

Here, f_i serves as the semantic key linking the navigation path to its functional and operational knowledge, while m_i summarizes the corresponding navigation procedure. The collection of these function–path pairs forms $\mathcal{K}_T$.

3.3.2 OPERATIONAL KNOWLEDGE CONSTRUCTION

Operational knowledge $\mathcal{K}_O$ records element-level action–effect evidence collected in $\mathcal{P}_{\text{local}}$. For each page-level sample set k_i, elements are represented using semantic and structural attributes such as class name, text, and content description; when these are unavailable, the frozen multimodal model derives a textual descriptor from the element and page context. SoM indices are used only for runtime grounding and are not stored as persistent element identities.

We aggregate the samples associated with each page into an operational entry

$$\omega_i = \{(u, A_{\text{chg}}(u), A_{\text{no-chg}}(u), \mathcal{E}(u)) \mid u \in U_{\text{local},i}\}, \tag{6}$$

where $A_{\text{chg}}(u)$ and $A_{\text{no-chg}}(u)$ record probed actions with and without detected interface changes, and $\mathcal{E}(u)$ stores the corresponding action–effect pairs. These records are page-specific interaction evidence rather than universal application rules. The resulting operational entries form $\mathcal{K}_O$ and are linked to transition entries whose paths reach the same terminal page.

3.3.3 FUNCTIONAL KNOWLEDGE CONSTRUCTION

Functional knowledge $\mathcal{K}_F$ organizes fine-grained functions into broader semantic categories for retrieval and pseudo-task generation. For each function descriptor f_i, the frozen multimodal model infers a broader functional category using its path representation m_i, terminal observation o_{E_i}, and activity prior $\mathcal{I}_{\text{act}}$. Functions assigned to the same semantic category are grouped together, forming the functional knowledge base $\mathcal{K}_F$.

Thus, each fine-grained function connects a semantic category with its observed navigation path and page-specific operational evidence. During refinement, edits target specific knowledge entries; linked entries remain unchanged unless explicitly revised.

3.4 KNOWLEDGE UPDATE

GUI-SKE refines structured external knowledge using weak supervision from pseudo-tasks while keeping model parameters fixed. Its linked functional, transition, and operational components provide explicit units for component-wise evaluation and targeted revision. Prompts and structured outputs are provided in Appendix C.2.

3.4.1 PSEUDO-TASK GENERATION

To obtain supervision without annotated target-task trajectories, GUI-SKE derives pseudo-tasks and verification requirements from interaction evidence collected during adaptation. For each fine-grained function f_i, we retrieve its associated transition knowledge m_i, operational knowledge ω_i, and terminal observation o_{E_i}. Together, these components provide environment-grounded evidence about the function being accessed, the observed navigation procedure, and the observed element-level interactions.

Conditioned on this evidence and the activity prior $\mathcal{I}_{\text{act}}$, the frozen multimodal model generates a pseudo-task instruction g_i' and a corresponding verification requirement y_i':

$$(g_i', y_i') = \Phi_\theta(f_i, m_i, \omega_i, o_{E_i}, \mathcal{I}_{\text{act}}). \tag{7}$$

The verification requirement specifies observable conditions associated with task completion, which are evaluated from runtime GUI observations containing both screenshots and structured UI information from the accessibility tree.

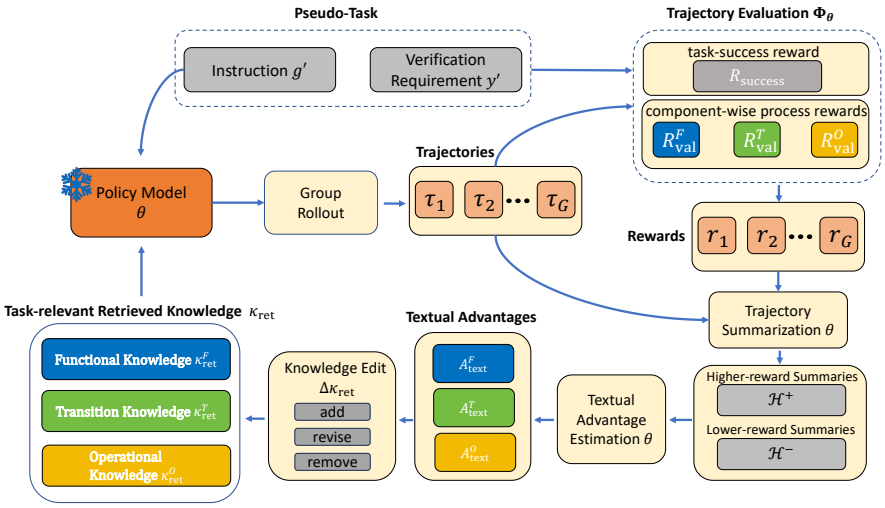

Figure 2: Group-wise trajectory comparison and knowledge editing in GUI-SKE. Multiple trajectories are evaluated with task-success and component-specific process feedback, producing component-wise textual signals for editing the corresponding knowledge entries.

The instruction–verification pairs are generated once after structured knowledge construction and remain fixed throughout refinement, so the same weak-supervision targets are used across update iterations. In a human evaluation, 93% of 100 sampled instruction–verification pairs satisfy both task feasibility and verification validity; see Appendix E.6.

3.4.2 Trajectory-Based Knowledge Refinement

Given a pseudo-task g', its verification requirement y', and initial state s_0, the agent retrieves the task-relevant structured knowledge $\mathcal{K}_{\text{ret}} = (\mathcal{K}^F_{\text{ret}}, \mathcal{K}^T_{\text{ret}}, \mathcal{K}^O_{\text{ret}})$. Conditioned on the same retrieved subset, the frozen policy generates multiple interaction trajectories. GUI-SKE uses task completion as the primary outcome feedback and component-wise process evaluation as auxiliary feedback, and derives textual refinement signals from relative trajectory quality.

3.4.3 Reward Design

Each trajectory is evaluated using a task-success reward together with component-wise process rewards:

$$r(\tau) = R_{\text{success}}(\tau, g', y') + \sum_{L \in \{F,T,O\}} \lambda_L R^L_{\text{val}}(\tau, g', \mathcal{K}_{\text{ret}}). \tag{8}$$

Here, $R_{\text{success}} = \Phi^{\text{ver}}_\theta$ is a binary task-completion score based on observable completion evidence, while $R^L_{\text{val}} = \Phi^{\text{eval},L}_\theta$ is a binary component-wise score evaluated against the executed trajectory. For $L \in \{F, T, O\}$, the corresponding evaluator assesses whether the observed execution supports the retrieved functional role, navigation procedure, and element-level interaction evidence, respectively. Both outcome and process evaluations are grounded in trajectory evidence that records the executed action and resulting GUI observation at each interaction step. In addition to the binary score, each component evaluator produces a textual rationale grounded in the observed trajectory, retaining the evidence underlying its assessment.

The task-success reward serves as the primary outcome signal, while the component-wise scores provide auxiliary signals for refinement. We choose the process-reward weights such that $\sum_{L \in \{F,T,O\}} \lambda_L < 1$. Since all reward terms are binary, this ensures that a trajectory with $R_{\text{success}} = 1$ receives a higher total reward than one with $R_{\text{success}} = 0$, regardless of the process scores. The binary component scores contribute to trajectory ranking, while their textual rationales provide additional evidence for subsequent comparison and refinement. Thus, a low component score is not treated by itself as a direct indication that the corresponding knowledge entry is incorrect.

3.4.4 GROUP-WISE KNOWLEDGE UPDATE

For each pseudo-task, the frozen policy generates G trajectories from the same initial state and retrieved knowledge subset. Following the group-relative comparison principle of Training-Free GRPO (Cai et al., 2025), we compute

$$\hat{A}_i = r_i - \bar{r}, \qquad \bar{r} = \frac{1}{G}\sum_{i=1}^{G} r_i. \tag{9}$$

To facilitate comparison of long trajectories, the frozen multimodal model summarizes each trajectory together with its evaluation feedback, retaining the task outcome, component-wise rationales, key interactions, state changes, and their relation to the retrieved knowledge. The summaries are divided into higher- and lower-reward groups according to the sign of $\hat{A}_i$; if all trajectories receive the same reward, the update for that group is skipped.

Otherwise, GUI-SKE compares the two groups separately for each knowledge component:

$$A^L_{\text{text}} = \Phi^{\text{cmp},L}_\theta\left(\mathcal{H}^+, \mathcal{H}^-, g', \mathcal{K}^L_{\text{ret}}\right), \qquad L \in \{F, T, O\}. \tag{10}$$

The resulting signal summarizes reward-associated execution differences and evaluation evidence relevant to component L. Because the retrieved knowledge is organized into functional, transition, and operational components, these signals can be associated with the corresponding task-relevant entries rather than applied as an undifferentiated rewrite of the full knowledge base. The comparison therefore uses both relative trajectory quality and component-specific rationales as evidence for refinement, without assuming that a low component score alone identifies an incorrect knowledge entry.

Conditioned on the component-wise refinement signals, the frozen model generates structured add, revise, or remove operations for the corresponding retrieved entries. These edits are deterministically applied to the full knowledge base while leaving unrelated entries unchanged.

3.5 INFERENCE

After adaptation, the agent retrieves task-relevant knowledge once at the beginning of each task. Given an instruction g, a frozen multimodal model selects the most relevant fine-grained function f from the functional knowledge. The agent then retrieves its associated navigation path m and operational knowledge ω. The resulting context $\mathcal{K}_g = \{f, m, \omega\}$ conditions the frozen policy throughout task execution. For steps not covered by the retrieved knowledge, the agent relies on the base policy. If no relevant function is identified, the agent proceeds without external knowledge. No further retrieval or knowledge updates are performed during execution.

4 EXPERIMENTS

4.1 DATASETS AND EVALUATION SETUP

We evaluate GUI-SKE on three mobile GUI benchmarks: AndroidWorld (Rawles et al., 2024), SPA-Bench (Chen et al., 2024), and our proposed ProMobile-Bench. We use GPT-4o as the primary base model and report task success rate as the main evaluation metric. For the main results, we average three independent full evaluations, each rerunning the complete adaptation pipeline from scratch, including environment exploration, knowledge construction, and knowledge refinement. We additionally evaluate GUI-SKE with Gemini 2.5 Flash on AndroidWorld as a second base-model setting.

AndroidWorld contains 116 tasks across 20 real-world Android applications with randomized task parameters and interface states. Following GUI-explorer (Xie et al., 2025), we evaluate SPA-Bench on the single-app English Level 3 setting, which contains multi-step workflows.

To further evaluate GUI agent adaptation on fine-grained editing operations and multi-step professional workflows, we introduce ProMobile-Bench. It contains 105 tasks across image, audio, and video editing applications. Implementation and compute details are provided in Appendix C, and ProMobile-Bench task composition and evaluation protocols are detailed in Appendix D.

Table 1: Task success rate on AndroidWorld. Training-free indicates no model-parameter updates in the target environment. GUI-SKE results are averaged over three independent full evaluations; results for prior methods are reported under their respective evaluation settings.

Agent	Input	Base Model	Training-free	Task Success Rate (%)
Human (Rawles et al., 2024)	screen	-	-	80.0
Ponder&Press (Wang et al., 2025)	screen	GPT-4o	×	34.5
Aria-UI (Yang et al., 2025)	screen	GPT-4o	×	44.8
AppAgent (Zhang et al., 2025)	screen + SoM	GPT-4o	√	14.9
M3A (Rawles et al., 2024)	screen + SoM	GPT-4o	√	40.5
AndroidGen (Lai et al., 2025)	screen + SoM	GPT-4o	√	46.8
GUI-explorer (Xie et al., 2025)	screen + SoM	GPT-4o	√	47.4
GUI-SKE	screen + SoM	GPT-4o	√	**52.3**
M3A (Rawles et al., 2024)	screen + SoM	Gemini 2.5 Flash	√	41.4
M3A + Code2World (Zheng et al., 2026)	screen + SoM	Gemini 2.5 Flash	√	50.9
GUI-SKE	screen + SoM	Gemini 2.5 Flash	√	**54.9**

Table 2: Task success rate on SPA-Bench single-app English Level 3 tasks. Training-free indicates no model-parameter updates in the target environment. The GUI-SKE result is averaged over three independent full evaluations. Baseline results are taken from GUI-explorer (Xie et al., 2025).

Agent	Input	Base Model	Training-free	Task Success Rate (%)
CogAgent (Hong et al., 2024)	screen	CogAgent	×	0.0
DigiRL (Bai et al., 2024)	screen	DigiRL	×	0.0
AutoDroid (Wen et al., 2024)	a11y tree	GPT-4o	√	12.0
SeeAct (Zheng et al., 2024)	screen + SoM	GPT-4o	√	12.0
AppAgent (Zhang et al., 2025)	screen + SoM	GPT-4o	√	14.0
MobileAgentV2 (Wang et al., 2024a)	screen + SoM	GPT-4o	√	20.0
M3A (Rawles et al., 2024)	screen + SoM	GPT-4o	√	42.0
GUI-explorer (Xie et al., 2025)	screen + SoM	GPT-4o	√	53.7
GUI-SKE	screen + SoM	GPT-4o	√	**56.7**

4.2 Main Results

AndroidWorld. With GPT-4o, GUI-SKE achieves an average task success rate of 52.3% over three independent full evaluations on AndroidWorld, exceeding the reported GUI-explorer result (47.4%) by 4.9 percentage points (Table 1). With Gemini 2.5 Flash, GUI-SKE achieves an average task success rate of 54.9% over three independent full evaluations, outperforming M3A (41.4%) and M3A + Code2World (50.9%) by 13.5 and 4.0 percentage points, respectively. These results support the effectiveness of GUI-SKE's non-parametric knowledge adaptation framework across both evaluated base models.

SPA-Bench. On the single-app English Level 3 split of SPA-Bench, GUI-SKE achieves an average task success rate of 56.7% over three independent full evaluations, exceeding the reported GUI-explorer result (53.7%) by 3.0 percentage points (Table 2). This result further supports the effectiveness of GUI-SKE in adapting agents to complex, multi-step application workflows.

ProMobile-Bench. On ProMobile-Bench, GUI-SKE achieves an average overall task success rate of 25.7% over three independent full evaluations, exceeding GUI-explorer (20.3%) by 5.4 percentage points (Table 3). It also achieves the highest average success rate among the evaluated methods across image, audio, and video editing tasks. These results demonstrate the effectiveness of GUI-SKE in specialized editing workflows, while the relatively low absolute success rates highlight the difficulty of these tasks.

4.3 Ablation Study

We conduct ablations on AndroidWorld using M3A with GPT-4o as the underlying agent implementation to examine the contributions of GUI-SKE's main components and knowledge composition.

Table 3: Task success rates on ProMobile-Bench. All methods use GPT-4o, and results are averaged over three independent runs over the full 105-task benchmark. For each run, Overall is computed over all 105 tasks rather than by averaging the three category-level success rates.

Method	Input	Image (%)	Audio (%)	Video (%)	Overall (%)
GPT-4o (Hurst et al., 2024)	screen + SoM	15.6	24.1	6.2	15.6
M3A (Rawles et al., 2024)	screen + SoM	16.3	25.3	7.4	16.5
GUI-explorer (Xie et al., 2025)	screen + SoM	21.1	29.9	8.6	20.3
GUI-SKE	screen + SoM	**25.9**	**39.1**	**11.1**	**25.7**

Table 4: Framework component ablation on AndroidWorld with GPT-4o. Structured Knowledge denotes the constructed external knowledge base; Group-wise Update adds group-wise knowledge refinement using the task-success reward; Component Feedback further adds the three component-wise process rewards.

Structured Knowledge	Group-wise Update	Component Feedback	Success (%)
×	×	×	40.5
√	×	×	43.1
√	√	×	48.3
√	√	√	**52.3**

Table 5: Knowledge composition ablation on AndroidWorld. We keep the constructed and updated knowledge base fixed and progressively provide functional, transition, and operational knowledge during inference.

Functional Knowledge	Transition Knowledge	Operational Knowledge	Success (%)
×	×	×	40.5
√	×	×	40.5
√	√	×	46.6
√	√	√	**52.3**

All ablations are conducted under the same fixed evaluation setting to support controlled comparisons among variants.

Table 4 shows progressive improvements from structured knowledge, group-wise refinement, and component-specific feedback, with the full framework reaching 52.3%. This indicates that both knowledge construction and refinement contribute to performance under this setting.

Table 5 shows that functional knowledge alone provides no improvement over the base agent, while transition and operational knowledge yield successive gains. This suggests that the latter two components provide complementary execution information beyond functional semantics.

5 Conclusion

We presented GUI-SKE, a non-parametric framework for GUI agent adaptation to unseen mobile environments. GUI-SKE organizes exploration-derived interaction evidence into linked functional, transition, and operational knowledge, and refines this knowledge using execution feedback from pseudo-tasks with verification criteria. Group-wise trajectory comparison combines task-success and component-wise process signals to guide edits to relevant knowledge entries while keeping model parameters fixed. Experiments across three mobile GUI benchmarks show improved task success rates under the evaluated settings.

Limitations. GUI-SKE relies on finite environment interactions, which may limit knowledge coverage and reliability, and introduces additional interaction and model-call costs during adaptation. Future work could improve these trade-offs through adaptive exploration, stronger verification, and more efficient interaction-budget allocation.

AI USE STATEMENT

We used generative AI tools solely to improve the clarity, grammar, and readability of the manuscript. All AI-assisted revisions were reviewed and verified by the authors. The authors take full responsibility for the final content of this paper.

REFERENCES

Hao Bai, Yifei Zhou, Mert Cemri, Jiayi Pan, Alane Suhr, Sergey Levine, and Aviral Kumar. Digirl: Training in-the-wild device-control agents with autonomous reinforcement learning. *Advances in Neural Information Processing Systems*, 37:12461–12495, 2024.

Yuzheng Cai, Siqi Cai, Yuchen Shi, Zihan Xu, Lichao Chen, Yulei Qin, Xiaoyu Tan, Gang Li, Zongyi Li, Haojia Lin, et al. Training-free group relative policy optimization. *arXiv preprint arXiv:2510.08191*, 2025.

Arthur Chen, Zuxin Liu, Jianguo Zhang, Akshara Prabhakar, Zhiwei Liu, Shelby Heinecke, Silvio Savarese, Victor Zhong, and Caiming Xiong. Test-time adaptation for llm agents via environment interaction. In *International Conference on Learning Representations*, volume 2026, pp. 39256–39288, 2026.

Jingxuan Chen, Derek Yuen, Bin Xie, Yuhao Yang, Gongwei Chen, Zhihao Wu, Li Yixing, Xurui Zhou, Weiwen Liu, Shuai Wang, et al. Spa-bench: A comprehensive benchmark for smartphone agent evaluation. In *NeurIPS 2024 Workshop on Open-World Agents*, 2024.

Yong Du, Yuchen Yan, Fei Tang, Zhengxi Lu, Chang Zong, Weiming Lu, Shengpei Jiang, and Yongliang Shen. Test-time reinforcement learning for gui grounding via region consistency. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 30593–30601, 2026.

Yue Fan, Handong Zhao, Ruiyi Zhang, Yu Shen, Xin Eric Wang, and Gang Wu. Gui-bee: Align gui action grounding to novel environments via autonomous exploration. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 33249–33266, 2025.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, et al. Cogagent: A visual language model for gui agents. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 14281–14290, 2024.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

Shilong Jin, Lanjun Wang, and Zhuosheng Zhang. Se-ga: Memory-augmented self-evolution for gui agents. *arXiv preprint arXiv:2605.16883*, 2026.

Quyu Kong, Xu Zhang, Zhenyu Yang, Nolan Gao, Chen Liu, Panrong Tong, Chenglin Cai, Hanzhang Zhou, Jianan Zhang, Liangyu Chen, et al. Mobileworld: Benchmarking autonomous mobile agents in agent-user interactive and mcp-augmented environments. *arXiv preprint arXiv:2512.19432*, 2025.

Hanyu Lai, Junjie Gao, Xiao Liu, Yifan Xu, Shudan Zhang, Yuxiao Dong, and Jie Tang. Androidgen: Building an android language agent under data scarcity. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2727–2749, 2025.

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. *arXiv preprint arXiv:2512.19396*, 2025.

Yanda Li, Chi Zhang, Wenjia Jiang, Wanqi Yang, Bin Fu, Pei Cheng, Xin Chen, Ling Chen, and Yunchao Wei. Appagent v2: Advanced agent for flexible mobile interactions. *arXiv preprint arXiv:2408.11824*, 2024.

Hongze Mi, Yibo Feng, WenJie Lu, Song Cao, Jinyuan Li, Yanming Li, Xuelin Zhang, Haotian Luo, Songyang Peng, He Cui, et al. Darwinian memory: A training-free self-regulating memory system for gui agent evolution. *arXiv preprint arXiv:2601.22528*, 2026.

Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, et al. Reasoningbank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, volume 2026, pp. 94327–94354, 2026.

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

Ram Ramrakhya, Andrew Szot, Omar Attia, Bogdan Mazoure, Anh Nguyen, Yuhao Yang, Zhe Gan, Harsh Agrawal, and Alexander Toshev. Scaling synthetic task generation for agents via exploration. In *International Conference on Learning Representations*, volume 2026, pp. 123630–123660, 2026.

Christopher 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. *arXiv preprint arXiv:2405.14573*, 2024.

Yucheng Shi, Wenhao Yu, Zaitang Li, Yonglin Wang, Hongming Zhang, Ninghao Liu, Haitao Mi, and Dong Yu. Mobilegui-rl: Advancing mobile gui agent through reinforcement learning in online environment. *arXiv preprint arXiv:2507.05720*, 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, 2024a.

Luyuan Wang, Yongyu Deng, Yiwei Zha, Guodong Mao, Qinmin Wang, Tianchen Min, Wei Chen, and Shoufa Chen. Mobileagentbench: An efficient and user-friendly benchmark for mobile llm agents. *arXiv preprint arXiv:2406.08184*, 2024b.

Yiqin Wang, Haoji Zhang, Jingqi Tian, and Yansong Tang. Ponder & press: Advancing visual gui agent towards general computer control. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 1461–1473, 2025.

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, 2025.

Leyang Yang, Ziwei Wang, Xiaoxuan Tang, Sheng Zhou, Dajun Chen, Wei Jiang, and Yong Li. Probench: Benchmarking gui agents with accurate process information. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 27547–27555, 2026.

Yuhao Yang, Yue Wang, Dongxu Li, Ziyang Luo, Bei Chen, Chao Huang, and Junnan Li. Aria-ui: Visual grounding for gui instructions. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 22418–22433, 2025.

Chaoyun Zhang, Shilin He, Jiaxu Qian, Bowen Li, Liqun Li, Si Qin, Yu Kang, Minghua Ma, Guyue Liu, Qingwei Lin, et al. Large language model-brained gui agents: A survey. *arXiv preprint arXiv:2411.18279*, 2024.

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.

Xiangyu Zhang, Lingling Fan, Sen Chen, Yucheng Su, and Boyuan Li. Scene-driven exploration and gui modeling for android apps. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1251–1262. IEEE, 2023.

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

Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. Synapse: Trajectory-as-exemplar prompting with memory for computer control. *arXiv preprint arXiv:2306.07863*, 2023.

Yuhao Zheng, Li'an Zhong, Yi Wang, Rui Dai, Kaikui Liu, Xiangxiang Chu, Linyuan Lv, Philip Torr, and Kevin Qinghong Lin. Code2world: A gui world model via renderable code generation. *arXiv preprint arXiv:2602.09856*, 2026.

Sibo Zhu, Wenyi Wu, Kun Zhou, Stephen Wang, and Biwei Huang. Hybrid self-evolving structured memory for gui agents. *arXiv preprint arXiv:2603.10291*, 2026.

A Illustrative Case Study

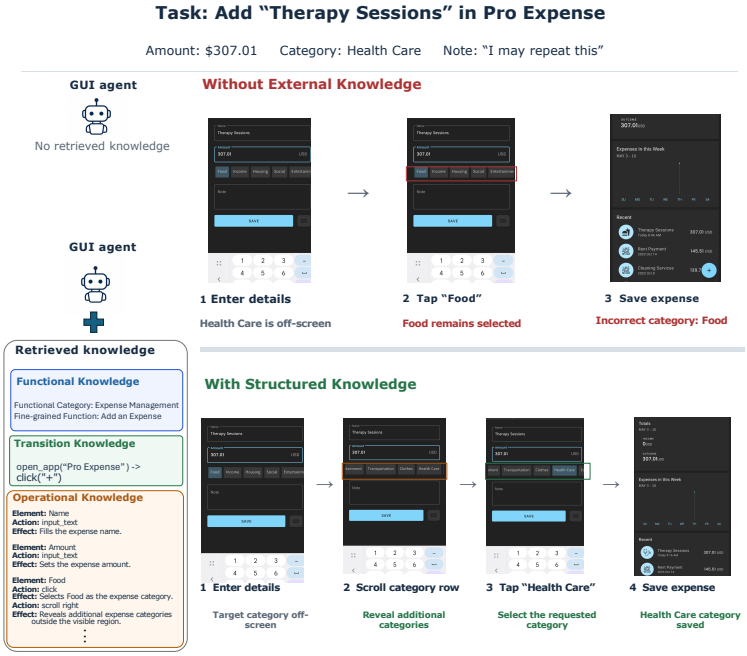

Figure 3: Illustration of knowledge-guided GUI task execution. Given the same expense-entry task, the agent without external knowledge selects an incorrect category, whereas the agent equipped with GUI-SKE's structured knowledge reveals the off-screen target category through horizontal scrolling and selects it correctly.

Figure 3 illustrates how structured knowledge supports GUI task execution in an unseen application. The task requires adding an expense with the category *Health Care*, which is initially outside the visible region of the category selector. Without external knowledge, the agent selects the visible *Food* category and saves the expense with an incorrect category.

With GUI-SKE, the agent selects the fine-grained function *Add an Expense* under the functional category *Expense Management*, and retrieves its linked transition and operational knowledge. These linked components provide the navigation procedure and page-specific interaction evidence used to guide execution. In particular, the operational knowledge indicates that horizontal scrolling on the category selector reveals additional options. Using this information, the agent reveals the previously hidden *Health Care* option, selects it, and saves the expense under the requested category. This example illustrates how structured environment-specific knowledge can organize complementary interaction evidence to support GUI operations that are not immediately apparent from the visible interface.

B Algorithmic Details and Pseudocode

B.1 Two-Level Environment Exploration

Algorithm 1 summarizes global exploration and local interaction probing. Paths are ordered sequences, with $\|$ denoting sequence concatenation. Following SceneDroid (Zhang et al., 2023), we define a structural signature $\sigma(o)$ from component types, resource identifiers, and UI hierarchy, while ignoring text-only changes. Previously visited pages are revisited by replaying their recorded

navigation paths from the initial state. Local sample sets retain their association with the page on which they were collected.

Algorithm 1 Two-Level Environment Exploration

Require: Initial state s_0, activity prior $\mathcal{I}_{\text{act}}$, maximum BFS depth D, atomic action set $\mathcal{A}$
Ensure: Global paths $\mathcal{P}_{\text{global}}$ and local interaction evidence $\mathcal{P}_{\text{local}}$

Obtain the initial observation o_0
Initialize FIFO queue $\mathcal{Q}$ with $(s_0, (), 0)$
Initialize visited-state set $\mathcal{V} \leftarrow \{\sigma(o_0)\}$
Initialize $\mathcal{P}_{\text{global}} \leftarrow \emptyset$, $\mathcal{S}_E \leftarrow \emptyset$
while $\mathcal{Q}$ is not empty **do**
    Dequeue (s, p, d) from $\mathcal{Q}$
    Revisit s by replaying p from s_0 and obtain $o = (I, U)$
    Filter candidate elements using the frozen multimodal model: $U_{\text{explore}} \leftarrow \Phi_\theta(o, \mathcal{I}_{\text{act}})$
    if $d = D$ or $U_{\text{explore}} = \emptyset$ **then**
        Add $p \| (s)$ to $\mathcal{P}_{\text{global}}$
        Add (s, p) to $\mathcal{S}_E$
        continue
    end if
    $\text{expanded} \leftarrow \text{false}$
    for each element $u \in U_{\text{explore}}$ **do**
        Revisit s by replaying p from s_0
        Execute navigation action a_u and obtain successor s' and observation o'
        if $\sigma(o') \notin \mathcal{V}$ **then**
            Add $\sigma(o')$ to $\mathcal{V}$
            Enqueue $(s', p \| ((s, a_u)), d + 1)$ in $\mathcal{Q}$
            $\text{expanded} \leftarrow \text{true}$
        end if
    end for
    if not expanded **then**
        Discard the branch without adding s to $\mathcal{S}_E$
        continue
    end if
end while
Initialize $\mathcal{P}_{\text{local}} \leftarrow \emptyset$
for each $(s_E, p_E) \in \mathcal{S}_E$ **do**
    Revisit s_E by replaying p_E from s_0 and obtain $o_E = (I_E, U_E)$
    Select elements $U_{\text{local}} \subseteq U_E$ for local probing
    Initialize page-level sample set $k \leftarrow \emptyset$
    for each element $u \in U_{\text{local}}$ **do**
        for each candidate action type $\bar{a}$ applicable to u **do**
            Revisit s_E by replaying p_E from s_0 and obtain current observation o_{pre}
            Instantiate $a \in \mathcal{A}$ from $\bar{a}$ using Φ_θ conditioned on o_{pre} and u
            Execute a on u and obtain observation o_{post}
            Detect observable interface change and infer the interaction effect: $(\delta, e) \leftarrow \Phi_\theta(o_{\text{pre}}, u, a, o_{\text{post}})$
            Add $\xi = (o_{\text{pre}}, u, a, o_{\text{post}}, \delta, e)$ to k
        end for
    end for
    Add k to $\mathcal{P}_{\text{local}}$, retaining its association with s_E
end for
return $\mathcal{P}_{\text{global}}, \mathcal{P}_{\text{local}}$

B.2 Structured Knowledge Construction

Algorithm 2 organizes the exploration evidence into functional, transition, and operational knowledge components. Each operational entry is constructed from a page-level sample set and linked to the transition entries that reach the same page. Functional categories organize fine-grained functions, which serve as semantic keys for connecting the corresponding transition and operational knowledge.

Algorithm 2 Structured Knowledge Construction

Require: Global paths $\mathcal{P}_{\text{global}}$, local interaction evidence $\mathcal{P}_{\text{local}}$, activity prior $\mathcal{I}_{\text{act}}$
Ensure: Functional knowledge $\mathcal{K}_F$, transition knowledge $\mathcal{K}_T$, and operational knowledge $\mathcal{K}_O$

Initialize $\mathcal{K}_F \leftarrow \emptyset, \mathcal{K}_T \leftarrow \emptyset, \mathcal{K}_O \leftarrow \emptyset$
for each page-level sample set $k_j \in \mathcal{P}_{\text{local}}$ **do**
  Identify its associated terminal page s_{E_j}
  Let $U_{\text{local},j}$ contain the elements probed in k_j
  Initialize operational entry $\omega_j \leftarrow \emptyset$
  for each element $u \in U_{\text{local},j}$ **do**
    Collect actions with detected observable changes:
    $A_{\text{chg}}(u) \leftarrow \{a \mid (o_{\text{pre}}, u, a, o_{\text{post}}, \delta, e) \in k_j,\ \delta = 1\}$
    Collect actions with no detected observable change:
    $A_{\text{no-chg}}(u) \leftarrow \{a \mid (o_{\text{pre}}, u, a, o_{\text{post}}, \delta, e) \in k_j,\ \delta = 0\}$
    Collect action–effect pairs:
    $\mathcal{E}(u) \leftarrow \{(a, e) \mid (o_{\text{pre}}, u, a, o_{\text{post}}, \delta, e) \in k_j\}$
    Add $(u, A_{\text{chg}}(u), A_{\text{no-chg}}(u), \mathcal{E}(u))$ to ω_j
  end for
  Add ω_j to $\mathcal{K}_O$, retaining its terminal-page association
end for
for each global path $p_i \in \mathcal{P}_{\text{global}}$ **do**
  Identify its terminal state s_{E_i} and recorded observation o_{E_i}
  Generate a function descriptor and textual navigation path:
  $(f_i, m_i) \leftarrow \Phi_\theta(p_i)$
  Add (f_i, m_i) to $\mathcal{K}_T$
  Link (f_i, m_i) to the operational entry associated with s_{E_i}
  Infer a broader functional category descriptor:
  $\tilde{F}_i \leftarrow \Phi_\theta(f_i, m_i, o_{E_i}, \mathcal{I}_{\text{act}})$
end for
Group functions with the same semantic category descriptor
Index the resulting categories by $c = 1, \ldots, C$
Let c_i denote the category assignment of f_i
Let F_c denote the descriptor of category c
Set $\mathcal{F}_c = \{f_i \mid c_i = c\}$ for each category c
Construct $\mathcal{K}_F = \{(F_c, \mathcal{F}_c)\}_{c=1}^{C}$
return $\mathcal{K}_F, \mathcal{K}_T, \mathcal{K}_O$

C Implementation and Reproducibility Details

C.1 Implementation Details

Observation format. The agent observes the current interface through a screenshot and structured UI information extracted from the accessibility tree. We use set-of-marks (SoM) annotations to associate interactable UI elements with visual indices overlaid on their screen regions. These indices allow the agent to specify target elements when generating actions. The runtime UI information is distinct from the static functional prior $\mathcal{I}_{\text{act}}$ extracted from the Android manifest.

Action space. We follow the AndroidWorld action schema (Rawles et al., 2024), representing actions as JSON-style operations with an `action_type` field and optional arguments. The element-level actions `click(index)` and `long_press(index)` operate on the element iden-

tified by its SoM index, while `input_text(text)` enters text into the focused field. The action `scroll(direction)` scrolls in one of four directions: `up`, `down`, `left`, or `right`. During local probing, the frozen multimodal model instantiates action arguments based on the target element and current page context.

For navigation, `navigate_home()` and `navigate_back()` send the corresponding key events, `keyboard_enter()` sends the enter key, and `open_app(app_name)` launches an application. The remaining actions are `wait()` for loading or interface transitions, `status(goal_status)` for reporting task status, and `answer(text)` for returning an answer. We use `unknown()` as a no-op for invalid or internal-error cases. At each execution step, the policy selects an action using the current observation, task instruction, and retrieved knowledge when available.

Hyperparameters. We set the maximum global exploration depth to $D = 5$. For knowledge refinement, we sample $G = 3$ trajectories per pseudo-task and perform $E = 3$ update epochs. The task-success reward has weight 1.0, while the functional, transition, and operational components of the process reward each have weight 0.2:

$$\lambda_F = \lambda_T = \lambda_O = 0.2. \tag{11}$$

The task-success reward and each process reward component are binary.

C.2 Knowledge Update Prompts and Structured Outputs

C.2.1 Pseudo-Task Generation

For each explored path, GUI-SKE forms an evidence bundle using the corresponding fine-grained function, transition knowledge, linked terminal-page operational knowledge, terminal GUI observation, and activity-level functional prior. The pseudo-task generator produces one executable task together with structured verification requirements. The verification requirements describe observable task-completion states rather than a prescribed action sequence.

Pseudo-Task Generation Prompt

```
You are a pseudo-task generator for GUI environment adaptation.

Given environment evidence associated with ONE explored GUI function
and path, generate exactly ONE executable GUI task together with
structured verification requirements that can later be used to
determine whether the task has been successfully completed.

The generated task and verification requirements will be used as weak
supervision for refining external GUI knowledge.

Rely on the provided environment evidence when constructing the task
and its verification requirements.

====================
INPUT
====================

Application:
{app_name}

Fine-grained function:
{fine_grained_function}

Transition knowledge:
{transition_knowledge}

Operational knowledge:
{operational_knowledge}

Terminal GUI observation:
- Screenshot: {terminal_screenshot}
```

```
- Accessibility tree: {terminal_accessibility_tree}

Activity-level functional prior:
{activity_prior}

====================
TASK GENERATION
====================

1. Generate exactly ONE task corresponding to the provided
   fine-grained function and its associated GUI evidence.

2. The task should be supported by the observed application
   functionality, navigation procedure, available UI elements, and
   element-level interaction effects.

3. Do not invent UI elements, entities, application capabilities,
   prerequisite states, or interaction effects that are not supported
   by the provided evidence.

4. The task must be executable using the interaction capabilities
   represented in the provided transition and operational knowledge.

5. Do not combine independently observed capabilities into a more
   complex task unless the provided evidence indicates that they
   belong to the same executable workflow.

6. Prefer a concrete task whose successful completion produces an
   observable change or state in the GUI.

====================
VERIFICATION REQUIREMENTS
====================

Generate structured verification checks that characterize successful
TASK COMPLETION.

The verification checks should describe observable GUI states rather
than prescribe how the task should be executed.

Do NOT use a required action, required navigation path, or required
step order as a success condition.

Executing an action such as click, input_text, or save does not itself
prove task completion. The verification requirements should instead
describe the observable GUI state expected after the requested
operation has been successfully carried out.

Each verification check consists of:

1. a target UI element specification; and
2. an expected observable state of that element.

When specifying a target UI element:

- Prefer accessibility attributes supported by the provided GUI
  observations, including class name, resource ID, text, hint text,
  and content description.
- Include only attributes useful for identifying the relevant element.
- Do not invent element attributes that are absent from the evidence.
- If accessibility attributes alone are insufficient, a short semantic
  description grounded in the observed GUI may be used.

Allowed predicates are:
```

```
- exists
- not_exists
- text_equals
- text_contains
- checked_equals
- selected_equals
- enabled_equals
- editable_equals

The downstream verifier will inspect a fixed final window of the
execution trajectory containing screenshots, accessibility-tree
information, and executed actions.

Therefore, construct checks that provide observable evidence of task
completion at or near the end of execution.

The downstream verifier may jointly use screenshot and accessibility
information. A verification check does not need to specify which
modality should be used.

All generated verification checks are required for task success.

====================
OUTPUT
====================

Return only a valid JSON object following the specified schema.
Do not include additional explanations.
```

Output schema. The generator returns one pseudo-task and a set of structured GUI-state predicates for subsequent trajectory verification.

Pseudo-Task Output Schema

```
{
  "app": "string",
  "task_requirement": "string",
  "verification_requirement": {
    "logic": "all",
    "checks": [
      {
        "check_id": "string",
        "target_ui_element": {
          "class_name": "string | null",
          "resource_id": "string | null",
          "text": "string | null",
          "hint_text": "string | null",
          "content_description": "string | null",
          "semantic_description": "string | null"
        },
        "predicate":
          "exists | not_exists | text_equals | text_contains |
           checked_equals | selected_equals | enabled_equals |
           editable_equals",
        "expected_value": "string | boolean | null"
      }
    ]
  }
}
```

C.2.2 Task-Success Evaluation

For each rollout, the task-success evaluator receives the generated pseudo-task, its structured verification requirements, and a fixed final window of the execution trajectory. The trajectory window

contains at most the last ten interaction steps. Each step contains the executed action together with the resulting GUI observation, represented by a screenshot and accessibility-tree information. The evaluator jointly uses the visual and structured UI evidence to determine whether the task has been successfully completed.

Task-Success Evaluation Prompt

```
You are a task-success evaluator for GUI agent trajectories.

Determine whether the provided execution trajectory contains sufficient
observable evidence that the requested task has been successfully
completed.

You are given:
- the task requirement;
- structured verification requirements; and
- a fixed final window of the trajectory containing at most the last
  10 interaction steps.

Each trajectory step contains:
- the executed action;
- the resulting  GUI screenshot; and
- the resulting accessibility-tree information.

====================
TASK
====================

{task_requirement}

====================
VERIFICATION REQUIREMENTS
====================

{verification_requirement}

====================
TRAJECTORY WINDOW
====================

{trajectory_window}

====================
EVALUATION RULES
====================

1. Determine whether all required verification conditions are supported
   by observable evidence in the provided trajectory window.

2. Use screenshots and accessibility-tree information jointly.
   Accessibility information provides structured element states and
   textual attributes, while screenshots may contain complementary
   visual evidence.

3. Task-completion evidence may appear in any observation within the
   provided trajectory window and does not have to remain visible in
   the final observation.

4. An executed action alone is not evidence of success.
   For example, clicking a save button or entering text does not prove
   completion unless the resulting GUI observations support the
   required task state.

5. Do not infer completion from application knowledge or from what an
   action would normally be expected to do.
```

```
   Base the judgment only on the provided trajectory evidence.

6. Do not combine mutually incompatible states from different steps to
   construct an artificial success state.

7. If later evidence explicitly indicates that an apparent completion
   state was not successfully committed or was subsequently
   ↪ invalidated,
   do not treat the earlier state alone as sufficient evidence.

8. If any required verification condition lacks sufficient observable
   evidence, judge the task as unsuccessful.

====================
OUTPUT
====================

Return only a JSON object with:
- "score": 1 if the task is successfully completed, otherwise 0;
- "reason": a concise explanation grounded in the observed trajectory.
```

C.2.3 Component-Wise Process Evaluation

The three component-wise process rewards evaluate the same execution trajectory but target different retrieved knowledge components. Specifically, the functional, transition, and operational evaluators assess whether the observed execution is consistent with the retrieved functional role, navigation procedure, and element-level interaction evidence, respectively. Each evaluator receives a fixed final window of the execution trajectory, where each step records the executed action together with the resulting GUI observation. For transition evaluation, we additionally provide the complete navigation action record, preserving the executed navigation sequence beyond the final trajectory window.

Component-Wise Process Evaluation Prompt

```
You are a component-wise evaluator for GUI agent trajectories.

Evaluate whether the observed execution provides evidence consistent
with the specified component of the retrieved structured knowledge.

You are given:
- the task requirement;
- the retrieved functional, transition, and operational knowledge;
- a fixed final window of the trajectory containing at most the last
  10 interaction steps; and
- for transition evaluation, the complete navigation action record
  from the execution trajectory.

Each trajectory step contains:
- the executed action;
- the resulting GUI screenshot; and
- the resulting accessibility-tree information.

====================
TASK
====================

{task_requirement}

====================
RETRIEVED KNOWLEDGE
====================

Functional knowledge:
```

```
{functional_knowledge}

Transition knowledge:
{transition_knowledge}

Operational knowledge:
{operational_knowledge}

====================
TRAJECTORY WINDOW
====================

{trajectory_window}

====================
NAVIGATION ACTION RECORD
====================

{navigation_action_record}

====================
EVALUATION TARGET
====================

Component:
{component_type}

Target knowledge:
{target_component_knowledge}

====================
COMPONENT-SPECIFIC CRITERION
====================

{component_criterion}

====================
GENERAL EVALUATION RULES
====================

1. Base the judgment on observable evidence from the provided
   trajectory.

2. Use the executed actions, screenshots, and accessibility-tree
   information jointly. For transition knowledge, also use the complete
   navigation action record to assess whether the retrieved navigation
   procedure is supported by the observed execution.

3. Evaluate only the specified knowledge component.
   Do not lower the score because another knowledge component appears
   incomplete or inconsistent.

4. Judge whether the observed trajectory provides evidence consistent
   with the target knowledge, rather than whether the overall task was
   successfully completed.

5. Do not assume that the retrieved knowledge is correct.
   If the observed execution contradicts the target knowledge, assign
   a negative judgment even when the trajectory itself is successful.

6. If the provided trajectory evidence does not contain sufficient
   support for the target component, do not infer consistency from
   prior application knowledge.

====================
```

```
OUTPUT
====================

Return only a JSON object with:
- "score": 1 if the observed execution supports the target knowledge
  component, otherwise 0;
- "reason": a concise explanation grounded in the observed trajectory.
```

Component-specific criteria. For functional knowledge, the evaluator examines whether the retrieved fine-grained function is semantically consistent with the task and the function exercised by the observed execution. For transition knowledge, it examines whether the retrieved navigation procedure is supported by the observed sequence of GUI states and actions. For operational knowledge, it examines whether the retrieved element–action effects are supported by the observed element-level interactions and resulting GUI state changes.

C.2.4 Component-Wise Trajectory Comparison

For each knowledge component $L \in \{F, T, O\}$, the comparator receives the pseudo-task, the corresponding retrieved knowledge component, and the higher- and lower-reward trajectory summaries. The summaries retain task outcomes, component-wise evaluation rationales, key interactions, observed state changes, and their relation to the retrieved knowledge. The comparator identifies reward-associated execution differences relevant to the target component and returns a structured component-wise comparison output A^L_{text}, containing an evidence summary, a group comparison, and a textual refinement signal. The comparison is evidence-based: a low component score alone is not treated as sufficient evidence that the corresponding knowledge entry is incorrect.

Component-Wise Trajectory Comparison

```
You are a component-wise comparator for GUI knowledge refinement.

Your goal is to compare higher-reward and lower-reward trajectory
summaries for the same pseudo-task and identify execution evidence
relevant to refining ONE specified component of the retrieved
structured knowledge.

You are given:
- the pseudo-task instruction;
- the target knowledge component type;
- the current retrieved knowledge component;
- summaries of higher-reward trajectories; and
- summaries of lower-reward trajectories.

Each trajectory summary contains:
- the task outcome;
- component-wise evaluation scores and textual rationales;
- key executed actions;
- observed GUI state changes; and
- evidence describing how the execution relates to the retrieved
  knowledge.

====================
TASK
====================
{task_requirement}

====================
TARGET COMPONENT
====================
Component type:
```

```
{component_type}

Current retrieved knowledge:
{target_component_knowledge}

====================
HIGHER-REWARD SUMMARIES
====================
{higher_reward_summaries}

====================
LOWER-REWARD SUMMARIES
====================
{lower_reward_summaries}

====================
COMPARISON RULES
====================

1. Focus only on the specified knowledge component.
   Do not propose changes to unrelated knowledge components.

2. Compare the two trajectory groups using observable execution
   evidence, including executed actions, GUI state changes, and
   component-wise evaluation rationales.

3. Do not assume that the current knowledge is correct or incorrect.
   Determine what is supported by the observed trajectories.

4. A low component score alone does not imply that the corresponding
   knowledge is incorrect. Use the accompanying trajectory evidence
   and textual rationale when interpreting the score.

5. Do not infer missing interactions, states, or application behavior
   from prior knowledge about the application.

6. Differences between higher- and lower-reward trajectories should
   be treated as evidence associated with execution quality, rather
   than proof that a particular knowledge entry caused task success
   or failure.

7. If the available trajectories do not provide sufficient evidence
   for a component-specific change, state this explicitly rather than
   forcing a refinement.

8. Summarize the relevant evidence in "evidence_summary", describe the
   reward-associated group differences in "group_comparison", and
   provide concise component-specific refinement guidance in
   "refinement_signal". Do not directly edit the knowledge entry.

====================
OUTPUT
====================

Return only a valid JSON object following this schema:

{
  "component_type": "functional | transition | operational",
  "evidence_summary": "string",
```

```
  "group_comparison": "string",
  "refinement_signal": "string"
}
```

C.2.5 STRUCTURED KNOWLEDGE EDITING

Given the full component-wise comparison output A^L_{text} and the corresponding retrieved knowledge component, the editor produces structured `add`, `revise`, or `remove` operations. The comparison output provides the evidence summary, group-level comparison, and refinement signal used to determine whether and how the target knowledge should be modified. Edits are restricted to entries relevant to the target component and are applied deterministically to the full knowledge base. If the available evidence does not support a knowledge change, the editor returns an empty edit list. For `revise` and `remove`, the target entry is selected directly from the provided retrieved knowledge component, allowing the corresponding existing entry to be updated without affecting unrelated entries.

Structured Knowledge Editing Prompt

```
You are a structured knowledge editor for GUI environment adaptation.

Your goal is to update ONE specified component of the structured GUI
knowledge using the full component-wise comparison output produced
from trajectory comparison.

You are given:
- the target knowledge component type;
- the current retrieved knowledge component; and
- the full component-wise comparison output.

You may generate structured add, revise, or remove operations.
If the available evidence does not justify a knowledge change,
return no edits.

====================
TARGET COMPONENT
====================
Component type:
{component_type}

Current retrieved knowledge:
{target_component_knowledge}

====================
COMPONENT-WISE COMPARISON OUTPUT
====================
{component_comparison_output}

====================
EDITING RULES
====================

1. Base every edit on the evidence and comparison results contained
   in the provided component-wise comparison output.
   Do not introduce application behavior, UI elements, navigation
   steps, or interaction effects that are not supported by the
   provided evidence.

2. Modify only the specified knowledge component and relevant entries.
   Do not change unrelated knowledge entries.
```

```
3. Use "revise" when an existing entry should be corrected or
   qualified based on the observed evidence.
   Preserve supported content whenever possible.

4. Use "add" when the comparison output provides evidence for relevant
   knowledge that is missing from the current component.
   Do not add information that is merely plausible but unobserved.

5. Use "remove" only when the available evidence indicates that an
   existing entry is misleading or contradicted by the observed
   executions.
   Lack of sufficient evidence alone is not a reason to remove an
   entry.

6. Do not determine an edit from the trajectory reward or the textual
   refinement signal alone.
   Consider the evidence summary, group comparison, and refinement
   signal jointly when deciding whether and how to edit the knowledge.

7. If the comparison output indicates insufficient or ambiguous
   evidence, or otherwise does not support a clear change, return an
   empty edit list.

8. Avoid duplicate additions.
   If the proposed information is already represented by an existing
   entry, revise that entry when appropriate instead of adding
   redundant knowledge.

9. When revising or removing an entry, select the target entry directly
   from the provided retrieved knowledge component and preserve its
   existing structural context.
   Do not invent a target entry that is not present in the provided
   knowledge.

====================
OUTPUT
====================

Return only a valid JSON object following this schema:

{
  "component_type": "functional | transition | operational",
  "edits": [
    {
      "operation": "add | revise | remove",
      "target_entry": "string | null",
      "updated_entry": "string | null",
      "reason": "string"
    }
  ]
}
```

For `add`, `target_entry` is null and `updated_entry` contains the new entry. For `revise`, both fields are provided. For `remove`, `target_entry` identifies the existing entry and `updated_entry` is null. An empty `edits` list indicates that no evidence-supported modification is proposed.

C.3 Compute Setup

Experiments use Android emulator workers, each configured with 8 CPU cores and 16 GB of memory. GPT-4o is the primary model used throughout GUI-SKE. For the additional AndroidWorld experiments with Gemini 2.5 Flash, all multimodal-model calls in both adaptation and task execution are replaced with Gemini 2.5 Flash. Model inference is performed through API calls. Throughout adaptation, model parameters remain fixed, and no local GPU training is required.

D ProMobile-Bench

D.1 Benchmark Overview and Task Composition

ProMobile-Bench evaluates GUI agents on specialized editing workflows across image, audio, and video applications. Table 6 compares its task scale and coverage with existing mobile GUI benchmarks.

Task composition. ProMobile-Bench contains 105 tasks: 49 in image editing, 29 in audio editing, and 27 in video editing. Tasks cover image cropping, filter and brightness adjustment, audio trimming and speed modification, video cutting and speed adjustment, and file export. The benchmark emphasizes fine-grained editing operations and multi-step professional workflows beyond general mobile operations.

Task difficulty. We divide the tasks into two difficulty levels. Easy tasks involve a single editing function or a short workflow, whereas hard tasks require combining multiple editing functions within a multi-step workflow. Easy and hard tasks account for 64.8% and 35.2% of the benchmark, respectively.

D.2 Evaluation Protocol

Task initialization and execution. Before each task, we clear the application cache and relaunch the application to obtain a clean initial state. All evaluated methods use the same initialization procedure and a maximum budget of 40 interaction steps. An episode terminates when the agent reports completion or reaches this budget.

Task completion is evaluated using two complementary verification stages. A task is counted as successful only if it passes both stages.

Activity-level verification. We record the foreground Android Activity at each execution step through ADB. Each task has an annotated target Activity corresponding to the editing interface required by the task. This check serves as a coarse process-level filter: the trajectory must reach the task-relevant editing context, while no specific navigation path is enforced.

Artifact-level verification. For each task, we pre-collect a ground-truth output artifact. After execution, the exported artifact is retrieved from the emulator through ADB and compared with the corresponding ground truth. The verification criterion depends on the task type. Content-oriented edits are evaluated using similarity- or feature-based comparisons, while task-specific properties such as file format, image dimensions, resolution, duration, and frame rate are checked directly when required by the instruction. Tolerance-based validation is applied where exact equality is inappropriate, for example due to compression, color-space conversion, or application-specific export behavior.

A task is counted as successful only if it passes both the activity-level and artifact-level verification.

Table 6: Comparison of mobile GUI benchmarks. Third-party Apps indicates coverage of non-system applications. Difficulty Levels indicates explicit task difficulty groupings. Pro-Domain Tasks indicates coverage of specialized application workflows beyond general daily mobile operations.

Benchmark	# Tasks	Third-party Apps	Difficulty Levels	Pro-Domain Tasks
AndroidWorld (Rawles et al., 2024)	116	√	√	×
SPA-Bench (Chen et al., 2024)	340	√	√	×
ProBench (Yang et al., 2026)	217	√	×	×
MobileWorld (Kong et al., 2025)	201	√	×	×
MobileAgentBench (Wang et al., 2024b)	100	√	√	×
ProMobile-Bench	105	√	√	√

E Additional Experiments and Efficiency Analysis

E.1 Adaptation Cost

We analyze the interaction requirements and measured overhead of the initial adaptation stage. This stage constructs and refines application-specific knowledge for reuse across subsequent tasks.

Interaction scaling. Global exploration depends on the number of visited pages and the candidate elements retained by activity-guided filtering. The maximum depth D bounds the search horizon, while candidate filtering restricts the branches explored on each page.

For local exploration, let $|\mathcal{S}_E|$ denote the number of terminal pages, $\bar{u}$ the average number of selected elements per page, and N_{type} the number of candidate atomic action types. The number of action probes is bounded by

$$O\left(|\mathcal{S}_E| \cdot \bar{u} \cdot N_{\text{type}}\right). \tag{12}$$

This bound counts probes; replaying recorded navigation paths to revisit pages incurs additional runtime overhead.

For N_{pseudo} pseudo-tasks, group size G, update epochs E, and maximum rollout length $T_{\max}$, the number of rollout interaction steps is bounded by

$$O\left(N_{\text{pseudo}} \cdot E \cdot G \cdot T_{\max}\right). \tag{13}$$

These bounds characterize interaction requirements as the exploration and rollout budgets increase. Actual runtime also depends on path replay, model-call latency, and emulator response time.

Measured adaptation overhead. Table 7 reports adaptation overhead using GPT-4o, averaged across all applications evaluated on AndroidWorld. Knowledge construction takes 0.67 hours and costs USD 3.80 per application on average, while knowledge update takes 1.90 hours and costs USD 18.75. The complete adaptation process takes 2.57 hours and costs USD 22.55. Knowledge update accounts for most of the measured overhead and involves repeated rollouts, verification, trajectory summarization, and group-wise comparison.

This cost is incurred during initial adaptation to an application. The resulting structured knowledge is reused across subsequent tasks without further knowledge updates or model-parameter optimization.

E.2 Inference Efficiency

Following GUI-explorer (Xie et al., 2025), we report inference-time efficiency using per-step time and API cost. We compare our measurements with the published overhead of GUI-explorer. For GUI-SKE, retrieval is performed once per task and its overhead is amortized over execution steps. Total time includes retrieval, ranking when applicable, policy inference, and other execution-time operations. API cost covers model calls during task execution, excluding the initial adaptation stage.

GUI-explorer retrieves and ranks knowledge at each step, with its ranking stage taking 28.462 seconds per step in the reported breakdown (Xie et al., 2025). GUI-SKE reuses the retrieved structured knowledge throughout execution without a separate ranking stage, avoiding both repeated retrieval and per-step ranking.

Table 7: Mean per-application adaptation overhead of GUI-SKE using GPT-4o across all applications evaluated on AndroidWorld. API costs are calculated from input and output token usage. Time is reported in hours.

Stage	Input Tokens	Output Tokens	Emulator Time	Wall-clock Time	API Cost
Knowledge construction	1.20M	0.08M	0.25	0.67	USD 3.80
Knowledge update	5.70M	0.45M	0.46	1.90	USD 18.75
Total adaptation	**6.90M**	**0.53M**	**0.71**	**2.57**	**USD 22.55**

Table 8: Per-step inference overhead. GUI-explorer values are taken from its published overhead analysis (Xie et al., 2025); GUI-SKE values are measured in our evaluation. For GUI-SKE, the one-time task-level retrieval overhead is amortized over execution steps.

Method	Retrieval Time (s/step) ↓	Total Time (s/step) ↓	API Cost (USD/step) ↓
GUI-explorer (reported)	7.12	66.378	0.064
GUI-SKE	2.19	35.631	0.041

As shown in Table 8, GUI-SKE has an amortized retrieval overhead of 2.19 seconds per step and a total overhead of 35.631 seconds per step, compared with the reported 7.12 and 66.378 seconds for GUI-explorer. Its API cost is USD 0.041 per step, compared with the reported USD 0.064. These measurements indicate lower per-step overhead under the respective evaluation settings.

E.3 Evaluation with a Weaker Static Functional Prior

We examine the effect of removing intent-filter information from the static functional prior on AndroidWorld. The weaker prior retains Activity information from the same Android manifest, providing coarse component-level functional cues. Runtime UI structures, navigation paths, and element-level interaction evidence are still acquired through environment exploration.

As shown in Table 9, removing intent-filter information reduces task success from 52.3% to 51.7%, a decrease of 0.6 percentage points. This comparison evaluates sensitivity to intent-filter information; both settings retain a static functional prior.

E.4 Effect of Removing the Activity Prior from Global Exploration

We further examine the role of the Activity-level functional prior in global exploration by removing it from candidate-element filtering. In this setting, the frozen multimodal model selects candidate exploration elements using only the current UI observation, while subsequent knowledge construction and refinement follow the default configuration.

As shown in Table 10, removing the Activity prior from global exploration reduces task success from 52.3% to 46.6%, a decrease of 5.7 percentage points. This result supports the utility of Activity-guided candidate filtering for retaining useful exploration branches under a fixed interaction budget.

E.5 Knowledge Base and Refinement Statistics

Knowledge-base scale. Table 11 summarizes the structured knowledge constructed on AndroidWorld. Across the 20 applications, GUI-SKE constructs 67 functional categories and 326 fine-grained functions, corresponding to 3.4 and 16.3 entries per application on average. The 326 fine-grained functions are associated with the same number of transition entries and pseudo-tasks, while the operational knowledge contains 1,763 element-level records. The larger number of operational records reflects the finer granularity of page-specific element–action interaction evidence.

Knowledge-edit distribution. We further analyze the structured edit operations produced during knowledge refinement. As shown in Table 12, `revise` accounts for 92.4% of all edits, while `add` and `remove` account for 4.9% and 2.7%, respectively. Thus, knowledge evolution primarily

Table 9: Effect of removing intent-filter information while retaining Activity information in the static prior. The comparison uses the fixed AndroidWorld evaluation setting of the main-text ablation study.

Setting	Task Success Rate (%)
GUI-SKE	52.3
GUI-SKE (Weak Prior)	51.7

Table 10: Effect of removing the Activity prior from global exploration on AndroidWorld. The comparison uses the same fixed evaluation setting.

Setting	Task Success Rate (%)
GUI-SKE	52.3
GUI-SKE w/o Activity prior in exploration	46.6

adjusts existing entries rather than extensively adding or removing knowledge. These frequencies characterize how the refinement mechanism operates, but do not by themselves measure knowledge quality, for which natural-language knowledge does not provide a unified objective criterion.

Retrieval granularity. GUI-SKE retrieves a matched fine-grained function together with its linked transition and operational knowledge as a single task-relevant context, rather than retrieving the three components independently. Therefore, separate component-wise retrieval frequencies are not defined under the current retrieval mechanism.

E.6 Evaluation of Pseudo-task Generation

Human evaluation of pseudo-task validity. We evaluate the quality of generated pseudo-tasks by randomly sampling 100 instruction–verification pairs from those generated across 20 applications. Each sample is manually assessed using two criteria. *Task feasibility* checks whether the task goal is achievable through a valid GUI interaction sequence in the corresponding application environment. For feasible tasks, *verification validity* checks whether the associated verification requirement is consistent with the task goal and sufficiently captures the conditions needed to determine task completion. A pseudo-task is considered valid only if it satisfies both criteria. This manual evaluation is used solely as a post-hoc quality check and does not affect pseudo-task selection or subsequent knowledge refinement.

Among the 100 sampled pseudo-tasks, 93 satisfy both criteria, corresponding to a validity rate of 93%. Of the remaining seven samples, three contain infeasible task goals, mainly because generation introduces prerequisite states or entities that are absent from the current environment. For example, a generated task may require replying to a message that has not been received. The other four samples contain feasible task goals but incomplete verification requirements. For instance, the verification may check the entered contact information without confirming that the contact is saved, or omit the requested recurrence setting for a calendar event. These results show that most generated pseudo-tasks satisfy the human evaluation criteria, while the remaining failures mainly arise from invalid environmental prerequisites or incomplete verification conditions.

Effect of exploration-derived information. We further examine whether grounding pseudo-task generation in exploration-derived information improves its utility for subsequent knowledge refinement. We compare the full GUI-SKE generation procedure with a baseline that uses only the same static functional prior and initial-state information, without access to information obtained through environment exploration.

For a controlled comparison, both conditions start knowledge refinement from identical copies of the same constructed knowledge base. The only difference is the information provided during pseudo-task generation: GUI-SKE additionally uses exploration-derived structured knowledge and observations, whereas the baseline does not. Both conditions generate the same number of pseudo-tasks using the same generation model and configuration, and use identical rollout, reward, and group-

Table 11: Scale of the constructed knowledge base on AndroidWorld.

Knowledge Type	Total	Avg. per App
Functional categories	67	3.4
Fine-grained functions	326	16.3
Transition entries	326	16.3
Operational element records	1,763	88.2
Pseudo-tasks	326	16.3

Table 12: Distribution of knowledge-edit operations on AndroidWorld.

Edit Operation	Proportion (%)
`add`	4.9
`revise`	92.4
`remove`	2.7

wise knowledge refinement settings. The resulting knowledge bases are evaluated under the same downstream evaluation protocol. We match the number of generated pseudo-tasks and refinement configurations rather than realized token consumption, since different pseudo-tasks can induce trajectories of different lengths.

As shown in Table 13, using exploration-derived information during pseudo-task generation improves downstream task success from 44.8% to 52.3% under the matched refinement setting. This result supports the utility of grounding pseudo-task generation in interaction evidence collected from the target environment.

E.7 Comparison of Global Exploration Strategies

We compare BFS and DFS using 400 exploration steps per application, excluding backtracking and path replay for page revisitation. Both strategies use the same maximum depth, candidate-element filtering, state-identification rule described in Appendix B.1, and environment settings, differing only in search order. Unique-state counts are averaged across all evaluated applications.

Table 14 shows that BFS discovers more unique states under the same exploration-step budget, supporting its use for global exploration in the evaluated setting.

Table 13: Effect of exploration-derived information in pseudo-task generation on downstream task success after knowledge refinement.

Pseudo-task generation	Task Success Rate (%)
Without exploration-derived information	44.8
GUI-SKE	**52.3**

Table 14: Mean number of unique GUI states discovered under a 400-step exploration budget per application, excluding backtracking and path replay.

Strategy	Mean Unique GUI States
DFS	15.6
BFS	**20.2**

F A Surrogate Interpretation of Group-Wise Textual Advantage Estimation

This section provides a surrogate interpretation of group-wise textual advantage estimation. It explains how relative trajectory quality motivates knowledge refinement, without establishing gradient equivalence or convergence guarantees for GUI-SKE.

F.1 Fixed-Snapshot Surrogate

Before refinement, GUI-SKE generates a fixed pseudo-task set $\mathcal{G}'$ from the initially constructed knowledge base $\mathcal{K}^{(0)}$. At knowledge-update iteration k, the agent selects a pseudo-task $(g', y') \in \mathcal{G}'$ and retrieves a task-relevant subset $\mathcal{K}_{\text{ret}}^{(k)}$ from $\mathcal{K}^{(k)}$. The instruction g', verification requirement y', and retrieved knowledge remain fixed during rollout sampling, evaluation, and comparison.

For conceptual analysis, let the retrieved knowledge have an implicit semantic representation

$$\mathbf{z}_{\text{ret}}^{(k)} = \text{Enc}_{\text{sem}}\left(\mathcal{K}_{\text{ret}}^{(k)}\right), \tag{14}$$

where Enc_{sem} is a conceptual semantic encoding function. This representation describes how textual knowledge conditions the frozen policy; it is not explicitly constructed or optimized in GUI-SKE.

Consider a hypothetical differentiable family of trajectory distributions $p_{\mathbf{z}}(\tau \mid s_0, g')$ parameterized by this semantic representation. We suppress the fixed conditioning variables (s_0, g') and write $p_{\mathbf{z}}(\tau)$. At iteration k, define the reward function using the current knowledge snapshot:

$$R^{(k)}(\tau) = R_{\text{success}}(\tau, g', y') + R_{\text{process}}\left(\tau, g', \mathcal{K}_{\text{ret}}^{(k)}\right). \tag{15}$$

Holding this reward function fixed as $\mathbf{z}$ varies gives the surrogate

$$\widetilde{J}^{(k)}(\mathbf{z}) = \mathbb{E}_{\tau \sim p_{\mathbf{z}}}\left[\text{sg}\left(R^{(k)}(\tau)\right)\right], \tag{16}$$

where $\text{sg}(\cdot)$ denotes stop-gradient treatment of the reward. The reward can differ across trajectories, but its knowledge-dependent evaluation criteria remain tied to $\mathcal{K}_{\text{ret}}^{(k)}$. Thus, the surrogate isolates how changes in the trajectory distribution affect expected quality under fixed evaluation criteria.

Assuming parameter-independent support and regularity conditions that permit differentiation under the expectation, the score-function identity gives

$$\nabla_{\mathbf{z}} \widetilde{J}^{(k)}(\mathbf{z}) = \mathbb{E}_{\tau \sim p_{\mathbf{z}}}\left[\text{sg}\left(R^{(k)}(\tau)\right) \nabla_{\mathbf{z}} \log p_{\mathbf{z}}(\tau)\right]. \tag{17}$$

This expression motivates examining execution patterns associated with different reward levels. It does not establish that textual trajectory comparison estimates the score-function gradient. GUI-SKE neither computes this gradient nor explicitly optimizes the semantic representation.

F.2 Connection to Group-Wise Textual Refinement

In the actual algorithm, the frozen policy samples G trajectories under the same pseudo-task, initial state, and retrieved knowledge. Given their rewards $r_i = R^{(k)}(\tau_i)$, GUI-SKE computes

$$\bar{r} = \frac{1}{G}\sum_{i=1}^{G} r_i, \qquad \hat{A}_i = r_i - \bar{r}. \tag{18}$$

Here, $\hat{A}_i$ identifies each trajectory's performance relative to the group mean. It is used as a comparison signal, not as an unbiased policy-gradient advantage estimator. If all rewards are equal, all relative advantages are zero, and the knowledge update for this group is skipped.

For each trajectory τ_i, the component-wise evaluators additionally produce textual rationales $\boldsymbol{\rho}_i = (\rho_i^F, \rho_i^T, \rho_i^O)$, where each ρ_i^L records the trajectory evidence underlying the assessment of knowledge component L. The frozen multimodal model then summarizes each trajectory together with its evaluation feedback:

$$h_i = \Phi_\theta^{\mathrm{sum}}\left(\tau_i, r_i, \boldsymbol{\rho}_i, g', \mathcal{K}_{\mathrm{ret}}^{(k)}\right), \tag{19}$$

where h_i retains the task outcome, component-wise evaluation rationales, key interactions, state changes, and their relation to the retrieved knowledge. The summaries are partitioned into

$$\mathcal{H}^+ = \left\{h_i \mid \hat{A}_i \geq 0\right\}, \qquad \mathcal{H}^- = \left\{h_i \mid \hat{A}_i < 0\right\}. \tag{20}$$

For each knowledge component $L \in \{F, T, O\}$, the model compares trajectories with rewards at or above the group mean against those below it:

$$A_{\mathrm{text}}^L = \Phi_\theta^{\mathrm{cmp},L}\left(\mathcal{H}^+, \mathcal{H}^-, g', \left(\mathcal{K}_{\mathrm{ret}}^{(k)}\right)^L\right). \tag{21}$$

The resulting signal summarizes reward-associated execution differences and evaluation evidence relevant to the corresponding knowledge component. These differences provide evidence for refinement, but do not by themselves establish that a particular knowledge entry caused an execution failure.

The component-wise textual signals guide structured edits to the retrieved knowledge, which are merged into the full knowledge base. The surrogate perspective therefore motivates comparison based on trajectory quality, while the implemented update operates through rollout evaluation, rationale-aware summarization, textual comparison, and structured knowledge editing without gradient computation.