GUI-explorer: Autonomous Exploration and Mining of Transition-aware Knowledge for GUI Agent

Anonymous ACL submission

Abstract

GUI automation faces critical challenges in dynamic environments. MLLMs suffer from two key issues: misinterpreting UI components and outdated knowledge. Traditional fine-tuning methods are costly for app-specific knowledge updates. We propose GUI-explorer, a training-free GUI agent that incorporates two fundamental mechanisms: **(1) Autonomous Exploration of Function-aware Trajectory**. To comprehensively cover all application functionalities, we design a **Function-aware Task Goal Generator** that automatically constructs exploration goals by analyzing GUI structural information (e.g., screenshots and activity hierarchies). This enables systematic exploration to collect diverse trajectories. **(2) Unsupervised Mining of Transition-aware Knowledge**. To establish precise screen-operation logic, we develop a **Transition-aware Knowledge Extractor** that extracts effective screen-operation logic through unsupervised analysis the state transition of structured interaction triples (observation, action, outcome). This eliminates the need for human involvement in knowledge extraction. With a task success rate of 53.7% on SPA-Bench and 47.4% on AndroidWorld, GUI-explorer shows significant improvements over SOTA agents. It requires no parameter updates for new apps. All data and code will be publicly available on Github after acceptance.

1 Introduction

Automation in graphical user interfaces (GUIs) has rapidly advanced (Su et al., 2024). This progress is driven by foundational models like large language models (LLMs) (Touvron et al., 2023; Achiam et al., 2023; Yang et al., 2024a) and multimodal large language models (MLLMs) (Chen et al., 2024a; Hurst et al., 2024; Google, 2025). These innovations enable agents (Zheng et al., 2024; Zhang et al., 2023; Wang et al., 2024a) to handle tasks. They require no extensive fine-tuning or pretraining. This demonstrates their potential for diverse applications.

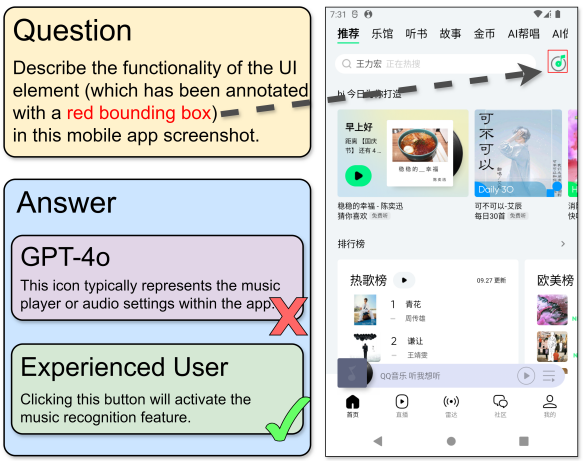

Figure 1: Comparison of GPT-4o and an user's interpretation of a UI element in QQ Music[3]. The red-bounded icon in the screenshot represents the music recognition feature, but GPT-4o misidentified it. This highlights the challenge of accurately interpreting UI elements in an ecosystem of diverse apps with distinct designs.

However, the practical deployment of these models faces significant challenges. These challenges stem from the long-tail distribution of app/website variants and their rapid iteration cycles. While core functionalities might appear similar across platforms, critical design divergences exist. For example: shopping cart features in Amazon.com and Temu[1] share similarities. In contrast, Pinduoduo[2] (China's dominant e-commerce platform) eliminates cart functionality entirely. This requires single-item purchases rather than batch checkout. Such inconsistencies extend beyond functionality to interface semantics. As shown in Figure 1, even advanced MLLMs such as GPT-4o (Hurst et al., 2024) can misinterpret the button's actual functionality. Human users familiar with the app, however,

[1] https://www.temu.com
[2] https://mobile.pinduoduo.com
[3] https://play.google.com/store/apps/details?id=com.tencent.qqmusic

correctly interpret it through learned interaction patterns. Compounding this challenge, apps/websites undergo frequent updates. Amazon Shopping alone released 30 version iterations in 2024[4]. This renders static model knowledge obsolete. Retraining or fine-tuning (M)LLMs for every change proves prohibitively expensive and latency-prone.

In this paper, we propose **Autonomous Exploration and Mining of Transition-aware Knowledge for GUI Agent (GUI-explorer)**. It synergizes two key components: **(1) Autonomous Exploration of Function-aware Trajectory**. To exhaustively cover all potential functions of target applications, we design a **Function-aware Task Goal Generator**. This module automatically constructs function-aware exploration goals by analyzing structural information of the GUI environment, including screenshots and activity lists from APK files. Through systematic exploration, we obtain diverse function-aware trajectories. **(2) Unsupervised Mining of Transition-aware Knowledge**. To establish precise screen-operation logic, we develop a **Transition-aware Knowledge Extractor**. This component extracts effective screen-operation logic through unsupervised analysis the state transition of structured interaction triples (observation, action, outcome). This eliminates the need for human involvement in knowledge extraction. Through multimodal state modeling that incorporates both visual patterns (e.g., element layouts) and semantic patterns (e.g., text labels), the extractor rigorously captures operation constraints and outcome dependencies, ultimately generating transition-aware knowledge with explicit action-effect correlations. Finally, by performing visual-semantic retrieval between current screen visuals and the knowledge vector store to construct Dynamic Guidance, it achieves two goals: suppressing the misinterpretation of UI components, and ensuring that action proposals align with actual UI states. This approach facilitates precise and goal-oriented prompt generation. These prompts guide the agent in effectively understanding and interacting with GUI elements.

Our main contributions are listed below:

- We propose GUI-explorer, a novel training-free agent that integrates two mechanisms: (1) Autonomous exploration of function-aware trajectory through environment-specific structural priors, and (2) Unsupervised mining of transition-aware knowledge that extracts atomic screen-operation logic from raw interaction traces.

- We conducted comprehensive evaluations of GUI-explorer across AndroidWorld and SPA-Bench benchmarks, our agent achieves 47.4% and 53.7% task success rates respectively, outperforming SOTA methods by 2.6%∼11.7% improvement. Through ablation studies, we verified that our framework's transition-aware knowledge integration approach reduces prior knowledge errors by 16.0%.

- We introduce a benchmark evaluating MLLMs' GUI understanding through 500 curated samples across 43 applications. Results reveal critical limitations in current models (15.2%∼22.8% prior knowledge inaccuracies).

2 Related Work

GUI Agents Modern GUI agents leverage MLLMs to interpret interface states and execute actions. SeeAct (Zheng et al., 2024) pioneers GPT-4V (OpenAI, 2023) for web task automation through visual understanding and HTML-guided action grounding. MobileAgentV2 (Wang et al., 2024a) implements multi-agent collaboration with memory units to track task progress and interface focus. M3A (Rawles et al., 2024) integrates ReAct-style (Yao et al., 2022) reasoning with Set-of-Mark (SoM) (Yang et al., 2023) visual annotations for Android device control, demonstrating zero-shot generalization across applications.

Exploration & Knowledge-aware Agents Autonomous exploration mechanisms vary in supervision requirements. AppAgent (Zhang et al., 2023) requires manually designed exploration tasks for knowledge acquisition, while AutoDroid (Wen et al., 2023) generates random action sequences for environment interaction. DigiRL (Zhou et al., 2024) employs reinforcement learning with Gemini-based (Google, 2025) trajectory filtering to collect successful demonstrations as training data.

Knowledge utilization strategies focus on experience retention and retrieval. CAT (Feng et al., 2024) employs retrieval-augmented generation with task-specific successful trajectories,

[4] https://www.apkmirror.com/uploads/?appcategory=amazon-shopping

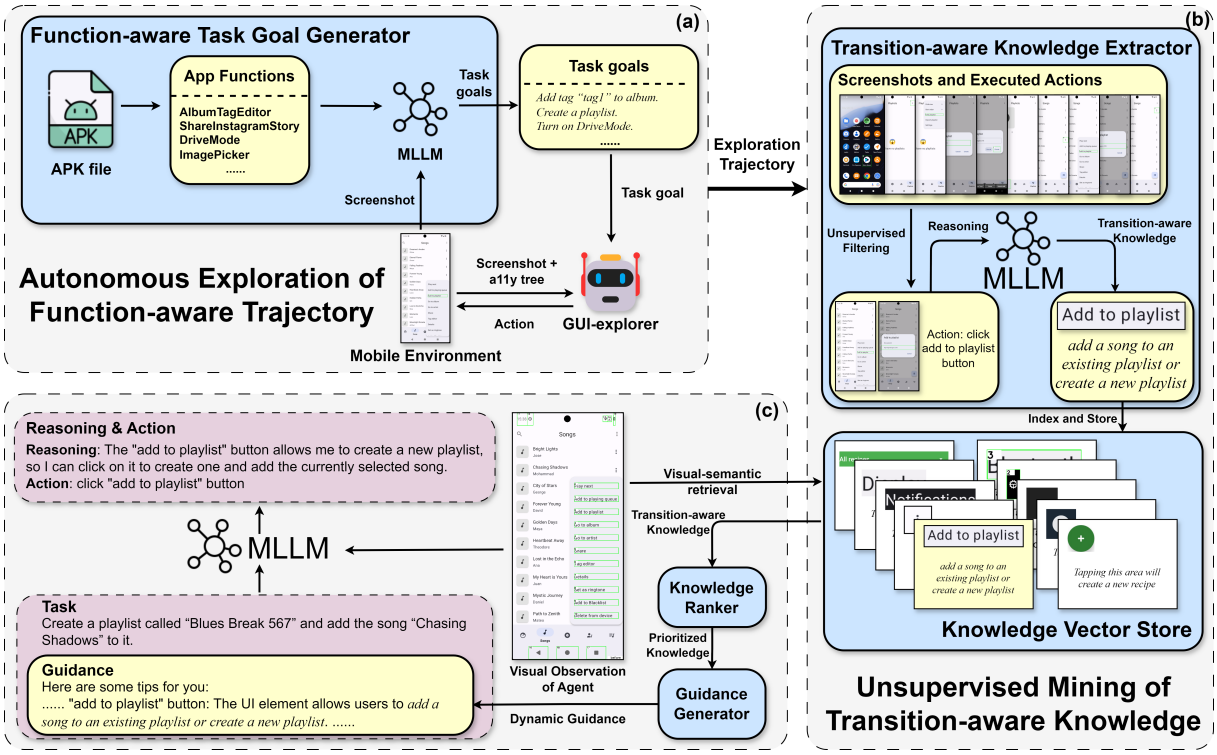

Figure 2: Overview of GUI-explorer. (a) Automatically constructing function-aware exploration goals by analyzing structural information from the GUI environment, followed by systematic exploration to collect diverse function-aware trajectories. (b) Extracting effective screen-operation logic through unsupervised analysis of structured interaction triples (observation, action, outcome), enabling unsupervised knowledge extraction. (c) Performing visual-semantic retrieval between screen visuals and the vector knowledge base to construct Dynamic Guidance achieves dual objectives: preventing UI misinterpretation and ensuring action proposals align with actual UI states.

though limited to pre-collected demonstrations. Synapse (Zheng et al., 2023) introduces trajectory-as-exemplar prompting with state abstraction to improve cross-task generalization. ICAL (Sarch et al., 2024) abstracts interaction traces into transferable knowledge through visual-language model summarization and human feedback.

While existing methods demonstrate progress, three critical limitations persist: (1) Exploration efficiency suffers from random action generation or manual task design; (2) Knowledge extraction relies on successful trajectories or human curation, limiting scalability; (3) Static knowledge bases struggle with rapidly evolving interfaces.

3 Autonomous Exploration and Mining of Transition-aware Knowledge for GUI Agent

As illustrated in Figure 2, GUI-explorer consists of two main components: autonomous exploration of function-aware trajectory and unsupervised mining of transition-aware knowledge. Building upon the dual components mentioned, we employ visual-semantic retrieval during the agent's task execution to extract relevant knowledge based on the current observation. This retrieval mechanism enables a dynamic knowledge integration process that enhances the agent's decision-making capabilities. Specifically, we construct task-specific guidance by synthesizing the retrieved knowledge with both the current task goal and observational data. This guidance framework facilitates sophisticated reasoning processes, allowing the agent to make more informed decisions while navigating complex task environments.

3.1 Autonomous Exploration of Function-aware Trajectory

The core of our method lies in autonomously generating diverse interaction trajectories without human supervision. This exploration is grounded in environment-specific structural priors. These priors suppress misinterpretations derived from MLLMs' obsolete domain priors. Algorithm 1 formalizes this process through two key components. First, anchor-guided task generation leverages interface semantics. Second, depth-first exploration incorporates state restoration mechanisms.

Algorithm 1: Autonomous Exploration of Function-aware Trajectory

Input: Environment E, max_branching_factor b, max_depth d, max_steps s

Function *Explore_DFS*($E, b, d, depth, task, s$)
  Task_Executor($E, task, s$);
  if *current_depth* $> d$ **then**
    return;
  current_state ← E.get_current_state();
  child_tasks ← Task_Generator(E, b);
  for $i = 0$ **to** $length(child\_tasks) - 1$ **do**
    if $i > 0$ **then**
      E.restore_to(current_state);
    Explore_DFS($E, b, d, depth + 1, child\_tasks[i], s$);

Function *Task_Generator*(E, k)
  anchors ← E.app_functions;
  p ← ConstructPrompt(E.observation, anchors);
  return *MLLM(p).sample_top_k(k)*;

Function *Task_Executor*($E, task, s$)
  for *round = 1* **to** s **do**
    action ← MLLM(task, E.observation);
    `/* We store the observation and action for knowledge vector store construction */`
    if *action == "END"* **then**
      return;
    E.step(action);

E.reset();
initial_state ← E.get_current_state();
tasks ← Task_Generator(E, b);
foreach *task in tasks* **do**
  Explore_DFS($E, b, d, 0, task, s$);
  E.restore_to(initial_state);

Given a target environment E, we first extract *Exploration Anchors* - structural primitives from E's ground-truth architecture. For mobile apps, functional modules declared in manifest files (e.g., "PaymentActivity"). These anchors serve as verifiable constraints during task generation, preventing MLLMs from proposing actions targeting non-existent components. The Task_Generator function constructs prompts (see Appendix D.1) containing current observation o_t and valid anchors, then samples up to k candidate tasks from MLLM outputs.

The exploration follows depth-first search (DFS) with configurable branching factor b and depth d. This strategy eliminates the first state restoration overhead when expanding child tasks. The elimination occurs because each branch naturally inherits the terminal state of its parent task. This differs from breadth-first search (BFS), which requires resetting to the parent state for each sibling task expansion. Starting from initial state $state_0$, each generated task initiates an exploration branch. After executing a task for up to s steps via Task_Executor, the environment rolls back to previous state $state_i$. This mechanism enables exhaustive traversal of interface pathways without manual reset. The executor terminates exploration branches under two conditions: when receiving an "END" action, or when reaching maximum steps. This balances thoroughness with computational efficiency.

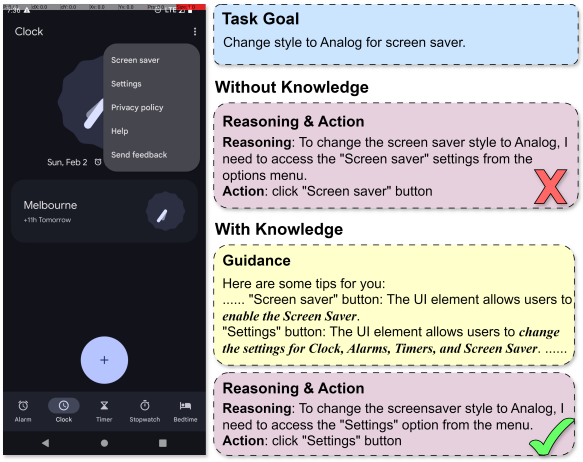

Figure 3: Without transition-aware knowledge as reliable prior information, MLLMs may fail to reason correctly due to outdated prior knowledge or diverse GUI designs.

This design achieves two critical properties: (1) **Semantic Grounding**: Anchors tether generated tasks to actual interface functions. (2) **Quadratic Coverage**: Each d-depth exploration with branching factor b yields $O(b^d)$ distinct trajectories, systematically capturing combinatorial interaction patterns.

3.2 Unsupervised Mining of Transition-aware Knowledge

The knowledge construction process focuses on mining atomic screen-operation logic. These logic are derived from exploration trajectories. Let $\xi = \langle o_1, a_1, ..., o_n, a_n \rangle$ denote an interaction trajectory. This trajectory is collected during autonomous exploration. We extract transition-aware GUI knowledge through a **Transition-aware Knowledge Extractor** function $F_{extract}$. This function operates on state-action transitions:

$$F_{extract} : (o_i, a_i, o_{i+1}) \rightarrow \{k_i : v_i\} \qquad (1)$$

where o_i and o_{i+1} represent consecutive observations, a_i denotes the action executed, and $\{k_i : v_i\}$ outputs a set of visual-semantic knowledge entries. Each entry consists of: (1) k_i: visual patch of the interacted UI element, (2) v_i: operational knowledge (e.g., "Clicking this button opens search history").

Unlike previous work (Zheng et al., 2023; Feng et al., 2024; Sarch et al., 2024; Qin et al., 2025), which requires successful trajectories for in-context learning or fine-tuning, our approach has different requirements. Specifically, we only need valid state transitions. Therefore, we implement a filtering mechanism termed **Transition Filtering** to filter out invalid state transitions: Discard transitions where $o_i \approx o_{i+1}$. This similarity is measured via perceptual hashing (Marr and Hildreth, 1980). Such transitions indicate ineffective actions. These occur in two scenarios: when a_i fails to alter the environment (invalid action) or when the environment fails to respond (execution error).

The knowledge vector store $\mathcal{K}$ is structured as a multi-modal index:

$$\mathcal{K} = \bigcup_{\xi \in \Xi} \bigcup_{t=1}^{|\xi|-1} F_{extract}(o_t, a_t, o_{t+1}) \quad (2)$$

where Ξ denotes all exploration trajectories and $|\xi|$ denotes the total steps of the trajectory ξ.

This knowledge construction process enables **Continuous Knowledge Refinement**. New explorations iteratively update $\mathcal{K}$ through:

$$\mathcal{K} = \begin{cases} \mathcal{K} \setminus \{(k_{\text{old}}, v_{\text{old}})\} \cup \{(k_{\text{old}}, v_{\text{old}} \oplus v_{\text{new}})\} & \text{if } \Phi \\ \mathcal{K} \cup \{(k_{\text{new}}, v_{\text{new}})\} & \text{otherwise} \end{cases} \quad (3)$$

where $\mathcal{K} \setminus \{(k_{old}, v_{old})\}$ denotes the removal of the original key-value pair from the knowledge vector store, $\oplus$ represents the concatenation of knowledge, condition Φ is formally defined as:

$$\exists (k_{\text{old}}, v_{\text{old}}) \in \mathcal{K} \quad \text{s.t.} \quad \begin{cases} \cos\left(Emb(k_{\text{new}}), Emb(k_{\text{old}})\right) \geq \delta_k \\ \cos\left(Emb(v_{\text{new}}), Emb(v_{\text{old}})\right) \leq \delta_v \end{cases} \quad (4)$$

where δ_k and δ_v are similarity thresholds for key matching (≥ 0.99) and value merging (≤ 0.1) respectively, $\cos(\cdot)$ is cosine similarity, and $\text{Emb}(\cdot)$ is the embedding function. This prevents redundant entries while capturing novel interface behaviors.

Figure 3 demonstrates the importance of transition-aware knowledge.

3.3 Dynamic Guidance for GUI Agent

The dynamic guidance mechanism connects acquired Transition-aware Knowledge to real-time task execution. This connection is achieved through a ranking architecture. As detailed in Algorithm 2, our approach uses a two-phase process. The first phase involves visual-semantic knowledge retrieval. The second phase performs instruction-aware prioritization.

Algorithm 2: Dynamic Guidance for GUI Agent

Input: Environment E, Instruction I, Knowledge_Vector_Store $\mathcal{K}$, Knowledge_Ranker $Ranker$, max_steps s

Function *Get_Guidance(obs, I, $\mathcal{K}$)*
- annot_scr ← Get_Annotated_Screenshot(obs);
- ui_elements ← Extract_UI_Elements(obs);
- all_knol ← $\emptyset$; `// all_knowledge`
- **foreach** *ui_element* ***in*** *ui_elements* **do**
  - all_knol.append(Retrieve_Knowledge($\mathcal{K}$, ui_element));
- prioritized_knol ← $Ranker(I, all\_knol)$;
- guidance ← Create_Guidance_Prompt(I, rioritized_knol, annot_scr);
- **return** *guidance*;

E.reset();
for $idx = 1$ **to** s **do**
- obs ← E.observation;
- operational_guid ← Get_Guidance(obs, I, $\mathcal{K}$);
- action ← MLLM(I, operational_guid, obs);
- **if** *action == "END"* **then**
  - **break**;
- E.step(action);

Knowledge Ranking Formulation Given an instruction I and candidate knowledge entries $\mathcal{C} = \{k_1, ..., k_n\}$, we define the optimal knowledge ordering $\mathcal{C}^*$ through pairwise utility comparison:

$$\mathcal{C}^* = \arg\max_{\pi \in \Pi(\mathcal{C})} \sum_{i=1}^{|\mathcal{C}|-1} int(u(k_{\pi(i)}, I) \geq u(k_{\pi(i+1)}, I)) \quad (5)$$

where $\Pi(\mathcal{C})$ denotes all permutations of $\mathcal{C}$, $int(\cdot)$ converts bool to integer (false as 0, true as 1), and utility function $u(k, I)$ measures the relevance between knowledge entry k and instruction I. We implement $u(\cdot)$ through an MLLM-based pairwise comparator:

$$u(k_a, I) > u(k_b, I) \Leftrightarrow f_{\text{rank}}(g(I, k_a, k_b)) = 1 \quad (6)$$

where $g(\cdot)$ constructs the ranking prompt (see Appendix D.3), and f_{rank} represents the MLLM's binary classification. When the classification result is 1, it indicates k_a is more helpful than k_b for this instruction. When the result is 2, it means k_b is more helpful than k_a. This formulation enables efficient sorting through a modified merge sort algorithm:

$$\text{Sort}(\mathcal{C}, I) = \begin{cases} \mathcal{C} & |\mathcal{C}| \leq 1 \\ \text{Merge}(\text{Sort}(\mathcal{C}_L, I), \text{Sort}(\mathcal{C}_R, I), I) & \text{otherwise} \end{cases} \quad (7)$$

The merge operation recursively compares head elements from sorted sublists using f_{rank}:

$$\text{Merge}(A, B, I) = \begin{cases} [a_0] \oplus \text{Merge}(A_{1:}, B, I) & f_{\text{rank}}(g(I, a_0, b_0)) = 1 \\ A \oplus B & A = \emptyset \vee B = \emptyset \\ [b_0] \oplus \text{Merge}(A, B_{1:}, I) & \text{otherwise} \end{cases} \quad (8)$$

where a_0 and b_0 denote the first elements of lists A and B respectively.

Operational Guidance Generation At each execution step t, the system: (1) Extracts UI elements $\mathcal{U}_t$ from current observation o_t; (2) Retrieves associated knowledge entries $\mathcal{K}_t \subseteq \mathcal{K}$; (3) Sorts entries via $\mathcal{K}_t^* = \text{Sort}(\mathcal{K}_t, I)$; (4) Constructs guidance prompt p_t with relevant knowledge.

As shown in Figure 2 (c), the dynamic guidance mechanism enables precise alignment between operational knowledge and real-time interface states.

4 GUI-Knowledge Reasoning Benchmark

We introduce the GUI-Knowledge Reasoning Benchmark (GUI-KRB). This benchmark evaluates MLLMs' accuracy in two areas: prior knowledge accuracy and dynamic UI comprehension for mobile environments. Existing benchmarks primarily focus on task completion. In contrast, GUI-KRB assesses models' fundamental understanding of UI elements and their behaviors. It contains 500 carefully curated samples spanning 43 applications across 8 categories. Appendix B shows the proportion of apps in each category.

Tasks and Metrics GUI-KRB includes two evaluation tasks: (1) **Prior Knowledge Assessment**: Models must identify the functionality of specified UI elements. They are given a single screenshot, its accessibility tree, and a task context about this element. This task simulates the planning phase in GUI automation. During planning, agents must understand element functionality before acting. Success here indicates effective use of prior training knowledge. (2) **Dynamic Comprehension Assessment**: Models analyze UI element functionality by comparing pre-interaction and post-interaction states within the task context of this transition. These states include screenshots and accessibility trees. This task evaluates reasoning about cause-effect logic in GUI interactions. It simulates the knowledge extraction method we use in this paper.

For both tasks, responses are evaluated against human-annotated keywords. A response is considered correct if it contains at least 50% of expert-identified keywords. This metric balances precision with flexibility for valid phrasings. (During keyword labeling, we include up to 50% synonyms to accommodate diverse responses.)

Annotation Process GUI-KRB was built through a rigorous multi-stage process: (1) **Trajectory Collection**: We utilized manually crafted tasks to enable the agent's execution in a mobile environment and collected over 300 task execution trajectories. These trajectories contain more than 7,000 interaction steps across diverse mobile applications. They capture authentic user interactions in real-world scenarios. (2) **Element Extraction**: From these trajectories, we extracted individual UI elements using bounding box information from accessibility trees. To ensure diversity and remove redundancy, we eliminated duplicate elements using perceptual hashing techniques (Marr and Hildreth, 1980). (3) **Keyword Annotation**: Human experts identified essential keywords uniquely associated with each UI element's functionality. These keywords capture both the element's immediate purpose and its broader role in the interface. (4) **Validation**: The authors conducted a comprehensive review of all annotations, verifying keyword accuracy and ensuring consistent annotation quality across the dataset.

The final dataset provides triplets of target UI elements, their corresponding screen states (before and after interaction), and expert-validated keyword sets. Example annotations are provided in Appendix C.

5 Experiments

5.1 Experimental Setup

5.1.1 Datasets

We evaluate GUI-explorer on two comprehensive, open-source benchmarks: MIT-licensed SPA-Bench (Chen et al., 2024b) and Apache-2.0-licensed AndroidWorld (Rawles et al., 2024).

SPA-Bench SPA-Bench is a benchmark simulating daily smartphone usage scenarios with 58 mainstream apps (e.g., Facebook and Gmail). It contains three progressively challenging task levels (Level 1-3), where Level 3 represents the most complex real-world workflows.

AndroidWorld AndroidWorld is an Android environment featuring 116 tasks across 20 real-world apps. The benchmark dynamically generates task variants through randomized parameters (e.g., message content, contact names, and calendar dates), creating millions of unique task instantiations.

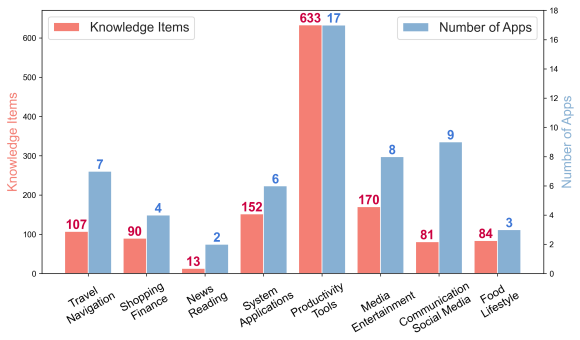

Figure 4: Distribution of transition-aware knowledge gained through autonomous exploration.

Agent	Input	Base model	Task success rate(%)
AppAgent (Zhang et al., 2023)	SoM	GPT-4o	14.0
AutoDroid (Wen et al., 2023)	a11y tree	GPT-4o	12.0
CogAgent (Hong et al., 2024)	screen	CogAgent	0
DigiRL (Zhou et al., 2024)	screen	DigiRL	0
MobileAgentV2 (Wang et al., 2024a)	SoM	GPT-4o	20.0
SeeAct (Zheng et al., 2024)	SoM	GPT-4o	12.0
T3A (Rawles et al., 2024)	a11y tree	GPT-4o	26.0
M3A (Rawles et al., 2024)	SoM	GPT-4o	42.0
GUI-explorer (Ours)	SoM	GPT-4o	**53.7**

Table 1: Performance comparison on SPA-Bench single-app English Level 3 tasks. Results for the first 8 agents are from the SPA-Bench (Chen et al., 2024b). SoM (Yang et al., 2023) utilizes the bounding boxes (bbox) recorded in the a11y tree to annotate UI elements with numerical labels in screenshots.

5.1.2 Implementation Details

To ensure fair evaluation across benchmarks, we carefully selected base models according to their characteristics. For SPA-Bench and AndroidWorld, we adopted GPT-4o (Hurst et al., 2024) as the unified base model, which has been the de facto standard model in prior works including, but not limited to, SPA-Bench (Chen et al., 2024b) and Aria-UI (Yang et al., 2024b), eliminating performance variance caused by heterogeneous model capabilities. In contrast, for GUI-KRB, we intentionally utilized the weakest-performing Qwen2-VL-72B-Instruct-GPTQ-Int4 (Wang et al., 2024b) as our base model, to rigorously validate the robustness of our method.

We configured the exploration process with a branching factor of 10, a maximum depth of 5, and a step limit of 30 for AndroidWorld and SPA-Bench. This setup facilitated the automated discovery of over 1,500 atomic knowledge entries (detailed distribution in Figure 4) across 50 applications, with an average exploration time of 1.7 hours per app. For visual-semantic retrieval, we utilized google/siglip-so400m-patch14-384[5] as the embedding model. Hardware configurations are provided in Appendix A.

5.1.3 Comparative Baselines

We select three baselines with exploration and knowledge extraction capabilities for comprehensive comparison. AppAgent (Zhang et al., 2023) requires manually designed exploration tasks to guide its interaction with GUI environments for knowledge acquisition, whereas AutoDroid (Wen et al., 2023) eliminates task-specific human effort by autonomously generating random action sequences to collect exploration trajectories. Both methods extract structured text-based knowledge from raw textual observations during exploration. DigiRL (Zhou et al., 2024) adopts a distinct reinforcement learning framework to iteratively explore environments while utilizing the Gemini (Google, 2025) model to filter successful trajectories as training data, enabling adaptive exploration with minimal human intervention. For completeness, we also report results from additional baselines in their respective benchmark papers as performance references.

[5] https://huggingface.co/google/siglip-so400m-patch14-384

5.2 Experimental Results

Our comprehensive evaluation demonstrates GUI-explorer's superior performance across multiple dimensions. As shown in Table 1, GUI-explorer achieves 53.7% task success rate on SPA-Bench single-app English Level 3 tasks. This represents a 28.1% absolute improvement over M3A, the previous state-of-the-art. Our transition-aware knowledge mining approach proves highly effective in complex, real-world scenarios.

The AndroidWorld results in Table 3 further validate GUI-explorer's generalizability. Our agent achieves 47.4% success rate. This surpasses vision-centric Aria-UI at 44.8%. It also outperforms multimodal M3A at 40.5%.

The GUI-KRB evaluation reveals critical insights about MLLMs' GUI reasoning limitations. GPT-4o shows an 18.2% prior knowledge error rate. These errors mainly stem from the misinterpreting of UI components and outdated interface understanding. Our method reduces these errors by 16.0% when applied to Qwen2-VL-72B-Instruct-GPTQ-Int4. This demonstrates the effectiveness of transition-aware knowledge. The dynamic comprehension assessment shows similar improvements. GUI-explorer-enabled models achieve 13.4% lower error rates than base models.

App Category	Retrieval Time per Step(sec)	Ranking Time per Step(sec)	Reasoning Time per Step(sec)	Total Time per Step(sec)	Ranking Cost per Step(0.1USD)	Reasoning Cost per Step(USD)	Total Cost per Step(USD)
Travel & Navigation	7.663	**33.084**	31.400	72.147	**0.017**	0.066	0.068
Shopping & Finance	**8.613**	24.922	**36.622**	70.157	0.013	0.063	0.065
News & Reading	8.123	17.317	29.272	54.712	0.008	0.053	0.053
System Applications	6.955	31.083	34.513	**72.552**	0.016	0.065	0.067
Productivity & Tools	7.136	28.091	28.382	63.609	0.016	0.064	0.066
Media v Entertainment	7.549	32.481	30.586	70.615	0.017	**0.066**	**0.068**
Communication & Social	**6.176**	25.662	**27.293**	59.130	0.013	0.057	0.058
Food & Lifestyle	6.304	**9.511**	30.481	**46.296**	**0.004**	**0.041**	**0.042**
Overall	7.120	28.462	30.796	66.378	0.015	0.062	0.064

Table 2: Per-Step Computational Overhead Analysis: Breakdown of time consumption (seconds) and API costs (USD) across application categories. Note that Ranking Cost per Step is presented in Dimes (0.1 USD) for better readability due to its small magnitude.

Agent	Input	Base model	Task success rate(%)
Human (Rawles et al., 2024)	*screen*	-	*80.0*
SeeAct (Rawles et al., 2024)	SoM	GPT-4-turbo	15.5
T3A (Rawles et al., 2024)	a11y tree	GPT-4-turbo	30.6
UGround (Gou et al., 2024)	screen	GPT-4o	32.8
Ponder&Press (Wang et al., 2024c)	screen	GPT-4o	34.5
Aguvis (Xu et al., 2024)	screen	GPT-4o	37.1
Aria-UI (Yang et al., 2024b)	screen	GPT-4o	44.8
AppAgent (Zhang et al., 2023)	SoM	GPT-4o	14.9
M3A (Rawles et al., 2024)	SoM	GPT-4o	40.5
GUI-explorer (Ours)	SoM	GPT-4o	**47.4**

Table 3: Performance comparison on AndroidWorld.

Model	Prior Knowledge error rate(%)	Dynamic Comprehension error rate(%)
Qwen2-VL (Wang et al., 2024b)	22.8	19.8
Qwen2.5-VL (Team, 2025)	16.6	14.0
UI-TARS (Qin et al., 2025)	18.0	14.2
Gemini 2.0 Flash (Google, 2025)	15.2	11.2
GPT-4o (Hurst et al., 2024)	18.2	13.4
GUI-explorer (w/o Ranker)	9.8	6.8
GUI-explorer	**6.8**	**6.4**

Table 4: Performance comparison on GUI-KRB. For all methods, we selected the highest-performing models within device VRAM constraints: Qwen2-VL-72B-Instruct-GPTQ-Int4 for Qwen2-VL, Qwen2.5-VL-7B-Instruct for Qwen2.5-VL, and UI-TARS-7B-DPO for UI-TARS.

5.3 Analysis and Discussion

The ablation study in Figure 5 quantifies the impact of our key components. Removing dynamic guidance construct by transition-aware knowledge causes a 12.2% performance drop. This emphasizes the critical role of transition-aware knowledge. Cross-Environment Guidance improves performance by 4.3% compared to No Guidance. This demonstrates that our transition-aware knowledge exhibits promising generalization capabilities. It effectively guides agent reasoning even in previously unseen scenarios. The knowledge learned can transfer across different UI environments.

Our computational overhead analysis appears in Table 2. It reveals practical tradeoffs. The ranking component contributes 42.9% of time. This comes primarily from MLLM-based pairwise comparisons. However, we use a merge sort implementation. This ensures $O(n \log n)$ complexity. This keeps practical costs acceptable (0.0015 USD/step average). Additionally, Table 4 shows another benefit. The ranking component reduced the error rate by 3% by prioritizing more relevant knowledge.

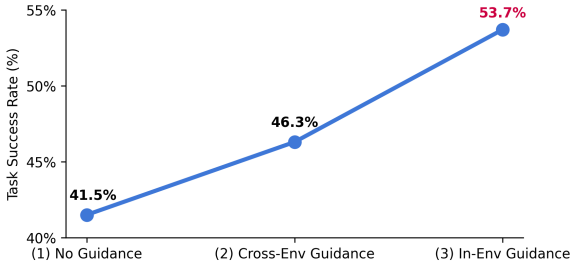

Figure 5: Ablation study on operational guidance configurations: (1) Baseline without dynamic guidance, (2) Guidance derived from cross-environment exploration (AndroidWorld), (3) Guidance generated through in-environment exploration (SPA-Bench).

The GUI-KRB results expose two fundamental limitations in current MLLMs. First, there are persistent prior knowledge gaps. Even Gemini 2.0 Flash (Google, 2025) has a 15.2% error rate. Second, there is limited dynamic reasoning capability.

The GUI-KRB Dynamic Comprehension task, equivalent to transition-aware knowledge mining, achieved 86.6% accuracy with GPT-4o, indicating comparable reliability in our GPT-4o-built Knowledge Vector Store.

6 Conclusion

We present GUI-explorer, a training-free GUI agent designed to address two key challenges: misinterpretation of UI components and knowledge obsolescence. Our approach achieves this through autonomous exploration and transition-aware knowledge mining. Experimental results demonstrate our SOTA performance across major benchmarks.We introduce the GUI-KRB benchmark, which reveals fundamental limitations in current MLLMs' interface understanding capabilities. Our dynamic guidance mechanism effectively mitigates these limitations.

Limitations

While GUI-explorer demonstrates significant advancements in GUI automation, several limitations warrant discussion. First, our current implementation of exploration anchors relies on mobile app manifest declarations (e.g., Android Activity components), which limits direct applicability to web and desktop environments. Second, although the current Knowledge Ranker takes only 28.5 seconds per step, it's still a bit slow. Future work will focus on extending this approach to web and desktop and speeding up Knowledge Ranker.

Ethics Statement

Our work introduces GUI-explorer, an autonomous agent for graphical user interface automation, and raises several ethical considerations inherent to AI-driven interaction systems. First, while our exploration process utilizes application screenshots and accessibility metadata, we strictly employ open-source or publicly available applications, ensuring no collection of private user data or infringement of intellectual property rights.

Second, our reliance on large multimodal models introduces potential risks of perpetuating societal biases embedded in their training data. Though our transition-aware knowledge mechanism mitigates the misinterpretation of UI components, we acknowledge that residual biases in element interpretation could lead to unintended operational consequences. We strongly advocate for human oversight in real-world deployments, particularly for sensitive applications in healthcare or finance domains.

The computational costs associated with our approach (average 66 seconds per interaction step) raise environmental concerns regarding energy consumption. While our method eliminates the need for model retraining—a significant carbon footprint contributor—future work must prioritize efficiency optimizations to enable sustainable scaling.

We recognize potential dual-use risks where autonomous GUI agents could be misused for malicious automation (e.g., credential stuffing or click fraud).

Finally, our benchmark construction followed ethical annotation practices, with contributors compensated at fair market rates and granted full rights to withdraw their participation.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Gongwei Chen, Leyang Shen, Rui Shao, Xiang Deng, and Liqiang Nie. 2024a. Lion: Empowering multimodal large language model with dual-level visual knowledge. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 26540–26550.

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

Sidong Feng, Haochuan Lu, Jianqin Jiang, Ting Xiong, Likun Huang, Yinglin Liang, Xiaoqin Li, Yuetang Deng, and Aldeida Aleti. 2024. Enabling cost-effective ui automation testing with retrieval-based llms: A case study in wechat. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1973–1978.

Google. 2025. Gemini 2.0 is now available to everyone.

Boyu Gou, Ruohan Wang, Boyuan Zheng, Yanan Xie, Cheng Chang, Yiheng Shu, Huan Sun, and Yu Su. 2024. Navigating the digital world as humans do: Universal visual grounding for gui agents. *arXiv preprint arXiv:2410.05243*.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, et al. 2024. Cogagent: A visual language model for gui agents. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14281–14290.

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

David Marr and Ellen Hildreth. 1980. Theory of edge detection. *Proceedings of the Royal Society of London. Series B. Biological Sciences*, 207(1167):187–217.

OpenAI. 2023. Gpt-4v(ision) system card.

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

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice

Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, et al. 2024. Androidworld: A dynamic benchmarking environment for autonomous agents. *arXiv preprint arXiv:2405.14573*.

Gabriel Sarch, Lawrence Jang, Michael J Tarr, William W Cohen, Kenneth Marino, and Katerina Fragkiadaki. 2024. Ical: Continual learning of multimodal agents by transforming trajectories into actionable insights. *arXiv e-prints*, pages arXiv–2406.

Yu Su, Diyi Yang, Shunyu Yao, and Tao Yu. 2024. Language agents: Foundations, prospects, and risks. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Tutorial Abstracts*, pages 17–24, Miami, Florida, USA. Association for Computational Linguistics.

Qwen Team. 2025. Qwen2.5-vl.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024a. Mobile-agent-v2: Mobile device operation assistant with effective navigation via multi-agent collaboration. *arXiv preprint arXiv:2406.01014*.

Peng Wang, Shuai Bai, Sinan Tan, Shijie Wang, Zhihao Fan, Jinze Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Yang Fan, Kai Dang, Mengfei Du, Xuancheng Ren, Rui Men, Dayiheng Liu, Chang Zhou, Jingren Zhou, and Junyang Lin. 2024b. Qwen2-vl: Enhancing vision-language model's perception of the world at any resolution. *arXiv preprint arXiv:2409.12191*.

Yiqin Wang, Haoji Zhang, Jingqi Tian, and Yansong Tang. 2024c. Ponder & press: Advancing visual gui agent towards general computer control. *arXiv preprint arXiv:2412.01268*.

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

Yiheng Xu, Zekun Wang, Junli Wang, Dunjie Lu, Tianbao Xie, Amrita Saha, Doyen Sahoo, Tao Yu, and Caiming Xiong. 2024. Aguvis: Unified pure vision agents for autonomous gui interaction. *arXiv preprint arXiv:2412.04454*.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2024a. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*.

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

Yuhao Yang, Yue Wang, Dongxu Li, Ziyang Luo, Bei Chen, Chao Huang, and Junnan Li. 2024b. Aria-ui: Visual grounding for gui instructions. *arXiv preprint arXiv:2412.16256*.

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

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

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

Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. 2023. Synapse: Trajectory-as-exemplar prompting with memory for computer control. In *The Twelfth International Conference on Learning Representations*.

Yifei Zhou, Hao Bai, Mert Cemri, Jiayi Pan, Alane Suhr, Sergey Levine, and Aviral Kumar. 2024. Digirl: Training in-the-wild device-control agents with autonomous reinforcement learning. In *Automated Reinforcement Learning: Exploring Meta-Learning, AutoML, and LLMs*.

A Hardware configurations

Hardware configurations were optimized for cost-effectiveness: Most experiments ran on a single NVIDIA GeForce RTX 4070 Laptop GPU (8GB VRAM). For GUI-KRB evaluations involving open-source MLLMs, we scaled to two NVIDIA L40S GPUs (48GB VRAM) to accommodate larger VRAM requirements.

B GUI-KRB Benchmark Distributions

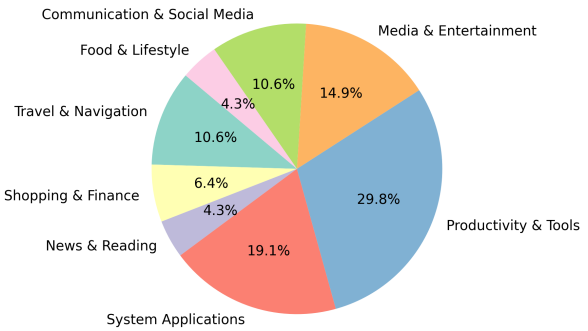

Figure 6: Distribution of apps in GUI-KRB.

Figure 6 shows the distribution of the number of apps in GUI-KRB.

C GUI-KRB Benchmark Sample Data

This section presents an example from the GUI-KRB benchmark to illustrate its data structure, as shown in Figure 7. Each sample consists of five main components. First, it contains screenshots captured before and after the interaction with the target element to demonstrate the visual state transition. Second, it includes the accessibility tree representation of the interface. Third, the broader task context describes the necessary interaction with the target element required to complete the task. Fourth, the transition-aware knowledge associated with the element is documented but excluded from the test input. Finally, for automated evaluation purposes, the sample includes evaluation keywords (also excluded from test input) that incorporate synonyms and related terms (such as "modify" and "Main") to accommodate various valid responses and reduce false judgments during model assessment.

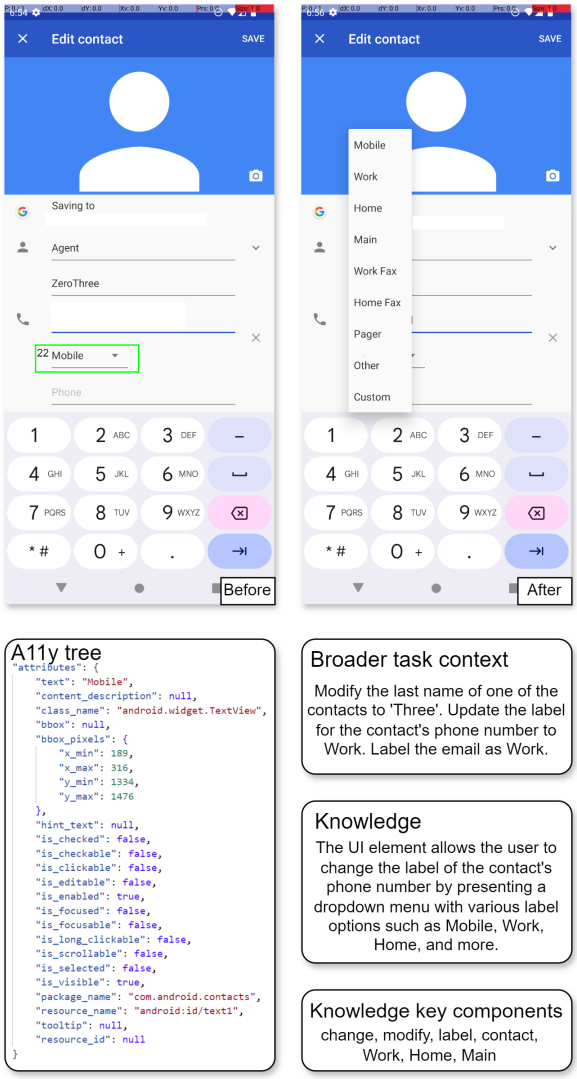

Figure 7: A comprehensive sample from GUI-KRB benchmark illustrating: (1) before/after screenshots of target element interaction, (2) accessibility tree representation, (3) broader task context, (4) transition-aware knowledge (excluded from test input), and (5) evaluation keywords with synonyms for robust assessment.

D Prompting Templates of GUI-explorer

D.1 Prompting Template of Function-aware Task Goal Generator

```
Given the screenshot of app name and its available activities, generate a comprehensive list of practical user tasks that:
1. Start from the current screen shown in the screenshot
2. Can be completed within 10-30 steps
3. Utilize the app's full feature set based on the activity list
4. Are concrete and specific (like searching for a particular item rather than just "search")
5. Cover different user interaction patterns (viewing, editing, sharing, etc.)
6. Include both basic and advanced features
7. Represent realistic user behaviors and goals
8. Avoid excessive steps on form-filling or scrolling pages
```

```
Important context:
- App name: app name
- Package name: package name
- Available activities (app screens/features):
ctivity list
Format requirements:
1. List only the tasks without explanations or commentary
2. Each task should be a single, clear directive
3. Use specific examples (e.g., concrete search terms, actions, settings)
4. Include the expected outcome where relevant
5. Tasks should follow this pattern: [Starting action] + [Specific steps] + [End goal]
Example tasks from other apps (for reference only):
1. Search for "ocean waves" white noise, then sort results by most played
2. Open the first recommended video, then post "Great content!" as a comment
3. Play the trending video, then add it to your "Watch Later" playlist
4. Navigate to the comments section of a featured video, then like the top comment
Generate diverse tasks that would help a user explore and utilize all major features visible in the screenshot and implied by the activity list.
```

D.2 Prompting Template of Unsupervised Mining of Transition-aware Knowledge

```
Objective: Describe the functionality of a specific UI element in a mobile app screenshot.
Input:
- Two screenshots: Before and after interacting with a UI element
- UI element marked with a numeric tag in the top-left corner
- Element number: numeric tag of element
- Broader task context: task description
- UI Element Attributes:
‘‘‘
ui element attributes
‘‘‘
Requirements for Functionality Description:
1. Concise: 1-2 sentences
2. Focus on general function, not specific details
3. Avoid mentioning the numeric tag
4. Use generic terms like "UI element" or appropriate pronouns
Example:
- Incorrect: "Tapping the element #3 displays David’s saved recipes in the results panel"
- Correct: "Tapping this element will initiates a search and displays matching results"
Guidance:
- Describe the core action and immediate result of interacting with the UI element
- Prioritize clarity and generality in the description
```

D.3 Prompting Template of Knowledge Ranker

```
Given the user instruction: task goal, determine which of the following two knowledge entries is more useful.
Respond ONLY with a integer value:
1 means Knowledge A is strictly better.
2 means Knowledge B is strictly better.
Knowledge A: knowledge a
Knowledge B: knowledge b
Please provide your response:
```

D.4 Prompting Template of Reasoning

```
## Role Definition
You are an Android operation AI that fulfills user requests through precise screen interactions.
The current screenshot and the same screenshot with bounding boxes and labels added are also given to you.
## Action Catalog
Available actions (STRICT JSON FORMAT REQUIRED):
1. Status Operations:
- Task Complete: {"action_type": "status", "goal_status": "complete"}
- Task Infeasible: {"action_type": "status", "goal_status": "infeasible"}
2. Information Actions:
- Answer Question: {"action_type": "answer", "text": "<answer_text>"}
3. Screen Interactions:
- Tap Element: {"action_type": "click", "index": <visible_index>}
- Long Press: {"action_type": "long_press", "index": <visible_index>}
4. Input Operations:
- Text Entry: {"action_type": "input_text", "text": "<content>", "index": <text_field_index>}
- Keyboard Enter: {"action_type": "keyboard_enter"}
5. Navigation:
- Home Screen: {"action_type": "navigate_home"}
- Back Navigation: {"action_type": "navigate_back"}
6. System Actions:
- Launch App: {"action_type": "open_app", "app_name": "<exact_name>"}
- Wait Refresh: {"action_type": "wait"}
## Current Objective
User Goal: task goal
## Execution Context
Action History:
history
Visible UI Elements (Only interact with *visible=true elements):
ui elements
## Core Strategy
1. Path Optimization:
- Prefer direct methods (e.g., open_app > app drawer navigation)
- Always use input_text for text entry
- Verify element visibility before interaction
2. Error Handling Protocol:
- Switch approach after ≥ 2 failed attempts
- Prioritize scrolling over force-acting on invisible elements
- Try opposite scroll direction if initial fails (up/down, left/right)
3. Information Tasks:
- MANDATORY: Use answer action for questions
- Verify data freshness (e.g., check calendar date)
## Expert Techniques
Here are some tips for you:
knowledge
## Response Format
STRICTLY follow:
Reasoning: [Step-by-step analysis covering:
- Visibility verification
- History effectiveness evaluation
- Alternative approach comparison]
Action: [SINGLE JSON action from catalog]
Generate response:
```

E Prompting Templates of GUI-KRB

E.1 Prompting Template of Prior Knowledge Task

```
Objective: Describe the functionality of a specific UI element in a mobile app screenshot.
Input:
- Two screenshots: Before and after interacting with a UI element
```

```
- UI element marked with a numeric tag in the top-left corner
- Element number: numeric tag of element
- Broader task context: task description
- UI Element Attributes:
‘‘‘
ui element attributes
‘‘‘
Requirements for Functionality Description:
1. Concise: 1-2 sentences
2. Focus on general function, not specific details
3. Avoid mentioning the numeric tag
4. Use generic terms like "UI element" or appropriate pronouns
Example:
- Incorrect: "Tapping the element #3 displays David's saved recipes in the results panel"
- Correct: "Tapping this element will initiates a search and displays matching results"
Guidance:
- Describe the core action and immediate result of interacting with the UI element
- Prioritize clarity and generality in the description
```

E.2 Prompting Template of Dynamic Comprehension Task

Same as Appendix D.2.

E.3 Prompting Templates for GUI-explorer (w/o Ranker)

E.3.1 Prompting Template of Prior Knowledge Task

```
Objective: Describe the functionality of a specific UI element in a mobile app screenshot.
Input:
- One screenshot: Before interacting with a UI element
- UI element marked with a numeric tag in the top-left corner
- Element number: numeric tag of element
- Broader task context: task description
- UI Element Attributes:
‘‘‘
ui element attributes
‘‘‘
- Similar UI Elements' Functionalities (retrieved based on visual similarity):
‘‘‘
similar element functionalities
‘‘‘
Requirements for Functionality Description:
1. Concise: 1-2 sentences
2. Focus on general function, not specific details
3. Avoid mentioning the numeric tag
4. Use generic terms like "UI element" or appropriate pronouns
5. Consider similar elements' functionalities as reference, but prioritize:
- Current screen context
- UI element attributes
- Task description
6. Only incorporate relevant patterns from similar elements if they align with the current context
Example:
- Incorrect: "Tapping the element #3 displays David's saved recipes in the results panel"
- Correct: "Tapping this element will initiates a search and displays matching results"
Guidance:
- Describe the core action and potential result of interacting with the UI element
- Infer functionality based on the current screen context
- Prioritize clarity and generality in the description
- Use similar elements' functionalities to validate and refine your description, not to simply copy them
```

E.3.2 Prompting Template of Dynamic Comprehension Task

```
Objective: Describe the functionality of a specific UI element in a mobile app screenshot.
Input:
- Two screenshots: Before and after interacting with a UI element
- UI element marked with a numeric tag in the top-left corner
- Element number: numeric tag of element
- Broader task context: task description
- UI Element Attributes:
‘‘‘
ui element attributes
‘‘‘
- Similar UI Elements' Functionalities (retrieved based on visual similarity):
‘‘‘
similar element functionalities
‘‘‘
Requirements for Functionality Description:
1. Concise: 1-2 sentences
2. Focus on general function, not specific details
3. Avoid mentioning the numeric tag
4. Use generic terms like "UI element" or appropriate pronouns
5. Consider similar elements' functionalities as reference, but prioritize:
- Current screen context
- UI element attributes
- Task description
6. Only incorporate relevant patterns from similar elements if they align with the current context
Example:
- Incorrect: "Tapping the element #3 displays David's saved recipes in the results panel"
- Correct: "Tapping this element will initiates a search and displays matching results"
Guidance:
- Describe the core action and immediate result of interacting with the UI element
- Infer functionality based on the current screen context
- Prioritize clarity and generality in the description
- Use similar elements' functionalities to validate and refine your description, not to simply copy them
```

E.4 Prompting Templates of Prior Knowledge Task for GUI-explorer

E.4.1 Prompting Template of Prior Knowledge Task

```
Objective: Describe the functionality of a specific UI element in a mobile app screenshot.
Input:
- One screenshot: Before interacting with a UI element
- UI element marked with a numeric tag in the top-left corner
- Element number: numeric tag of element
- Broader task context: task description
- UI Element Attributes:
‘‘‘
ui element attributes
‘‘‘
- Similar UI Elements' Functionalities (ranked by relevance to task description):
‘‘‘
similar element functionalities
‘‘‘
Note: Elements are sorted by relevance, with most task-relevant functionalities listed first
Requirements for Functionality Description:
1. Concise: 1-2 sentences
2. Focus on general function, not specific details
3. Avoid mentioning the numeric tag
```

```
4. Use generic terms like "UI element" or appropriate pronouns
5. Consider similar elements' functionalities as reference, with priority:
- Higher-ranked (more relevant) reference functionalities
- Current screen context
- UI element attributes
- Task description
6. Only incorporate relevant patterns from similar elements if they align with the current context
Example:
- Incorrect: "Tapping the element #3 displays David's saved recipes in the results panel"
- Correct: "Tapping this element will initiates a search and displays matching results"
Guidance:
- Describe the core action and potential result of interacting with the UI element
- Infer functionality based on the current screen context
- Prioritize clarity and generality in the description
- Pay special attention to higher-ranked similar functionalities as they are more likely to be relevant
- Use similar elements' functionalities to validate and refine your description, not to simply copy them
```

E.4.2 Prompting Template of Dynamic Comprehension Task

```
Objective: Describe the functionality of a specific UI element in a mobile app screenshot.
Input:
- Two screenshots: Before and after interacting with a UI element
- UI element marked with a numeric tag in the top-left corner
- Element number: numeric tag of element
- Broader task context: task description
- UI Element Attributes:
‘‘‘
ui element attributes
‘‘‘
- Similar UI Elements' Functionalities (ranked by relevance to task description):
‘‘‘
similar element functionalities
‘‘‘
Note: Elements are sorted by relevance, with most task-relevant functionalities listed first
Requirements for Functionality Description:
1. Concise: 1-2 sentences
2. Focus on general function, not specific details
3. Avoid mentioning the numeric tag
4. Use generic terms like "UI element" or appropriate pronouns
5. Consider similar elements' functionalities as reference, with priority:
- Higher-ranked (more relevant) reference functionalities
- Current screen context
- UI element attributes
- Task description
6. Only incorporate relevant patterns from similar elements if they align with the current context
Example:
- Incorrect: "Tapping the element #3 displays David's saved recipes in the results panel"
- Correct: "Tapping this element will initiates a search and displays matching results"
Guidance:
- Describe the core action and potential result of interacting with the UI element
- Infer functionality based on the current screen context
- Prioritize clarity and generality in the description
- Pay special attention to higher-ranked similar functionalities as they are more likely to be relevant
- Use similar elements' functionalities to validate and refine your description, not to simply copy them
```