KnowAct-GUIClaw: Know Deeply, Act Perfectly, Personal GUI Assistant with Self-Evolving Memory and Skill

Lychee Team[*], Harbin Institute of Technology, Shenzhen
AI Training Platform, Shenzhen Loop Area Institute

Codes Website Experimental Logs

Abstract

OpenClaw has emerged as a leading agent framework for complex task automation, yet its existing variants face two core bottlenecks: insufficient cross-platform GUI interaction support and no built-in self-evolution mechanism. These flaws limit its adaptation to heterogeneous device ecosystems and prevent performance improvements through continuous learning from execution experience. To resolve these issues, we propose the "**Know Deeply, Act Perfectly**" paradigm for personal assistants, which holds that accumulated human-machine interaction and task-running experience directly improve execution accuracy and efficiency, unifying cognitive comprehension and operational execution. Based on this paradigm, we introduce **KnowAct-GUIClaw**, a novel Know-Route-Act-Reflect framework designed to address OpenClaw's GUI manipulation deficits and break through its cross-platform and recursive self-improvement constraints. First, the host agent leverages accumulated interaction experience and task-relevant knowledge for long-horizon task decomposition and allocation (Know). Second, a pluggable GUI subagent with an experience-attributable memory system (Know) and self-evolving skill library (Act), enabling seamless cross-platform migration and fast-path integration. Especially, this framework continuously stores user profiles and feedback to improve the accuracy of task decomposition and tool calls. Extensive experiments across Android, iOS, HarmonyOS and Windows show that KnowAct-GUIClaw achieves superior UI manipulation efficiency, accuracy and cross-platform adaptability. Especially, the GUIClaw with open-source Kimi-2.6 models achieves the best performance (64.1%) on the long-horizon MobileWorld benchmark, beating all agential frameworks and closed-source agentical models, e.g., Seed-2.0-Pro and GPT-5.5. Additionally, the knowledgeable memory and execution skills supported by our framework are transferable across diverse base models, improving by 8.5% with Kimi-2.6 and 16.2% with Qwen3.5-35B-A3B.

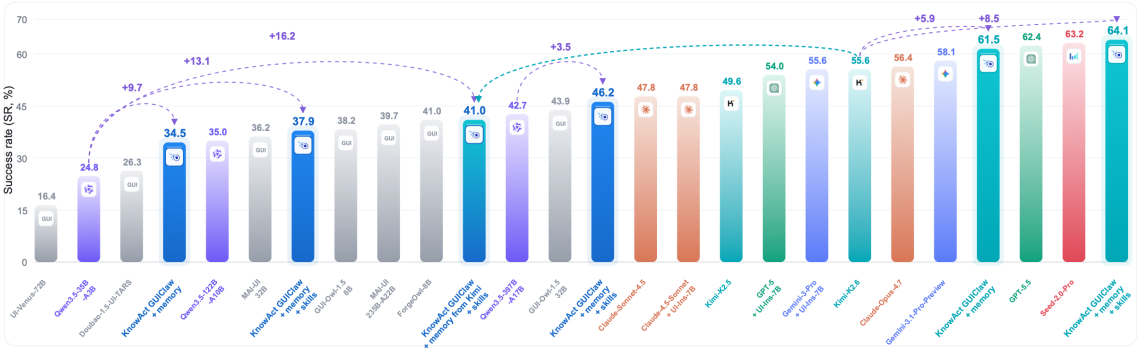

Figure 1: The success rate (SR) comparison on MobileWorld GUI-Only tasks. The bars summarize Table 1 together with the additional Kimi-based KnowAct-GUIClaw runs; gray bars denote specialized GUI models, colored external bars denote general model families, and highlighted bars denote KnowAct-GUIClaw variants with memory and skills. The experimental results show that KnowAct-GUIClaw achieves SOTA performance and that the memory and skill are effective for different base models.

[*]Contributors are shown in Sec. 7

Contents

1 Introduction **3**

2 Related Work **4**
2.1 Personal Assistant Agent 4
2.2 GUI Agents 4
2.3 Memory and Skill Reuse 5

3 Preliminaries **5**
3.1 GUI Automation as a POMDP 5
3.2 Host-Centric Multi-Agent Systems 5

4 KnowAct-GUIClaw **5**
4.1 Overview: A Know–Route–Act–Reflect Stack 5
4.2 Know: Context Gathering and Host Control 6
4.3 Route: Task Decomposition and Information Contracts 7
4.4 Act: Hybrid GUI–Fast Path Execution 7
4.5 Reflect: Trajectory Distillation and Skill Evolution 8

5 Experiments **9**
5.1 Benchmarks and Metrics 9
5.2 Main Results of MobileWorld 10
5.3 Ablation and Efficiency Analysis 10
5.4 AndroidDaily Results 12
5.5 Case Study 12
5.6 Cross-Platform Checks 15
5.7 Reproducibility 16

6 Conclusion **16**

7 Contributors **16**

A Supplementary Case Studies **20**

B Action Space **20**

C Skill Extraction and Prompt Contracts **21**

D Skill and Shortcut Examples **26**

1 Introduction

Large language model (LLM) agents are transitioning from one-off dialogue bots to stateful, long-running personal assistants. They process user requests from diverse input channels, preserve persistent workspace state, invoke external tools, coordinate auxiliary subagents, and resume interrupted long-duration workflows. Two dominant architectural lines guide their development: ReAct-based agents unify logical reasoning and environmental action execution (Yao et al., 2023), while general autonomous agent frameworks modularize planning, memory, tool utilization, and multi-agent dialogue as interchangeable runtime building blocks (Zhou et al., 2023). Practical deployed systems–OpenClaw, Nanobot, and Hermes–integrate these foundational designs into local-first assistant platforms with built-in channel routing, tool extensions, configurable skills, session persistence, and structured memory storage (OpenClaw Contributors, 2026; HKUDS, 2026; Nous Research, 2026). As capable autonomous personal assistants, these platforms autonomously decompose and fulfil complex user objectives through targeted external tool calls.

However, numerous real-world user tasks demand interaction with graphical user interfaces (GUIs), rather than well-structured, standardized application programming interfaces (APIs). For instance, a personal assistant may be required to inspect mobile applications, migrate data across different apps, handle permission pop-up dialogues, or execute multi-step workflows within login-protected environments. Recent benchmarks spanning web, mobile, and desktop platforms collectively indicate that GUI manipulation poses unique challenges that mandate visual grounding, sequential decision-making, and resilient execution amid dynamically shifting interface states (Deng et al., 2023; Zhou et al., 2024; Koh et al., 2024; Xie et al., 2024; Rawles et al., 2025; Li et al., 2026). A suite of dedicated GUI agents—including AppAgent, Mobile-Agent, CogAgent, OS-Atlas, ShowUI, UI-TARS, and Aguvis—have delivered remarkable advances in multimodal GUI action generation grounded solely on screen snapshots (Zhang et al., 2023; Wang et al., 2024; Hong et al., 2024; Wu et al., 2024b; Lin et al., 2024; Qin et al., 2025; Xu et al., 2025). Hence, *a natural research question arises: how can we endow OpenClaw-style agents with the ability to efficiently interact with visual graphical environments?*

A straightforward approach is to integrate a standalone GUI agent into the OpenClaw framework. When deployed for long-horizon personal assistant tasks, this approach suffers from severe inefficiency and fragility due to four drawbacks: *Firstly*, high-level user instructions frequently span multiple disjoint applications. Concise free-text summaries often fail to retain intermediate data values extracted from one app for subsequent cross-app operations. Conversely, rigidly enforcing explicit target application labels for vague tasks without accounting for device environments often leads to spurious app-allocation hallucinations. *Secondly*, GUI observations are inherently incomplete. Each individual data modality—screen captures, accessibility trees, foreground app IDs, historical action trajectories, and internal model reasoning logs—only reveals fragments of the full underlying device state. This necessitates the host agent to record historical trajectories and provide actionable guidance for the lightweight GUI Agent. *Thirdly*, successful and failed trajectories are typically discarded once a task terminates. When re-running analogous tasks, the agent is forced to re-launch target applications, redo redundant navigation steps, and re-learn known shortcuts and recurring failure patterns from scratch. *Finally*, most GUI workflows do not integrate faster non-visual shortcuts, such as web search tools, Android deep links, system intents, and reusable predefined action sequences. And such shortcuts cannot be safely repurposed as persistent long-term skills without validation against the real-time interface page.

This work presents KnowAct-GUIClaw, an agent framework augmented with structured knowledge and executable skills, built upon an OpenClaw-style host runtime and GUI-centric agent execution engine. The framework comprises two core functional components: an attribution policy-enhanced personalized memory system and a self-evolving skill library that supports rapid skill invocation and iterative optimization. The design follows a simple principle: Knowing Deeply, Acting Perfectly. Before acting, the agent retrieves task-relevant app candidates, tools, policies, and GUI hints from all agents' memories. During action, it treats GUI control as a partially observable decision process and records structured trajectory evidence or skills. After action, it converts useful traces into a skill library or generalized memory, so future runs can use operational knowledge rather than repeating exploratory behaviour. Specifically, KnowAct-GUIClaw treats GUI automation as a collaboration between a capable host agent and a lightweight GUI executor, not as a single monolithic GUI agent. The host owns the user-facing context—conversation, workspace memory, the user profile, external tools, and orchestration—and decides what to do and which cli tool could be used; the GUI executor owns screenshot perception, action normalization, device backends, skill validation, and trajectory recording, and decides how to carry it out on screen. Framing the problem this way makes efficiency a first-class objective: **the system invokes the GUI subagent only when visual interaction is genuinely required, reuses validated operational skills and fast CLI tools whenever possible**.

On this foundation, KnowAct-GUIClaw contributes four connected mechanisms.

- **Two-tier host–executor collaboration.** We design a structured, resumable GUI task interface. The host can assign limited GUI tasks to the executor and retrieve standardized outputs (fully completed, partially completed, or blocked), along with progress updates and resumption cues. Incomplete or stalled task runs act as reusable checkpoints: rather than sending one fuzzy command, the host either resumes unfinished execution or rearranges subsequent workflows. We also adopt an adaptive host involvement rule, where the host directly handles qualified information-gathering subtasks with tools instead of forwarding all subtasks to the GUI executor. This boosts task success rates and cuts computational overhead.
- **Memory-grounded routing and information transfer.** A built-in routing mechanism classifies user requests as single-app tasks or cross-app workflows spanning multiple applications. For cross-app sequences, every subtask explicitly defines its input and output data. A temporary shared data board transmits these structured values across subtasks, while persistent routing memory provides fixed candidate applications and auxiliary reference context.
- **Knowledge- and skill-augmented GUI execution.** KNOWACT-GUICLAW distills trajectories into parameterized, state-validated skills, letting the executor commit to a reusable action prefix as a single decision. The same abstraction covers frequent click-then-type patterns and Android deeplink and intent shortcuts, which are taken only when their launch behavior and target page state satisfy the skill's contract.
- **Trajectory-derived memory and skill evolution.** Post-run reflection summarizes traces, distills success and failure lessons into retrievable experience memory, and refines the skill set as new evidence accumulates. This knowledge feeds back to the memory-grounded router before task decomposition and to the executor during execution, closing the loop from past runs to future decisions.

Extensive experimental results demonstrate that our model attains SOTA performance on the challenging MobileWorld benchmark, while enabling cross-platform deployment as well as memory and skill transfer.

2 Related Work

2.1 Personal Assistant Agent

LLM-agent runtimes pair language models with action interfaces, memory, and control loops, as in ReAct (Yao et al., 2023), the Agents framework (Zhou et al., 2023), and OS-Copilot (Wu et al., 2024a).Recently Two salient design philosophies recur when this pattern is packaged into local-first assistants. OpenClaw and Nanobot are configuration-centric: the host exposes message channels, external tools and Model Context Protocol (MCP) servers, conversation history and session state, long-lived workspace memory, and user-authored skills as first-class, declarable components, so that most behavior is specified around the model rather than left to it (OpenClaw Contributors, 2026; HKUDS, 2026). Hermes instead foregrounds adaptation, growing its competence over time by turning prior interactions into reusable routines that the assistant can later invoke (Nous Research, 2026). Both directions show that a capable personal assistant benefits from treating control logic, memory, tools, and history as managed state around the model.

2.2 GUI Agents

GUI and computer-use agents study how language models act through visual interfaces rather than clean APIs. On the web, Mind2Web and WebArena expose long-horizon navigation across realistic sites through HTML and accessibility-tree observations (Deng et al., 2023; Zhou et al., 2024), while VisualWebArena adds visually grounded decision making over rendered pages (Koh et al., 2024), and SeeAct and WebVoyager explore screenshot- and page-structure-based control (Zheng et al., 2024; He et al., 2024). Beyond the browser, Android in the Wild and AndroidWorld provide large-scale demonstrations and dynamic, programmatically checked tasks (Rawles et al., 2023, 2025), OSWorld and OmniACT cover broader desktop and web computer-use settings (Xie et al., 2024; Kapoor et al., 2024), and AppAgent and Mobile-Agent solve smartphone tasks with human-like actions and multimodal perception (Zhang et al., 2023; Wang et al., 2024). Model-centric systems such as CogAgent, OS-Atlas, ShowUI, UI-TARS, and Aguvis further improve perception, grounding, and visual action generation (Hong et al., 2024; Wu et al., 2024b; Lin et al., 2024; Qin et al., 2025; Xu et al., 2025).These works have substantially improved GUI task execution, their primary focus is often on task completion within benchmarked interaction environments.

2.3 Memory and Skill Reuse

Agent memory and skill reuse let agents improve without weight updates (Vu et al., 2018) in both language-only and embodied settings. Reflexion stores verbal feedback to guide later attempts (Shinn et al., 2023), Generative Agents maintain a reflective memory stream for coherent long-term behavior (Park et al., 2023), Voyager builds an executable skill library for open-ended embodied learning (Wang et al., 2023), and KnowAgent uses an action knowledge base to curb planning hallucination (Zhu et al., 2025). ReasoningBank distills reusable success and failure rationales into retrievable memory for self-evolving agents (Ouyang et al., 2026), GUI-specific work such as LearnAct and CUA-Skill highlights the value of demonstrations, retrieved knowledge, and engineered skills for computer-use agents (Liu et al., 2025; Chen et al., 2026), and a recent survey frames agent memory as a write–manage–read loop coupled with perception and action (Du, 2026). Across these lines, textual memory and executable skills address related but distinct reuse problems: memory preserves guidance, rationales, and context, whereas skills compress procedures into reusable behavior.

3 Preliminaries

3.1 GUI Automation as a POMDP

GUI automation is naturally modeled as a partially observable Markov decision process (POMDP) (Kaelbling et al., 1998):

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{O}, T, \Omega, R, \gamma), \tag{1}$$

where $\mathcal{S}$ is the hidden device state, $\mathcal{A}$ the action space, $\mathcal{O}$ the observation space, T and Ω the transition and observation functions, R the task reward, and γ the discount factor. The defining property of GUI control is partial observability: the agent never observes $\mathcal{S}$ directly but only an observation o_t—typically a screenshot with optional screen metadata, the foreground app, and a bounded action history—that exposes a projection of hidden state such as navigation stacks, login and permission status, asynchronous loading, and off-screen form values. A GUI agent must therefore act and recover from errors under this uncertainty rather than assume a fully known interface.

3.2 Host-Centric Multi-Agent Systems

Multi-agent systems distribute problem solving across agents with distinct roles, capabilities, and local information, coordinating through communication and shared environments rather than a single monolithic policy (Wooldridge, 2009). Recent LLM agent frameworks adopt this view, pairing a central orchestrator that maintains conversation state, memory, and tool access with specialized environment- or tool-facing agents to which it delegates subtasks (Zhou et al., 2023). For GUI automation, this host-orchestration pattern motivates separating long-horizon task management from low-level interaction with a partially observed device; Section 4 instantiates this as a concrete execution stack for mobile GUI tasks.

4 KnowAct-GUIClaw

4.1 Overview: A Know–Route–Act–Reflect Stack

Recent mobile-agent systems and benchmarks point to three requirements: hybrid GUI–shortcut action spaces improve efficiency when shortcuts are available (Zhao et al., 2026); perception, memory, and action work as one execution stack (Ren et al., 2026); and a GUI-agent framework should learn from the execution trajectories (Tang et al., 2026). KnowAct-GUIClaw organizes new long-horizon task execution as a four-stage loop: Know, Route, Act, and Reflect. Figure 2 summarizes this loop and the persistent stores that feed every stage.

GUI execution should serve as a subagent within the personal user assistant framework, specifically to address scenarios where no standardized API endpoints are available and graphical interfaces are dynamically rendered in real time. Accordingly, the GUI task interface acts as the formal boundary between the host agent and the UI subagent. Each GUI task returns a structured output consisting of three components: its completion status, a concise execution summary, and the final screen state upon task termination. The host agent retains responsibility for user-facing task orchestration and long-horizon context maintenance, while the GUI subagent operates as a self-contained engine dedicated to low-level device control.

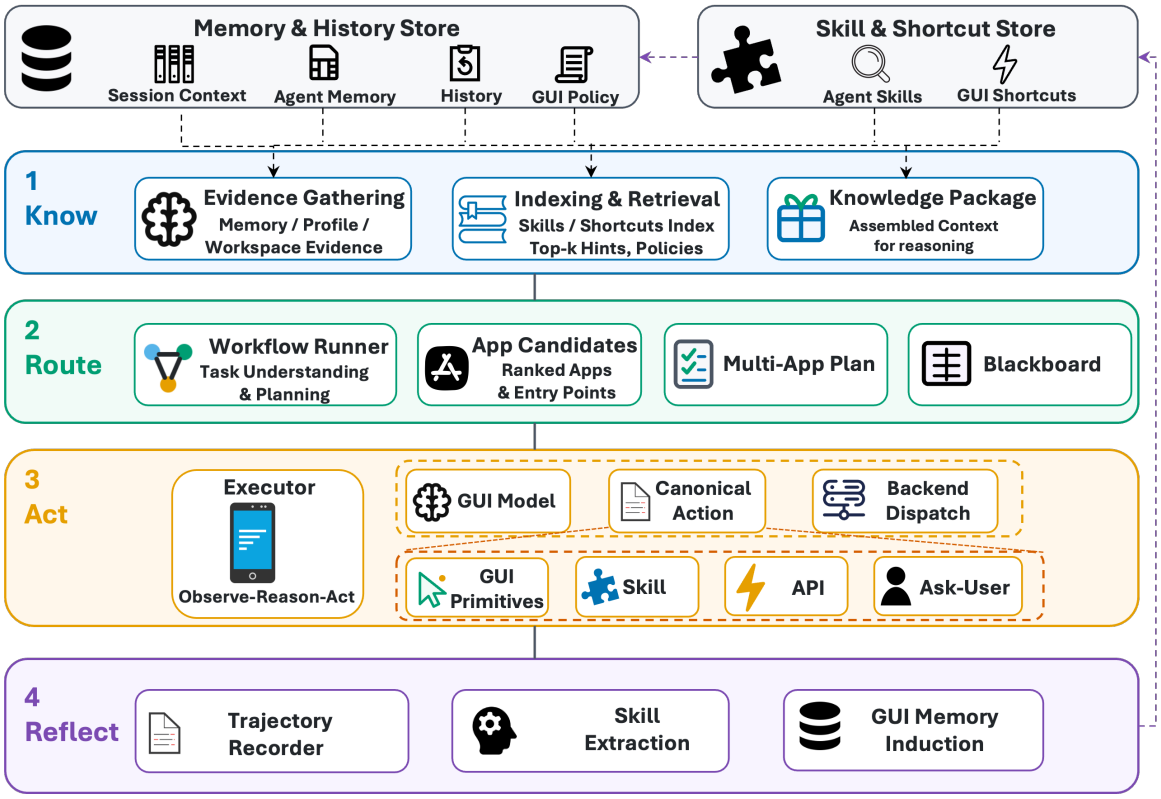

Figure 2: Overview of the KNOWACT-GUICLAW execution loop. Two persistent stores—a memory and history store and a skill and shortcut store—supply advisory context to every stage. Know gathers evidence and assembles a reasoning context; Route ranks app candidates and turns the request into either a single GUI task or an ordered multi-app workflow whose subtasks exchange typed values through a blackboard; Act runs GUIClaw's observe–reason–act loop over the hybrid action space of GUI primitives, skills, deeplink/intent shortcuts, and intervention actions; and Reflect distills each trajectory into updated skills and experience memory that feed back into the stores.

4.2 Know: Context Gathering and Host Control

Active retrieval and policy injection. Before any GUI action, the Know stage gathers most of its context actively: prior GUI memories and candidate skills are retrieved by semantic similarity and kept advisory, never overriding the current instruction, while policy memory is injected directly rather than ranked. Figure 3 illustrates this advisory role on a Mastodon task: the retrieved lesson redirects the task from unsupported mobile-app settings to the web administration panel without replaying an old trajectory.

Host-held context and active recall. The host agent keeps the running session context and forwards only what a GUI task needs when it issues that task, and it recalls its own stored memories, such as session history, agent memory, and the user profile, only when it judges them relevant. When the inputting instruction is fuzzy, it draws on these memories to propose defaults as explicit, overridable assumptions. *This capability is critical for an agentic assistant to achieve continuous capability improvement when deployed across mobile and desktop platforms.*

Host-centric Control. The host agent delegates selectively. It answers a subtask itself when conversation context, recalled memory, or non-GUI tools suffice and no device-local GUI state must be observed or changed, and it issues a GUI task only when the subtask needs a live app session, visual grounding, cross-app manipulation, text entry, or on-device verification. Resolving eligible subtasks directly reduces hand-offs and token-heavy GUI traces.

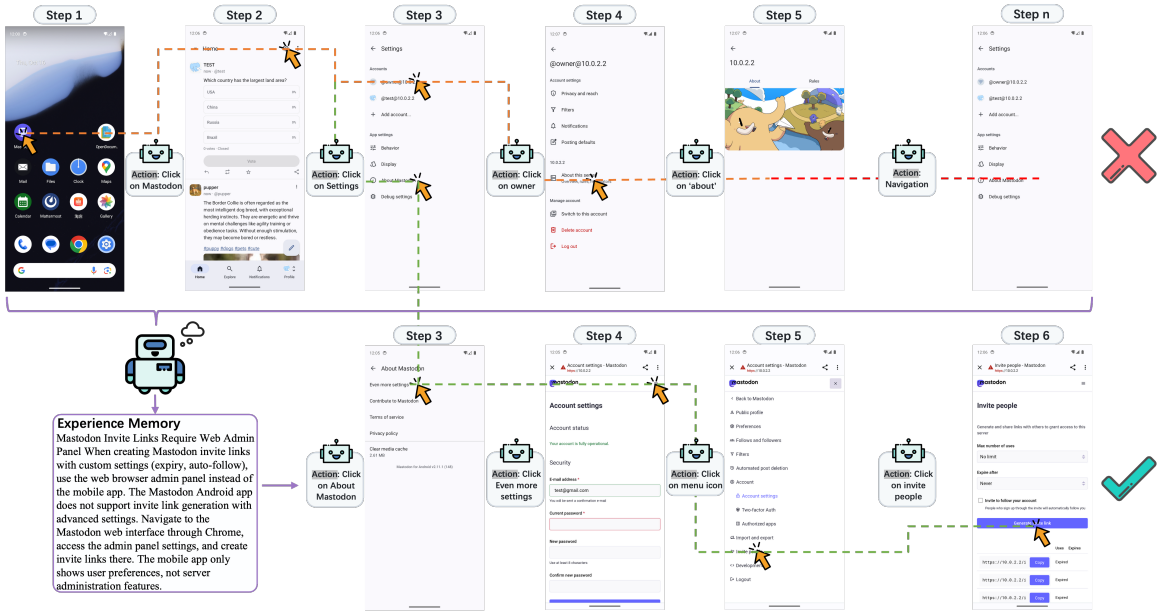

Figure 3: Experience memory improves a GUI task by changing the task context before low-level control begins. Without the retrieved memory (Top), GUIClawinvites continues through Mastodon's mobile settings and reaches a nonproductive path for invite-link creation. With the retrieved memory (bottom), the Know stage supplies an advisory lesson that invite links with advanced settings that require the web administration panel; GUIClaw then opens the web interface, navigates to account settings, and reaches the invite-people page. The example shows that experience memory guides app choice, decomposition, and recovery while live screen observations still ground each action.

4.3 Route: Task Decomposition and Information Contracts

Task decomposition. The routing policy emits either a single GUI task or an ordered multi-app workflow. In the multi-app case, each subtask is a goal-level tuple (g_i, h_i, I_i, O_i): g_i is the app-scoped goal, h_i optionally narrows the app, I_i names the inputs it requires, and O_i names the values it should return. The router does not predict screen sequences. Because each subtask has its corresponding app, the relevant memory and skills are retrieved afresh for that subtask rather than inherited from the top-level route.

Information transfer. The blackboard makes cross-app data flow explicit. Let G denote GUIClaw, which executes a subtask over the POMDP, let E denote the evidence-to-value mapping, and let B_i denote the blackboard after subtask i. A subtask sees only the declared inputs already on B_{i-1}, and only its declared outputs are written back from the trajectory evidence:

$$G(g_i, h_i, B_{i-1}[I_i]) \to \tau_i, \qquad E(\tau_i, O_i) \to B_i[O_i]. \tag{2}$$

If a required input or a declared output is missing, the workflow fails closed rather than running a subtask on incomplete state or letting a later one infer or fabricate the value. This rule matters for cross-app transfer, comparison, and fact-collection tasks. Figure 4 traces the typed tuple contract, while Figure 5 (Section 5) shows the same mechanism in a real cross-app trajectory.

4.4 Act: Hybrid GUI–Fast Path Execution

Hybrid action space. GUIClaw executes each routed subtask over a hybrid action space:

$$\mathcal{A} = \mathcal{A}_{\text{gui}} \cup \mathcal{A}_{\text{skill}} \cup \mathcal{A}_{\text{shortcut}} \cup \mathcal{A}_{\text{ask}}. \tag{3}$$

$\mathcal{A}_{\text{gui}}$ contains human-like primitives such as tap, swipe, scroll, text input, navigation, app open/close, and wait; $\mathcal{A}_{\text{skill}}$ contains skills distilled from reusable GUI behavior. $\mathcal{A}_{\text{shortcut}}$ contains Android deeplinks and intents (Android Developers, 2026a,b) which bypass lengthy navigation workflows when verified as valid on the target device. Predefined action sequences with historically calibrated interaction parameters can also be stored in this set to eliminate the visual grounding overhead of the GUI subagent, though such rigid shortcut

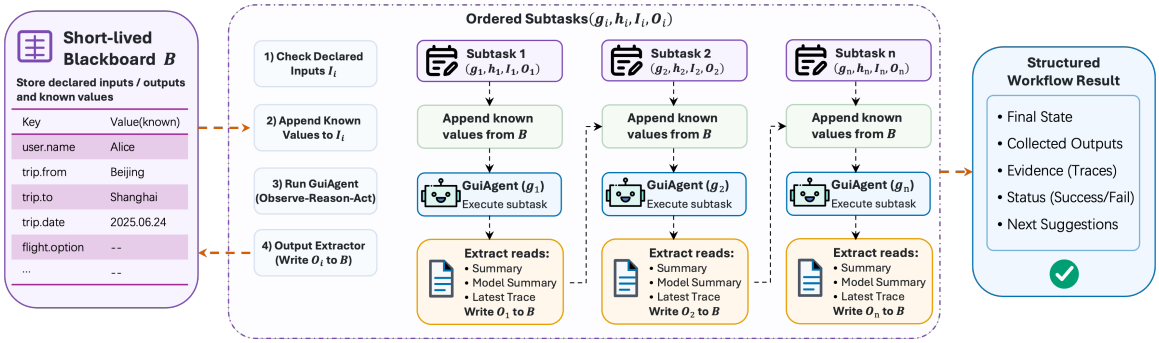

Figure 4: Blackboard-mediated execution in the Route stage. The short-lived blackboard B stores typed inputs and outputs known so far. Each subtask (g_i, h_i, I_i, O_i) checks its declared inputs I_i, reads their values from B, runs GUIClaw's observe–reason–act loop to produce a trajectory τ_i, and writes only its declared outputs O_i back to B (2). A missing required input or output makes the workflow fail closed, so later subtasks consume observed typed values rather than free-form summaries.

schemes suffer from limited generalization capability. $\mathcal{A}_{\text{ask}}$ contains intervention actions for cases that require user input or authorization. These types trade efficiency against reliability, so the executor validates skills and shortcuts against the current state before using them. Appendix B gives the unified action space, by platform, that realizes $\mathcal{A}_{\text{gui}} \cup \mathcal{A}_{\text{ask}}$.

Observe–Reason–Act loop. Within this action space, GUIClaw runs a grounded control loop:

$$p_t \to o_t \to a_t \to e_t \to o_{t+1}, \tag{4}$$

where p_t is the prompt, o_t is the current observation, a_t is the action, and e_t its result on the device. The prompt carries the subtask, task's blackboard, policy memory, interface hints, and available skills, and a normalization step maps each model's output into a common action format. Appendix C gives the prompt contracts used to expose, induce, and validate these skills.

Skills. Skills turn repeated interaction into reusable procedures. Each stored skill contains an identifier, app and platform scope, description, parameters, reliability counters, and an ordered sequence of steps. Stable fields such as app package names, coordinates, components, and intent payloads are fixed in advance, while task-dependent values are represented as placeholders and grounded at run time. Before each subtask, the GUI subagent retrieves app-scoped candidates by lexical and embedding signals, then uses a lightweight applicability prompt to select one useful prefix or reject all candidates. Retrieved skills (Top-5) are advisory: KnowAct-GUIClaw may apply one or fall back to ordinary actions. Before each step, it checks the expected state through a deterministic state contract when available and through a visual valid-state check otherwise; on a mismatch, it either runs a bounded recovery subgoal, skips an optional obstacle step, or returns control to ordinary GUI execution. Appendix D gives concrete skill records, including a multi-step skill with per-step state validation.

Deeplink and intent shortcuts. $\mathcal{A}_{\text{shortcut}}$ contains page-validated Android deeplinks and intents that bypass long GUI navigation. During execution, GUIClaw treats them as one-step skills with `open_deeplink` or `open_intent`, and uses them only after their target page, required parameters, and app state have been validated. This prevents broad Android constants such as `SEARCH`, `SEND`, or `GET_CONTENT` from being executed merely because a manifest exposes them. Appendix D shows representative skill and shortcut entries.

4.5 Reflect: Trajectory Distillation and Skill Evolution

Post-run summarization. After each GUI task, reflection condenses the trajectory into a short note of where execution ended and what remains, so a partial or blocked task becomes a checkpoint the host can resume without redoing finished work. The post-run processor summarizes the trace and, when enabled, evaluates the outcome before learning from it. Runs with no useful signal, e.g., empty, cancelled, timed out before progress, or completed entirely by an already reused skill, are not written to the long-term stores.

Skill extraction. For trajectories selected for skill learning, reflection extracts a reusable procedure rather than storing raw actions for replay. It rewrites GUI events into structured evidence: the task, platform, observed apps, action sequence, action parameters, screenshots, target-control hints, and inferred state contracts. A

vision LLM receives this evidence under a restricted prompt that permits only declarative action sequences. The candidate skill is normalized and checked for supported actions, declared parameters, executable fixed fields, reusable app scope, and valid-state coverage. Accepted steps inherit the state contracts inferred from the trajectory, so later execution validates screen state instead of replaying an ungrounded script. Offline extraction uses the same agent after filtering short or abnormal traces, keeping bounded successful prefixes, clustering structurally identical skills, and recording success counts for retrieval priority.

Skill evolution. KNOWACT-GUICLAW separates repair from new extraction. When a trajectory shows a reused skill failing, reflection records the failed step, screen evidence, error, original skill, and prior feedback, then asks an evolution prompt to update that same skill in place: it may narrow the description, add guarded optional obstacle handling, or refresh stale targets and state contracts, but it may not replace the skill with an unrelated workflow or add destructive terminal actions. Repair therefore takes precedence over new extraction; only when no failed reused skill is present does the system mine a new skill from reusable behavior.

Shortcut validation. Shortcut candidates are mined from app manifests as discovery evidence, not trusted actions. A validation run launches candidate variants, records the foreground app, ADB output, UI tree, and screenshot, and asks a verifier to return usability, page status, payload preservation, parameters, and a natural-language capability description. Only page-validated records are promoted into one-step skills; merely launchable records remain candidates unless explicitly allowed by the validation configuration. Appendix C gives the validation prompt.

Experience memory. Experience memory stores textual policies derived from running trajectories. Inspired by ReasoningBank (Ouyang et al., 2026), KNOWACT-GUICLAW formats each trace into a concise action-and-UI summary and uses separate success and failure prompts to induce at most a few actionable memory items. The memory inducer skips short or abnormal traces, resolves outcome and app at the per-trace level, caps the number of retained items per task, and drops near-duplicates within the same app. These lessons are distinct from executable skills and feed later tasks at both levels: routing draws on them to choose apps and decomposition patterns, while the GUI agent uses them as hints about layout, shortcut reliability, and recovery. Appendix C gives the corresponding induction prompts and filtering rules.

5 Experiments

5.1 Benchmarks and Metrics

We evaluate KNOWACT-GUICLAW on two mobile GUI benchmarks that include diverse real-user tasks. MobileWorld (Kong et al., 2025) is our primary benchmark: it emphasizes long-horizon mobile tasks, many of which span multiple applications. From its 201 tasks over 20 applications, we use the 117-task GUI-Only subset, scored by the benchmark's native deterministic evaluators. AndroidDaily (StepFun, 2025) is a complementary end-to-end benchmark that groups tasks by type, complexity, and ambiguity; many of its tasks require returning an explicit answer rather than only manipulating the UI. Lacking a native evaluator, we score its GUI-only tasks with `qwen3.5-flash` as an LLM judge and its answer-returning tasks with two human experts, with correct tasks scoring 1.0 and partial tasks 0.5.

Setup. Unless noted, KNOWACT-GUICLAW pairs a Qwen3.5-397B-A17B host with a Qwen3.5-35B-A3B GUI executor. In the Kimi-K2.6 configurations of Table 1, Kimi-K2.6 serves as both the host and GUI executor. The Kimi-to-Qwen transfer configuration retains the default Qwen host–executor pairing and instead uses a joint set of experience memory and skills distilled from trajectories generated by the Kimi-K2.6 host–executor configuration. All MobileWorld configurations use the same 117-task GUI-Only subset, native deterministic evaluators, and 50-step cap. Following the public leaderboard, our primary MobileWorld metric is the single-run success rate (SR), i.e. pass@1; we report pass@3 only as a repeated-attempt upper bound, in a separate table (Table 5). For AndroidDaily we report a resolved setting over the 194 available tasks and an all setting that scores the 41 unavailable entries as 0 across all 235 tasks. The AndroidDaily evaluation uses iOS devices.

Efficiency metrics. We report SR together with execution cost: GUI steps, the number of GUI task invocations, executed GUI-trace tokens, and host tokens. The total column measures the executed GUI traces and is comparable across systems; host total measures the additional generation that the host spends to route, coordinate, call external tools such as web search, and resolve eligible subtasks.

Table 1: MobileWorld GUI-Only success rate (pass@1). External rows are grouped by the public MobileWorld leaderboard categories (Kong et al., 2025). All KnowAct-GUIClaw rows are our own runs; the Kimi-to-Qwen transfer row equips the 35B executor with experience memory and skills distilled from Kimi-K2.6 trajectories. Rows matching configurations B, C, and F are analyzed in Table 2, and Table 5 reports the repeated-attempt upper bound.

Model	SR (pass@1, %)
General models	
Seed-2.0-Pro (Bytedance Seed, 2026)	63.2
GPT-5.5 (OpenAI, 2026)	62.4
Gemini-3.1-Pro-Preview (Google DeepMind, 2026)	58.1
Claude-Opus-4.7 (Anthropic, 2026)	56.4
Kimi-K2.6 (Moonshot AI, 2026b)	55.6
Kimi-K2.5 (Moonshot AI, 2026a)	49.6
Claude-Sonnet-4.5 (Anthropic, 2025)	47.8
Qwen3.5-397B-A17B (Qwen Team, 2026)	42.7
Qwen3.5-122B-A10B (Qwen Team, 2026)	35.0
Qwen3.5-35B-A3B (Qwen Team, 2026)	24.8
Qwen3-VL-235B-A22B (Bai et al., 2025)	12.8
Agentic systems	
Gemini-3-Pro + UI-Ins-7B (Google DeepMind, 2026; Chen et al., 2025)	55.6
GPT-5 + UI-Ins-7B (Team, 2026; Chen et al., 2025)	54.0
Claude-4.5-Sonnet + UI-Ins-7B (Anthropic, 2025; Chen et al., 2025)	47.8
Specialized GUI models	
GUI-Owl-1.5-32B (Xu et al., 2026)	43.9
ForgeOwl-8B (Liu et al., 2026)	41.0
MAI-UI-235B-A22B (Zhou et al., 2025)	39.7
GUI-Owl-1.5-8B (Xu et al., 2026)	38.2
MAI-UI-32B (Zhou et al., 2025)	36.2
Doubao-1.5-UI-TARS (Qin et al., 2025)	26.3
UI-Venus-72B (Gu et al., 2025)	16.4
GUI-Owl-7B (Ye et al., 2025)	7.7
Ours (Open-source Qwen3.5)	
35B + host & memory	**34.5**
35B + host, memory & skills	**37.9**
35B + host, Kimi-derived memory & skills	**41.0**
397B host acts directly	**46.2**
Ours (Open-source Kimi-K2.6)	
+ host & memory	**61.5**
+ host, memory & skills	**64.1**

5.2 Main Results of MobileWorld

Table 1 compares KnowAct-GUIClaw with the public MobileWorld leaderboard. The Qwen3.5-35B-A3B GUI executor reaches 24.8 SR on its own; running it inside KnowAct-GUIClaw with the host and experience memory raises pass@1 to 34.5, and enabling skills on top raises it to 37.9. Equipping the same 35B executor with experience memory and skills distilled from Kimi-K2.6 trajectories further raises SR to 41.0, a 16.2-point gain over the base executor and 3.1 points above the standard 35B host–memory–skills configuration. With Kimi-K2.6 as the base model, KnowAct-GUIClaw reaches 61.5 with the host and experience memory and 64.1 after enabling skills, the highest SR in the table. Letting the 397B Qwen host act directly reaches 46.2; using only open Qwen3.5 models, this configuration remains above the public Qwen3.5-397B-A17B plain-GUI score (42.7) and every specialized end-to-end GUI model listed. We isolate the host–memory and skills increments next.

5.3 Ablation and Efficiency Analysis

Table 2 reports a system-level ablation along two executor scales.

Table 2: MobileWorld GUI-Only ablation and efficiency (our runs). Rows A–C use the Qwen3.5-35B-A3B GUI executor and rows D–F the Qwen3.5-397B-A17B; **+host & mem** adds the KnowAct-GUIClaw host, router, and experience memory, and **+skills** enables skills on top of that setting. In E–F the 397B host resolves eligible subtasks directly rather than delegating each to the 35B executor (Section 4.2). Row D is our plain-GUI 397B run, distinct from its 42.7 public leaderboard score. Total is GUI-trace tokens per task; host total is the host's own generation, including external tool calls, and is the extra token cost KnowAct-GUIClaw adds on top. Highlighted rows mark the host- and skill-augmented configurations.

Config	Setting	SR (%)	GUI steps	GUI tasks	Total	Host Total
GUI executor: Qwen3.5-35B-A3B						
A	exec. only	24.8	26.7	1.0	281,266	–
B	+host & mem	34.5	26.8	2.3	279,211	65,224
C	+skills	37.9	25.1	2.5	278,289	63,792
GUI executor: Qwen3.5-397B-A17B						
D	exec. only	40.7	26.1	1.0	254,459	–
E	+host & mem	43.3	26.8	1.4	273,352	10,096
F	+skills	**46.2**	**23.7**	1.7	**260,516**	10,982

Host and memory raise accuracy at a modest comparable cost. Adding the host, router, and experience memory raises SR at both executor scales (A→B, $24.8 \rightarrow 34.5$; D→E, $40.7 \rightarrow 43.3$). Total tokens stay flat for the 35B executor (281k vs. 279k) but rise about 7% for the 397B executor (254k to 273k), where the host splits a task into more GUI subtrajectories. Because total tokens are dominated by visual observation, the accuracy gain reflects better task organization rather than a larger GUI-executor budget; the host total is $65{,}224$ tokens per task when the host must coordinate the weaker 35B executor across 2.3 GUI task invocations (about 23% of total), but only $10{,}096$ with the 397B host at 1.4 invocations (under 4%).

Skills cut steps and total tokens (B→C, E→F). Enabling skills on top of the host–memory setting lowers GUI steps and total tokens—modestly for the 35B executor, and more clearly for the 397B host, where steps fall from 26.8 to 23.7 and total tokens by about 5%—while SR rises rather than falls. The saving comes from shorter trajectories and fewer screenshot-heavy observations. Table 3 isolates this effect on the tasks that actually invoke a skill.

Experience memory and skills transfer across base models. Table 1 includes a cross-model transfer test in which the Qwen3.5-35B-A3B executor uses experience memory and executable skills distilled from Kimi-K2.6 trajectories. This transferred configuration reaches 41.0 SR, compared with 37.9 for the standard 35B host–memory–skills configuration and 24.8 for the base executor. The result shows that both textual experience rationales and state-validated executable behaviors can carry task knowledge across model families rather than remaining tied to the model that generated the source trajectories. Together with the Kimi-K2.6 results (61.5 with host and memory; 64.1 with skills), this establishes transferability in the tested Kimi-to-Qwen direction without assuming universal model-independent transfer.

A capable host should not route everything through the GUI (F). Letting the 397B host resolve eligible information-query subtasks directly, rather than delegating each to the 35B executor, yields the best SR (46.2) while lowering both cost components relative to the delegated configuration C: the host answers eligible subtasks itself instead of spawning a GUI task for each (1.7 GUI task invocations versus 2.5 for C), so total tokens fall to $260{,}516$ (from $278{,}289$) and host total to $10{,}982$ tokens per task (from $63{,}792$). F thus delivers the best accuracy at the lowest GUI-trace execution cost among the Qwen ablations, while keeping host overhead minimal. For external context, Qwen3.7-Plus reaches 43.6 SR with 30.0 steps, while the public Kimi-K2.6 baseline reaches 55.6; our Kimi host–memory and skill-augmented configurations reach 61.5 and 64.1, respectively (Table 1).

Skill reuse, measured where it applies. Restricting to the tasks that invoke a skill concentrates the same effect (Table 3): reuse removes about three GUI steps per task (-3.3), cuts total and prompt tokens by roughly 6%, and improves SR ($+4.8$ and $+1.9$ points).

Section 5.5 complements these aggregate comparisons with qualitative evidence for routing and information transfer, experience-memory correction, navigation compression through skills and shortcuts, and host recovery and direct participation.

Table 3: Skill-reuse effect on the MobileWorld tasks that invoke at least one skill (83 with the 35B executor, 87 with the 397B host), each compared against the same tasks run without skills. SR and pass@3 deltas are in percentage points.

	Qwen3.5-35B-A3B executor			Qwen3.5-397B-A17B executor		
Metric (skill-using tasks)	+ skills	no skills	Δ	+ skills	no skills	Δ
GUI steps / task	25.7	29.0	−3.3	22.3	25.6	−3.3
total tokens / task	284,279	303,014	−6.2%	242,930	258,084	−5.9%
single-run SR (%)	40.6	35.7	+4.9	50.2	48.3	+1.9
pass@3 (%)	54.2	53.0	+1.2	63.2	63.2	+0.0

5.4 AndroidDaily Results

Table 4: AndroidDaily (end-to-end) performance by task type, complexity, and ambiguity. Our AndroidDaily evaluation is conducted on iOS devices. For KNOWACT-GUICLAW, resolved excludes the 41 unavailable entries and rates the remaining 194 tasks, while all scores those entries as 0 over all 235 tasks.

	Task Type			Complexity			Ambiguity			
Model / Setting	Filter	Query	Analyze	Atomic	Comp.	Cond.	Low	Mid	High	Total
UI-TARS-1.5	57.64	65.97	36.71	61.41	13.64	60.38	57.05	54.90	57.89	56.64
Step-GUI-4B	44.77	64.29	33.72	54.03	19.61	42.86	51.21	38.32	59.52	49.06
Step-GUI-8B	52.50	63.82	32.95	59.09	14.00	42.86	54.08	44.55	61.54	52.50
Ours (Resolved)	80.56	76.88	77.08	81.25	62.50	75.00	78.89	77.50	78.95	**78.61**
Ours (All)	68.40	66.13	51.39	70.98	41.67	53.23	64.94	65.96	62.50	64.89

We conduct the AndroidDaily evaluation on iOS devices. AndroidDaily contains many long-horizon tasks that require cross-application analysis and an explicit answer rather than a single visual goal. Table 4 shows that KNOWACT-GUICLAW's largest margins over GUI-only baselines fall on Analyze and complex (Comp.) tasks, where explicit subtask decomposition, blackboard information transfer, and retrieved memory reduce repeated trajectory solving. The difference between the resolved and all settings reflects app availability and environment mismatch: these factors, rather than policy quality alone, account for a substantial fraction of end-to-end failure.

5.5 Case Study

This case study uses representative workflows to illustrate four characteristics of KNOWACT-GUICLAW: routing and information transfer, experience-memory correction, navigation compression through skills and shortcuts, and host recovery and direct participation. The screenshots come from actual executions; host cards, step labels, per-step thoughts, and blackboard pills are overlays added for readability.

Routing and information transfer. Long-horizon, cross-app tasks amplify memory decay and goal drift when one GUI trajectory must retain every intermediate summary. KNOWACT-GUICLAW instead decomposes the request into app-scoped GUI tasks and passes only declared outputs through the blackboard. Figure 6(a) isolates this handoff in an Email-to-Clock workflow, while Figure 7 shows a longer Email-to-Messages-to-Maps workflow in which the resolved address becomes an explicit input to both downstream GUI tasks. The cart-to-SMS case in Appendix A similarly transfers product names, an order number, and a recipient phone number from shopping to messaging (Figure 8). These bounded handoffs reduce the history each GUI task must preserve and keep later goals tied to explicit inputs.

Experience-memory correction. Experience memory supplies prior failure lessons as advisory task context before low-level control begins. Figure 3 contrasts an invite-link task with and without retrieved memory. Without memory, GUIClaw remains in Mastodon's mobile settings and follows a nonproductive path; with memory, it recalls that advanced invite links require the web administration panel and switches to the productive interface. The retrieved lesson corrects app and route selection without replacing live screen evidence.

Table 5: MobileWorld repeated-attempt upper bound for the A–F configurations of Table 2. "Any of 3" counts a task solved if at least one of three runs succeeds; "all 3" requires all three. SR (pass@1) is repeated for reference. This shows our model steadily boosts accuracy without large random volatility.

Metric	A	B	C	D	E	F
single-run SR (%)	24.8	34.5	37.9	40.7	43.3	46.2
pass@3, any of 3 (%)	34.2	47.9	51.3	51.3	59.0	59.8
pass@3, all 3 (%)	15.4	18.8	26.5	30.8	29.1	32.5

Navigation compression through skills and shortcuts. Skills compress repeated procedures, while validated shortcuts eliminate navigation when they can reach a task-relevant page directly. In Figure 5, one validated JD search shortcut lands on the product-results page, whereas the corresponding Taobao branch requires five ordinary GUI steps. In Figure 8, a validated messaging shortcut opens the SMS compose view with the recipient phone number and message body already populated, leaving GUIClaw to verify and send the message. These cases visualize the navigation savings measured more broadly on skill-using tasks in Table 3.

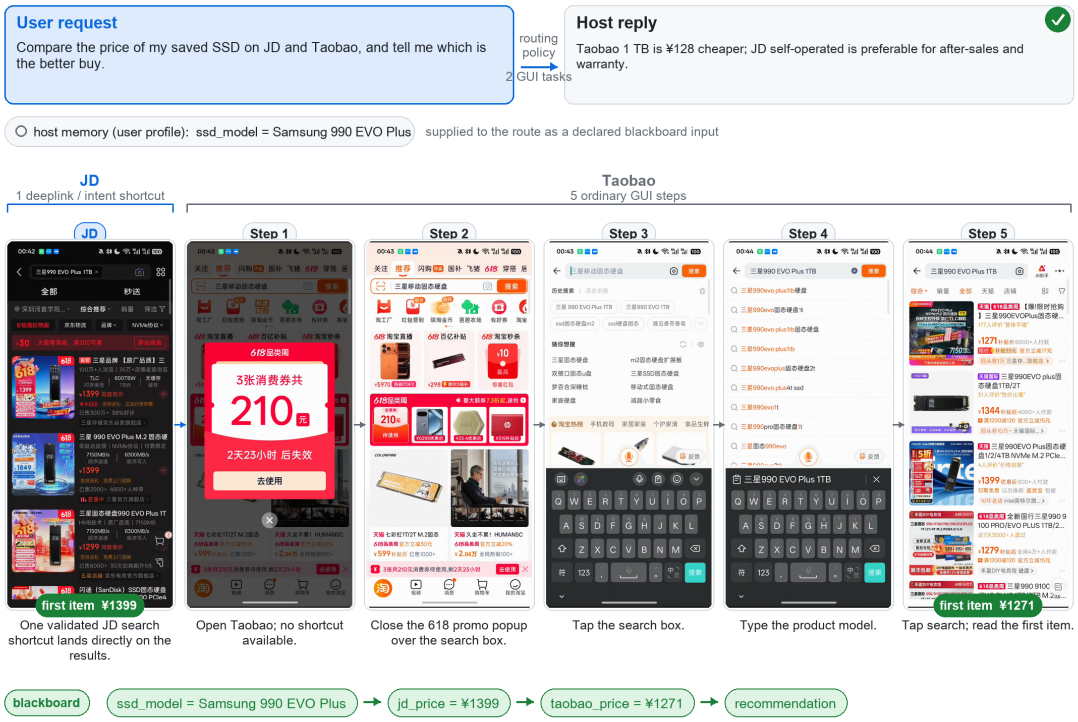

Figure 5: KnowAct-GUIClaw execution of a cross-app price comparison. The host supplies the product model from the user profile and routes two GUI tasks. A validated JD search shortcut lands directly on the results page, whereas Taobao requires five ordinary GUI steps. The blackboard carries both observed prices into the host's recommendation, illustrating the step and token savings in Table 3.

Host recovery and direct participation. The host intervenes when GUI execution fails or returns a value that does not satisfy a downstream contract. Figure 6(b) shows failure-driven re-planning: after a message-by-name attempt fails because the contact is absent, the host records the failed lookup, recovers the phone number from a resume, and delegates a new Messages GUI task. Figure 7 shows direct tool participation after the Email GUI task returns only the ambiguous hotel name "Harvard Square Hotel." The host resolves the full street address through web search before delegating the SMS and Maps GUI tasks, avoiding a longer GUI-only address lookup; the final Maps task reports a 13-minute walk. The two trajectories distinguish host-side re-planning from tool-assisted information resolution while preserving typed subtask boundaries.

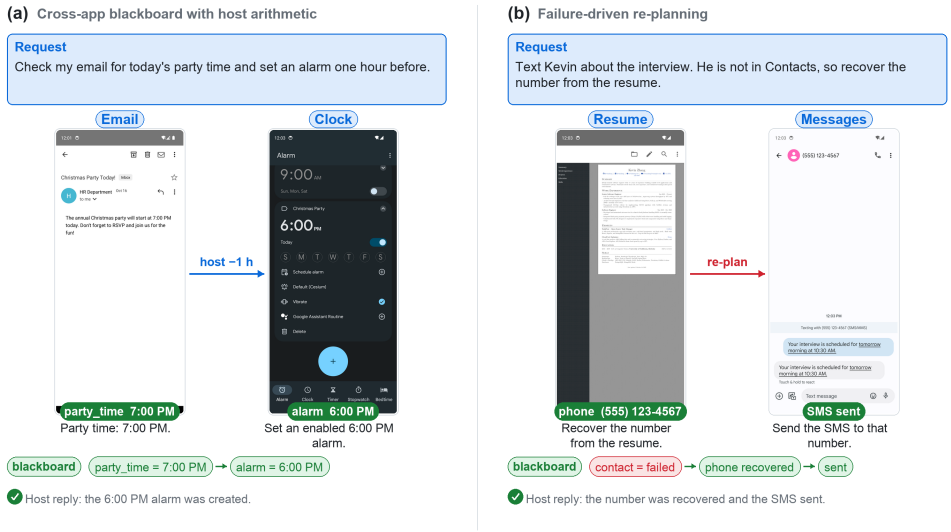

Figure 6: Cases of our workflow with attribution experience. **(a)** Email-to-alarm transfers the observed party time through the blackboard; the host subtracts one hour before the GUI subagent sets the alarm. **(b)** Failure-driven re-planning records a failed contact lookup, recovers the phone number from a resume, and delegates the final Messages GUI task.

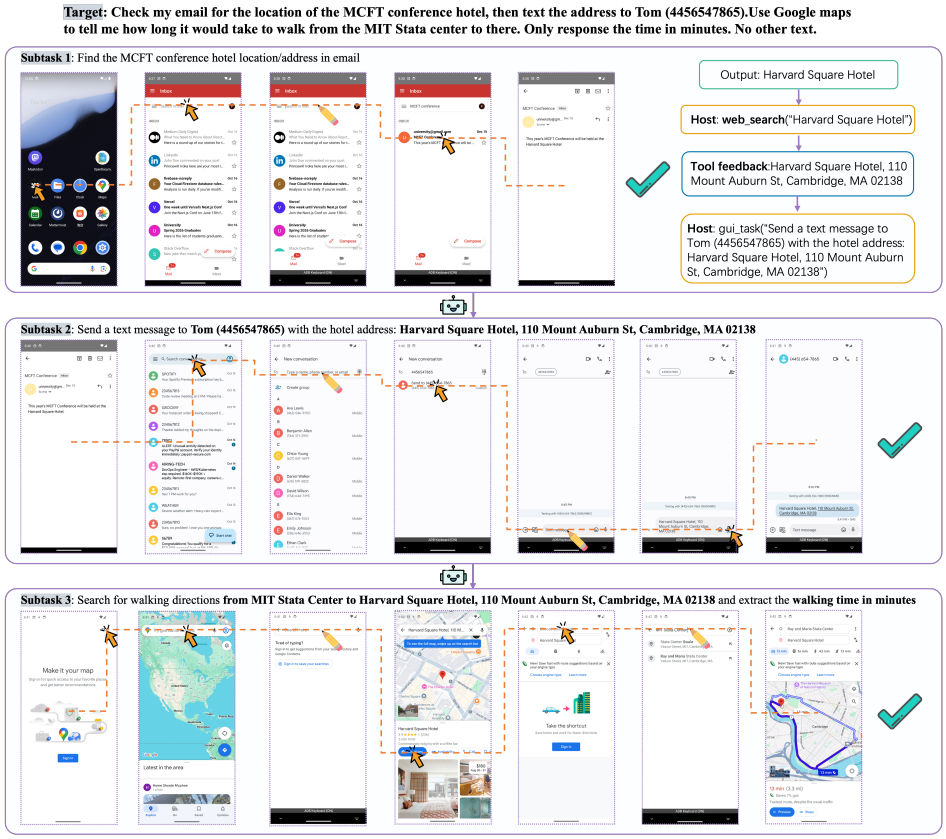

Figure 7: Host-mediated recovery in a conference-location task. The Email GUI task returns only the hotel name, so the host resolves the full address through web search before delegating the Messages and Maps GUI tasks. Maps reports a 13-minute walk. The workflow combines partial GUI evidence with external tools while preserving typed subtask boundaries.

5.6 Cross-Platform Checks

We further test whether the same KnowAct-GUIClaw interface remains usable outside the Android evaluation stack. On HarmonyOS, we run MobileWorld-derived tasks that are not tied, or only weakly tied, to Android-specific mirrored app states; weakly tied cases are manually initialized before evaluation, and outcomes are judged with `qwen3.5-flash`. On Windows, we use a manually designed desktop set to check basic usability across browser, file, office, terminal, and system-control workflows. *Under this protocol,* KnowAct-GUIClaw *solves* $48/63$ *HarmonyOS tasks (*76.2%*) and* $21/30$ *Windows tasks (*70.0%*).* Table 6 reports ten source-matched cases from each platform.

Table 6: Representative cross-platform usability cases. HarmonyOS instructions are translated into English when necessary while preserving their source requirements. Windows instructions name one concrete application wherever the source offered alternatives or referred to a generic application.

Instruction	Success
HarmonyOS mobile checks	
Next Saturday from 10:00 a.m. to 12:30 p.m., I will travel to Shanghai Hongqiao Railway Station. Add a Calendar event named "Business Trip." Find attractions within 10 km of the station that I can visit before work on Monday, and place them in the event description as "attraction name: address," separated by commas.	Yes
Check my calendar and send a WeChat message to Su with the dates of my arrival in Shanghai. The message must contain only the two dates in `MM/DD/YYYY` format, separated by a comma.	Yes
I received a coffee invitation for 3:00 p.m. tomorrow. Check my calendar; if I am available, reply "OK" to Ping An Xi Le in QQ and create the corresponding calendar event. Otherwise, reply "Not available in this time slot."	Yes
Plan exactly one shortest taxi route in Chengdu from Chengdu Shuangliu International Airport Terminal 2 to my hotel at No. 8, South Section of Chunxi Road, Jinjiang District, visiting exactly Kuanzhai Alley and Jinli Ancient Street in either order. Send Ping An Xi Le in QQ the names and coordinates of all four locations as "name: longitude, latitude," the shortest visit order, the three driving distances in meters, and the total driving distance in meters; separate each ordered list with commas.	Yes
An error message on my phone contains the keyword `TurnOffWifi`. Search the `google-research/android_world` GitHub repository for related issues. If similar issues exist, send every issue link, separated by commas, to Ping An Xi Le in QQ; otherwise send "no turnoffwifi issues."	Yes
Find the resume file downloaded most recently within the past month in Downloads, and send it to my HR colleague with the subject `candiaditaes_cv`.	Yes
Take a selfie and share it with Jimmy via email.	Yes
Increase the font size and icons on my phone to the maximum setting.	Yes
Set a weekend alarm for 8:25 a.m. with the ringtone "beebeep" and vibration off.	No
Halloween is approaching. Use TaoBao to place an order for a set of temporary tattoos, and hand control back to me when the payment page appears.	No
Windows desktop checks	
Open the Ctrip website and search for a one-way flight from Beijing to Tokyo on the Friday after next. Filter for nonstop flights, sort by price in ascending order, and select the cheapest option.	Yes
Open Baidu Maps in Google Chrome and search for "Starbucks near Tiananmen Square." Select the highest-rated result from the left-hand list, open its details, copy its building or street address, and save the address to `address.txt` on the desktop using Notepad.	Yes
Open `loop.py` in Visual Studio Code, add a breakpoint on line 3, open Run and Debug, and start debugging.	Yes
Use Google Chrome to find the current Apple (AAPL) share price, calculate the value of 100 shares, and save the total to `stock.txt` on the desktop using Notepad.	Yes
Open Windows Settings, retrieve the detailed operating-system version, and save it in a new draft email in Microsoft Outlook.	Yes
Open the PDF report in Documents with Microsoft Edge, translate the first paragraph of Chapter 1 into English using Google Translate, and save the translation as a plain-text file on the desktop using Notepad.	Yes
Open GitHub in Google Chrome, search for the `nanobot` project, copy its latest commit hash, and save it to `commit_record.txt` on the desktop using Notepad.	Yes
Open Microsoft PowerPoint, create a blank slide, insert a local image, and apply a "Fly In" animation to the image.	Yes
I will send a test message in Slack within the next minute. Keep the desktop idle; when the Slack notification appears, click it immediately, open the conversation, and reply "Received."	No
Open the locally installed WeChat, open File Transfer Assistant, send the text "Test emoji," and then use the built-in emoji panel to send a "laughing" emoji.	No

HarmonyOS failure modes. The failed HarmonyOS cases expose both low-level control difficulties and state-semantic mismatches. In the weekend-alarm case, the minute field uses an inertial wheel picker. GUIClaw repeatedly swipes within a nearby range but cannot settle on 25 minutes, so it exhausts the step budget without setting 8:25 a.m. In the temporary-tattoo case, the agent stops on the order-confirmation screen before submitting the order, rather than handing control back after reaching the subsequent payment page. This premature termination reflects application-level UI diversity: after a payment method is selected, TaoBao presents a "Pay Now" button, which the model misinterprets as evidence that the current screen is already the payment page; in fact, the payment page appears only after that button is pressed. The full set also shows system-UI mismatches: several quick-setting tasks repeatedly open the left notification pane even though the relevant HarmonyOS controls require a swipe from the right.

Windows failure modes. Windows failures expose three distinct grounding bottlenecks. First, the WeChat case reaches File Transfer Assistant and sends the text correctly, but selects a different face from the built-in panel instead of the requested laughing emoji. This error reveals a limitation in mapping a natural-language affect label to the corresponding visually similar, unlabeled icon. Second, the notification case requires sustained observation followed by an immediate click on a short-lived toast, exposing weak temporal grounding. The remaining failures primarily involve continuous geometric control, deep application-specific controls, and targets outside the current viewport, as seen in free-form canvas manipulation, window arrangement, and bottom-of-page actions. Thus, desktop reliability depends not only on pointer precision and timing, but also on semantic grounding between language and graphical symbols.

5.7 Reproducibility

MobileWorld results use the benchmark's deterministic evaluators on the 117 GUI-Only tasks at a 50-step cap. We log GUI-executor traces and host calls separately, so that the GUI-trace total and host total columns of Table 2 can be reproduced independently. AndroidDaily scoring (`qwen3.5-flash` for GUI-only tasks, two human experts under the $1.0/0.5/0$ rule for answer-returning tasks) and the resolved/all splits follow the protocol described above. You can see the experimental logs at https://github.com/HITsz-TMG/KnowAct/releases/tag/Result.

6 Conclusion

This work proposes KnowAct-GUIClaw, a cross-platform GUI agent framework built on the "Know Deeply, Act Perfectly" paradigm to address two core limitations of mainstream OpenClaw-style agent systems: insufficient cross-device GUI interaction capacity and the absence of native self-evolution mechanisms. The framework instantiates a four-stage Know–Route–Act–Reflect closed-loop pipeline that decouples high-level task orchestration (host agent) from low-level visual device manipulation (GUI subagent). It introduces two core reusable storage modules: attribution-aware persistent experience memory and state-validated self-evolving skill libraries, alongside a typed blackboard information transfer protocol to standardize cross-application data flow and eliminate fuzzy free-text context loss. Future work targets tighter native integration of the Knowledge module, external general-purpose tools, and the GUI subagent to eliminate rigid sequential pipeline handoffs. We will design a unified joint planner to concurrently weigh knowledge retrieval, non-visual tool calls and GUI actions within one decision cycle and cut host-subagent communication overhead.

7 Contributors

Core Contributors

Yunxin Li, Jinchao Li, Baotian Hu, Min Zhang

Contributors

Shibo Su, Zhenran Xu, Chenrui Zhao, Tongshu Bian, Xiaoman Liang, Meishan Zhang

Corresponding Author

Baotian Hu

Harbin Institute of Technology, Shenzhen; Shenzhen Loop Area Institute

Email: hubaotian@hit.edu.cn

References

Android Developers. Android Developers: Create deep links to app content. https://developer.android.com/training/app-links/deep-linking, 2026a. Accessed 2026-06-10.

Android Developers. Android Developers: Intents and intent filters. https://developer.android.com/guide/components/intents-filters, 2026b. Accessed 2026-06-10.

Anthropic. Claude sonnet 4.5, 2025. URL https://www.anthropic.com/news/claude-sonnet-4-5.

Anthropic. Claude opus 4.7, 2026. URL https://www.anthropic.com/news/claude-opus-4-7.

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, et al. Qwen3-vl technical report, 2025. URL https://arxiv.org/abs/2511.21631.

Bytedance Seed. Seed2.0 model card: Towards intelligence frontier for real-world complexity, 2026. URL https://arxiv.org/abs/2607.00248.

Liangyu Chen, Hanzhang Zhou, Chenglin Cai, Jianan Zhang, Panrong Tong, Quyu Kong, Xu Zhang, Chen Liu, Yuqi Liu, Wenxuan Wang, Yue Wang, Qin Jin, and Steven Hoi. Ui-ins: Enhancing gui grounding with multi-perspective instruction-as-reasoning, 2025. URL https://arxiv.org/abs/2510.20286.

Tianyi Chen, Yinheng Li, Michael Solodko, Sen Wang, Nan Jiang, Tingyuan Cui, Junheng Hao, Jongwoo Ko, Sara Abdali, Leon Xu, Suzhen Zheng, Hao Fan, Pashmina Cameron, Justin Wagle, and Kazuhito Koishida. CUA-Skill: Develop skills for computer using agent, 2026. URL https://arxiv.org/abs/2601.21123.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2Web: Towards a generalist agent for the web, 2023. URL https://arxiv.org/abs/2306.06070.

Pengfei Du. Memory for autonomous llm agents: Mechanisms, evaluation, and emerging frontiers, 2026. URL https://arxiv.org/abs/2603.07670.

Google DeepMind. Gemini 3.1 pro, 2026. URL https://deepmind.google/models/gemini/pro/.

Zhangxuan Gu, Zhengwen Zeng, Zhenyu Xu, et al. Ui-venus technical report: Building high-performance ui agents with rft, 2025. URL https://arxiv.org/abs/2508.10833.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. WebVoyager: Building an end-to-end web agent with large multimodal models, 2024. URL https://arxiv.org/abs/2401.13919.

HKUDS. nanobot: The ultra-lightweight personal ai agent. https://github.com/HKUDS/nanobot, 2026. Software repository, accessed 2026-05-13.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxuan Zhang, Juanzi Li, Bin Xu, Yuxiao Dong, Ming Ding, and Jie Tang. CogAgent: A visual language model for gui agents, 2024. URL https://arxiv.org/abs/2312.08914.

Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. Artificial Intelligence, 101(1–2):99–134, 1998.

Raghav Kapoor, Yash Parag Butala, Melisa Russak, Jing Yu Koh, Kiran Kamble, Waseem Alshikh, and Ruslan Salakhutdinov. OmniACT: A dataset and benchmark for enabling multimodal generalist autonomous agents for desktop and web, 2024. URL https://arxiv.org/abs/2402.17553.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Chong Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan Salakhutdinov, and Daniel Fried. VisualWebArena: Evaluating multimodal agents on realistic visual web tasks, 2024. URL https://arxiv.org/abs/2401.13649.

Quyu Kong, Xu Zhang, Zhenyu Yang, Nolan Gao, Chen Liu, Panrong Tong, Chenglin Cai, Hanzhang Zhou, Jianan Zhang, Liangyu Chen, Zhidan Liu, Steven Hoi, and Yue Wang. MobileWorld: Benchmarking autonomous mobile agents in agent-user interactive and MCP-augmented environments, 2025. URL https://arxiv.org/abs/2512.19432.

Jinchao Li, Yunxin Li, Chenrui Zhao, Zhenran Xu, Baotian Hu, and Min Zhang. Windowsworld: A process-centric benchmark of autonomous gui agents in professional cross-application environments. arXiv preprint arXiv:2604.27776, 2026.

Kevin Qinghong Lin, Linjie Li, Difei Gao, Zhengyuan Yang, Shiwei Wu, Zechen Bai, Weixian Lei, Lijuan Wang, and Mike Zheng Shou. ShowUI: One vision-language-action model for gui visual agent, 2024. URL https://arxiv.org/abs/2411.17465.

Guangyi Liu, Pengxiang Zhao, Liang Liu, Zhiming Chen, Yuxiang Chai, Shuai Ren, Hao Wang, Shibo He, and Wenchao Meng. LearnAct: Few-shot mobile gui agent with a unified demonstration benchmark, 2025. URL https://arxiv.org/abs/2504.13805.

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

Moonshot AI. Kimi k2.5 technical report, 2026a. URL https://github.com/MoonshotAI/Kimi-K2.5/blob/master/tech_report.pdf.

Moonshot AI. Kimi k2.6 tech blog: Advancing open-source coding, 2026b. URL https://www.kimi.com/blog/kimi-k2-6.

Nous Research. Hermes Agent: The agent that grows with you. https://github.com/NousResearch/hermes-agent, 2026. Software repository, accessed 2026-05-13.

OpenAI. Introducing gpt-5.5, 2026. URL https://openai.com/index/introducing-gpt-5-5/.

OpenClaw Contributors. OpenClaw: Personal ai assistant. https://github.com/openclaw/openclaw, 2026. Software repository, accessed 2026-05-13.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. Reasoningbank: Scaling agent self-evolving with reasoning memory, 2026. URL https://arxiv.org/abs/2509.25140.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior, 2023. URL https://arxiv.org/abs/2304.03442.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, Wanjun Zhong, Kuanye Li, Jiale Yang, Yu Miao, Woyu Lin, Longxiang Liu, Xu Jiang, Qianli Ma, Jingyu Li, Xiaojun Xiao, Kai Cai, Chuang Li, Yaowei Zheng, Chaolin Jin, Chen Li, Xiao Zhou, Minchao Wang, Haoli Chen, Zhaojian Li, Haihua Yang, Haifeng Liu, Feng Lin, Tao Peng, Xin Liu, and Guang Shi. Ui-tars: Pioneering automated gui interaction with native agents, 2025. URL https://arxiv.org/abs/2501.12326.

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

Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. Android in the wild: A large-scale dataset for android device control, 2023. URL https://arxiv.org/abs/2307.10088.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy Lillicrap, and Oriana Riva. AndroidWorld: A dynamic benchmarking environment for autonomous agents, 2025. URL https://arxiv.org/abs/2405.14573.

Xiaoming Ren, Ru Zhen, Chao Li, Yang Song, Qiuxia Hou, Yanhao Zhang, Peng Liu, Qi Qi, Quanlong Zheng, Qi Wu, Zhenyi Liao, Binqiang Pan, Haobo Ji, and Haonan Lu. X-omniclaw technical report: A unified mobile agent for multimodal understanding and interaction, 2026. URL https://arxiv.org/abs/2605.05765.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://arxiv.org/abs/2303.11366.

StepFun. Step-GUI technical report, 2025. URL https://arxiv.org/abs/2512.15431.

Fei Tang, Zhiqiong Lu, Boxuan Zhang, Weiming Lu, Jun Xiao, Yueting Zhuang, and Yongliang Shen. ClawGUI: A unified framework for training, evaluating, and deploying gui agents, 2026. URL https://arxiv.org/abs/2604.11784.

OpenAI Team. Openai gpt-5 system card, 2026. URL https://arxiv.org/abs/2601.03267.

Tu Vu, Baotian Hu, Tsendsuren Munkhdalai, and Hong Yu. Sentence simplification with memory-augmented neural networks. In Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers), pp. 79–85, 2018.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models, 2023. URL https://arxiv.org/abs/2305.16291.

Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. Mobile-Agent: Autonomous multi-modal mobile device agent with visual perception, 2024. URL https://arxiv.org/abs/2401.16158.

Michael Wooldridge. An Introduction to MultiAgent Systems. John Wiley and Sons, 2009.

Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. OS-Copilot: Towards generalist computer agents with self-improvement, 2024a. URL https://arxiv.org/abs/2402.07456.

Zhiyong Wu, Zhenyu Wu, Fangzhi Xu, Yian Wang, Qiushi Sun, Chengyou Jia, Kanzhi Cheng, Zichen Ding, Liheng Chen, Paul Pu Liang, and Yu Qiao. OS-ATLAS: A foundation action model for generalist gui agents, 2024b. URL https://arxiv.org/abs/2410.23218.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments, 2024. URL https://arxiv.org/abs/2404.07972.

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

Yiheng Xu, Zekun Wang, Junli Wang, Dunjie Lu, Tianbao Xie, Amrita Saha, Doyen Sahoo, Tao Yu, and Caiming Xiong. Aguvis: Unified pure vision agents for autonomous gui interaction, 2025. URL https://arxiv.org/abs/2412.04454.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629.

Jiabo Ye, Xi Zhang, Haiyang Xu, Haowei Liu, Junyang Wang, Zhaoqing Zhu, Ziwei Zheng, Feiyu Gao, Junjie Cao, Zhengxi Lu, Jitong Liao, Qi Zheng, Fei Huang, Jingren Zhou, and Ming Yan. Mobile-agent-v3: Fundamental agents for gui automation, 2025. URL https://arxiv.org/abs/2508.15144.

Chi Zhang, Zhao Yang, Jiaxuan Liu, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. AppAgent: Multimodal agents as smartphone users, 2023. URL https://arxiv.org/abs/2312.13771.

Pengxiang Zhao, Guangyi Liu, YaoZhen Liang, Weiqing He, Zhengxi Lu, WenHao Wang, Yuehao Huang, Yuxiang Chai, Zhaolu Kang, Yaxuan Guo, Hao Wang, Kexin Zhang, Liang Liu, and Yong Liu. Mas-bench: A unified benchmark for shortcut-augmented hybrid mobile gui agents, 2026. URL https://arxiv.org/abs/2509.06477.

Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. GPT-4V(ision) is a generalist web agent, if grounded, 2024. URL https://arxiv.org/abs/2401.01614.

Hanzhang Zhou, Xu Zhang, Panrong Tong, Jianan Zhang, Liangyu Chen, Quyu Kong, Chenglin Cai, Chen Liu, Yue Wang, Jingren Zhou, and Steven Hoi. Mai-ui technical report: Real-world centric foundation gui agents, 2025. URL https://arxiv.org/abs/2512.22047.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents, 2024. URL https://arxiv.org/abs/2307.13854.

Wangchunshu Zhou, Yuchen Eleanor Jiang, Long Li, Jialong Wu, Tiannan Wang, Shi Qiu, Jintian Zhang, Jing Chen, Ruipu Wu, Shuai Wang, Shiding Zhu, Jiyu Chen, Wentao Zhang, Xiangru Tang, Ningyu Zhang, Huajun Chen, Peng Cui, and Mrinmaya Sachan. Agents: An open-source framework for autonomous language agents, 2023. URL https://arxiv.org/abs/2309.07870.

Yuqi Zhu, Shuofei Qiao, Yixin Ou, Shumin Deng, Shiwei Lyu, Yue Shen, Lei Liang, Jinjie Gu, Huajun Chen, and Ningyu Zhang. KnowAgent: Knowledge-augmented planning for llm-based agents, 2025. URL https://arxiv.org/abs/2403.03101.

A Supplementary Case Studies

Figure 8 supplements the routing and navigation-compression cases in Section 5.5. The first GUI task extracts the product names, order number, and recipient phone number from TaoDian as explicit outputs. A validated messaging shortcut then opens the SMS compose view with the recipient and message body populated, leaving GUIClaw to verify and send the message.

Figure 9 provides an annotated Chinese companion for the host-mediated recovery case in Figure 7. It shows the same control flow in which an underspecified GUI output is routed back to the host, grounded through web search, and then consumed by downstream Messages and Maps GUI tasks.

B Action Space

Table 7 gives the unified action space GUIClaw exposes to the GUI executor, realizing $\mathcal{A}_{\text{gui}} \cup \mathcal{A}_{\text{ask}}$ from Section 4.4. The two platforms share a common core of pointer, text, navigation, and task-control actions; desktop adds key combinations, an explicit pointer swipe, and application launch and close, while mobile adds a dedicated enter key. This largely shared action surface is what lets a single host drive both platforms through the same GUI task interface.

Table 7: Unified action space of GUIClaw across platforms. **Shared** actions are available on both mobile and desktop; **Desktop** and **Mobile** list the platform-specific additions. Together they realize $\mathcal{A}_{\text{gui}} \cup \mathcal{A}_{\text{ask}}$ (Section 4.4); skills ($\mathcal{A}_{\text{skill}}$), Android deeplink/intent shortcuts ($\mathcal{A}_{\text{shortcut}}$), and external Model Context Protocol (MCP) tool calls extend this set through the runtime.

Environment	Action	Definition
Shared	Click(x, y)	Clicks or taps at coordinates (x, y).
	DoubleTap(x, y)	Double-clicks or double-taps at (x, y).
	LongPress(x, y)	Long-presses at (x, y).
	Drag(x_1, y_1, x_2, y_2)	Drags from (x_1, y_1) to (x_2, y_2).
	Scroll$(x, y,$ direction)	Scrolls at (x, y) in the given direction.
	Type(content)	Types the specified content.
	Back()	Returns to the previous screen.
	Home()	Returns to the home screen.
	Wait()	Pauses for a brief moment.
	Finished()	Returns the final answer or marks the task complete or infeasible.
	CallUser()	Requests user intervention.
Desktop	Hotkey(key)	Presses the specified key combination.
	Swipe(x_1, y_1, x_2, y_2)	Swipes the pointer from (x_1, y_1) to (x_2, y_2).
	OpenApp(name)	Launches the specified application.
	CloseApp(name)	Closes the specified application.
Mobile	PressEnter()	Presses the "enter" key.

C Skill Extraction and Prompt Contracts

KnowAct-GUIClaw uses prompt contracts at both execution time and reflection time. The execution prompt below is the runtime MobileWorld-style template used for prompt-side skill selection (Kong et al., 2025): GUIClaw retrieves top-k relevant skills, formats them as the `Compact skills` catalog, and lets the GUI executor choose a listed skill or continue with ordinary actions. Reflection then turns GUI experience into reusable knowledge through additional contracts rather than unrestricted trace replay. First, trajectory evidence is normalized into a task, an app scope, ordered actions, screenshots, target hints, and state contracts. Second, a skill-extraction agent compresses this evidence into one reusable, parameterized procedure and rejects outputs that are not executable or not state guarded. Third, a separate evolution agent repairs a previously reused skill only when a failure trace identifies that skill as the failure source. Finally, a memory-induction agent writes textual success and failure lessons that guide future routing and execution without becoming executable actions. The boxes below reproduce the runtime source prompts without shortening; braces denote runtime placeholders and structured inputs supplied by the caller.

(a) Execution Prompt

```
# Role: Android Phone Operator AI
You are an AI that controls an Android phone to complete user requests. Your responsibilities:
- Answer questions by retrieving information from the phone.
- Perform tasks by executing precise actions.

# Action Framework
Respond with EXACT JSON format for one of these actions:
| Action          | Description                                | JSON Format Example
                                    |
|-----------------|--------------------------------------------
     |-------------------------------------------------------------------------|
| `click`         | Tap visible element (describe clearly)   | `{"action_type": "click", "
    coordinate": [x, y]}`   |
| `double_tap`    | Double-tap visible element (describe clearly)   | `{"action_type": "double_tap",
     "coordinate": [x, y]}`   |
| `long_press`    | Long-press visible element (describe clearly) | `{"action_type": "long_press",
    "coordinate": [x, y]}`          |
| `drag`          | Drag from visible element to another visible element (describe both clearly) |
    `{"action_type": "drag", "start_coordinate": [x1, y1], "end_coordinate": [x2, y2]}`
    |
| `input_text`    | Type into field | `{"action_type":"input_text", "text":"Hello"}|
| `answer`        | Respond to user                          | `{"action_type":"answer", "text":"It'
    s 25 degrees today."}`              |
```

```
| `navigate_home` | Return to home screen                    | `{"action_type": "navigate_home"}`
                                  |
| `navigate_back` | Navigate back                            | `{"action_type": "navigate_back"}`
                                  |
| `scroll`        | Scroll direction (up/down/left/right)    | `{"action_type":"scroll", "direction
    ":"down"}`                    |
| `status`        | Mark task as `complete` or `infeasible`  | `{"action_type":"status", "
    goal_status":"complete"}`                 |
| `wait`          | Wait for screen to update                | `{"action_type":"wait"}`
                                  |
| `ask_user`      | Ask user for information                 | `{"action_type":"ask_user", "text":"
    what is the exact requirements do you need?"}`       |
| `keyboard_enter`   | Press enter key          | `{"action_type":"keyboard_enter"}`              |
| `use_skill`     | Run a listed compact GUI skill prefix when it clearly matches the task | `{"
    action_type":"use_skill","skill_id":"listed_skill_id","arguments":{}}` |
| `click_then_type` | Preferred one-step action for visible text fields: tap a coordinate and type
    text. Use auto_enter true only for search submission. | `{"action_type":"click_then_type","
    coordinate":[x,y],"text":"Hello","auto_enter":false}` |
| `click_multi` | Tap multiple visible coordinates in sequence | `{"action_type":"click_multi","
    coordinates":[[x1,y1],[x2,y2]]}` |

Note:
- The coordinate is the center of the element to be clicked/long-pressed/dragged.
- x, y are coordinates in the screen, the origin is the top-left corner of the screen.
- x, y are numbers, the range is normalized to [0, 1000].

# Execution Principles
1. Communication Rule:
   - ALWAYS use 'answer' action to reply to users - never assume on-screen text is sufficient
   - Please follow the user instruction strictly to answer the question, e.g., only return a single
     number, only return True/False, only return items separated by comma.
   - NEVER use 'answer' action to indicate waiting or loading - use 'wait' action instead
   - Note that `answer` will terminate the task immediately.

2. Efficiency First:
   - Choose simplest path to complete tasks
   - If action fails twice, try alternatives (e.g., long_press instead of click)

3. Smart Navigation:
   - Gather information when needed (e.g., open Calendar to check schedule)
   - For scrolling:
     * Scroll direction is INVERSE to swipe (scroll down to see lower content)
     * If scroll fails, try opposite direction

4. Text Operations:
   - You MUST first click the input box to activate it before typing the text.
   - When a text field is visible, prefer `click_then_type` (one action) over separate `click`+`
     input_text` to activate and type known text.
   - For text manipulation:
     1. Long-press to select
     2. Use selection bar options (Copy/Paste/Select All)
     3. Delete by selecting then cutting

5. Ask User:
    - If you think you have no enough information to complete the task, you should use `ask_user`
     action to ask the user to get more information.


# Decision Process
0. Before a manual GUI action, check the compact skill list; if one clearly matches the requested
     app/workflow, choose `use_skill` (it may open the target app internally, so do not open it
     manually first).
0a. Use listed composite actions only when they exactly match the next local operation. Prefer `
     click_multi` when several targets on screen need the same tap and none opens a confirmation
     dialog. Prefer `click_then_type` over separate `click`+`input_text` when tapping a visible text
      field to enter known text -- unless the field is already focused, existing text must be
     cleared, tapping opens a picker to observe first, or the text depends on the tap result.
1. Analyze goal, history, and current screen
2. Determine if task is already complete (use `status` if true)
3. If not, choose the most appropriate action to complete the task.
4. Output in exact format below, and ensure the Action is a valid JSON string:
5. The action output format is different for GUI actions and MCP tool actions. Note only one tool
     call is allowed in one action.

# Optional Compact GUI Skills
Optionally pick ONE listed compact skill as a single action when it clearly matches
the task. If none clearly matches, keep using the normal GUI actions above.

Skill action format:
```

```
`{"action_type":"use_skill","skill_id":"listed_skill_id","arguments":{"param":"value"}}`

Rules:
- Prefer `use_skill` over manual navigation when a listed skill clearly matches the
  requested app/workflow; copy its `skill_id` exactly.
- A skill may open/navigate the target app internally, so the target app need not
  already be on screen.
- Fill `arguments` only with values the task makes obvious; otherwise use `{}`.

Compact skills:
- skill_id={skill_id}; skill_name={skill_name}; description={description}; app={app}; parameters={
    comma_separated_parameters}

# Expected Output Format (`Thought: ` and `Action: ` are required):
Thought: [Analysis including reference to key steps/points when applicable]
Action: [Single JSON action]

# Output Format Example
## for GUI actions:
Thought: I need to ... to complete the task.
Action: {"action_type": "type", "text": "What is weather like in San Francisco today?"}
```

(b) Skill extraction prompt

```
Extract a reusable GUI skill as Python code from this trajectory.

Target format:
from opengui.skills.flat import C, R, action, skill, tag

@skill(app="{app}", platform="{platform}", name="short_name", description="One sentence summary")
async def skill_name(device, param1, param2):
    await action("open_app", target="{app}", fixed=True, fixed_values={"text": "{app}"},
                 valid_state="No need to verify")
    await action("tap", target="search button", fixed=True,
                 fixed_values={"x": 540, "y": 960},
                 valid_state="target is visible and clickable",
                 state_contract=C(app="...[app_package]...", required=[R(resource_id="target_id",
     visible=True)]))
    await action("input_text", target="{{param1}}",
                 valid_state="input field is focused")

Rules:
- Extract ONE cohesive skill that covers the core action sequence. Do NOT split into multiple tiny
    @skill functions.
- fixed=true + fixed_values: static UI (nav bars, toolbar, system actions, open_app). Copy exact x/
    y/text from the trajectory step params shown after "|".
- Use only these action types: tap, long_press, double_tap, drag, swipe, scroll, input_text, hotkey,
     screenshot, wait, open_app, open_deeplink, open_intent, close_app, back, home, enter,
    app_switch, done, request_intervention. Do not invent actions such as read_text, press_key,
    navigate_back, or navigate_to_folder.
- The skill function body must contain only await action(...) statements. Do not add if/for/while/
    try blocks, assignments, calculations, helper calls, return values, or non-action awaits.
- Do not use f-strings, string concatenation, arithmetic expressions, comparisons, comprehensions,
    dict/list expressions with computed values, or nested function calls in action arguments. Use
    literal strings or {{param}} placeholders only.
- fixed_values may contain only executable action fields such as x, y, x2, y2, text, key, pixels,
    direction, duration_ms, component, package, intent_action, mime_type, categories, extras,
    relative, status, auto_enter. Never put selectors such as resource_id, content_desc, class, or
    class_name in fixed_values.
- fixed=false: dynamic content (search results, variable input). Use {{param}} placeholders, omit
    fixed_values.
- target: Every required interactive step must have a natural-language target and valid_state. Use
    a concise natural-language grounding hint, e.g. "search button", "search input field", "
    matching video result", "skip ad button". Do not use raw class/resource_id as target unless it
    is also visible user-facing text.
- Collapse all app-launch steps into ONE open_app as the first step. For open_app, prefer the
    trajectory app package for both target and fixed_values.text when available, and always use
    valid_state="No need to verify".
- valid_state: Every required interactive step must have a specific present-tense valid_state, e.g.
     "search field is visible and enabled". For input_text, use "input field is focused" if no
    better state is available. If a required step has no verifiable state, remove or regenerate
    that step instead of leaving valid_state empty.
- state_contract: do not invent selectors. Copy only the exact contract provided by trajectory/
    codegen for the matched step; omit if no contract is provided. The extractor postprocess will
    align contracts from codegen.
- R(...) supports resource_id, text, content_desc, class_, xpath, visible, clickable, enabled,
    focused, and scrollable only. Do not use class_name.
- Drop duplicate/redundant clicks, exploratory taps, and pointless scrolls.
```

```
- Transient popups (ads, permissions, consent): keep as optional=True step. Executor skips them
    when absent.
- Keep transient blockers and benign app confirmations such as skip/close/save/done as guarded
    optional=True steps when they appear. Omit destructive or externally visible confirmations such
     as pay, delete, send, submit order, publish, or irreversible consent unless the original user
    task explicitly requires them.
- description: MUST be generic and reusable. Mention app name, capability, and broad feature-level
    route only. Use parameter roles like query, media item, contact, or item. NEVER include literal
     values/entities, exact titles/names, or narrow qualifiers such as specific, official, first
    result, or top result. Avoid tap-by-tap UI actions.

For failure trajectories, insert:
FAILURE trajectory: keep the reusable succeeded prefix. If the failure screen clearly shows one
    safe corrective next action, append at most one non-fixed corrective step with natural-language
     target and valid_state. Do not invent coordinates or state_contract for that corrective step.
    Use optional=True only for transient blockers/popups, and never add pay/delete/send/submit/
    publish/irreversible confirmation actions.

## Trajectory
{code_text}

Output ONLY the Python code. No markdown fences, no JSON object, no explanation.
```

(c) Skill evolution prompt

```
You are improving one existing GUI automation skill after it failed during reuse.

Task:
{task}

Original skill JSON:
{skill_json}

Failure case:
{failure_case_json}

Prior feedback for this skill:
{feedback_json}

Full trajectory:
{trajectory_json}

Return ONLY a JSON object for the improved skill, using the same schema as the original skill.
Rules:
- Improve the original skill in place. Do not create a different skill or unrelated workflow.
- Keep the same high-level intent unless the failure shows the description caused a wrong match;
    then narrow the description/preconditions.
- If a popup or optional obstacle appeared, add a guarded optional step before the blocked step:
  {"action_type": "tap", "target": "Close", "parameters": {"optional": true}, "valid_state": "popup
     close button is visible"}
  Optional steps are skipped when their valid_state is not present.
- If an action target, parameter, valid_state, or state_contract is stale for the new UI, update
    that step.
- Preserve reusable parameters as {param_name} placeholders.
- Do not include final destructive actions such as pay, delete, send, submit, or irreversible
    confirmation unless the original skill already required them.
- Omit skill_id; the caller will preserve the original skill_id.
```

(d) Shortcut validation prompt

```
Verifier system prompt:
You validate Android deeplink/intent probe results. Return only one JSON object with fields: usable
     (boolean), status ('page_validated' or 'launchable' or 'failed'), name (short English function
     name, e.g. 'bili_search', 'taobao_cart'; required when usable=true), description (short
    natural-language capability, e.g. 'Bili video search'), parameters (array of strings),
    payload_preserved (boolean), reason (short string), next_variant_hint (string or null). Set
    usable=true only if the screenshot/UI indicates the intended app page opened and the payload,
    such as query text, was preserved when relevant. When usable=true for the intended page, set
    status='page_validated'; use status='launchable' only for target-package launches whose page
    purpose is unclear. The `name` field must be a concise English identifier (snake_case, max 30
    chars) derived from the app and page function, e.g. 'bili_scan', 'taobao_search'. Do not treat
    a query visible only in search history or suggestions as payload_preserved; payload_preserved
    means the current input/result page reflects that payload. Use a concise natural-language
    description such as 'Bili video search'. Do not mention adb, URI, component, or implementation
    details in description or name.
```

```
Verifier user payload:
{
  "task": "{task}",
  "candidate": "{candidate_to_dict(candidate)}",
  "variant": "{variant_to_dict(variant)}",
  "foreground": "{foreground_app}",
  "adb_output": "{summarized_adb_output}",
  "ui_sample": ["..."],
  "ui_tree_excerpt": "{summarized_ui_tree}"
}
If available, the screenshot is attached as an image_url message part.

Metadata refinement system prompt:
You refine shortcut skill names/descriptions. Return only JSON.

Metadata refinement user prompt:
{
  "task": "Refine Android shortcut skill metadata after validation. Return JSON only: {\"records
     \":[{\"index\":0,\"name\":\"...\",\"description\":\"...\"}]}. Do not change payload fields,
     packages, status, parameters, valid_state, or indices. All promoted shortcut records must keep
     valid_state='No need to verify'. Prefer concise distinct names and descriptions that help an
     agent choose the right shortcut.",
  "records": "{compact_records}"
}
```

(e) Experience-memory induction prompt

```
Success system prompt:
You are an expert in Android GUI automation. You will be given a user task query
and the corresponding trajectory that represents **how an agent successfully
accomplished the task**.

## Guidelines
You need to extract and summarize useful insights in the format of memory items
based on the agent's successful trajectory.  The goal of summarized memory items
is to be helpful and generalizable for future similar tasks.

## Important notes
- You must first think why the trajectory is successful, and then summarize
  the insights.
- You can extract *at most 3* memory items from the trajectory.
- You must not repeat similar or overlapping items.
- Prefer concrete, actionable procedures over abstract principles.  Do not
  embed specific product names, queries, or literal string contents from the
  task.
- For Android GUI tasks, focus on: app navigation patterns, form-filling
  strategies, common UI pitfalls that were avoided, and efficient action
  sequences.
- When UI snippets are present in the trajectory, ground lessons in the actual
  visible/clickable controls rather than only repeating the agent's own
  reasoning.

## Output Format
Your output must strictly follow the Markdown format shown below:

'''
# Memory Item i
## Title <the title of the memory item>
## Description <one sentence summary describing when to use the memory item>
## Content <1-3 sentences describing the insights learned to successfully
  accomplish similar tasks in the future>
'''

Failure system prompt:
You are an expert in Android GUI automation. You will be given a user task query
and the corresponding trajectory that represents **how an agent attempted to
resolve the task but failed**.

## Guidelines
You need to extract and summarize useful insights in the format of memory items
based on the agent's failed trajectory.  The goal of summarized memory items is
to be helpful and generalizable for future similar tasks.

## Important notes
- You must first reflect and think why the trajectory failed, and then
  summarize what lessons you have learned or strategies to prevent the failure
  in the future.
```

```
- You can extract *at most 3* memory items from the trajectory.
- You must not repeat similar or overlapping items.
- Prefer concrete, actionable recovery procedures over abstract principles.
  Do not embed specific product names, queries, or literal string contents
  from the task.
- For Android GUI tasks, focus on: navigation mistakes, form-filling errors,
  premature task completion, app state assumptions that were wrong, and
  specific UI patterns that caused trouble.
- When UI snippets are present in the trajectory, ground lessons in the actual
  visible/clickable controls rather than only repeating the failed agent's own
  reasoning.

## Output Format
Your output must strictly follow the Markdown format shown below:

'''
# Memory Item i
## Title <the title of the memory item>
## Description <one sentence summary describing when NOT to use this approach>
## Content <1-3 sentences describing the insights learned to avoid such
  failures in the future>
'''

User prompt:
{trajectory_text}
```

D Skill and Shortcut Examples

The three boxes below show excerpts from KnowAct-GUIClaw's Android skill and shortcut store, abbreviated for readability: timestamps and some keyword arguments are dropped, 64-character state-contract hashes and long component names are truncated, and one Chinese accessibility label is glossed in ASCII. They illustrate the three forms covered by the single skill abstraction (Section 4.4): (a) a parameterized click-then-type pattern in $\mathcal{A}_{\text{skill}}$; (b) a multi-step skill whose steps carry an expected state and, where available, a structural state contract checked before each step; and (c) validated deeplink and intent shortcuts in $\mathcal{A}_{\text{shortcut}}$. Stable values, such as package names, button coordinates, and intent components, are fixed in advance, while request-dependent values, such as the query string, are supplied at run time. The `validated` tag marks shortcut candidates that passed on-device validation rather than static manifest discovery alone.

(a) Parameterized pattern: click-then-type

```python
@skill(name='click_then_type', platform='android', tags=['compact_action'],
       skill_id='compact:action:click_then_type',
       description='Tap a [0,999] coordinate, then type text.')
async def click_then_type(device, coordinate, text):
    await action('click_then_type', target=coordinate, text=text,
                 auto_enter=False, valid_state='No need to verify')
```

(b) Multi-step skill with per-step state validation (Pinduoduo)

```python
@skill(name='search_product', app='com.xunmeng.pinduoduo', platform='android',
       skill_id='flat:search_product',
       description='Search for a product in Pinduoduo.')
async def search_product(device, query):
    await action('open_app', target='com.xunmeng.pinduoduo',
                 valid_state='No need to verify')
    await action('tap', target='search bar', fixed_values={'x': 242, 'y': 88},
                 valid_state='search bar is visible and clickable',
                 state_contract=C.from_dict({
                     'anchor': {'app_package': 'com.xunmeng.pinduoduo'},
                     'signature': {'required': [{
                         'selector': {'content_desc': 'search'},  # ASCII gloss
                         'state': ['visible', 'enabled']}]},
                     'fingerprint': '60c6a0fb...c950947b'}))  # sha256, truncated
    await action('input_text', target=query,
                 valid_state='input field is focused',
                 state_contract=C.from_dict({
                     'anchor': {'app_package': 'com.xunmeng.pinduoduo'},
                     'signature': {'required': [{
                         'selector': {'class': 'android.widget.EditText',
                                      'resource_id': 'com.xunmeng.pinduoduo:id/pdd'},
```

```
                        'state': ['visible', 'enabled', 'focused']}]},
                    'fingerprint': 'cbd6ef8c...4e7e2861'}))  # sha256, truncated
    await action('tap', target='search button', fixed_values={'x': 446, 'y': 96},
                 valid_state='search button is visible and clickable')
```

(c) Validated deeplink and intent shortcuts (JD)

```
@skill(name='jd_splash', app='com.jingdong.app.mall', platform='android',
       tags=['shortcut', 'deeplink', 'validated'], success_count=1)
async def jd_splash(device):
    await action('open_deeplink', target='JDAnalytics:',
                 package='com.jingdong.app.mall', valid_state='No need to verify')

@skill(name='jd_search', app='com.jingdong.app.mall', platform='android',
       tags=['shortcut', 'intent', 'validated'], success_count=1)
async def jd_search(device, query):
    await action('open_intent', intent_action='android.intent.action.SEND',
                 package='com.jingdong.app.mall',
                 component='com.jingdong.app.mall/...SearchBridgeActivity',
                 mime_type='text/plain',
                 extras=[['android.intent.extra.TEXT', '{{query}}']],
                 valid_state='No need to verify')
```

These entries populate the skill and shortcut store that Know retrieves and Act applies during GUI task execution.

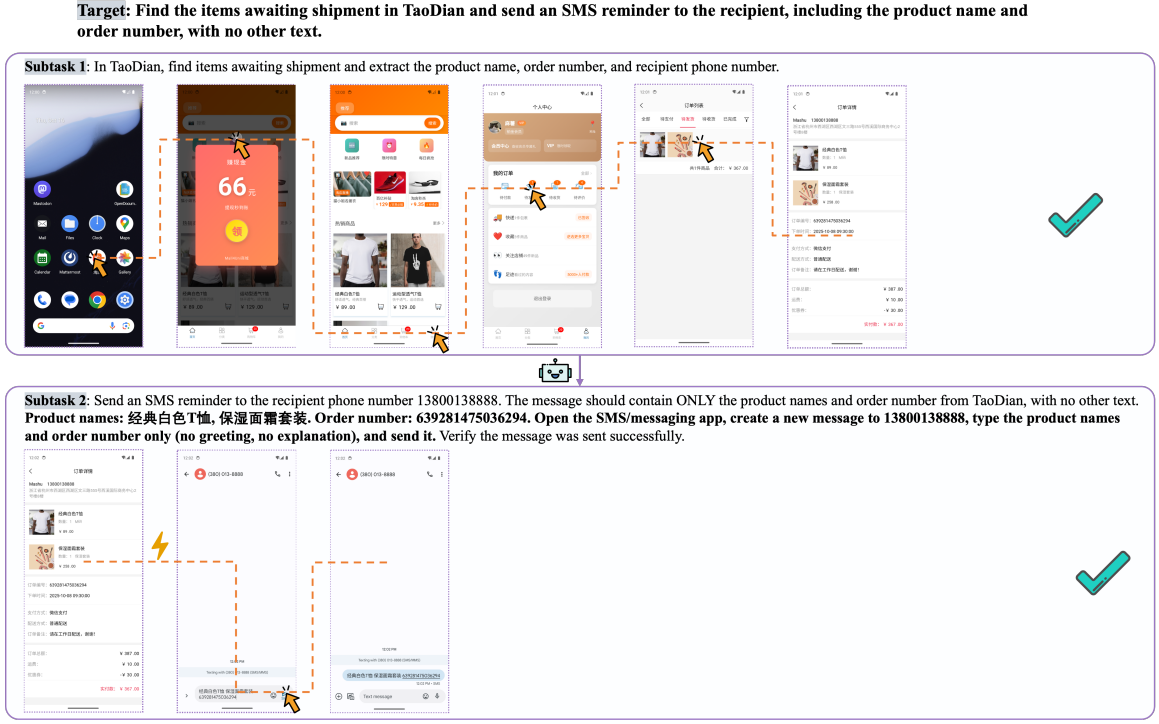

Figure 8: Cart-to-SMS cross-app execution. The TaoDian GUI task extracts the product names, order number, and recipient phone number and transfers them to the downstream messaging task. A validated messaging shortcut opens the SMS compose view with the recipient and message body already populated, illustrating both blackboard information transfer and navigation compression.

Figure 9: Chinese annotated companion for the host-mediated recovery case in Figure 7. The panel highlights the same control flow in which an underspecified GUI output is routed back to the host, grounded through web search, and then consumed by downstream Messages and Maps GUI tasks.