TAP-AGENT: FROM ONE-WAY HANDOFFS TO TWO-SIDED PREPARATION FOR DEVICE–CLOUD GUI AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Capable graphical user interface (GUI) agents increasingly rely on cloud-hosted multimodal large language models for high-level planning and lightweight on-device models for visual grounding and execution. However, existing device–cloud frameworks treat each cloud response as a rigid handoff from planning to execution. This serialization forces the device to ground targets only after a plan arrives while leaving its computation underutilized during cloud reasoning, resulting in inaccurate execution and increased interaction latency. To address these limitations, we propose **TAP-Agent** (**T**wo-sided **A**nticipatory **P**reparation), which prepares on-device computation on both sides of each cloud response. After a response arrives, plan-conditioned visual experts provide heterogeneous visual evidence, which is compiled into action-ready representations that associate semantic elements with screen geometry. While awaiting the next response, a compact experience buffer of verified interactions guides the device to anticipate likely operations and prefetch supporting visual evidence. An alignment mechanism reuses this computation when it is consistent with the returned plan, while UI-changing actions remain governed by confirmed instructions. Together, these mechanisms coordinate computation across time, improving execution reliability while shifting reusable device inference off the critical path. Experiments on AndroidWorld and OSWorld with multiple on-device executors show that TAP-Agent improves task success by **5.7%** and reduces end-to-end interaction latency by **10.4%** over state-of-the-art device–cloud baselines. The code will be publicly available.

1 INTRODUCTION

Recent advances in increasingly large and capable multimodal large language models (MLLMs) have substantially expanded the capabilities of graphical user interface (GUI) agents, enabling them to interpret natural-language instructions, understand visual interfaces, and execute multi-step operations across mobile and desktop applications (Qin et al., 2025). By interacting directly with graphical interfaces, these agents offer a general approach to automating tasks without application-specific APIs or manually designed workflows. However, the computational demands of MLLMs often necessitate cloud deployment, where repeated inference and communication incur substantial computational costs and interaction latency. These overheads become particularly pronounced in long-horizon tasks that require frequent observation and action (Abhyankar et al., 2026).

Device–cloud collaboration balances model capability with deployment efficiency by combining powerful cloud models with lightweight local agents (Jiang & Huang, 2026; Zhou et al., 2025). Interleaving reasoning and actions with environmental feedback supports interactive decision-making (Yao et al., 2023). A practical design assigns high-level planning to the cloud and visual grounding and action execution to the device (Li et al., 2026; Gou et al., 2025). Some recent works further integrate the two into an alternating loop: the cloud generates a plan for the next stage of a task, the device executes the corresponding interactions, and execution feedback informs subsequent cloud planning. For example, EcoAgent (Yi et al., 2026) implements a Planner-Executor-Observer architecture that combines cloud planning, local execution, and expectation-based verification to support iterative replanning and recovery.

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

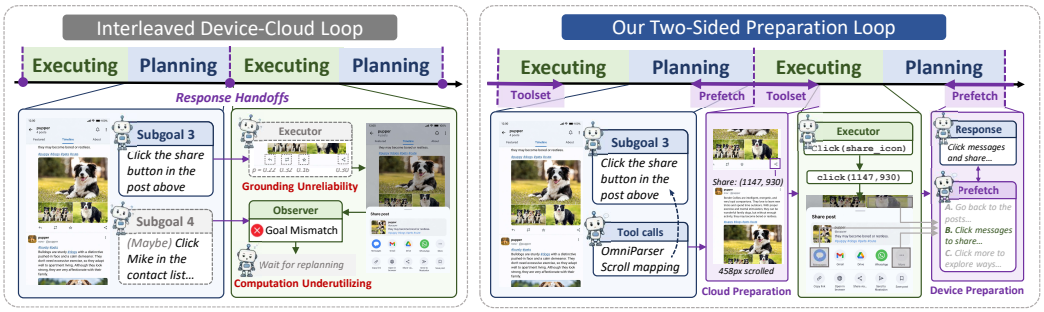

Figure 1: Two interaction loops for sharing a post with Mike. **Left:** Conventional systems suffer from unreliable target grounding and device idling. **Right:** TAP-Agent couples plan-conditioned visual expert tool orchestration with alignment-gated asynchronous action prefetching. This two-sided preparation strengthens semantic grounding and overlaps local computation with cloud reasoning.

Despite their efficiency benefits, existing device–cloud frameworks treat the cloud response as a rigid boundary between planning and execution, creating two critical bottlenecks: (1) *Unreliable post-response grounding.* Upon receiving a high-level plan, a resource-constrained on-device model must identify and localize the referenced targets to derive concrete actions, often resulting in inaccurate execution. (2) *Underutilized pre-response computation.* While waiting for cloud planning, the device remains largely idle and defers perception to the execution phase, increasing end-to-end latency. Such issues lead to wasted planning effort and underutilized device capacity, impeding the orchestration of device–cloud computation across time.

To address these challenges, we propose **TAP-Agent** (**T**wo-sided **A**nticipatory **P**reparation), which transforms the device–cloud interaction from a one-way plan-to-execution handoff into two-sided preparation around each cloud response. Rather than treating plan arrival as the point at which on-device computation begins, it uses cloud guidance to prepare the current execution and device-side anticipation to prepare a candidate action for the next plan. After a response arrives, its high-level plan orchestrates on-device visual experts to map GUI pixels to semantic elements with executable coordinates, yielding an action-ready representation for target grounding. While awaiting the next response, the device anticipates likely operations and prefetches an executable action from the post-action screen, reusing it when the prefetched semantic goal is consistent with the returned goal. Through this two-sided preparation, TAP-Agent coordinates device–cloud computation across time, reducing post-response grounding uncertainty while turning pre-response waiting into useful on-device computation.

Specifically, TAP-Agent introduces cloud-guided visual expert tools that instantiate plan-conditioned visual intent specifications as task-specific calls to local experts, such as element detection and text recognition. To provide the executor with an action-ready interface, heterogeneous visual evidence compilation normalizes spatial outputs into a unified element-level representation that associates semantic descriptions with screen geometry, while preserving non-spatial outputs as complementary state evidence. Through element-level action-space lifting, this representation augments the executor's native pixel-level action space with element-level operations, enabling it to select semantic targets and recover their physical coordinates from the associated geometry. For pre-response preparation, TAP-Agent maintains a compact online experience distilled from locally verified interactions and initiates next-round cloud planning while the current action is in flight. During cloud replanning, the experience supports candidate prediction on the realized post-action screen, and the device invokes visual experts to prefetch an executable action. When the revised plan arrives, semantic-gated action reuse compares the prefetched and returned semantic goals and directly executes the prefetched action when they are consistent and necessary checks pass. Together, these mechanisms realize two-sided preparation: the cloud-guided visual experts provide action-ready grounding for semantic subgoals, while alignment-gated action prefetching enables critical-path latency reduction by shifting reusable grounding into the cloud-reasoning interval.

We evaluate the effectiveness and generalizability of TAP-Agent on AndroidWorld (Rawles et al., 2025) for mobile interaction and OSWorld (Xie et al., 2024) for long-horizon desktop tasks. We further examine its applicability across model scales by pairing TAP-Agent with different on-device executors. Across these settings, TAP-Agent improves task success by **5.7%** and reduces end-to-end interaction latency by **10.4%** on successful episodes against the state-of-the-art device–cloud methods, demonstrating consistent gains in both execution effectiveness and system efficiency.

Contributions. **(1)** We formulate device–cloud GUI collaboration as two-sided anticipatory preparation around each cloud response, replacing the reactive plan-to-execution handoff with coordinated post-response grounding and pre-response computation. **(2)** We develop cloud-guided visual expert tools that compile heterogeneous visual evidence into an action-ready representation, enabling lightweight executors to perform element-level action selection beyond native coordinate prediction. **(3)** We introduce alignment-gated asynchronous action prefetching, which initiates next-round planning while the current action is in flight, uses verified online experience to anticipate likely operations during cloud reasoning, and reuses prefetched actions only when their semantic goals match the returned goal. **(4)** We evaluate TAP-Agent across mobile and desktop GUI benchmarks and multiple models, demonstrating consistent improvements in task success and end-to-end latency while validating its applicability across platforms and model scales.

2 Related Work

2.1 GUI agents.

GUI agents automate software tasks by observing interfaces and translating natural-language instructions into executable actions. Some systems couple visual perception with action generation, allowing a model to interpret a screenshot and propose the next interaction (Qin et al., 2025; Hong et al., 2024). Others separate action prediction from target grounding, making localization a distinct part of the decision process (Zheng et al., 2024). Generalist models extend these capabilities across platforms and interface types (Wu et al., 2025; Xu et al., 2025). Training advances include larger cross-platform datasets (Liu et al., 2026) and reinforcement learning for computer use and visual grounding (Lai et al., 2026; Kang et al., 2026). Planning and grounding methods help agents choose successive steps and identify on-screen targets in longer tasks (Agashe et al., 2025; Yang et al., 2026; Chen et al., 2026b). Evaluation spans element grounding (Cheng et al., 2024) and web interactions (Deng et al., 2023; Zhou et al., 2024). Interactive benchmarks further test full task completion in mobile and desktop environments (Rawles et al., 2025; Xie et al., 2024).

On-device deployment motivates compact models such as ShowUI and GoClick (Lin et al., 2025; Li et al., 2026), alongside dedicated grounding components such as UGround (Gou et al., 2025). OpenPhone (Jiang & Huang, 2026) and MAI-UI (Zhou et al., 2025) selectively invoke cloud assistance, while EcoAgent (Yi et al., 2026) couples cloud planning with local execution and verification. AAPT precomputes conditional policy trees during idle periods to accelerate event-triggered execution (Dong et al., 2026); TAP-Agent couples post-response grounding with action prefetching during cloud reasoning, gating reuse by semantic goal alignment.

2.2 Visual tool experts.

Visual tool experts provide task-specific evidence that complements agents' native screenshot understanding. Text recognizers recover textual content and locations (Liao et al., 2020; Li et al., 2023), with PP-OCR emphasizing lightweight pipelines (Du et al., 2020; Li et al., 2022; Cui et al., 2026). Screen parsing recovers UI elements and their relationships (Wu et al., 2021), while OmniParser associates interactable elements with semantic descriptions (Lu et al., 2024). Screenshot-based visual question answering provides complementary semantic evidence (Lee et al., 2023; Baechler et al., 2024). Compact vision-language models such as SmolVLM reduce the memory requirements, supporting deployment on resource-constrained devices (Marafioti et al., 2025).

Beyond static parsing, regional zooming (Wu & Xie, 2024; Wu et al., 2024) and learned tool invocation (Chen et al., 2026a) enable finer visual inspection. SAM 3 supports concept-prompted segmentation (Carion et al., 2026) and helps to identify task-relevant regions. Related approaches introduced action-effect verification (Zhang et al., 2026) and interface display-defect assessment (Xiang et al., 2026), which further examine execution outcomes and interface conditions. In TAP-Agent, these capabilities produce heterogeneous evidence, which is compiled into semantic elements with executable geometry, retaining non-spatial outputs for state assessment.

3 Preliminaries

Partially Observable Markov Decision Process. Following prior work (Xie et al., 2024; Xu et al., 2025), we formulate instruction-conditioned GUI interaction as an episodic Partially Observable Markov Decision Process (POMDP) $\mathcal{M}_I = (\mathcal{S}, \mathcal{O}, \mathcal{A}^{\text{pix}}, P, R_I)$, where $\mathcal{S}$ is the latent device-state

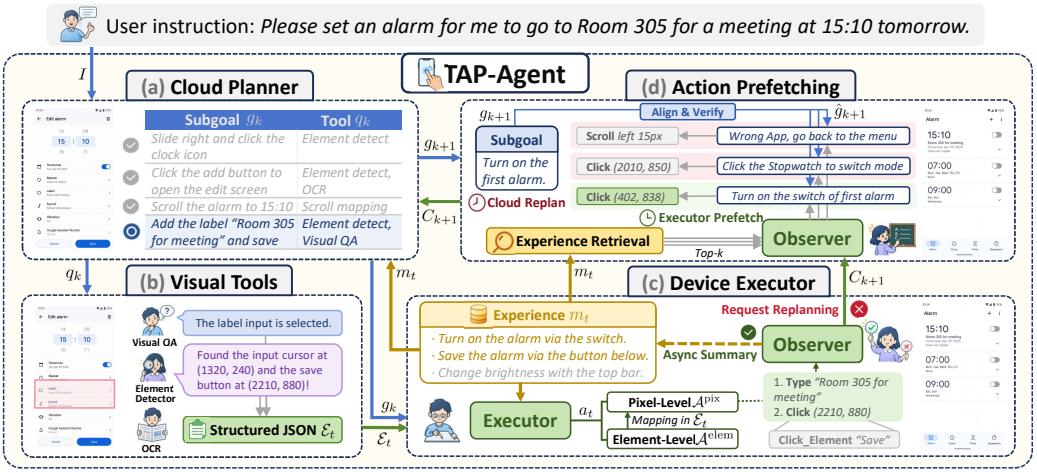

Figure 2: **Overview of the TAP-Agent loop.** (a) Cloud planner generating subgoals and tool-use specifications. (b) Local visual experts producing action-ready representations for element-level execution. (c) On-device executor action space lifting and observer verification. (d) Experience retrieval and action prefetch during cloud replanning.

space, $\mathcal{O}$ is the screenshot-observation space, $\mathcal{A}^{\text{pix}}$ is the pixel-level action space, P is the transition kernel, and R_I is the task reward associated with instruction I. At interaction step t, the agent observes a rendering of the latent state, selects an action from the observable history, and transitions to the next state:

$$o_t = \text{Render}(s_t), \qquad a_t^{\text{pix}} \sim \pi(\cdot \mid I, h_t), \qquad s_{t+1} \sim P(\cdot \mid s_t, a_t^{\text{pix}}), \tag{1}$$

where $h_t = (o_0, a_0^{\text{pix}}, \ldots, a_{t-1}^{\text{pix}}, o_t)$ denotes the interaction history. The action space includes primitive operations such as clicking, typing, and scrolling. The objective is to maximize the expected task reward, $\mathbb{E}_{\tau\sim(\pi,P)}[R_I(\tau)]$, over the interaction trajectory τ.

Device-Cloud Collaboration. Device–cloud GUI agents operate at two temporal granularities: the cloud plans over semantic subgoals, whereas the device resolves each subgoal through fine-grained visual grounding and pixel-level interaction. At planning round k, the cloud planner generates a subgoal g_k and its expected outcome e_k from instruction I and execution context C_k, which contains the current screenshot and feedback from preceding interactions:

$$(g_k, e_k) = \pi_{\text{cloud}}(I, C_k), \qquad \mathcal{T}_k = \left(a_{k,1}^{\text{pix}}, \ldots, a_{k,K_k}^{\text{pix}}\right), \tag{2}$$

where $\mathcal{T}_k$ is the sequence of K_k on-device actions used to realize g_k. At local step j, the device executor selects an action from the current screenshot and the active subgoal, after which the observer compares the resulting screen with the expected outcome:

$$a_{k,j}^{\text{pix}} \sim \pi_{\text{dev}}(\cdot \mid o_{k,j}, g_k), \qquad v_{k,j} = \pi_{\text{obs}}(o_{k,j+1}, e_k). \tag{3}$$

The verification signal $v_{k,j}$ determines whether local execution proceeds or the updated context is returned to the cloud for replanning.

4 Methodology

In this paper, we propose **TAP-Agent**, a temporally coordinated device–cloud framework that reformulates the hierarchical division between cloud planning and on-device execution as *two-sided anticipatory preparation* around each cloud response. It incorporates two complementary designs: a cloud-guided visual expert toolset that translates coarse-grained semantic subgoals into action-ready visual grounding for pixel-level execution, and alignment-gated prefetching that uses cloud-waiting intervals to anticipate likely subsequent operations and precompute their visual evidence without changing the GUI state.

4.1 Plan-Conditioned Visual Expert Tool Orchestration

Existing device–cloud GUI agents require a lightweight executor to translate each cloud-generated subgoal directly into pixel-level actions without explicit visual guidance, making post-response

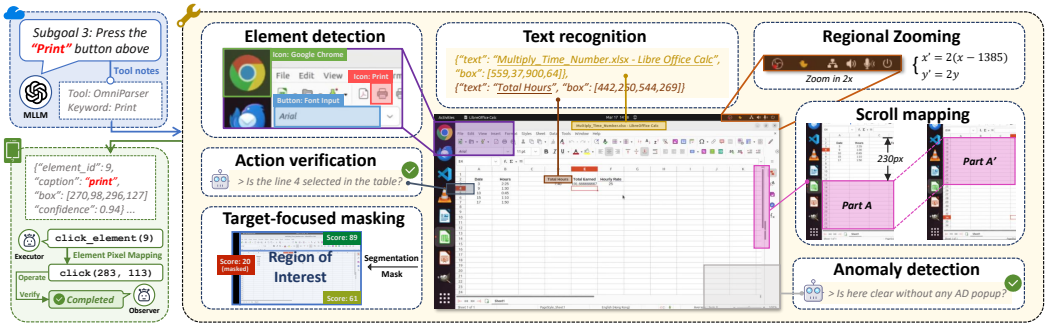

Figure 3: **Visualization of TAP-Agent's on-device visual expert tools.** The panels illustrate seven visual tools and their representative outputs on a desktop GUI. The left example shows how cloud guidance and tool outputs support element-level target selection and coordinate-based execution.

grounding error-prone. We therefore orchestrate on-device visual experts as plan-conditioned tools that transform a semantic subgoal into an executable element-level interface. To this end, the cloud response specifies which experts to invoke and what evidence to acquire; their spatial and state outputs are compiled into a unified semantic–spatial representation that lifts the executor from coordinate prediction to element-level action selection, while expert-provided geometry resolves the selected element into native click, type, or scroll operations.

Plan-Conditioned Visual Intent Specification. Based on the step-wise planning, we augment each cloud response with a plan-conditioned visual intent specification. At planning round k, the cloud planner receives the task instruction I, the execution context C_k, descriptions of the registered visual tools, and a structured response schema. It then returns

$$r_k = (g_k, q_k), \qquad q_k = \left[(f_{k,\ell}, \xi_{k,\ell})\right]_{\ell=1}^{n_k}, \tag{4}$$

where g_k specifies the semantic transition to be achieved and q_k specifies the visual evidence required to ground and execute it. Each request in q_k pairs a registered visual tool $f_{k,\ell}$ with a task-specific evidence query $\xi_{k,\ell}$, such as a target keyword or a region of interest.

To fulfill these evidence requests, the device maintains a registry of seven complementary visual tools, as illustrated in Fig. 3. *Element detection* and *text recognition* localize actionable components and visible text; *regional zooming*, *target-focused masking*, and *scroll mapping* refine evidence acquisition in small, cluttered, or scrollable regions; and *action verification* and *anomaly detection* assess transition outcomes and unexpected interface states. Rather than executing a fixed processing pipeline, q_k assembles a task-specific subset of these tools to acquire the evidence required for target grounding and state assessment. Implementation details are provided in Appendix A.1.

This formulation decomposes each cloud response into complementary semantic and perceptual specifications. The semantic goal g_k defines the interface transition to be achieved, whereas the visual intent specification q_k identifies the evidence required to ground that transition on the current screen. The device executes the tool requests in q_k over its local observation and passes their heterogeneous outputs to the subsequent compilation stage, which normalizes them into a unified executable representation.

Heterogeneous Visual Evidence Compilation. The requests in q_k may invoke experts with different output structures, semantic granularity, and coordinate systems. For example, detection and recognition tools return localized elements, whereas verification and anomaly tools produce assessments of the interface state. Directly exposing these heterogeneous outputs would require the executor to interpret a different interface for each expert. TAP-Agent instead compiles them into a common representation that separates spatial elements from complementary non-spatial evidence.

At a local step t within planning round k, the device evaluates q_k on the observation o_t and constructs

$$(\mathcal{E}_t, \mathcal{V}_t) = \mathrm{CompileVis}(o_t, q_k), \quad \mathcal{E}_t = \{(d_{t,i}, b_{t,i}, \gamma_{t,i})\}_{i=1}^{N_t}, \quad \mathcal{V}_t = \{v_{t,z}\}_{z=1}^{N_t^{\mathrm{aux}}}, \tag{5}$$

where each element in $\mathcal{E}_t$ associates a semantic description or recognized text $d_{t,i}$ with screen geometry $b_{t,i}$ and an optional confidence score $\gamma_{t,i}$. When an expert operates on a cropped, zoomed, or masked input, its predicted geometry is transformed back to the full-screen coordinate system

before being inserted into $\mathcal{E}_t$. Consequently, all spatial outputs share a common reference frame and can be addressed through the same downstream interface.

Outputs without meaningful spatial support are preserved in $\mathcal{V}_t$ rather than being assigned artificial geometry. These outputs include, for example, whether an action achieved the expected transition or whether the current screen exhibits an anomalous state. The pair $(\mathcal{E}_t, \mathcal{V}_t)$ forms an executable visual intermediate representation: $\mathcal{E}_t$ provides plan-relevant semantic elements with physical locations, while $\mathcal{V}_t$ supplies state evidence for action selection and verification. Because this representation is constructed from the requests in q_k, it captures the evidence required by the active subgoal instead of requiring an exhaustive description of the entire screen.

Element-Level Action-Space Lifting. Although the native device interface accepts physical operations such as coordinate clicks, text entry, and scrolling, the cloud subgoal refers to targets semantically. A coordinate-only executor must therefore infer both which interface element satisfies the subgoal and where that element is located. We reduce this grounding burden by lifting the executor's action space with operations defined over the compiled elements:

$$\mathcal{A}_t^{\mathrm{dev}} = \mathcal{A}^{\mathrm{pix}} \cup \mathcal{A}^{\mathrm{elem}}(\mathcal{E}_t), \tag{6}$$

where $\mathcal{A}^{\mathrm{elem}}(\mathcal{E}_t)$ contains operations such as `click_element` and `type_into_element` over the elements available in $\mathcal{E}_t$. Conditioned on the active subgoal and compiled evidence, the executor can select a semantic target directly rather than predicting its coordinates from raw pixels.

Before interacting with the environment, the selected element-level action is lowered to the native device-control interface:

$$a_t^{\mathrm{pix}} = \mathrm{Resolve}\left(a_t^{\mathrm{elem}}, \mathcal{E}_t\right). \tag{7}$$

The resolved action is executed through the same native primitives, including clicking, typing, and scrolling, so no change to the environment action protocol is required. By separating semantic target selection from coordinate resolution, this lifted interface reduces the grounding burden on the on-device executor while preserving precise pixel-level control.

4.2 Alignment-Gated Asynchronous Action Prefetching

The preceding visual-expert grounding module converts a returned cloud subgoal into an executable device action, but a sequential schedule still places this computation after cloud reasoning. TAP-Agent instead pipelines adjacent planning rounds: it initiates the next cloud request while the current action is in flight, and then uses the realized post-action interface to prefetch the most likely next action before cloud reasoning completes. The returned cloud goal gates whether this prefetched action can be reused, allowing speculative computation to reach an action-ready result without transferring execution authority from the cloud to the device.

In-Flight Next-Round Planning. Consider a local action a_t^{pix} executed under the current cloud response $r_k = (g_k, q_k)$. Upon dispatching the action, TAP-Agent immediately constructs the next-round request context from the existing execution context, the pre-action observation, the active semantic goal, and the in-flight action:

$$C_{k+1}^{\mathrm{req}} = \mathrm{UpdateCtx}\left(C_k, o_t, g_k, a_t^{\mathrm{pix}}\right). \tag{8}$$

Given I, C_{k+1}^{req}, and the visual-tool schema from Sec. 4.1, the cloud begins generating $r_{k+1} = (g_{k+1}, q_{k+1})$ before the outcome of a_t^{pix} is observed. Because C_{k+1}^{req} encodes the pre-action screen and the in-flight action, the cloud plans for the expected transition while the device completes the action and obtains the realized post-action observation o_{t+1}. This temporal offset allows next-round cloud planning to overlap with current-action execution and subsequent device-side preparation.

Experience-Guided Candidate Prediction. After each completed interaction, TAP-Agent updates its on-device experience memory with a compact reflection of the verified transition,

$$m_t = \mathrm{Reflect}\left(I, g_k, a_t^{\mathrm{pix}}, o_t, o_{t+1}, \zeta_t\right), \tag{9}$$

where ζ_t summarizes the transition outcome and execution evidence. The resulting experience is added to on-device memory, from which TAP-Agent retrieves a top-k relevant subset $\mathcal{R}_{k+1}$ according to the current task and execution context. Conditioned on the retrieved experience and the post-action observation, the on-device predictor generates an ordered set of M candidate responses:

$$\widehat{r}_{k+1,c} = (\widehat{g}_{k+1,c}, \widehat{q}_{k+1,c}), \qquad [\widehat{r}_{k+1,c}]_{c=1}^{M} = \pi_{\text{pref}}\left(I, C_{k+1}^{\text{req}}, o_{t+1}, \mathcal{R}_{k+1}\right), \tag{10}$$

where $\widehat{g}_{k+1,c}$ predicts a possible next semantic transition and $\widehat{q}_{k+1,c}$ specifies the visual evidence that would be required to ground it. The selected candidates reuse exactly the plan-conditioned visual interface defined in Sec. 4.1. Its visual intent is evaluated on o_{t+1}, after which the executor selects an action from the compiled evidence.

Thus, prefetching proceeds beyond candidate prediction or feature caching: it completes visual-expert invocation, evidence compilation, target selection, and coordinate resolution. The resulting action is ready for execution but remains withheld until cloud response arrives.

Semantic-Gated Action Reuse. Upon receiving the confirmed response $r_{k+1} = (g_{k+1}, q_{k+1})$, a device-side matcher compares its goal with the prefetched goal:

$$\alpha_{k+1} = \text{Match}_{\text{dev}}\left(g_{k+1}, \widehat{g}^{\star}_{k+1}\right) \in \{0, 1\}. \tag{11}$$

If $\alpha_{k+1} = 1$ and the action passes executor-validity and fresh-screen checks (Appendix B), TAP-Agent executes $\widehat{a}^{\text{pix}}_{k+1}$ without repeating visual grounding; otherwise it evaluates the confirmed intent q_{k+1} through the standard tools. The observer separately assesses the in-flight action's outcome for replanning.

Critical-Path Latency Reduction. The latency benefit arises from moving reusable device-side grounding into the cloud-reasoning interval. For a generic planning round k, let τ_k^{cloud} denote cloud reasoning latency, τ_k^{inf} the visual inference and grounding normally performed after receiving the response, and τ_k^{exec} the physical action-execution latency. When the prefetched grounding finishes before the cloud response and passes the semantic gate, only a lightweight on-device check with cost τ_k^{check} remains after the response. Thus the two schedules incur

$$L_k^{\text{seq}} = \tau_k^{\text{cloud}} + \tau_k^{\text{inf}} + \tau_k^{\text{exec}}, \quad L_k^{\text{async}} = \tau_k^{\text{cloud}} + \tau_k^{\text{check}} + \tau_k^{\text{exec}}. \tag{12}$$

Hence, the latency reduction for a successfully reused prefetch result is

$$L_k^{\text{seq}} - L_k^{\text{async}} = \tau_k^{\text{inf}} - \tau_k^{\text{check}}. \tag{13}$$

The improvement therefore comes from moving the same plan-conditioned visual grounding defined in Sec. 4.1 into the cloud reasoning interval. A successful match resumes directly from the resolved pixel action, whereas a mismatch falls back to the original cloud-conditioned visual tool experts without changing the execution interface.

5 Experiments

The evaluation is designed to test whether two-sided preparation improves task completion and interaction efficiency in device–cloud GUI agents. We examine overall performance across platforms and executors, isolate the contributions of post-response grounding and pre-response preparation, and analyze when prefetched computation can be reused.

5.1 Experimental Setup

Benchmarks and protocol. Following the settings in EcoAgent (Yi et al., 2026), we conduct our experiments on AndroidWorld (Rawles et al., 2025). Configured with Android 13, AndroidWorld contains 116 parameterized task templates across 20 Android applications, with programmatic initialization, evaluation, and cleanup. We further extend to OSWorld (Xie et al., 2024), a desktop benchmark containing 369 Ubuntu tasks spanning operating-system operations, office applications, everyday applications, professional software, and multi-application workflows. Both online benchmarks are fully contained in an emulator or a virtual machine, providing a consistent, reproducible real-device environment and automated evaluation mechanism.

Table 1: Task success rates (%, ↑) with different device executors on AndroidWorld and OSWorld. Pro. and Work. denote Professional and Workflow categories in OSWorld.

Method	Executor	Params	AndroidWorld SR (%) ↑				OSWorld SR (%) ↑					
			Easy	Medium	Hard	Avg.	OS	Office	Daily	Prof.	Work.	Avg.
Fully cloud	–	–	82.0	77.8	63.2	77.6	75.0	76.0	66.7	85.7	61.4	71.3
EcoAgent	UI-Mate	9B	62.3	58.3	42.1	57.8	41.7	51.3	50.0	65.3	44.6	50.4
TAP-Agent	UI-Mate	9B	68.9	66.7	57.9	**66.4**	58.3	57.3	54.5	75.5	47.5	**56.5**
EcoAgent	UI-TARS	7B	39.3	27.8	10.5	31.0	25.0	27.3	33.3	57.1	14.9	29.0
TAP-Agent	UI-TARS	7B	47.5	36.1	21.0	**39.7**	37.5	31.6	39.7	61.2	18.8	**34.1**
EcoAgent	OS-Atlas	4B	32.8	22.2	0.0	24.1	16.7	18.8	26.3	42.9	5.9	19.9
TAP-Agent	OS-Atlas	4B	41.0	27.8	5.3	**31.0**	20.8	23.1	29.5	53.1	7.9	**24.1**
EcoAgent	ShowUI	2B	26.2	16.7	0.0	19.0	12.5	16.2	24.4	32.7	4.0	16.5
TAP-Agent	ShowUI	2B	29.5	19.4	0.0	**21.6**	16.7	19.7	28.8	40.8	5.0	**20.2**

Compared methods and model selection. We compare TAP-Agent with EcoAgent (Yi et al., 2026) under matched device–cloud configurations and include a fully cloud reference. To evaluate TAP-Agent across different model scales and GUI-agent designs, we adopt four representative on-device executors: UI-Mate-9B (Ding et al., 2026) is an open-weight foundation GUI agent designed for long-horizon computer use with live-screen grounding and structured computer actions; UI-TARS-1.5-7B (Qin et al., 2025) is a general-purpose multimodal GUI agent with strong reasoning and action capabilities; OS-Atlas-4B (Wu et al., 2025) is a compact generalist action model trained for cross-platform GUI grounding and interaction; and ShowUI-2B (Lin et al., 2025) is a lightweight vision-language-action model designed for efficient visual GUI interaction. We adopt Doubao Seed 2.1 Pro (ByteDance Seed, 2026) as the common high-capacity MLLM across all configurations for cloud planning. For local observation, we use Qwen3-VL-2B (Bai et al., 2025) as a lightweight multimodal observer with visual and spatial understanding capabilities.

Method	Executor	AndroidWorld	OSWorld
Fully cloud	–	440.6 / 610.7	508.9 / 707.2
EcoAgent	UI-Mate	93.2 / 243.8	110.9 / 279.1
TAP-Agent	UI-Mate	86.9 / 208.9	98.7 / 251.5
EcoAgent	UI-TARS	128.5 / 350.3	160.7 / 366.1
TAP-Agent	UI-TARS	101.3 / 311.6	142.3 / 345.1
EcoAgent	OS-Atlas	136.9 / 374.5	157.9 / 391.9
TAP-Agent	OS-Atlas	120.5 / 347.9	139.8 / 375.2
EcoAgent	ShowUI	192.7 / 401.1	206.3 / 409.8
TAP-Agent	ShowUI	182.5 / 392.2	197.7 / 399.0

Figure 4: Average execution time (s) on successful examples / all examples. TAP-Agent consistently reduces time cost across all executor–benchmark pairs.

Implementation details. We run TAP-Agent on a single RTX 4090. Episodes are limited to 30 interaction steps on AndroidWorld and 50 on OSWorld with a 450 s timeout. The cloud-based reference is set with a 1200 s timeout. For failed episodes, the agent exceeds the timeout limit, and the time cost is recorded as the timeout value. All experiences are cleared at the beginning of the benchmark, while those generated during evaluation are retained. The on-device action predictor uses the same model as the observer and generates $M = 3$ candidate responses from experience retrieval, which uses a weighted sum with coefficients $(1, 1, 0.25)$ for lexical overlap, confidence, and recency. Details of the visual expert tools and lightweight models are provided in Appendix A.

5.2 Main Comparisons

Task completion across platforms and executors. Table 1 compares task success across four on-device executors on AndroidWorld and OSWorld. Under matched device–cloud configurations, TAP-Agent consistently outperforms EcoAgent, achieving an average improvement of **5.7%** across the eight executor–benchmark pairs. These gains span model scales from 2B to 9B and both mobile and desktop environments, indicating that the benefits extend beyond a particular executor or platform. Although the fully cloud reference achieves higher SR, TAP-Agent consistently narrows this performance gap while retaining local grounding and execution.

End-to-end interaction efficiency. Table 4 reports average execution time over successful episodes and all episodes. TAP-Agent reduces both measures in every evaluated configuration, with average relative savings of **7.2%** over all episodes and **10.4%** over successful episodes compared with EcoAgent. These averages give equal weight to the eight executor–benchmark pairs. Successful-episode means focus on completed trajectories, although each method contributes its own successful set. The ablation in Table 2 indicates that the visual experts drive the SR gain, while prefetching reduces time with and without these experts.

5.3 Ablation Study

Table 2 presents a 2×2 ablation with UI-TARS-1.5-7B, where disabling both components recovers EcoAgent. The results show the contributions of each component:

Tool	Prefetch	AndroidWorld SR	AndroidWorld Time	OSWorld SR	OSWorld Time
		31.0	128.5 / 350.3	29.0	160.7 / 366.1
✔		39.7	133.2 / 324.2	33.8	165.4 / 353.8
	✔	30.2	110.7 / 347.5	29.0	150.5 / 363.1
✔	✔	**39.7**	**101.3 / 311.6**	**34.1**	**142.3 / 345.1**

Table 2: Ablation study of visual tool experts and action prefetching with UI-TARS-1.5-7B. Time: average execution time on successful examples / all examples (s).

Cloud-guided visual expert tools. Without prefetching, enabling the visual experts improves SR on both benchmarks, with a modest time cost in successful-episode time consistent with the additional visual processing. Nevertheless, all-episode mean time decreases, yielding a favorable time trade-off despite this overhead. This suggests that the perceptual benefits of plan-conditioned tool orchestration not only raise task success but also reduce wasted execution on failed trajectories, thereby compensating for the extra computation introduced by tool invocations.

Alignment-gated asynchronous prefetching. With the visual experts enabled, prefetching reduces successful-episode time by **24.0%** on AndroidWorld and **14.0%** on OSWorld, and also lowers all-episode time, while leaving SR nearly unchanged (within $\pm1\%$). The stable SR suggests that the alignment gate can retain suitable prefetched actions and reject mismatched ones even when predictions are imperfect, while the lower time reflects computation overlapped with cloud reasoning.

5.4 Prefetch Hit Analysis

Figure 5 compares prefetch hit probability with and without the visual experts as the number of prediction candidates M varies. A hit occurs when at least one prefetched action passes alignment with the returned plan and is accepted for reuse after cloud confirmation, i.e. $\alpha_k = 1$. Increasing M improves hit probability on both benchmarks, but the gains become marginal beyond $M = 3$, supporting this default as a compact candidate budget. The tool-enabled variant consistently achieves higher hit probability; notably, with only three candidates, it outperforms Prefetch Only with five candidates on both benchmarks. These observations reveal a mutually reinforcing relationship: prefetching performs tool calls during cloud reasoning to prepare visual evidence ahead of execution, while more accurate tool-assisted interactions can supply more reliable experience for subsequent anticipation.

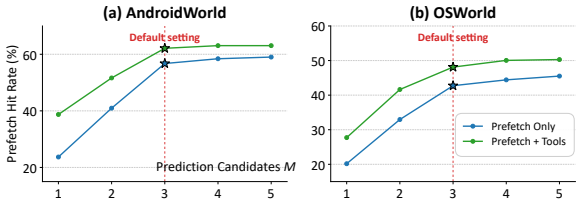

Figure 5: Prefetch hit probability versus candidate count M, with and without visual tools. Stars mark the default $M = 3$.

6 Conclusion

In this work, we propose TAP-Agent, a framework that organizes device–cloud GUI collaboration through two-sided anticipatory preparation around each cloud response. By combining the compilation of heterogeneous visual evidence from expert tools with in-flight next-round planning and semantic-gated action reuse, TAP-Agent strengthens element-level grounding and shifts reusable computation into the cloud-reasoning interval. Long-horizon experiments on AndroidWorld and OSWorld with different on-device executors demonstrate higher task success rate and lower end-to-end interaction latency. We hope these findings inspire further research on coordinating device and cloud computation over time to build more reliable and efficient GUI agents.

AI USE STATEMENT

We used generative AI tools to polish the language of the manuscript and improve its readability. We did not use these tools for generating experimental data and results. The authors reviewed all AI-assisted edits and take full responsibility for the final manuscript, including its text, claims, and results.

ETHICS STATEMENT

We used the publicly available AndroidWorld and OSWorld benchmarks for evaluation. All experiments were conducted by a single researcher in isolated virtual machines or emulators on dedicated servers. The study did not involve human participants or real user accounts, and the experiments caused no damage to production systems or live environments. Data uploaded to cloud services consisted only of simulated data and contained no private information. Our evaluation was limited to these controlled settings; deployment in real environments would require safeguards against unintended actions and unauthorized access. We plan to release our code as open source.

REPRODUCIBILITY STATEMENT

Our method is described in Section 4, and the experimental setup and evaluation protocol are detailed in Section 5. We use the publicly available AndroidWorld and OSWorld benchmarks for evaluation. We will release all code used in this work to support reproduction of the reported results.

REFERENCES

Reyna Abhyankar, Qi Qi, and Yiying Zhang. OSWorld-Human: Benchmarking the efficiency of computer-use agents. In *Proceedings of Machine Learning and Systems*, volume 8, 2026.

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

Gilles Baechler, Srinivas Sunkara, Maria Wang, Fedir Zubach, Hassan Mansoor, Vincent Etter, Victor Cărbune, Jason Lin, Jindong Chen, and Abhanshu Sharma. ScreenAI: A vision-language model for UI and infographics understanding. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, pp. 3058–3068, 2024.

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, et al. Qwen3-VL technical report. *arXiv preprint arXiv:2511.21631*, 2025.

ByteDance Seed. Seed2.1: A next-generation agent for real-world productivity, 2026.

Nicolas Carion, Laura Gustafson, Yuan-Ting Hu, Shoubhik Debnath, Ronghang Hu, Didac Suris Coll-Vinent, Chaitanya Ryali, Kalyan Vasudev Alwala, Haitham Khedr, Andrew Huang, et al. Sam 3: Segment anything with concepts. In *International conference on learning representations*, volume 2026, pp. 138846–138923, 2026.

Chen Chen, Jiawei Shao, Dakuan Lu, Haoyi Hu, Xiangcheng Liu, Hantao Yao, and Wu Liu. GUI-Eyes: Tool-augmented perception for visual grounding in GUI agents. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(35):29350–29358, 2026a.

Liangyu Chen, Hanzhang Zhou, Chenglin Cai, Jianan Zhang, Panrong Tong, Xu Zhang, Quyu Kong, Chen Liu, Yuqi Liu, Wenxuan Wang, Yue Wang, Qin Jin, and Steven Hoi. UI-Ins: Enhancing GUI grounding with multi-perspective instruction as reasoning. In *International Conference on Learning Representations*, 2026b.

Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Yantao Li, Jianbing Zhang, and Zhiyong Wu. SeeClick: Harnessing GUI grounding for advanced visual GUI agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9313–9332, 2024.

Cheng Cui, Yubo Zhang, Ting Sun, Xueqing Wang, Hongen Liu, Manhui Lin, Yue Zhang, Tingquan Gao, Changda Zhou, Jiaxuan Liu, et al. Pp-ocrv5: A specialized 5m-parameter model rivaling billion-parameter vision-language models on ocr tasks. *arXiv preprint arXiv:2603.24373*, 2026.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2Web: Towards a generalist agent for the web. In *Advances in Neural Information Processing Systems*, volume 36, pp. 28091–28114, 2023.

Zihan Ding, Longxu Dou, Qi Gao, Xiangwu Guo, Shengchao Hu, Zilong Huang, Zihang Jiang, Lei Ke, Mengcheng Lan, Weixian Lei, Hanxuan Li, Honglin Li, Xiyun Li, Zaitang Li, Leowei Liang, Xin Luo, Haozhe Ma, Jiayi Mao, Zhoujie Pan, Can Qin, Tianyuan Qu, Weiqi Wang, Wenkai Wang, Yonglin Wang, Yuxin Wang, Chenxu Wu, Yingchen Yu, Chenyu Zhang, and Yuhao Zheng. UI-Mate: Advancing open-weight foundation GUI agents with in-context demonstrations. *arXiv preprint arXiv:2608.15930*, 2026.

Zihan Dong, Rui Qian, Qishi Zhan, Dongshen Peng, Kaixin Li, and Yu Li. Why are GUI agents correct but late? decode on the decision-time critical path, tested with pre-compiled policy trees. *arXiv preprint arXiv:2607.28399*, 2026.

Yuning Du, Chenxia Li, Ruoyu Guo, Xiaoting Yin, Weiwei Liu, Jun Zhou, Yifan Bai, Zilin Yu, Yehua Yang, Qingqing Dang, et al. PP-OCR: A practical ultra lightweight OCR system. *arXiv preprint arXiv:2009.09941*, 2020.

Boyu Gou, Ruohan Wang, Boyuan Zheng, Yanan Xie, Cheng Chang, Yiheng Shu, Huan Sun, and Yu Su. Navigating the digital world as humans do: Universal visual grounding for GUI agents. In *International Conference on Learning Representations*, 2025.

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

Yangqin Jiang and Chao Huang. OpenPhone: Mobile agentic foundation models. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 30362–30380. Association for Computational Linguistics, 2026.

Weitai Kang, Bin Lei, Gaowen Liu, Caiwen Ding, and Yan Yan. GuirlVG: Incentivize GUI visual grounding via empirical exploration on reinforcement learning. In *International Conference on Learning Representations*, 2026.

Hanyu Lai, Xiao Liu, Yanxiao Zhao, Han Xu, Hanchen Zhang, Bohao Jing, Yanyu Ren, Shuntian Yao, Yuxiao Dong, and Jie Tang. ComputerRL: Scaling end-to-end online reinforcement learning for computer use agents. In *International Conference on Learning Representations*, 2026.

Kenton Lee, Mandar Joshi, Iulia Raluca Turc, Hexiang Hu, Fangyu Liu, Julian Martin Eisenschlos, Urvashi Khandelwal, Peter Shaw, Ming-Wei Chang, and Kristina Toutanova. Pix2Struct: Screenshot parsing as pretraining for visual language understanding. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 18893–18912. PMLR, 2023.

Chenxia Li, Weiwei Liu, Ruoyu Guo, Xiaoting Yin, Kaitao Jiang, Yongkun Du, Yuning Du, Lingfeng Zhu, Baohua Lai, Xiaoguang Hu, Dianhai Yu, and Yanjun Ma. PP-OCRv3: More attempts for the improvement of ultra lightweight OCR system. *arXiv preprint arXiv:2206.03001*, 2022.

Hongxin Li, Yuntao Chen, and Zhaoxiang Zhang. GoClick: Lightweight element grounding model for autonomous GUI interaction. *arXiv preprint arXiv:2604.23941*, 2026.

Minghao Li, Tengchao Lv, Jingye Chen, Lei Cui, Yijuan Lu, Dinei Florencio, Cha Zhang, Zhoujun Li, and Furu Wei. TrOCR: Transformer-based optical character recognition with pre-trained models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 13094–13102, 2023.

Minghui Liao, Zhaoyi Wan, Cong Yao, Kai Chen, and Xiang Bai. Real-time scene text detection with differentiable binarization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 11474–11481, 2020.

Kevin Qinghong Lin, Linjie Li, Difei Gao, Zhengyuan Yang, Shiwei Wu, Zechen Bai, Stan Weixian Lei, Lijuan Wang, and Mike Zheng Shou. ShowUI: One vision-language-action model for GUI visual agent. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 19498–19508, 2025.

Zhaoyang Liu, JingJing Xie, Zichen Ding, Zehao Li, Bowen Yang, Zhenyu Wu, Xuehui Wang, Qiushi Sun, Shi Liu, Weiyun Wang, Shenglong Ye, Qingyun Li, Zeyue Tian, Gen Luo, Xiangyu Yue, Biqing Qi, Kai Chen, Bowen Zhou, Yu Qiao, Qifeng Chen, and Wenhai Wang. ScaleCUA: Scaling open-source computer use agents with cross-platform data. In *International Conference on Learning Representations*, 2026.

Yadong Lu, Jianwei Yang, Yelong Shen, and Ahmed Awadallah. OmniParser for pure vision based GUI agent. *arXiv preprint arXiv:2408.00203*, 2024.

Andrés Marafioti, Orr Zohar, Miquel Farré, Merve Noyan, Elie Bakouch, Pedro Cuenca, Cyril Zakka, Loubna Ben Allal, Anton Lozhkov, Nouamane Tazi, Vaibhav Srivastav, Joshua Lochner, Hugo Larcher, Mathieu Morlon, Lewis Tunstall, Leandro von Werra, and Thomas Wolf. Smolvlm: Redefining small and efficient multimodal models. *arXiv preprint arXiv:2504.05299*, 2025.

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. *arXiv preprint arXiv:2501.12326*, 2025.

Chris 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. In *International Conference on Learning Representations*, 2025.

Jason Wu, Xiaoyi Zhang, Jeff Nichols, and Jeffrey P. Bigham. Screen parsing: Towards reverse engineering of UI models from screenshots. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pp. 470–483. Association for Computing Machinery, 2021.

Penghao Wu and Saining Xie. V*: Guided visual search as a core mechanism in multimodal LLMs. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 13084–13094, 2024.

Yixuan Wu, Yizhou Wang, Shixiang Tang, Wenhao Wu, Tong He, Wanli Ouyang, Philip Torr, and Jian Wu. DetToolChain: A new prompting paradigm to unleash detection ability of MLLM. In *European Conference on Computer Vision*, pp. 164–182. Springer, 2024.

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: Foundation action model for generalist GUI agents. In *International Conference on Learning Representations*, 2025.

Wei Xiang, Yexinrui Wu, Xinli Chen, Xinran Li, and Shi Chen. UI-Lens: Assessing general MLLMs' potential to automate UI display quality assurance. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 25882–25892, 2026.

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. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

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. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 69772–69805. PMLR, 2025.

Yan Yang, Dongxu Li, Yutong Dai, Yuhao Yang, Ziyang Luo, Zirui Zhao, Zhiyuan Hu, Junzhe Huang, Amrita Saha, Zeyuan Chen, Ran Xu, Liyuan Pan, Caiming Xiong, and Junnan Li. GTA1: GUI test-time scaling agent. In *International Conference on Learning Representations*, 2026.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

Biao Yi, Xueyu Hu, Yurun Chen, Shengyu Zhang, Hongxia Yang, and Fan Wu. EcoAgent: An efficient device-cloud collaborative multi-agent framework for mobile automation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(35):29838–29846, 2026.

Yuzhe Zhang, Xianwei Xue, Xingyong Wu, Mengke Chen, Chen Liu, Xinran He, Run Shao, Feiran Liu, Huanmin Xu, Qiutong Pan, and Haiwei Wang. Don't act blindly: Robust GUI automation via action-effect verification and self-correction. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 28924–28941. Association for Computational Linguistics, 2026.

Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. GPT-4V(ision) is a generalist web agent, if grounded. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 61349–61385. PMLR, 2024.

Hanzhang Zhou, Xu Zhang, Panrong Tong, Jianan Zhang, Liangyu Chen, Quyu Kong, Chenglin Cai, Chen Liu, Yue Wang, Jingren Zhou, and Steven Hoi. MAI-UI technical report: Real-world centric foundation GUI agents. *arXiv preprint arXiv:2512.22047*, 2025.

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. In *International Conference on Learning Representations*, 2024.

TAP-Agent: Appendix

The appendix is organized as follows:

- Appendix A describes the visual tools, agent interfaces, deployment, response parsing, and coordinate adaptation.
- Appendix B details observation, prefetch validation, and action selection.
- Appendix C specifies procedural experience construction, retrieval, and episode boundaries.
- Appendix D provides representative prompts for the toolchain, prefetching, and experience.

A Visual Tools and Execution Interfaces

This section specifies the evidence produced by local tools and the interfaces that connect planning to execution.

A.1 Visual Tool Registry

We register seven on-device visual tools, each with an explicit input and output interface. The cloud-provided requests q_k specify tool selections and task-dependent arguments, which the device applies to the available observations:

Tool	Evidence and implementation details
Element detection	OmniParser (Lu et al., 2024) parses a screenshot into candidate GUI elements with bounding boxes and semantic descriptions. Task-specific keyword filtering and ranking select instruction-relevant candidates for downstream element-level grounding. Evidence includes descriptions, local identifiers, and boxes; the reference configuration returns at most 30 elements.
Text recognition	This tool recognizes text in a full screenshot or selected region, returning text strings, bounding boxes, and confidence scores. By associating recognized content with screen locations, it supports identifying text-defined targets and reading interface values. Evidence includes visible text, recognition scores, and optionally boxes. The configured standalone backend uses English PP-OCRv5 (Cui et al., 2026) through PaddleOCR, on CPU, with minimum score 0.3, at most 120 lines, image upscale 1.5, and detector side limit 960. This is distinct from OCR inside OmniParser.
Regional zooming	This tool crops and enlarges a specified screen region to facilitate inspection of small controls and fine text, enabling more precise inspection especially when important controls or labels are too small to recognize reliably at the original screen resolution. Evidence includes a magnified crop and its full-screen bounding box; margin 0.25, target longest side 1,344 pixels, minimum crop size 96, and JPEG quality 92.
Target-focused masking	Given the screenshot and task target, SAM3 (Carion et al., 2026) segments the screen into regions and assigns each region an interest score for target relevance. The tool dims low-interest regions while retaining the screen layout. Evidence includes region masks, their interest scores, and the dimmed composite view. These are interest scores, not action-confidence scores. Reference settings: dim factor 0.35 for non-target regions, mask dilation 5 px, and output at original resolution.

Tool	Evidence and implementation details
Scroll mapping	To track elements in long pages, this tool compares screenshots captured before and after scrolling, estimates scroll offsets, tracks elements across views, and constructs a stitched page representation. Evidence includes stateful comparison of a stored pre-scroll frame and the current frame, including displacement and match diagnostics. The wrapper reports whether a stitched image is available; it does not deliver that image as executor input in the inspected path.
Action verification	SmolVLM-500M (Marafioti et al., 2025) performs visual question answering (VQA) on the outcome screenshot of an executed action and returns a binary judgment of whether the intended effect is visible. This short answer provides nonspatial feedback on the individual action. Reference generation budget: 128 tokens, temperature 0.
Anomaly detection	The same SmolVLM-500M VQA backend examines the current screenshot for blocking dialogs, errors, authentication walls, or loading failures. It returns the detected condition, supporting evidence, and recovery suggestions, providing diagnostic context without executing corrective actions. Reference generation budget: 160 tokens, temperature 0.

Localized outputs supply candidate action targets, while transformed views, cross-scroll context, and execution assessments provide complementary evidence. All tools operate on captured observations, separating visual evidence acquisition from GUI-changing actions.

A.2 Agent Roles and Interfaces

The cloud planner receives the task and visible desktop context and produces an ordered plan. Each entry contains `thought`, `step`, and `expectation`; optional fields are `tool_use` and `check_only`. The expectation specifies a visible postcondition. Tool requests are attached to the action that consumes their evidence, rather than creating an additional GUI-changing "localization" step. Observation-only entries must not also change the interface.

The following is an illustrative schema example, not a recorded trajectory:

```
{
  "Step1": {
    "thought": "Locate the field before entering the query.",
    "step": "CLICK the visible search field",
    "expectation": "The search field is focused.",
    "tool_use": [{"name": "omniparser", "args": {}}],
    "check_only": false
  }
}
```

A.3 Reference Deployment and Decoding

Component	Inspected reference setting
Cloud planner	`doubao-seed-2-1-pro-260915` (ByteDance Seed, 2026); temperature 0; maximum output 4,096 tokens; thinking disabled.
Executor	UI-Mate-9B (Ding et al., 2026) / UI-TARS-1.5-7B (Qin et al., 2025) / OS-Atlas-4B (Wu et al., 2025) / ShowUI-2B (Lin et al., 2025), BF16; temperature 0; maximum output 2,048 tokens; thinking disabled in the serving request and the adapter instruction.

Observer	Qwen3-VL-2B-Instruct (Bai et al., 2025), served under the alias Qwen3-VL; temperature 0; description 256, summary 384, and matching 256 output tokens.
Visual tools	OmniParser (Lu et al., 2024) service; CPU PaddleOCR; shared SmolVLM-500M (Marafioti et al., 2025) instance for VQA and anomaly detection.
Desktop	Ubuntu guest; 1,920 by 1,080 screenshots; four virtual CPUs; 8 GiB guest RAM; QEMU software emulation in the inspected deployment.

A.4 Cloud Output Formatting and Parsing

The desktop planning prompt requests a brief **Description**, a brief **Thought**, and a **Plan** containing a complete JSON object. Thus the response envelope is not necessarily a single JSON document. The parser must locate the intended plan and validate its structure before executing any step. Required fields, boolean types, step ordering, and a closed object should be verified rather than inferred from an arbitrary prefix of the response.

A decoding cap can truncate an otherwise useful response before its closing braces. The 4,096-token setting is an output-token cap; it is not the model's advertised context-window size. End-to-end plans, repeated tool arguments, and descriptive fields all consume this budget. Disabling model thinking does not eliminate malformed JSON or guarantee that an entire plan fits. Increasing a cap changes latency and cost and must be recorded as a protocol change. Changing the cloud model identifier alone also does not establish that truncation or schema errors have been solved.

A.5 Geometry and Executor Adaptation

Spatial evidence must identify its coordinate system explicitly. A box in normalized image coordinates, a crop-local point, and a screen-pixel point are not interchangeable. The element adapter converts normalized boxes to screen coordinates, orders their endpoints, clips them to image boundaries, and derives centers. Element identifiers are local to a tool result and can be renumbered after filtering. They must not be treated as persistent widget identities across screenshots.

For a crop whose upper-left screen point is (x_0, y_0) and whose original size is (w_c, h_c), a point (u, v) predicted on a resized crop of size (w_z, h_z) maps back to the screen as

$$x = x_0 + u\frac{w_c}{w_z}, \qquad y = y_0 + v\frac{h_c}{h_z}. \tag{14}$$

The transformation must use the actual crop after clipping and margin expansion. Mixing the requested crop with the actual crop creates systematic localization errors near image boundaries.

B Prefetch Validation and Execution

This section describes how the observer assesses screen changes and how a prepared action is selected for reuse.

B.1 Observation, Stagnation, and Screen Validity

Description, transition assessment, recovery proposal, and speculative matching use the same Qwen3-VL-2B-Instruct service. The matching response contains a candidate identifier and confidence, while other observer calls retain their requested brief explanation.

The implementation also compares screenshots numerically. It crops the vertical range from 5.5% to 96.5% of image height, converts to grayscale, and resizes to 64×64. For resulting images A and B, define

$$D(A, B) = \frac{1}{255N}\sum_{j=1}^{N}|A_j - B_j|, \tag{15}$$

$$S(A, B) = 1 - \frac{1}{N}\sum_{j=1}^{N}\mathbf{1}\{|A_j - B_j| > 12\}, \qquad N = 4096. \tag{16}$$

The second quantity is a thresholded pixel agreement. Candidate screen validation requires $D \leq 0.02$ and $S \geq 0.97$. Decode failure rejects reuse. A separate no-op detector uses $D \leq 0.006$ and $S \geq 0.995$, with replanning after three consecutive detected no-ops and termination after six.

B.2 Selection and Execution

For each predicted next-round response, the device evaluates its visual-tool requests on the realized post-action screenshot. It compiles the resulting evidence, selects a target, and resolves an action in screen coordinates. This preparation does not issue GUI events; the action remains pending until the cloud response arrives.

The returned cloud goal is compared with each prepared candidate's semantic goal. A candidate must also satisfy the active executor's action constraints. When the alignment gate accepts a candidate, the device checks that the current screen still matches the screenshot used for grounding, as described in Section B.1, and executes its resolved action. A tool request in the returned cloud step does not itself prevent reuse: visual-tool inference, evidence compilation, and coordinate resolution have already been completed for the accepted candidate. If alignment or screen validation fails, the device discards the candidate and evaluates the returned visual intent through the standard toolchain.

C Procedural Experience

This section describes how transition lessons are recorded, retrieved, and carried across episodes.

C.1 Construction, Ranking, and Lifetime

The experience worker summarizes local transition events into textual records. A record contains reusability, application scope, trigger information, a lesson, recommended action, action to avoid, and confidence. The reflection request uses text summaries rather than image inputs and inherits the cloud planner model in the reference configuration. Thus experience storage is local, but experience reflection is not exclusively on-device inference.

The queue capacity is 64; field and event-payload limits are 1,200 and 12,000 characters. Reflection uses temperature 0, a 512-token output budget, a 45-second request timeout, and a configured retry limit of one. Queue insertion is nonblocking: a full queue can drop an event, with an error record. Closing the worker waits up to 50 seconds and is a best-effort operation, not a guarantee that every outstanding event has been persisted.

C.2 Exact retrieval score

Let $T(q)$ and $T(m)$ be the distinct lexical terms extracted from the query and experience text. The tokenizer lowercases English text and retains matches of `[a-zA-Z][a-zA-Z0-9_]{2,}`, excluding the fixed stopword set {the, and, for, with, that, this, from, into, task, step}. In particular, this is neither an embedding retriever nor a multilingual tokenizer. The overlap is an unnormalized intersection count:

$$O(q, m) = |T(q) \cap T(m)|. \tag{17}$$

For confidence $C(m)$ and recency position i in the reverse traversal of the startup snapshot, the implemented score is

$$R_i = \max(0, 1 - i/500), \tag{18}$$

$$\text{score}(q, m) = O(q, m) + B(q, m) + C(m) + 0.25R_i. \tag{19}$$

The three headline coefficients for lexical overlap, confidence, and recency are therefore $(1, 1, 0.25)$. They are not normalized mixture probabilities. The additional trigger bonus B contributes 2 for a matching state substring, 1.5 for a matching task-family substring, and 0.75 per matching screen keyword. Omitting B changes the actual ranking rule.

Records are filtered by platform and compatible application scope, a minimum confidence of 0.35, and reusability. A record with neither lexical overlap nor trigger evidence is rejected even if confidence is high. Recency is line position, not elapsed wall-clock time; positions can include lines

later rejected during parsing or filtering. General retrieval returns up to three records, while the speculative-experience branch in the current configuration requests one. The injected experience block is limited to 3,000 characters.

D PROMPT EXAMPLES

The following prompts illustrate tool-guided replanning, action grounding, speculative prefetch, and procedural experience used by TAP-Agent. One example is shown for each function. Task-specific inputs are omitted.

D.1 CLOUD REPLANNING PROMPT

```
Original plan:
{original_plan}

Screen history:
{descriptions}

Latest observer result for the failed step:
{summary}

Current desktop description:
{description}

Create a corrected plan for the REMAINING work toward: {goal}

Explain the failure only inside each step's short `thought`; return a Plan JSON object in
↪  the same schema as the initial planner.
- Start from the current real desktop state, not the original start state.
- Do not repeat a no-effect target, coordinate, gesture, or strategy.
- Preserve already completed effects and repair only what remains.
- If a popup or menu blocks the target, dismiss or handle that visible blocker first.
- If the original plan is still correct and the failed expectation was simply checked too
↪  early, return exactly `No need to replan.`

This is an Ubuntu desktop controlled through OSWorld. Work only from visible screen
↪  evidence.
- Produce a complete end-to-end, step-by-step plan from the CURRENT screen to the FINAL
↪  user goal. The plan is not merely the next action.
- Cover every necessary intermediate action, including opening the relevant application or
↪  page, editing the requested content or setting, and saving/submitting when the goal
↪  requires it.
- Do not end the plan at an intermediate milestone such as launching an app, opening a
↪  menu, focusing a field, or reaching a settings page. A one-step plan is valid only when
↪  that single action visibly completes the entire original goal.
- Retrieved experiences describe local step transitions, not complete solutions. They may
↪  help choose an action but must never shorten or replace the remaining end-to-end plan.
- The task setup normally opens the relevant application and files. Do not reopen or reset
↪  applications unless the screenshot shows that it is necessary.
- Ubuntu's left dock contains pinned applications and its bottom grid button opens Show
↪  Applications; do not infer a Trash icon from dock position alone. For Trash tasks, open
↪  the visible Files application and use its `Trash` sidebar entry unless a Trash label is
↪  explicitly visible.
- Break text entry into CLICK field, optional DELETE_TEXT, then INPUT [text]. Use ENTER
↪  only when submission is required.
- Every plan entry must contain exactly one GUI-changing action. Never join CLICK, HOTKEY,
↪  DELETE_TEXT, INPUT, or ENTER with "then" or commas in the same step.
- Use ordinary single CLICK for buttons, tabs, fields, menus, and links. Use DOUBLE_CLICK
↪  for desktop files/folders only when opening them requires it. Use RIGHT_CLICK only when
↪  a context menu is required.
- Use SCROLL for document/list/page navigation and DRAG for sliders, selections, moving
↪  objects, or resizing.
- Describe the semantic target, not screen coordinates. The grounding model chooses
↪  coordinates from the current screenshot.
- Use HOTKEY [CTRL+S], HOTKEY [CTRL+F], HOTKEY [CTRL+L], HOTKEY [CTRL+Z], or similar only
↪  when the shortcut is appropriate for the active application.
- In Visual Studio Code, CTRL+H opens Replace and CTRL+ALT+ENTER runs Replace All; prefer
↪  these standard shortcuts when a small icon is hard to ground reliably.
- Treat toggles, checkboxes, and sliders as stateful. Inspect their current visible state
↪  and do not change them if the goal is already satisfied.
- Never use terminal/shell commands, OSWorld internals, accessibility APIs, browser
↪  developer tools, or programmatic file manipulation to bypass GUI interaction.
- Never claim success from the plan alone. Each step expectation must name a visible state
↪  that can be checked after the action.
- Use the exact step `ANSWER` only when the user asks to return information as text. Do not
↪  append ANSWER to normal UI-operation tasks.
```

```
Allowed planner actions: CLICK, DOUBLE_CLICK, RIGHT_CLICK, SCROLL, DRAG, INPUT [text],
↪  DELETE_TEXT, ENTER, HOTKEY [keys], PRESS_ESCAPE, WAIT [seconds], ANSWER.

Available auxiliary tools for `tool_use`:
- `omniparser`: locate visible desktop controls and return element ids and pixel boxes.
- `sam`: segment the screenshot into regions and assign each an interest score for the task
↪  target; use it to emphasize relevant regions.
- `zoom_in`: magnify a small or crowded region; provide normalized `bbox`.
- `ocr`: read visible text; set `include_boxes: true` only when coordinates are needed.
- `vqa`: answer a concise non-coordinate visual-state question.
- `anomaly_detection`: inspect for blocking dialogs, errors, login walls, or loading
↪  failures.
- `scroll_tracker`: use only with scrolling. On the first scroll use
↪  `{{"name":"scroll_tracker","args":{{"mode":"store"}}}}`; before a later scroll use
↪  `{{"name":"scroll_tracker","args":{{"mode":"track"}}}}` to compare the current screen
↪  with the stored pre-scroll screen.

Tool rules:
- `tool_use` runs on the screenshot at the start of that same step.
- Attach localization tools to the action that consumes their result; never make "locate" a
↪  separate step.
- A pure observation step must use only OCR/VQA/anomaly_detection and set `"check_only":
↪  true`.
- Never combine `check_only: true` with a GUI-changing action.

Return exactly this structure and no extra text:
Description: <brief current desktop state>
Thought: <brief task approach>
Plan: <one complete valid JSON object>

Each plan entry must contain `thought`, `step`, and `expectation`. It may contain `tool_use`
↪  and boolean `check_only`.
The Plan must cover all remaining work needed to satisfy the original user goal. Do not
↪  output a partial next-action-only plan.
Example:
```json
{{
  "Step1": {{"thought": "Focus the search field.", "step": "CLICK the visible search
  ↪  field", "expectation": "The search field is focused.", "tool_use": []}},
  "Step2": {{"thought": "Enter the requested query.", "step": "INPUT [quarterly report]",
  ↪  "expectation": "The search field contains quarterly report.", "tool_use": []}},
  "Step3": {{"thought": "Submit the search.", "step": "ENTER", "expectation": "Search
  ↪  results for quarterly report are visible.", "tool_use": []}}
}}
```
All JSON strings must use double quotes. `check_only` must be a JSON boolean, never a
↪ quoted string. Keep each step description concise so the complete end-to-end plan fits
↪ and the JSON object closes correctly.
```

D.2 Action Grounding Prompt

```
You are grounding one immediate action on an Ubuntu desktop screenshot. Execute only the
↪  requested immediate step.

Output exactly:
Thought: one concise English sentence under 20 words
Action: exactly one action

Action space (coordinates are absolute screenshot pixels):
- click(start_box='(x,y)')
- click_element(element_id='id') only when current tool results provide that id
- double_click(start_box='(x,y)')
- right_click(start_box='(x,y)')
- long_press(start_box='(x,y)')
- type(content='text')
- scroll(start_box='(x,y)', direction='up|down|left|right')
- drag(start_box='(x1,y1)', end_box='(x2,y2)')
- hotkey(keys=['ctrl','s'])
- press_enter()
- press_escape()
- finished(content='done')

Desktop grounding rules:
- Click the center of the visible target. Do not guess coordinates when it is not visible.
- Do not infer a Trash icon from the bottom of Ubuntu's dock. Use Files and its labeled
↪  Trash sidebar for Trash tasks.
```

```
- Single-click normal controls. Double-click only desktop files/folders that require
↪  opening. Right-click only for a context menu.
- Use scroll for page/list/document navigation; use drag only for sliders, selections,
↪  moving, or resizing.
- Keep coordinates within the screenshot. Never use image search, screenshots, shell,
↪  filesystem APIs, or accessibility APIs.
- `finished` is valid only when this immediate instruction explicitly says no further
↪  action is needed.
- Return one complete Action line, without markdown or additional actions.

Immediate task: {goal}
```

D.3 SPECULATIVE MATCH PROMPT

```
You are validating speculative GUI actions after the authoritative cloud replan has
↪  returned. Select a candidate only when it performs the SAME immediate UI transition as
↪  the cloud step on the current screenshot.

Cloud step: {cloud_step}
Cloud expectation: {cloud_expectation}
Candidates: {candidates}

Matching rules:
- Operation, semantic target, required argument, and expected visible effect must agree.
- Similar wording is not enough. Reject candidates that skip a prerequisite, assume another
↪  screen, use a different target, or perform a broader/narrower action.
- The candidate's grounded action must be a plausible realization of the cloud step.
- Be conservative: uncertainty means no match.

Return valid JSON only. Do not include a reason field or explanatory prose:
{{"candidate_id":"candidate id or empty string","confidence":0.0}}
```

D.4 EXPERIENCE REFLECTION PROMPT

```
You are maintaining a collection of reusable experiences for a GUI agent.

You receive one successful step transition from an experiment. It contains only
text summaries, not screenshots. Extract a reusable lesson only if the transition
contains a pattern that could help future GUI tasks.

Return valid JSON only with this schema:
{{
  "is_reusable": true or false,
  "apps": ["canonical app name whose UI or behavior this lesson depends on"],
  "scope": "short domain or app family",
  "trigger": {{
    "task_family": "short task family if inferable",
    "screen_keywords": ["keyword1", "keyword2"],
    "state": "compact state signature"
  }},
  "lesson": "what was learned",
  "recommended_action": "what to do next time",
  "avoid": "what to avoid, or empty string",
  "confidence": 0.0
}}

Prefer conservative experiences. Do not invent details not supported by the event.
In "apps", include only the concrete app or apps whose app-specific UI or
behavior the lesson actually depends on. Identify the app used in this step,
not every app mentioned by a multi-app task. Use [] for genuinely app-agnostic
GUI actions. Use "android_system" only for Android system UI or Settings, and
use "osworld_system" only for Ubuntu desktop or system behavior.
If this is merely a routine tap/type with no reusable pattern, set
"is_reusable": false and keep the fields concise.

Successful step transition:
{event_json}
```