UIAnchor: Anchoring UI Perception and Action Execution for Reliable Service-Composed Mobile Task Automation with GUI Agents

WENTAO ZHOU, Northwestern Polytechnical University, China
SICONG LIU*, Northwestern Polytechnical University, China
ZIMU ZHOU, City University of Hong Kong, China
YIMENG DUAN, Northwestern Polytechnical University, China
YONGYAN CAI, Northwestern Polytechnical University, China
WEIYE WU, Northwestern Polytechnical University, China
TENG LI, Northwestern Polytechnical University, China
DAQING ZHANG, Institut Polytechnique de Paris, France
ZHIWEN YU, Harbin Engineering University, China

Mobile task automation aims to streamline *multi-step, cross-app* interactions on smartphones and in-vehicle systems. Recent LLM-based GUI agents have advanced rapidly, yet reliability remains limited because modern mobile workflows are increasingly *service-composed* (app switching, forms, pop-ups/permissions, copy-paste), requiring *fine-grained* operations on dense, dynamic, and heterogeneous UIs, often in distraction-sensitive contexts (*e.g.*, hands-busy or attention-limited use). Through an in-the-wild failure analysis of deployable GUI agents, we identify two dominant bottlenecks: agents often miss or misread actionable UI elements, and they execute actions without verifying target correctness or outcomes. We present UIAnchor, a *modular* multi-agent system that improves mobile GUI automation by *anchoring* both perception and execution. UIAnchor combines a two-stage UI parser for high-recall element capture and context-aware semantics with a meta-controller that performs pre-action verification, post-action outcome perception, per-step state tracking, and targeted recovery. We further introduce an L1-L5 task taxonomy based on step length, cross-app scope, and UI granularity, showing that L5 remains beyond today's practical frontier. On the hardest practical tier, L4 tasks (20–30 steps, multi-app, targets $< 100 \times 100$ px, ~5 mm on typical phones), UIAnchor improves success by 31.5% over GPT-4o and 16.6% over Mobile-Agent-v3. With edge/cloud assistance, UIAnchor runs at ~2 s per step, comparable to human operation, while reducing per-step latency by up to 75.5% and energy by 52.4%. It also generalizes to unseen apps and cross-platform GUIs.

CCS Concepts: • **Human-centered computing** → **Ubiquitous and mobile computing**; • **Computing methodologies** → **Machine learning**.

*Corresponding author.

Authors' Contact Information: Wentao Zhou, Northwestern Polytechnical University, Xi'an, China, never_gone.oldcat@mail.nwpu.edu.cn; Sicong Liu (corresponding author), Northwestern Polytechnical University, Xi'an, China, scliu@nwpu.edu.cn; Zimu Zhou, School of Data Science, City University of Hong Kong, Hong Kong, China and City University of Hong Kong Shenzhen Research Institute, Shenzhen, China, zimuzhou@cityu.edu.hk; Yimeng Duan, Northwestern Polytechnical University, Xi'an, China, 2935752563@mail.nwpu.edu.cn; Yongyan Cai, Northwestern Polytechnical University, Xi'an, China, yongyanc@mail.nwpu.edu.cn; Weiye Wu, Northwestern Polytechnical University, Xi'an, China, maxwellwwy@mail.nwpu.edu.cn; Teng Li, Northwestern Polytechnical University, Xi'an, China, liteng4735@mail.nwpu.edu.cn; Daqing Zhang, Telecom SudParis, Institut Polytechnique de Paris, Paris, France and Peking University, Beijing, China, daqing.zhang@telecom-sudparis.eu; Zhiwen Yu, Harbin Engineering University, Harbin, China and Northwestern Polytechnical University, Xi'an, China, zhiwenyu@nwpu.edu.cn.

ACM 2474-9567/2026/9-ART197
https://doi.org/10.1145/3832008

ACM Reference Format:
Wentao Zhou, Sicong Liu, Zimu Zhou, Yimeng Duan, Yongyan Cai, Weiye Wu, Teng Li, Daqing Zhang, and Zhiwen Yu. 2026. UIAnchor: Anchoring UI Perception and Action Execution for Reliable Service-Composed Mobile Task Automation with GUI Agents. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 10, 3, Article 197 (September 2026), 30 pages. https://doi.org/10.1145/3832008

1 INTRODUCTION

Human-computer interaction is undergoing a major *shift*: from users *operating interfaces* to *delegating intent* and letting agents automatically execute **multi-step**, **cross-app**, **mm-scale** UI actions. This shift is amplified by today's *service-composed* mobile workflows, where routine tasks increasingly involve permission pop-ups, copy–paste, form filling, and multi-app handoffs. It is particularly valuable on mobile and in-vehicle devices, where interaction often occurs in **hands-busy** or **attention-limited** contexts (*e.g.*, driving, on-the-move, or for users with visual impairments; see Fig. 1), making *dense cross-app* tapping *inconvenient*, *unsafe*, or *inaccessible*.

Motivated by this trend, *mobile task automation* aims to autonomously carry out *multi-step, cross-app* tasks, such as reordering a frequent meal (app 1) and inviting a friend (app 2), or messaging a contact (app 1) and launching navigation (app 2) while driving. While existing platforms offer shortcuts [5] and scripted routines (*e.g.*, Bixby Routines [35]), they typically require *manual* configuration and mainly support pre-defined, single-app, low-granularity operations, or rely on brittle UI assumptions. Thus, they remain fragmented and struggle to generalize across diverse *third-party* apps and *evolving* interfaces.

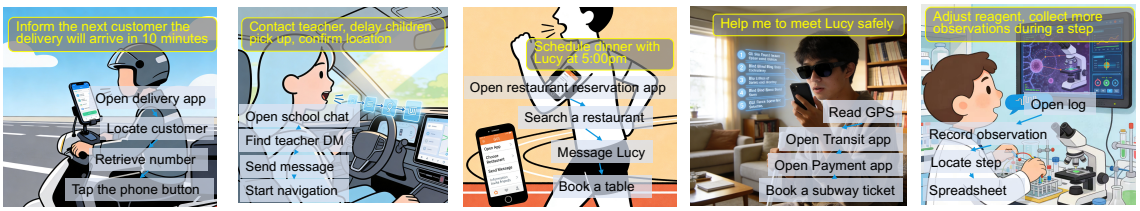
GUI agents [1, 9, 12, 15, 25–27, 41–43, 46, 50] provide a more general path to automation by operating *through the interface* the same way humans do. Given a natural-language instruction, a GUI agent observes the current screen, decides the next step, and executes touch actions such as tapping, typing, and scrolling. This paradigm avoids privileged APIs and can generalize across system apps and third-party apps. With recent advances in vision-language models (VLMs) [7, 10, 37, 38], GUI agents have become increasingly capable at interpreting mobile screenshots and following a *UI parse-action plan-action execution* loop for task execution [26, 27, 41, 50].

However, even strong VLM-based GUI agents often fail on hard tasks, such as permission pop-ups, copy-paste, form filling, and multi-app handoffs. More critically, these errors *compound* in multi-step or cross-app workflows: a single mis-click can lead the agent to an unintended page, from which it continues with an incorrect understanding of the UI, spiraling into long failure chains. This propagation of errors is a major barrier to deploying GUI agents as reliable assistants, particularly in high-stakes or accessibility-critical scenarios.

Despite substantial advances in GUI grounding, deployment failures remain common in state-of-the-art VLM-driven GUI agents [26]. Our analysis shows that these failures arise from the interaction between perception and execution, where residual perception errors propagate through long-horizon mobile workflows. In real apps, small front-end errors easily cascade in *service-composed* worklfows (*i.e.*, multi-step, cross-app, mm-scale), turning minor slips into long failure chains.

• **Fragile UI anchors: ambiguous perception in dense mobile UIs.** Mobile UIs are dense and gesture-driven, where actionable targets are often mm-scale, visually similar, and text-sparse (*e.g.*, icon-only toolbars). Although recent GUI grounding models have substantially improved perception, existing single-stage parsers [9, 19, 26, 27, 41, 50, 51] still encounter two challenges: *coverage failures*, where small or transient elements are missed, and *semantic failures*, where detected elements remain ambiguous without sufficient context. Such errors may propagate downstream and lead to mis-grounded actions.

• **Limited runtime execution verification: open-loop actions without per-step state verification.** Many agents [9, 12, 19, 51] act in an open-loop manner, with limited checks that an action is valid or succeeded. Yet mobile apps hinge on transient or partially non-visual state changes (*e.g.*, toasts, short loaders, keyboard shifts, activity transitions, login/session changes). Without per-action verification, agents act on stale states, fail silently, and drift from the true UI, leading to brittle recovery and long failure chains.



(a) Cycling (b) Driving (c) In motion (d) Inaccessible (blind) (e) Science agent

Fig. 1. Illustration of **hands-busy** and **attention-limited** scenarios where automated GUI agents are especially valuable: manually performing dense, **multi-step**, **cross-app**, **mm-scale** interactions are inconvenient, unsafe, or inaccessible.

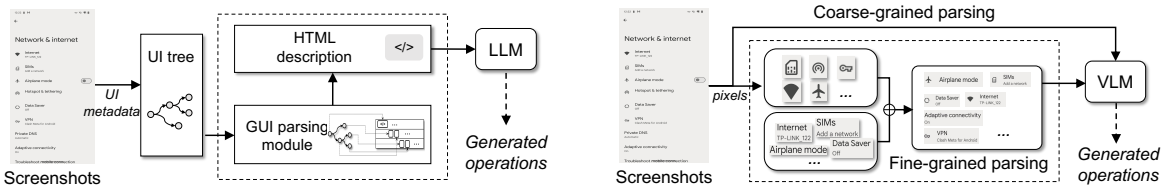
To address these, we present UIAnchor, a mobile GUI agent that improves reliability by *jointly anchoring both What to act on and Whether actions succeed*. We retain the standard parse-plan-act paradigm while strengthening UI grounding and redesign the runtime per-action coordination to mitigate the failure modes above.

• **First, two-stage UI parsing for high coverage and context-aware semantics.** We design a two-stage parser to *anchor* GUI automation with both high coverage and context-grounded semantics, instead of forcing a single model to "detect+understand" in one shot. In the *capture* stage, we use segmentation-style extraction to maximize recall, preserving mm-scale, low-contrast, and icon-only candidates with optional OCR. In the *interpretation* stage, we infer each candidate's functional intent from *joint* evidence (appearance, OCR, and surrounding UI context), resolving icon ambiguity across pages. We output a structured UI state with confidence scores, enabling robust grounding and allowing downstream agents to rank, abstain, or re-ground before errors cascade.

• **Second, meta-controller for per-action state perception and targeted recovery.** We design a lightweight meta-controller that anchors each action with state evidence and prevents error cascades in long mobile workflows. It performs *pre-action verification* to ensure the target is confidently groundable, and *post-action outcome perception* to verify success and detect state transitions. A runtime *meta-state memory* tracks transient signals (*e.g.*, toasts, activity transitions, overlay/keyboard shifts), while state differencing distinguishes real progress from silent failure. Based on this, the meta-controller diagnoses failures and applies targeted recovery (*e.g.*, re-ground, retry, or navigation correction) instead of re-planning from scratch.

We implement UIAnchor as a *modular* multi-agent system with four components: a UI Parser, Planner, Executor, and Meta-Controller. We build the *UI Parser* as a lightweight pipeline for high-recall, structured UI extraction. For the remaining agents (Planner/Executor/Meta-Controller), we adopt a shared LVLM backbone over screenshots/UI states and specialize each role via *Multi-LoRA*, without maintaining multiple full models.

We prototype UIAnchor on smartphones, tablets, and in-vehicle systems, and evaluate it on AndroidWorld [34], MobileWorld [23], ScreenSpot-v2 [44], MMBench-GUI [40], and our meta-state-aware MobileTask episodes. We stratify GUI tasks into five levels (L1–L5) by step length, cross-app scope, and UI granularity (Sec. 7.2.1). L5 approaches the practical ceiling of current frontier V/LLMs, while L4 captures the hardest attainable real-world workflows, including permission pop-ups, copy-paste, form filling, and multi-app handoffs. On L4 tasks, UIAnchor improves accuracy by 20% over *GPT-4o* and 18% over Mobile-Agent-v3. The gain comes from a self-verifying *meta-controller* that performs pre-action checks, post-action outcome perception, and targeted recovery. UIAnchor also explicitly tracks transient meta-states, such as permission dialogs, toasts, activity transitions, and login/session changes, enabling more reliable execution and recovery. Despite these gains, UIAnchor remains compact: it uses a 4B backbone with 83M LoRA parameters and runs at ~2 s/step, comparable to human operation. It reduces per-step latency by up to 75.5% and energy by 52.4% over baselines, while generalizing to unseen apps and evolving cross-platform GUIs. On resource-limited servers such as Jetson-class devices, where model size dominates



(a) LLM-based (b) VLM-based

Fig. 2. Comparison of LLM- and VLM-based GUI agents.

latency, our 4B model avoids VRAM/bandwidth bottlenecks and cuts end-to-end latency by 65% compared with 7B/8B GUI agents. Our main contributions are summarized as follows.

- We revisit two complementary sources of unreliability in deployable mobile GUI agents *i.e.*, incomplete/ambiguous UI perception and the lack of per-action closed-loop coordination, and show how they lead to compounding execution errors.
- We integrate structured UI grounding with runtime execution verification for reliable mobile GUI execution. A two-stage parser produces confidence-aware UI states, and a lightweight meta-controller enables closed-loop execution through per-action verification and targeted recovery.
- Extensive experiments show that UIAnchor achieves strong accuracy on the hardest practical tier L4 (cross-app, mm-scale, multi-step tasks) with only a 4B backbone, while running at human-like speed.

2 PRELIMINARIES

2.1 Mobile Task Automation via GUI Agents

Mobile task automation enables devices to autonomously execute multi-step workflows on behalf of users, which is particularly valuable when manual interaction is inconvenient, unsafe, or inaccessible (*e.g.*, while driving, during emergencies, or for users with motor or visual impairments). Among various technical approaches, *GUI-based* automation [27, 32, 42, 43, 50] prevails because it can operate across diverse apps and versions without relying on developer-provided APIs, making it broadly compatible with rapidly evolving mobile ecosystems.

A *GUI agent* aims to translate a high-level natural-language instruction (*e.g.*, "text Alice I'm running late") into a sequence of low-level UI interactions (*e.g.*, `tap`, `swipe`, and `type`) that manipulate on-screen elements to complete the task. This process is commonly modeled as a *sequential decision-making* loop. At each step t, the agent observes the current screen state S_t and produces an executable action A_t, which transitions the device to a new state S_{t+1}. The loop repeats until the task terminates (successfully or not):

$$\langle S_t, I \rangle \xrightarrow{\text{Agent}} A_t \xrightarrow{\text{Execute}} S_{t+1}. \tag{1}$$

Most GUI agents implement this loop with three conceptual stages.

(1) **UI Parsing:** Extract actionable UI elements and their attributes from the current screen (*e.g.*, element type, text, icon semantics, and location).
(2) **Action Planning:** Decide the next action or interaction needed to advance the instruction under the current state.
(3) **Action Execution:** Emit a low-level operation (*e.g.*, `click(x,y)`, `swipe`, `input`) and apply it to the device to obtain the next state.

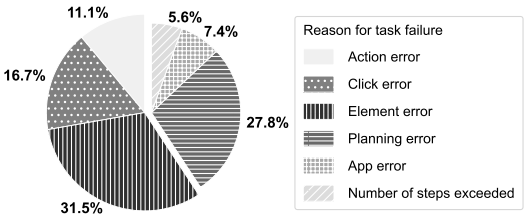


Fig. 3. Distribution of failure reasons for the compact model Qwen3-VL-4B [37] and GUI-Owl-7B [50] on the AndroidWorld [34] and self-collected datasets.

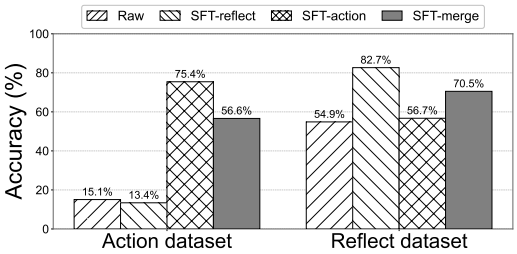


Fig. 4. Accuracy of specialized vs. merged fine-tuning. SFT-reflect/action/merge are fine-tuned on reflection-only, action-only, and mixed data.

2.2 Limitations of Current GUI Agents

Existing GUI agents fall into two architectural paradigms based on *what* they use for *screen representation* (Fig. 2).

- **LLM-based (UI-tree-based) agents.** These methods [26, 42, 43] rely on structured UI metadata (typically an accessibility hierarchy) as the main observation. They employ a GUI parsing module to map the UI tree, based on predefined rules, into a textual form (e.g., HTML-like descriptions) and use an LLM to reason about the instruction, select a target element, and generate the next action. Their effectiveness depends on the availability and fidelity of UI metadata.
- **VLM-based (screenshot-based) agents.** These methods [27, 32, 50] treat a rendered screenshot as the main observation and use a vision-language model to infer the next interaction directly from pixels. They avoid reliance on UI trees but require fine-grained visual grounding on small, dense mobile screens.

To understand where representative compact open-source existing GUI agents fail in real mobile interactions, we analyze error categories for the open-source model Qwen3-VL-4B [37] and GUI-Owl-7B [50] on AndroidWorld [34] (1.1k traces, 20 apps), a dynamic benchmark for GUI agents, and our self-collected dataset (2k traces, 32 apps). Fig. 3 shows that failures fall into two patterns.

- **For representative compact GUI agents, fine-grained UI perception remains the dominant bottleneck.** A major failure source comes from `element` (31.5%), `click` (16.7%), and `action` (11.1%), which counts for 59.3%. This indicates that, for representative compact open-source GUI agents, a major barrier is not high-level intent understanding, but reliably identifying *what* to interact with and *where* to interact on the screen. While static GUI grounding benchmarks [44] are approaching saturation, long-horizon mobile interactions still require robust perception under changing UI states and resource constraints. This motivates designs that explicitly strengthen *UI parsing* by improving coverage of actionable elements and disambiguating icon semantics in context.
- **Cross-step coordination is fragile, leading to drift, stalls, and compounding errors.** A second cluster of failures reflects *multi-step control issues*: `planning` (27.8%), `App` (7.4%) and `number of steps exceeded` (5.6%), which counts for 40.7%. This implies that even if individual actions are plausible, the agent can drift from the intended trajectory, miss transient state changes, or get stuck in loops without timely verification and recovery. This motivates designs that improve *cross-step coordination*, *i.e.*, deciding when to trust the current state, when to re-assess, and how to detect and correct execution anomalies before errors accumulate.

As a separate note on **modularity**, most prior systems implement UI parsing, planning, and execution by repeatedly prompting a single LLM/VLM [27, 50], which couples heterogeneous functions and can amplify both perception and coordination errors. Fig. 4 shows this effect under a *cross-domain task* setting, where we compare

Table 1. Comparison with representative prior GUI-agent systems.

Method	Architecture	Modularity	Structured parsing	Verification	Reflection/Recovery	Runtime memory	Meta-controller
AutoDroid [42]	LLM-based	✓	✓(A11y tree)	✓	✗	✓	✗
MobileGPT [26]	LLM-based	✓	✓(A11y tree)	✗	✓	✓	✗
V-Droid [12]	LLM-based	✓	✓(A11y tree)	✓	✓	✓	✗
OmniParser [31]	VLM-based	✓	✓(Screenshot)	✗	✗	✗	✗
UGround [15]	VLM-based	✓	✓(Screenshot)	✗	✗	✗	✗
MobileUse [27]	VLM-based	✓	✗	✓	✓	✓	✗
Mobile-Agent-v3 [50]	VLM-based	✓	✗	✓	✓	✓	✗
UIAnchor	**VLM-based**	✓	**✓(Screenshot)**	✓	✓	✓	✓

a unified base model against specialized, functionally decomposed agents. The results show that decoupling perception (action tasks) from reasoning (reflection tasks) into separate modules yields substantial accuracy gains. Specifically, SFT-action attains 75.42% on action tasks, and SFT-reflect reaches 82.65% on reflection tasks. In contrast, a merged model (SFT-merge) shows balanced but lower peak performance (70.52% on reflection; 56.63% on action), illustrating a *specialization-generalization trade-off*. It suggests that implementing the workflow as *multiple agents* can improve task success. We note that this analysis is based on representative compact open-source GUI agents and is intended to motivate design choices for the compact mobile-agent regime; stronger larger-scale systems may exhibit different error distributions. More broadly, this modularity is not meant to oppose the long-term trend toward end-to-end agents. While stronger foundation agents may internalize some routine verification and correction behaviors, modularity may still remain useful for handling complex failures. In the long term, manually designed modular structures may evolve into more advanced and automated forms rather than disappear entirely.

3 SOLUTION OVERVIEW

To address the dominant grounding and cross-step coordination failures identified in Sec. 2.2, we propose UIAnchor, a *multi-agent* GUI system that extends the VLM-based GUI agent paradigm by strengthening **UI parsing** and **per-action runtime coordination** within the standard **parse-plan-act** loop.

3.1 Architecture

UIAnchor follows the VLM-based (screenshot-based) paradigm [9, 19, 27, 32, 50]. This is because it operates directly on the *rendered screen*, rather than relying on an accessibility UI tree, so it can function when UI metadata is missing or unreliable (*e.g.*, dynamic layouts, or incomplete accessibility annotations). Conceptually, UIAnchor follows the conventional parse-plan-act pipeline [27, 32, 50], but makes two functions *explicit and robust*: *(i)* robust UI parsing that constructs a UI state for precise grounding, and *(ii)* cross-step coordination that verifies state transitions and triggers targeted recovery around every action.

In particular, UIAnchor consists of three functional layers.

- **Two-Stage UI Parsing Agent (Sec. 4).** Given the current screenshot I_t, the UI parsing agent produces a structured UI state $\mathcal{U}_t = \{e_k\}_{k=1}^{m}$, where each element $e_k = (b_k, t_k, \hat{y}_k, s_k)$ contains a bounding box, optional OCR text, an inferred function label, and a confidence score. Unlike existing approaches that directly act from pixels [9, 19, 27, 50], UIAnchor separates *component localization* from *component-level semantic inference* stage, improving coverage of small actionable elements and reducing confusions among visually similar icons. The contribution lies not in the two-stage paradigm itself (also used in parsers such as OmniParser [31]), but in our parser design and its integration with anchored execution.
- **Planner and Executor (we follow standard components).** On top of $\mathcal{U}_t$, a *planner* decomposes the user instruction into a sequence of sub-tasks and determines the appropriate next sub-task to address, and an *executor* grounds that the current sub-task into an atomic UI action (`click`, `swipe`, `input`) on the device,

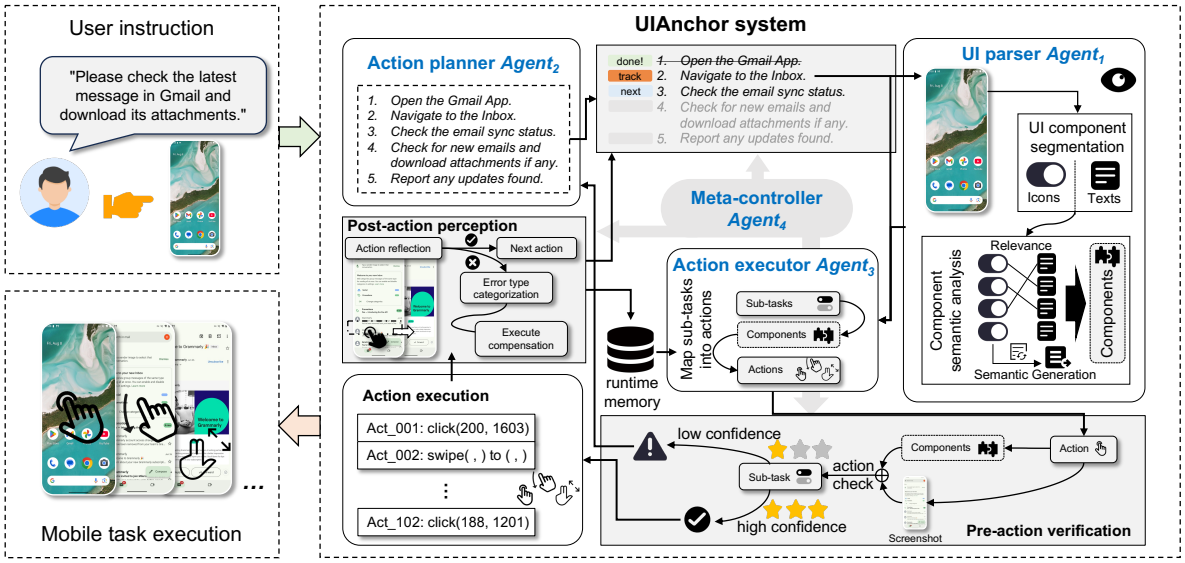


Fig. 5. Overview of UIAnchor workflow.

based on the actionable elements identified in $\mathcal{U}_t$. These two modules implement the standard GUI-agent backbone. The focus of UIAnchor is to make their *inputs* (UI state) and *handoffs* (coordination) reliable.

- **Meta-Controller for Per-Action State Perception (Sec. 5).** To prevent drift and compounding errors in multi-step execution, UIAnchor introduces a *meta-controller* that performs *per-action state perception*. It maintains a contextual runtime memory (including device meta-state signals), verifies readiness and grounding confidence before executing an action, and infers outcomes and detects anomalies after execution.

As discussed in Sec. 2.2, these designs address two key failure modes: unreliable UI perception and unverified action execution. Tab. 1 shows that, while prior systems include some components individually, UIAnchor uniquely integrates **structured parsing**, **verification**, **recovery**, **runtime memory**, and **meta-control** into a unified framework for *service-composed* mobile workflows.

3.2 Workflow

At each step t, UIAnchor runs the following loop (Fig. 5).

(1) **Decompose user instruction and plan sub-task sequence.** The planner decomposes the overall user instruction into a linear sequence of sub-tasks.
(2) **Determine the current sub-task.** The meta-controller tracks execution progress and determines which sub-task should be tackled next.
(3) **Parse the current UI on-demand.** The meta-controller decides whether to invoke the UI parsing agent to obtain an updated $\mathcal{U}_t$ from I_t or to reuse a cached UI state when the screen is stable.
(4) **Pre-action verification and act only when confident.** Before executing, the meta-controller checks whether the planned behavior can be grounded to the current interface with sufficient confidence, using $\mathcal{U}_t$ and runtime context. If confidence is low, it intervenes (*e.g.*, re-parse, re-decide, or re-plan) instead of executing a risky action.
(5) **Execute the grounded action.** The executor performs the atomic UI action on-device.

(6) **Post-action perception and targeted recovery.** After execution, the meta-controller updates runtime memory and infers whether the intended state transition occurred. If anomalies are detected (no-op, unexpected pop-up, wrong navigation, stalled progress), it triggers targeted recovery. Only when recovery fails does it escalate to re-planning.

We implement parsing, planning, execution, and meta-control as multiple lightweight agents rather than a single VLM-based agent. This enables role specialization and allows the meta-controller to invoke expensive components only when needed. We detail this multi-agent implementation in Sec. 6.

4 TWO-STAGE UI PARSING AGENT

The UI parsing agent is the perception front-end of UIAnchor. Given a raw mobile screenshot I, it outputs a UI state $\mathcal{U} = \{e_k\}_{k=1}^{m}$, where each element $e_k = (b_k, t_k, \hat{y}_k, s_k)$ contains a bounding box b_k, optional OCR text t_k, an inferred function label $\hat{y}_k$ (*e.g.*, `search`, `back`, `share`), and a confidence score s_k. As discussed in Sec. 2.2, over half of the failures arise from perception errors, *i.e.*, either **mislocalizing** the target element or **misinterpreting** an icon. Accordingly, this agent aims to explicitly strengthen both element localization and function disambiguation.

Challenges. Reliable UI parsing requires **robust element localization** under heterogeneous layouts and icon styles, and **context-dependent functional inference** for visually ambiguous icons. This challenge arises because identical icons (*e.g.*, a "heart" or "star") can represent different functions in different apps, and app-specific layouts, color schemes, and semantics vary widely across platforms. For example, a user tapping a button expecting a “favorite” action may instead "save for later," "bookmark and share," or even "add to cart" depending on the app’s context. As a result, UI parser must go beyond visual recognition and reason about icon meaning in context. Existing studies often satisfy only one side, leading to systematic failures.

- *Detector-based parsers* (*e.g.*, YOLO+OCR such as OmniParser [31]) rely on learned region proposals and can fail under icon-style or app-specific domain shifts.
- *Single-VLM-based parsers* (*e.g.*, CogAgent [19], SeeClick [9], UI-TARS [32], MobileUse [27]) map screenshots directly to actions or coordinates. Their parsing is implicit, which can overlook small UI components and entangle localization with semantic reasoning in a single model prediction.
- *LLM-based parsers* (*e.g.*, AutoDroid [42], MobileGPT [26], V-Droid [12], AutoGLM [28]) rely on HTML/XML abstractions. They discard key visual cues (shape, style, relative layout) and thus struggle with stylized or obfuscated icons whose intent is missing or under-specified.

Our Methods. To address these issues, we implement UI parsing as a **two-stage** pipeline: *(i)* UI component capture (Sec. 4.1) aims at localization of icons and text blocks; and *(ii)* component-level semantic analysis (Sec. 4.2) assigns a function label to each localized component. This separation makes the failure points explicit, avoiding a single VLM [9, 19, 32] that must solve both problems implicitly.

4.1 Segmentation-Based UI Component Capture

To maximize coverage across heterogeneous UIs, we adopt a *segmentation-based* decomposition instead of detector-based region proposals [31]. Compared with detector-based region proposals[31], this choice lets us start from physical UI boundaries and avoid additional icon-detector retraining. The key observation is that many actionable UI elements exhibit clear visual boundaries (*e.g.*, icon contours, button outlines), which makes segmentation a stable primitive for zero-shot generalization when UI styles shift.

Concretely, given screenshot I, we apply training-free SAM2 [33] to generate a set of mask proposals $\{M_i\}$, which are converted into icon candidates with bounding boxes $\{b_i^{icon}\}$ after filtering tiny masks and suppressing near-duplicates (see Figure 6). In parallel, we run OCR to extract text spans and text boxes $\{(t_j, b_j^{txt})\}$. The union of icon and text candidates forms a high-recall UI inventory that is passed to the semantic stage.

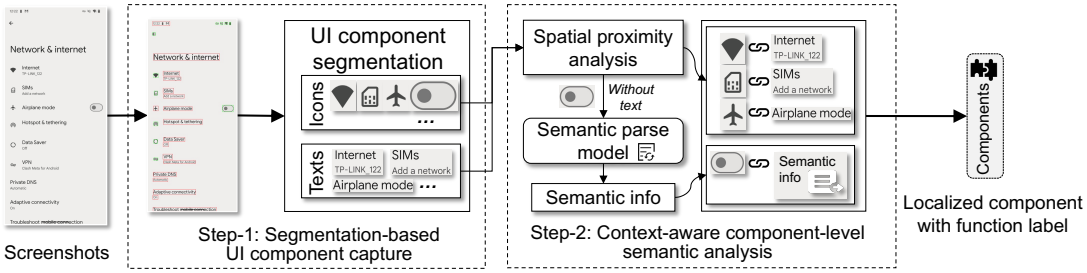


Fig. 6. Illustration of the two-stage UI parser agent pipeline.

Table 2. VLM vs Detector vs UIAnchor: element detection recall on screenspot-V2 datasets.

Methods	Mobile app		Desktop		Web		Average
	Text	Icon	Text	Icon	Text	Icon	
Qwen2.5VL-72B [7]	25.52%	9.95%	32.47%	23.57%	4.27%	4.43%	16.50%
OmniParser-v2 [31]	93.45%	85.31%	91.75%	92.14%	83.76%	80.79%	87.90%
Ours	**98.97%**	**91.47%**	96.39%	85.71%	86.32%	80.30%	**90.57%**

Tab. 2 shows that our design achieves strong zero-shot generalization on ScreenSpot-V2 [44], a benchmark for visual element grounding in mobile app interfaces. For example, on mobile apps, UIAnchor reaches 98.97% text recall and 91.47% icon recall without any dataset-specific tuning, outperforming the detector-based OmniParser-v2 [31] by 5.52% (text) and 6.16% (icon). A general-purpose VLM (Qwen2.5VL-72B) attains only 25.52% text recall and 9.95% icon recall, indicating that model scale alone does not guarantee accurate perceptual grounding for mobile GUIs. These results validate that our method mitigates a dominant failure mode of single-VLM GUI agents [9, 19, 32], where small actionable elements are missed due to implicit parsing.

4.2 Context-Aware Component-Level Semantic Analysis

After reliable component localization, we resolve *functional ambiguity* by combining textual cues with learned visual semantics. We first associate each icon candidate b_i^{icon} with a nearby text box using a distance-based heuristic. The heuristic operates as follows: for each icon candidate, we identify the text block with the smallest pairwise distance, as measured by a minimum separation metric d, defined as the minimum Euclidean distance between the boundaries of the two bounding boxes:

$$d(b_i^{\text{icon}}, b_j^{\text{txt}}) = \min_{\substack{p \in \partial(b_i^{\text{icon}}) \\ q \in \partial(b_j^{\text{txt}})}} \|p - q\|_2 \tag{2}$$

where $\partial(\cdot)$ denotes the set of points on the boundary of a bounding box. Let $j^*(i) = \arg\min_j d(b_i^{icon}, b_j^{txt})$ be the nearest text block. If $d(b_i^{icon}, b_{j^*(i)}^{txt}) < \tau$, where τ is a threshold set to 60 pixels based on empirical UI design conventions, we treat $t_{j^*(i)}$ as contextual metadata for the icon. We then infer the function label via two complementary ways.

- *Text-guided inference.* When the nearby text is informative (*e.g.*, "Favorites", "Share"), we directly infer the semantic meaning of the icon from its adjacent text label. This leverages a common UI design pattern where local text labels disambiguate visually similar icons.
- *Visual-context inference.* When no reliable textual cue exists, we invoke a fine-tuned Florence-2-base model [45] to predict actionable semantics from the icon crop together with its local layout context (*e.g.*, an

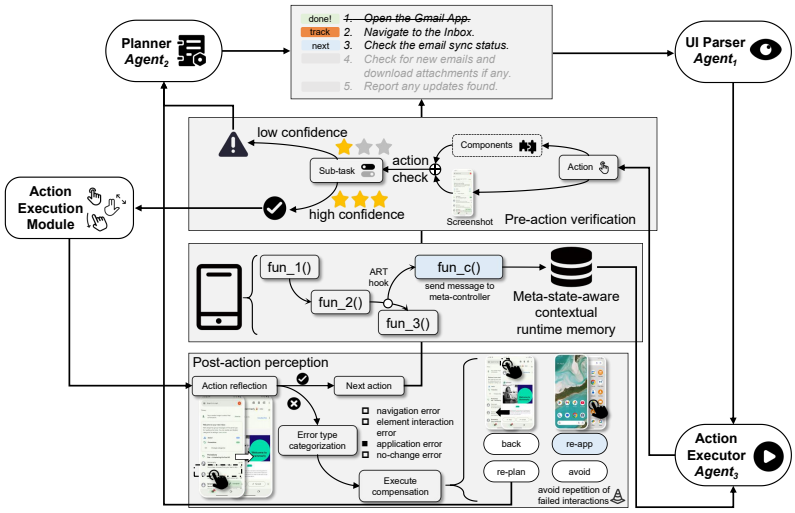


Fig. 7. Illustration of the mechanism of Meta-Controller.

expanded region around the icon). This yields a function label $\hat{y}_k$ with confidence s_k, complementing text-guided inference when UI labels are absent.

This dual strategy addresses two limitations in prior designs. *(i)* Single-VLM-based methods [9, 19, 32] mix localization and semantics, which can cause small elements to be ignored or misinterpreted. *(ii)* UI-tree-based methods [12, 26, 42] lack sufficient context for stylized icons when metadata is missing or under-specified.

By explicitly constructing a high-recall UI inventory and then inferring functions with local context, the UI parsing agent provides the planner agent with a reliable UI state $\mathcal{U}$ for downstream decision-making.

5 META-CONTROLLER FOR PER-ACTION STATE PERCEPTION

The meta-controller is the cross-agent coordinator of UIAnchor. While agents handle *what to do* (task planner), *what is on screen* (UI parser), and *how to do it* (action executor), the meta-controller ensures the system *acts on the right state* and *verifies what actually happened* at every step. Concretely, it performs *state perception (i)* before each action to decide whether the current screen is ready, and *(ii)* after each action to infer the outcome, detect anomalies, and trigger recovery when needed.

Challenges. Per-action state perception in mobile GUIs is difficult for two reasons.

- **Action-state is partially observable from screenshots.** Many critical cues for deciding whether an action is feasible or successful are *non-visual* (*e.g.*, foreground app, system time) or *transient* (*e.g.*, toasts, short-lived pop-ups), and are easily missed by fixed-interval screen sampling.
- **Closed-loop control must be selective to remain efficient.** A naive solution is to re-plan before every action and re-parse after every action. However, frequent planner invocations are expensive, and over-triggering repairs can break task flow. Thus the controller must gate actions only when confidence is low, and recover with targeted handlers rather than restarting the workflow.

Prior modular GUI automation frameworks, such as MobileGPT [26], treat error repair as a separate, user-invoked mode. This design interrupts task flow and fails to handle transient errors autonomously in a closed-loop manner. Other systems, like Mobile-Agent-v3 [50], attempt to mitigate the coordination problem by invoking the task planner before each action to re-determine the current sub-task. However, this design introduces significant

Fig. 8. Illustration of missed critical frames due to fixed-interval sampling in VLM-based GUI agents.

computational overhead due to frequent planner invocations, undermining system efficiency. Moreover, existing approaches predominantly adopt an open-loop architecture [9, 28, 42, 43, 51], which inherently lacks a mechanism to monitor execution validity in real time, adapt agent behavior based on runtime feedback, or autonomously trigger context-aware recovery. Our meta-controller addresses these limitations by providing a lightweight, closed-loop coordination mechanism with selective intervention.

Our Methods. We propose a lightweight, rule-augmented meta-controller which consists of *(i)* a contextual runtime memory (Sec. 5.1) that continuously integrates *device-level meta-state* and execution history, and *(ii)* a per-action state perception loop (Sec. 5.2) that performs *pre-action* verification, *post-action* outcome inference, and targeted *error handling*.

5.1 Meta-State-Aware Contextual Runtime Memory

We define the **meta-state** M_t as a real-time stream of contextual information beyond pixels, including persistent device state (*e.g.*, current foreground application, system time) and ephemeral interface signals (*e.g.*, toast notifications, transient pop-ups).

Access to M_t is crucial for reliable action-state perception because visual frames captured only before/after an operation can be identical even when the action succeeded (*e.g.*, a success toast appears briefly and disappears), and time-dependent tasks (*e.g.*, "next Thursday") require accurate device time. As illustrated in Fig. 8, VLM-based GUI agents typically capture the device state by sampling the frames at fixed intervals after actions. These intervals cannot be arbitrarily short as the UI often requires time to complete rendering and settle into a stable visual state. Such sampling often misses critical frames containing transient notifications (e.g., success toasts), which are vital for accurately assessing task progress. For example, sampled frames taken before (Frame-1) and after (Frame-4) adding a song to a playlist may appear identical, depriving the model of reliable cues to determine whether the addition succeeded.

5.1.1 Meta-State Monitor Implementation. To capture such ephemeral signals, we implement a **meta-state monitor** by leveraging the LSPosed framework [29] to hook key Android runtime (ART) methods in real-time. This allows us to intercept system-level events without modifying the target applications. Specifically, we hook:

Table 3. Structure of the Contextual Runtime Memory (H_c).

Field	Data Type	Description
Timestamp	`DateTime`	The precise system time when the action was executed. Provides temporal ordering for memory entries.
Action Type	`String (Enum)`	Type of the executed action. *e.g.*, `CLICK` (click at coordinates), `TYPE` (type text into input), `SWIPE` (swipe between coordinates).
Action Content	`Serializable Object`	Specific parameters of the executed action. *e.g.*, `CLICK`: {"coordinate": [x, y]}, `TYPE`: {"text": "example", "clear_text": true}.
Screenshot Paths	`List<String>`	File paths of screenshots capturing UI states: pre-action (state before execution) and post-action (state after execution).
Meta-state	`Context Object`	Real-time contextual information, including persistent device state (e.g., current foreground application, system time) and ephemeral interface signals (e.g., toast notifications, pop-up dialogs).
Execution Context	`Context Object`	Structured snapshot containing: current sub-task goal, recent action history with outcomes, encountered error states and recovery attempts.

- `Toast.makeText()` to capture transient notification text and duration.
- `View.performClick()` to detect UI click events and extract view metadata.
- Additional hooks for system intents, activity lifecycle events, and permission dialogs.

These hooks are registered during the application loading phase (`XC_LoadPackage`), ensuring comprehensive coverage of both system and application-level interactions. The intercepted meta-information (including event type, content, timestamp, and source application) is immediately serialized and streamed to the meta-controller agent via a lightweight IPC channel.

5.1.2 Contextual Runtime Memory. The collected meta-state feed is continuously integrated into a structured **contextual runtime memory** H_c, which maintains the controller's execution context. As detailed in Tab. 3, H_c is organized as a temporal sequence of memory entries, each containing:

- **Timestamp** and **Action Type/Content**: When and what action was executed.
- **Screenshot Paths**: Visual records before and after execution.
- **Meta-state**: Runtime device and interface context.
- **Execution Context**: Task progress and action history.

By unifying the real-time meta-state with execution history, H_c provides a coherent view of the evolving environment and supports consistent decision-making between agents. This design enables the system to reason about events that are visually imperceptible but semantically critical for task progress assessment.

5.2 Per-Action State Perception and Closed-Loop Control

We use **per-action state perception** to decide *when to act* and *how to react*. For each planned step, the controller runs a pre-action verification stage (Sec. 5.2.1) and a post-action perception stage (Sec. 5.2.2), and invokes a targeted handler (Sec. 5.2.3) when anomalies are detected.

5.2.1 Pre-Action Verification via Grounding Confidence. Before executing any planned action, the controller estimates whether the executor can ground the prescribed sub-task to the current interface with high confidence.

We compute a grounding confidence score $C_{\text{ground}} \in [0, 1]$ as a weighted combination of semantic alignment and contextual feasibility:

$$C_{\text{ground}} = \alpha \cdot S_{\text{sem}}(d_{\text{plan}}, d_{\text{UI}}) + (1 - \alpha) \cdot F_{\text{ctx}}(M_t), \tag{3}$$

where S_{sem} measures the semantic similarity between the planner's target description d_{plan} and the textual attributes d_{UI} of candidate UI components extracted by the UI parser (see Sec. 4). Specifically, we compute the cosine similarity between their corresponding embeddings generated by a lightweight pre-trained sentence transformer (*e.g.*, all-MiniLM-L6-v2 [39]). $F_{\text{ctx}}(M_t) \in [0, 1]$ encodes context feasibility derived from the real-time

meta-state M_t. It is a heuristic function that evaluates factors such as the visibility of target UI elements (via accessibility tree), the absence of blocking overlays (*e.g.*, permission dialogs detected via the meta-state monitor), and navigational readiness (*e.g.*, whether the correct application screen is foreground). The weighting factor α is empirically set to 0.6 to balance the contributions of semantic and contextual signals.

We empirically set thresholds $\tau_{\text{low}} = 0.3$ and $\tau_{\text{high}} = 0.7$ based on an ablation study detailed in Sec. 7. If $C_{\text{ground}} \geq \tau_{\text{high}}$, execution proceeds with high certainty. If $C_{\text{ground}} \leq \tau_{\text{low}}$, the controller triggers on-demand re-planning by supplying the current M_t and relevant excerpts from H_c to the planner. When the score falls within the intermediate range $(\tau_{\text{low}}, \tau_{\text{high}})$, the executor proceeds but logs the action for potential future verification. This two-threshold mechanism prevents the execution of low-certainty actions, which often propagate errors, while avoiding the computational overhead of re-planning at every step.

5.2.2 Post-Action Outcome Perception via State Differencing. After executing an action, the controller performs a differential analysis between the *pre-action* state S_{t-1} and the *post-action* state S_t, supplemented by meta-state transition $\Delta M = M_t - M_{t-1}$. It classifies the action outcome into one of four categories $O \in \{O_1, O_2, O_3, O_4\}$:

$$O = \begin{cases} O_1, & \text{action successful and sub-task completed,} \\ O_2, & \text{action successful but sub-task incomplete,} \\ O_3, & \text{erroneous action leading to an error page/application,} \\ O_4, & \text{no detectable interface change.} \end{cases}$$

This outcome classification is produced by the *action reflector*, a dedicated module that leverages a multi-modal model. The reflector implements the decision function f by constructing a comparative prompt that encapsulates the observed changes and contextual information, and then calls a VLM to obtain the outcome classification:

$$O = f\Big(\phi_{\text{visual}}(S_{t-1}, S_t),\ \phi_{\text{textual}}(S_{t-1}, S_t),\ \Delta M,\ H_c\Big), \tag{4}$$

where ϕ_{visual} captures pixel/structure changes, ϕ_{textual} compares semantic content (OCR text and UI labels), ΔM captures meta-state transitions, and H_c provides task-progress context. The controller updates H_c with the pre/post observations and the inferred O. If $O \in \{O_1, O_2\}$, execution continues (advance the sub-task for O_1, or continue within the sub-task for O_2). If $O \in \{O_3, O_4\}$, the controller invokes targeted recovery.

5.2.3 Targeted Error Handling. When $O \in \{O_3, O_4\}$, the controller refines diagnosis by inferring an error type:

$$E_{\text{type}} = \arg\max_{e \in E_{\text{set}}} P\big(e \mid O, \phi_{\text{visual}}, \phi_{\text{textual}}, \Delta M, H_c\big), \tag{5}$$

where E_{set} includes error types such as *navigation error*, *element interaction error*, *application error*, and *no-change error*. Based on E_{type} and context in H_c, the controller selects a recovery policy π_{rec} from a small set Π_{rec}:

$$\pi_{\text{rec}} = g(E_{\text{type}}, H_c) \in \Pi_{\text{rec}} = \{\pi_{\text{back}}, \pi_{\text{replan}}, \pi_{\text{reapp}}, \pi_{\text{avoid}}\},$$

where: π_{back} performs system back navigation; π_{redecide} re-selects the current action from candidates, marking the previous as erroneous; π_{reapp} re-launches or switches to the target application; π_{avoid} re-decides while explicitly avoiding repetition of failed interactions. In particular, for π_{redecide} and π_{avoid}, the executor re-evaluates the candidate set $A_{\text{candidate}}$ under updated constraints:

$$A_{\text{valid}} = \{a \in A_{\text{candidate}} \mid a \neq a_{\text{failed}},\ \text{feasibility}(a, S_t, H_c) > \theta\},$$

and selects the highest-confidence action from A_{valid}. All recovery steps are guided by H_c to maintain task continuity without restarting the workflow.

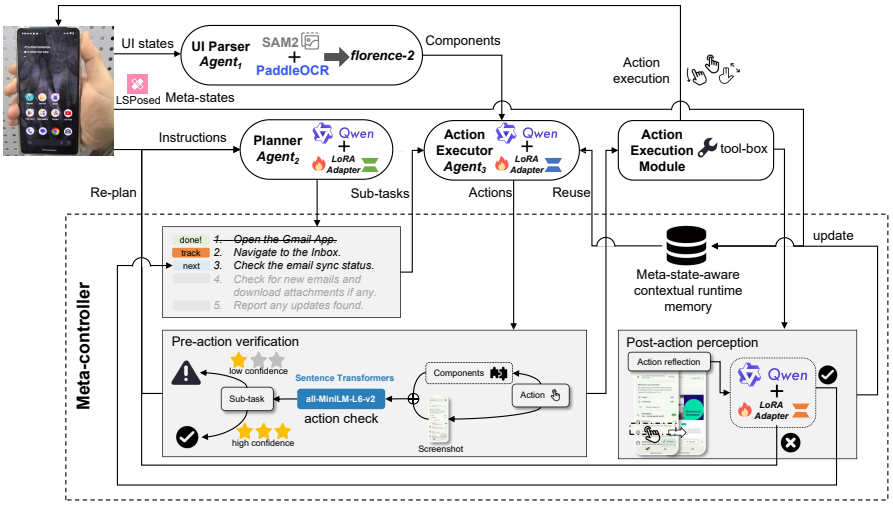


Fig. 9. Implementation of UIAnchor: system integration, data flow, and model deployment in the multi-agent framework.

6 MULTI-AGENT IMPLEMENTATION

This section describes the implementation of UIAnchor (see Fig. 9), including our Multi-LoRA agent instantiation (Sec. 6.1), the training pipeline for role-specific adapters (Sec. 6.2), and key deployment details for execution and evaluation (Sec. 6.4).

6.1 Multi-LoRA Specialized Agent Implementation

We implement UIAnchor as a modular multi-agent system with four components: a **UI Parser**, a **Planner**, an **Executor**, and a **Meta-Controller**. These agents follow two implementation styles that correspond to their roles. The **UI Parser** is implemented as a lightweight perception pipeline to maximize recall and structured UI extraction. The remaining three agents (**Planner/Executor/Meta-Controller**) require flexible reasoning over screenshots and UI states, and we implement them using a shared Large Vision-Language Model (LVLM) backbone with **Multi-LoRA** specialization for efficient deployment. Specifically, as shown in Fig. 10, the Planner, Executor, and Meta-Controller share a single frozen LVLM backbone (`Qwen3-VL-4B-Instruct` [37]), while each role is instantiated by activating a small, task-specific LoRA adapter [21]. At runtime, we switch roles by loading only the corresponding adapter, avoiding multiple full model instances. We inject low-rank updates into the transformer linear layers while keeping the visual encoder frozen, ensuring consistent visual grounding across agents. Unless otherwise specified, we use rank r=16 and scaling α=32.

- **UI Parser (perception pipeline).** Given a screenshot, the UI Parser segments UI elements using SAM2 [33], extracts on-screen text via PaddleOCR [11], and associates nearby icon-text pairs by spatial proximity. It then uses a fine-tuned Florence-2 model [45] to produce concise semantic labels (*e.g.*, "send button"), yielding a structured UI component list (bounding boxes, OCR text, types, and labels) for downstream grounding.
- **Planner (LoRA adapter on `Qwen3-VL-4B-Instruct` [37].** The Planner converts a user instruction into an ordered sub-task sequence.

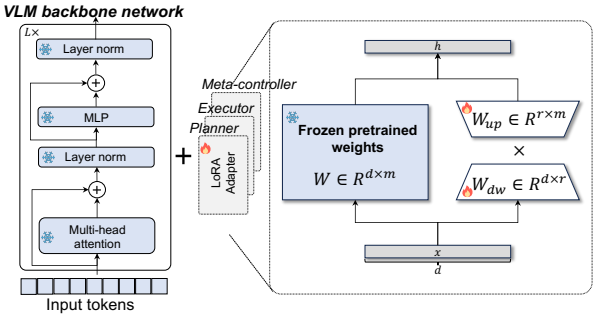


Fig. 10. Illustration of the specialized multi-agent (*i.e.*, Planner, Executor, Meta-controller), which is implemented by fine-tuning lightweight LoRA adapters on a shared VLM backbone.

- **Executor (LoRA adapter on `Qwen3-VL-4B-Instruct` [37]).** The Executor grounds each sub-task into a concrete UI action (*e.g.*, `click`, `input_text`, `scroll`) and its target component, conditioned on the parsed UI state and runtime memory.
- **Meta-Controller (hybrid verifier).** The Meta-Controller improves reliability through pre-action verification and post-action outcome perception. It uses a lightweight verifier (`all-MiniLM-L6-v2` [39]) for pre-action confidence estimation, and a LoRA-specialized `Qwen3-VL-4B-Instruct` [37] verifier for post-action perception.

6.2 Training Pipeline for Specialized Agents

We train each LoRA-based agent (Planner/Executor/Meta-Controller) using a three-stage pipeline inspired by prior mobile GUI agent training recipes (*e.g.*, MobileRL [48], Mobile-Agent-v3 [50], Mobile-R1 [16]). The UI parser, based on pre-trained models, remains fixed and is not fine-tuned throughout the training process.

- *Stage 1: Action-Only SFT.* We first fine-tune each agent's LoRA adapter on expert action trajectories (20k steps across 32 apps, see Section 6.3 for dataset details) without reasoning traces. This stage teaches basic mobile GUI interaction patterns and substantially reduces exploration cost for subsequent reinforcement learning.
- *Stage 2: Reasoning-Augmented SFT.* We then enrich the training corpus with intermediate reasoning steps generated by a strong instruction-following model through iterative refinement (see Section 6.3 for the construction process). Each agent is further fine-tuned on the enriched data with role emphasis: the Planner focuses on subgoal decomposition and structured plans, while the Executor focuses on action grounding and state transitions.
- *Stage 3: GRPO with Failure-Aware Curriculum.* Building on the SFT-initialized adapters, we perform online multi-turn reinforcement learning using Group Relative Policy Optimization (GRPO) [36] with curriculum learning. We implement the curriculum via *Failure Curriculum Filtering (FCF)* [8, 48, 50], which dynamically re-weights task sampling based on per-task failure statistics, *i.e.*, tasks that consistently yield zero reward are gradually down-weighted to avoid wasteful exploration and stabilize training.

6.3 Training and Evaluation Datasets

We split the dataset into disjoint training and evaluation sets, with no overlap in applications or instantiated task descriptions. The training set covers 32 applications, including the 20 AndroidWorld apps and additional

real-world mobile apps. Following AndroidWorld, we instantiate task descriptions by randomizing entities, input contents, and field values from shared templates.

Training Set Construction: We collect over 20k action trajectories from 200 tasks across 32 applications using GPT-4o and other state-of-the-art models. Human annotators review the collected samples to remove failed, ambiguous, or inconsistent trajectories and correct obvious annotation errors. The training data includes three components:

- *Action Trajectories:* The foundational data comprises raw, multi-step action sequences (*e.g.*, tap, type, swipe) paired with corresponding screen states and task instructions.
- *Plan Data:* To train the Planner agent, we employ GPT-4o to analyze the temporal and logical relevance within action sequences, summarizing them into structured, high-level subgoal plans. These generated plans undergo rigorous manual review to ensure correctness and coherence.
- *Reflect & Action Data:* To train the Executor and Meta-Controller, we first manually annotate the Reflect data based on the generated plans and current execution state. The action-level training data is then constructed by determining the appropriate atomic action for the current sub-task, guided by the outcome of the reflection process.

Testing Set: The testing set is specifically designed to assess the agent's capability in meta-status-aware scenarios. It comprises 51 tasks spanning 9 applications across 5 categories: *Writing, Science Education, Food, Office,* and *Health*. These 9 applications are held out from training and do not overlap with the 32 training applications. The task types are diverse and challenging, including: handling transient system *toasts*, performing multi-field *data entry*, making dynamic *state judgments*, completing *compositional tasks*, and operating within *complex UI layouts* that require advanced visual understanding. Although the evaluation tasks are designed to be similar in task type to the training set (e.g., *data entry*, *edit*, and *delete*), their concrete task descriptions are independently instantiated with different entity names and field values, and do not overlap with those used in training. Each evaluation task is manually reviewed and validated to ensure that the goal condition is executable and unambiguous on the target app version.

6.4 Prototype Deployment

We implement UIAnchor as a hybrid on-device and edge/cloud-assisted system. On-device inference runs via `llama.cpp` [14] with 8-bit quantization to meet latency and memory constraints, while mobile UI interactions are executed through a Python-based automation library that exposes action APIs and UI querying utilities.

- **Training and Serving.** We train all LoRA adapters using the `Swift` framework [52] and serve models with `vLLM` [24] for high-throughput inference with dynamic batching.
- **Devices and Infrastructure.** We test UIAnchor on physical devices (Google Pixel 7, Xiaomi 17 Ultra, Huawei Mate 70 Pro) and Android Virtual Device (AVD) emulators to cover diverse hardware/software configurations. We use an NVIDIA Jetson AGX Orin for edge assistance and an NVIDIA RTX 3090 server (with 2×3090 GPUs) for cloud assistance. Training is conducted on a server with 8×NVIDIA RTX 3090 GPUs and 4×NVIDIA L20 GPUs.

7 EVALUATION

7.1 Experiment Setup

Tasks and Datasets. We evaluate UIAnchor on four complementary benchmarks that cover dynamic interaction, long-horizon workflows, cross-platform grounding, and real-world mobile complexity:

- **AndroidWorld** [34] evaluates end-to-end interactive execution under dynamic UI states, with 116 tasks across 20 apps and millions of randomized task variants.

- **MobileWorld** [23] is a challenging mobile-use benchmark with 201 cross-app tasks across 20 applications, featuring agent-user interaction and MCP-augmented hybrid tool-use scenarios.
- **ScreenSpot-v2** [44] focuses on cross-platform GUI grounding, containing 1.2k re-annotated tasks spanning mobile, desktop, and web interfaces.
- **MMBench-GUI** [40] is a multimodal benchmark for evaluating AI agents' understanding and reasoning on graphical user interfaces (GUIs). It assesses both static screen comprehension and dynamic task execution across mobile, desktop, and web platforms.
- **Self-collected MobileTask Dataset** targets practical mobile edge cases, including complex multi-layer interfaces, modal dialogs, toast pop-ups, and dynamic UI changes, with 200 tasks across 30+ apps.

Baselines. To ensure a comprehensive and fair comparison, we select representative baselines that cover the mainstream approaches to screen understanding, action generation, and execution control.

- **Holistic VLM-driven GUI Agents** directly leverage VLM to interpret screenshots and generate UI actions in an end-to-end manner. They emphasize strong visual grounding and generalization.
  - **UI-TARS** [32]: a fine-tuned VLM designed for end-to-end GUI interaction.
  - **GUI-Owl** [50]: a foundational GUI agent model (7B/32B) built upon Qwen2.5-VL.
  - **MAI-UI** [53]: a family of foundation GUI agents (2B/8B/32B) built upon Qwen3-VL, with native support for agent-user interaction and MCP-style tool calls.
- **Step-wise GUI Agents** introduce structured intermediate representations or multi-step decision policies to improve execution stability and long-horizon reasoning. They explicitly model state transitions or verification but typically rely on predefined structures or UI representations.
  - **AutoDroid** [42]: a memory-augmented agent with step-wise policies and explicit state tracking.
  - **MobileUse** [27]: employs hierarchical reflection for autonomous mobile operation.
  - **V-Droid** [12]: a verifier-driven approach that parses XML-based UI representations.
  - **Mobile-Agent-v3** [50]: a multi-agent framework endowed with capabilities for knowledge evolution, task planning, sub-task execution, and reflective reasoning.

Metrics. We evaluate the GUI agent performance along complementary dimensions of effectiveness, efficiency, and on-device cost:

- **Success Rate (SR)** measures end-to-end task completion and serves as the primary effectiveness metric. Task completion follows the criteria of DroidTask and LlamaTouch, requiring correct action subsequences or state transitions.
- **Action Type Accuracy** evaluates the precision of high-level action category prediction by comparing the predicted action type (*e.g.*, 'click', 'swipe', 'input_text') against the ground-truth type for each step.
- **Action Accuracy** measures the exact match of both the action type and its associated parameters (e.g., coordinates, text content, direction) with the ground truth.
- **LLM Inference Latency** captures end-to-end response time, reflecting real-time usability in on-device settings.
- **Token Consumption** measures computational cost and scalability by counting input and output tokens.

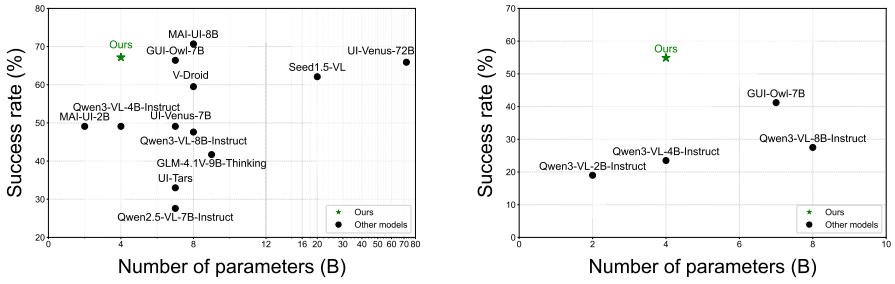
7.2 Main Result

7.2.1 Performance Across Complexity Levels (L1-L5). We evaluate UIAnchor and baselines under five progressive task-complexity levels (L1–L5), defined by three factors: **task length** (average execution steps), **apps** involved, and **UI perception granularity** (pixel size of the critical UI component at the failure step for unsuccessful tasks).
Level definitions.

- **L1**: < 10 steps, single app, large critical components ($> 100 \times 100$ pixel), clear semantics, no distractors.

Table 4. Performance comparison of UIAnchor and baselines across task complexity levels (L1-L5). The "Meta-state" denotes contextual signals including foreground app, system time, and LSPosed-hooked toast notifications and transient pop-ups. "-" denotes undisclosed (typically very large) model sizes.

	Methods	Size	Screenshot	Ally tree	Meta-state	L1	L2	L3	L4	L5
Models	Claude-Sonnet-4-20250514-thinking [3]	-	✓	✓	✗	45.8%	44.9%	13.0%	20.4%	0.0%
	GPT-4o-2024-11-20 [22]	-	✓	✓	✗	37.5%	46.9%	16.7%	20.4%	7.7%
	Qwen3-VL-4B-Instruct [37]	4B	✓	✗	✗	66.7%	59.2%	31.5%	5.6%	0.0%
	Qwen3-VL-2B-Instruct [37]	2B	✓	✗	✗	62.5%	50.0%	12.3%	3.7%	0.0%
Agents	Mobile-Agent-v3 (GUI-Owl-7B) [50]	7B	✓	✗	✗	75.0%	87.8%	49.6%	35.2%	0.0%
	UIAnchor (Qwen3-VL-4B-Instruct-grpo)	**4B**	✓	✗	✓	**87.5%**	**87.8%**	**52.8%**	**51.9%**	3.8%



(a) AndroidWorld testing (b) Self-collected MobileTask testing

Fig. 11. Comparison of UIAnchor and baselines in terms of success rate-parameter size trade-off.

- **L2**: < 10 steps, single app, large components ($> 100 \times 100$ pixel), with distractors.
- **L3**: $10 - 20$ steps, single app, small components ($< 100 \times 100$ pixel), with distractors.
- **L4**: $20 - 30$ steps, multi-app, small components ($< 100 \times 100$ pixel), with distractors.
- **L5**: ≥ 30 steps, multi-app, small components ($< 100 \times 100$ pixel), with distractors.

On a typical smartphone screen, 100 pixel corresponds to roughly **mm-level** in physical size. Both AndroidWorld (L1–L5: 20, 42, 31, 14, 9; total 116) and our MobileTask testing dataset (L1–L5: 4, 7, 23, 13, 4; total 51) are partitioned accordingly, and aggregated success rates are reported in Tab. 4. *First*, UIAnchor is robust across practical levels. UIAnchor achieves the best results in four of five levels: 87.5% (L1), 87.8% (L2, tied with Mobile-Agent-v3), 52.8% (L3), and 51.9% (L4). The gains at L3/L4 highlight the effect of our **UI parser** for small/occluded targets and the **meta-controller** for long-horizon planning and cross-app transitions. *Second*, standalone lightweight VLMs fail to scale with horizon and app transitions. Qwen3-VL-4B-Instruct remains competitive on L1-L3, but drops sharply on L4 (5.6%) and L5 (0.0%), indicating that fine-grained UI grounding and long-horizon state management require dedicated agent modules beyond a single VLM. *Third*, extreme long-horizon tasks remain hard for current LLMs. On L5 (≥30 steps), UIAnchor reaches 3.8% and even GPT-4o only 7.7%, while other lightweight VLMs drop to 0%. Despite this, UIAnchor outperforms GPT-4o on L1/L3/L4 and surpasses Mobile-Agent-v3 (GUI-Owl-7B) by +16.7% on L4. **Summary**. L5 remains a hard bottleneck for today's V/LLM modes and agents. On L4, the hardest attainable tier aligned with users' service-composed usage trends, UIAnchor improves success by **31.5%** over GPT-4o and **16.6%** over Mobile-Agent-v3, showing that a meta-controller plus a fine-grained UI parser scales better than brute-force scaling.

7.2.2 Performance Comparison. We evaluate UIAnchor, built upon the *Qwen3-VL-4B-Instruct* backbone, on two benchmarks: the standard *AndroidWorld*, the challenging *MobileWorld* [23], and our *self-collected MobileTask testing*

Table 5. Success rates (%) on AndroidWorld (mobile GUI tasks). The "Meta-state" denotes contextual signals including foreground app, system time, and LSPosed-hooked toast notifications and transient pop-ups. "-" denotes undisclosed (typically very large) model sizes. Notably, AndroidWorld contains many challenging tasks (*e.g.*, multi-step data entry, pop-ups/detours), where even careful human execution achieves only ~80% success under the benchmark protocol. AndroidWorld spans **L1-L5** complexity levels.

	Methods	Size	Screenshot	Ally tree	Meta-state	Mobile-deployable	Success rates (%)
	Human	-	✓	✗	✓	-	80.0
Models	GPT-4o-2024-11-20 [22]	-	✓	✗	✗	✗	34.5
	Claude-Sonnet-4-20250514-thinking [3]	-	✓	✗	✗	✗	41.0
	UI-Tars-1.5 [32]	-	✓	✗	✗	✗	64.2
	UI-Venus-72B [17]	72B	✓	✗	✗	✗	65.9
	Seed1.5-VL [18]	20B	✓	✓	✗	✗	62.1
	GLM-4.1V-9B-Thinking [20]	9B	✓	✓	✗	✗	41.7
	Qwen3-VL-8B-Instruct [37]	8B	✓	✗	✗	✗	47.6
	MAI-UI [53]	8B	✓	✗	✗	✗	**70.7**
	GUI-Owl-7B [50]	7B	✓	✗	✗	✓	66.4
	UI-Venus-7B [17]	7B	✓	✗	✗	✓	49.1
	UI-Tars-7B [32]	7B	✓	✗	✗	✓	33.0
	Qwen2.5-VL-7B-Instruct [7]	7B	✓	✗	✗	✓	27.6
	Qwen3-VL-4B-Instruct [37]	4B	✓	✗	✗	✓	45.3
	MAI-UI [53]	2B	✓	✗	✗	✓	49.1
	Qwen3-VL-2B-Instruct [37]	2B	✓	✗	✗	✓	36.4
Agents	Agent S2 (Agent S2) [1]	-	✓	✗	✗	✗	54.3
	GUI-Explorer (GPT-4o) [46]	-	✓	✓	✗	✗	47.4
	AndroidGen (GPT-4o) [25]	-	✗	✓	✗	✗	46.8
	UGround (GPT-4o) [15]	-	✓	✗	✗	✗	32.8
	AndroidWorld (GPT-4o Turbo) [34]	-	✗	✓	✗	✗	30.6
	AutoDroid (GPT-4o) [42]	-	✗	✓	✗	✗	15.7
	MobileUse (Qwen2.5-VL-72B) [27]	72B	✓	✗	✗	✗	62.9
	V-Droid (V-Droid) [12]	8B	✗	✓	✗	✓	59.5
	UIAnchor **(Qwen3-VL-4B-Instruct-grpo)**	**4B**	✓	✗	✓	✓	**67.2**

Table 6. Success rates (%) on MobileWorld GUI-only. The "Meta-state" denotes contextual signals including foreground app, system time, and LSPosed-hooked toast notifications and transient pop-ups.

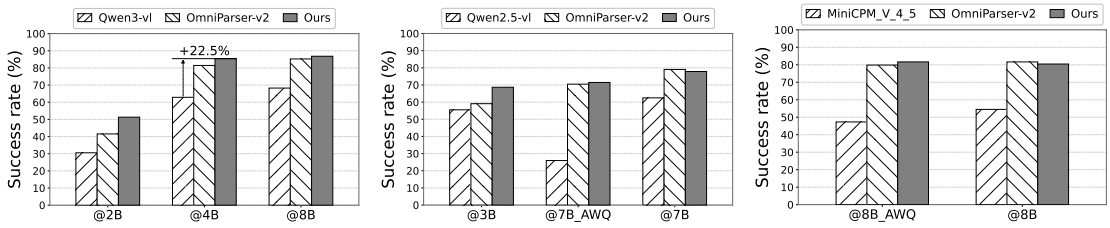
Methods	Parameter size	Screenshot	Ally tree	Meta-state	Mobile-deployable	GUI-only (%)
GUI-Owl-7B [50]	7B	✓	✗	✗	✓	7.7
GUI-Owl-32B [50]	32B	✓	✗	✗	✗	8.5
UI-Venus-7B [17]	7B	✓	✗	✗	✓	8.5
UI-Venus-72B [17]	72B	✓	✗	✗	✗	16.4
Qwen3-VL-4B-Instruct [37]	4B	✓	✗	✗	✓	7.7
Qwen3-VL-8B-Instruct [37]	8B	✓	✗	✗	✗	9.4
Qwen3-VL-32B-Instruct [37]	32B	✓	✗	✗	✗	11.9
MAI-UI-2B [53]	2B	✓	✗	✗	✓	10.3
MAI-UI-8B [53]	8B	✓	✗	✗	✗	27.5
MAI-UI-32B [53]	32B	✓	✗	✗	✗	**36.2**
GELab-Zero-4B [49]	4B	✓	✗	✗	✓	16.1
Qwen3-VL-4B-Instruct-grpo	4B	✓	✗	✗	✓	12.0
UIAnchor **(Qwen3-VL-4B-Instruct-grpo)**	**4B**	✓	✗	✓	✓	**23.9**

dataset. Since UIAnchor focuses on GUI-based automation and does not natively support MCP tool invocation, we evaluate on its GUI-only subset (117 tasks) for fair comparison. MobileTask targets *meta-state-aware* interactions, including transient toast handling, multi-step information input, and error recovery. We compare UIAnchor against a broad set of baselines, including proprietary models (GPT-4o [22], Claude-Sonnet-4 [3]), open-source VLMs (V-Droid [12], UI-TARS-7B [32], UI-Venus-7B/72B [17], Seed1.5-VL [18], GLM-4.1V-9B-Thinking [20], MAI-UI [53], GUI-Owl-7B [50], and the Qwen3-VL/Qwen2.5-VL series [7, 37]), as well as recent agent frameworks

Table 7. Success rate (%) on MMBench-GUI-L1 test tasks across diverse platforms. "-" denotes undisclosed (typically very large) model sizes. MMBench-GUI-L1 contains only single-step tasks, thus covering only **L1-L2** in our taxonomy.

Model	Parameter size	Windows	MacOS	Linux	iOS	Android	Web	Overall
Task difficulty (easy)								
GPT-4o [22]	-	62.47	67.89	62.38	58.52	56.41	58.51	60.16
Claude-3.5-Sonnet [2]	-	41.34	50.04	41.61	42.03	38.96	41.79	41.54
Claude-3.7-Sonnet [4]	-	34.66	49.05	39.37	42.76	37.45	40.80	39.08
Qwen-Max-VL [6]	-	69.05	72.51	69.91	70.82	63.09	69.46	68.15
Qwen2.5-VL-72B [7]	72B	65.86	75.23	73.02	67.24	58.09	72.08	66.98
UI-TARS-72B-DPO [32]	72B	41.59	28.52	35.16	31.08	52.25	35.33	40.18
InternVL3-72B [54]	72B	74.67	78.72	79.16	83.57	80.10	81.18	79.15
GUI-Owl-7B [50]	7B	82.96	84.52	85.57	82.61	83.28	88.13	84.50
Qwen3-VL-2B [37]	2B	69.26	62.94	55.49	61.71	64.54	72.14	65.15
Qwen3-VL-4B [37]	4B	88.91	86.82	86.56	84.40	84.17	84.89	85.98
Ours (Qwen3-VL-2B-Instruct-grpo)	2B	83.04	77.10	78.24	76.70	81.60	86.08	81.41
Ours (Qwen3-VL-4B-Instruct-grpo)	4B	92.40	91.74	92.24	88.82	88.57	90.91	**90.72**
Task difficulty (medium)								
GPT-4o [22]	-	56.33	63.13	59.70	54.06	57.69	54.98	57.24
Claude-3.5-Sonnet [2]	-	39.28	47.63	45.97	44.57	42.03	34.33	41.26
Claude-3.7-Sonnet [4]	-	39.34	39.23	42.28	39.45	36.05	36.17	38.39
Qwen-Max-VL [6]	-	63.40	73.85	66.90	68.02	63.66	64.59	65.44
Qwen2.5-VL-72B [7]	72B	66.29	72.73	72.63	59.27	66.24	68.24	67.45
UI-TARS-72B-DPO [32]	72B	38.83	41.60	37.14	41.72	54.74	31.55	41.77
InternVL3-72B [54]	72B	71.46	78.58	79.88	78.43	81.36	78.67	77.89
GUI-Owl-7B [50]	7B	88.89	88.10	91.24	84.35	85.25	83.56	86.86
Qwen3-VL-2B [37]	2B	81.01	66.45	67.88	52.25	70.26	65.12	69.35
Qwen3-VL-4B [37]	4B	93.32	89.30	90.14	77.40	84.97	82.58	86.84
Ours (Qwen3-VL-2B-Instruct-grpo)	2B	90.22	77.18	85.57	67.84	84.00	76.19	82.18
Ours (Qwen3-VL-4B-Instruct-grpo)	4B	93.32	84.53	95.40	85.23	90.97	83.01	**89.75**
Task difficulty (hard)								
GPT-4o [22]	-	60.69	60.38	52.42	45.27	50.93	50.83	53.49
Claude-3.5-Sonnet [2]	-	37.40	42.70	34.07	40.86	36.96	38.11	37.55
Claude-3.7-Sonnet [4]	-	32.99	34.48	31.97	39.20	36.99	38.92	35.65
Qwen-Max-VL [6]	-	66.64	67.59	65.80	60.23	58.78	65.34	63.69
Qwen2.5-VL-72B [7]	72B	70.68	68.91	70.98	57.59	53.94	68.10	64.56
UI-TARS-72B-DPO [32]	72B	31.48	35.87	24.19	36.33	58.13	19.94	35.78
InternVL3-72B [54]	72B	75.08	77.44	76.19	70.37	75.73	78.11	75.70
GUI-Owl-7B [50]	7B	87.78	96.43	94.33	87.83	88.85	94.06	90.90
Qwen3-VL-2B [37]	2B	78.49	81.88	91.30	73.81	82.30	83.36	82.25
Qwen3-VL-4B [37]	4B	91.91	93.02	84.41	93.84	92.48	89.54	91.29
Ours (Qwen3-VL-2B-Instruct-grpo)	2B	88.60	87.05	94.41	81.03	88.59	90.34	89.02
Ours (Qwen3-VL-4B-Instruct-grpo)	4B	94.79	95.35	94.89	87.91	92.85	92.68	**93.29**

(AndroidWorld [34], AutoDroid [42], UGround [15], AndroidGen [25], GUI-Explorer [46], MobileUse [27], Agent S2 [1]). Tab. 5 and Fig. 11 show the results. *First*, on AndroidWorld, UIAnchor achieves the best success-size tradeoff (Fig. 11a) with 67.2% success, surpassing larger models such as UI-Venus-72B (65.9%) and MobileUse (62.9%), and improving over the base Qwen3-VL-4B-Instruct (45.3%). *Second*, on MobileTask, UIAnchor reaches 54.9% accuracy, significantly outperforming Qwen3-VL-4B-Instruct (23.5%) and other comparable or larger models (*e.g.*, Qwen3-VL-8B-Instruct at 27.5%, GUI-Owl-7B at 41.2%), highlighting strong robustness in state-aware scenarios. *Third*, on MobileWorld GUI-only subset, UIAnchor achieves 23.9% success rate (Table 6), substantially outperforming the base Qwen3-VL-4B-Instruct (7.7%) and surpassing larger models including GUI-Owl-32B (8.5%) and UI-Venus-72B (16.4%). UIAnchor remains competitive with MAI-UI-8B (27.5%) despite using only a 4B backbone. *Fourth*, UIAnchor is highly parameter-efficient: it contains 4.083B parameters in total, adding only 83M LoRA (2.1% overhead) while delivering a >2.3× improvement over the base model. **Summary**. UIAnchor's **meta-controller** coordination beats brute-force scaling, enabling reliable mobile GUI automation with a 4B backbone. It relies on screenshots (no A11y tree), remaining robust when accessibility trees are unavailable.



(a) Qwen3-VL (b) Qwen2.5-VL (c) MiniCPM_V_4_5

Fig. 12. Success rate of Qwen3-VL/Qwen2.5-VL/MiniCPM_V_4_5 with supplemental UI element context on screenSpotV2-mobile (Ablation of the baseline model versus its performance when augmented with pre-processed UI element information from OmniParser-v2 and our method, across 2B/3B/4B/7B/8B scales).

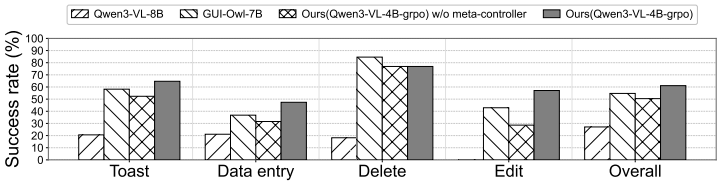


Fig. 13. Success rate (%) across different models and task categories on unseen applications.

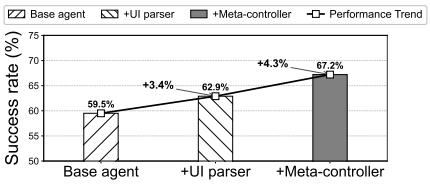


Fig. 14. Success rate (%) ablation study of UI parser and meta-controller.

7.2.3 Cross-Platform Performance. We evaluate single-step GUI understanding on *MMBench-GUI-L1*, a QA benchmark spanning six platforms (*i.e.*, Windows, MacOS, Linux, iOS, Android, Web). Importantly, UIAnchor is trained *only* on our *MobileTask Train Dataset* (trajectory data; Sec. 6.3), and the MMBench test set is strictly held out (no overlap with training data). As shown in Tab. 7, UIAnchor achieves state-of-the-art accuracy across Easy/Medium/Hard: 90.72%/89.75%/93.29%, outperforming all baselines including InternVL3-72B and GUI-Owl-7B. Gains are consistent across platforms, with strong results on Windows and Linux. Moreover, our 2B variant remains competitive (81.41/82.18/89.02), surpassing multiple larger models (*e.g.*, Qwen2.5-VL-72B and both Claude variants). **Summary**. Trajectory-only training yields transferable GUI understanding, enabling strong cross-platform generalization without benchmark-specific supervision.

7.3 Performance of UIAnchor's UI Parser

We test UIAnchor 's UI parser by augmenting *Qwen3-VL* models (2B/4B/8B) with structured UI elements and testing on *ScreenSpotV2-Mobile* [44] *without* any fine-tuning on the benchmark data. We compare the vanilla VLM [37], OmniParser-v2 [31], and UIAnchor 's parser (Fig. 12). *First*, UIAnchor consistently improves success rate across all scales, with the largest gain on 4B: +22.53%(62.87%→85.40%). *Second*, the augmented 4B model nearly matches the 8B baseline (85.40% *vs.* 86.8%; 1.4% gap), indicating structured UI context can offset reduced capacity. *Third*, UIAnchor outperforms OmniParser-v2 at every scale (up to +9.78% on 2B). *Finally*, our segmentation-based parser achieves 91.47% icon recall (Tab. 2), +6.16% over YOLO-based methods, consistent with the accuracy gains. **Summary**. UIAnchor's high-recall structured UI parsing substantially boosts visual grounding, reducing dependence on model scale for mobile GUI interaction.

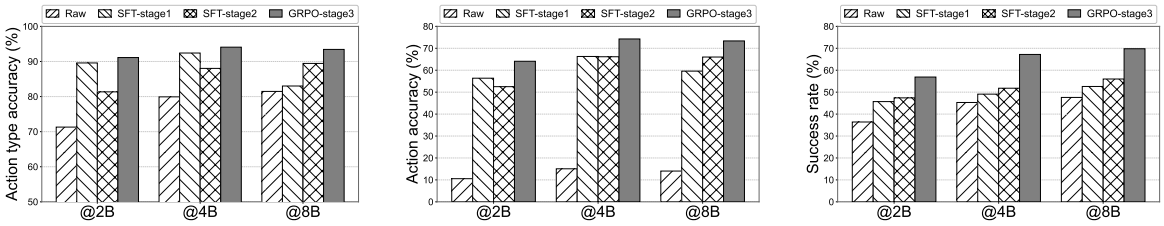


Fig. 15. Improvements of task success rates by incrementally applying SFT and GRPO to the base models.

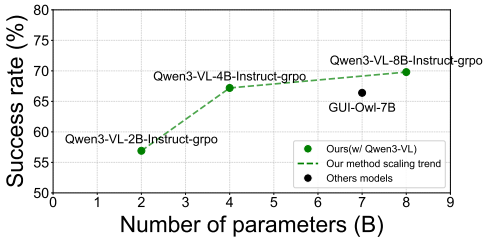


Fig. 16. Performance-parameter trade-off in vision-language models.

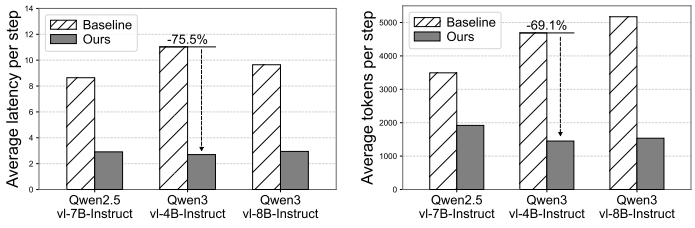


Fig. 17. System overhead comparison: UIAnchor vs. M3A implementation in AndroidWorld (evaluated on 2×3090 GPUs).

7.4 Performance on Unseen Apps/Traces

We test UIAnchor on *37* held-out tasks across *9* unseen applications. None of these tasks appear in training. Fig. 13 shows the resullts. *First*, UIAnchor with the meta-controller achieves the best overall success of 61.1%, outperforming Qwen3-VL-8B (27.1%) and GUI-Owl-7B (54.7%), demonstrating strong generalization with a compact 4B backbone. *Second*, removing the meta-controller drops success to 50.41% (-10.69% points), with the largest degradations on state-sensitive categories: *toast handling* (64.7%→52.35%), *data entry* (47.4%→31.58%), and *edit operations* (57.1%→28.6%). *Third*, even without the meta-controller, UIAnchor remains competitive and often exceeds Qwen3-VL-8B, indicating the benefit of specialized multi-agents, while meta-controller is key to consistently surpassing all baselines on unseen tasks. **Summary**. UIAnchor generalizes effectively to unseen apps, and the meta-controller is essential for robust state tracking and recovery in complex interactions.

7.5 Ablation Study and Micro-Benchmarks

7.5.1 Impact of UI Parser. We quantify the benefit of structured UI parsing by augmenting a base agent (Planner+Executor) with element-level screen representations. On AndroidWorld, this raises success from 59.5% to 62.9% (+3.4%). The gain indicates that converting raw screenshots into structured UI elements reduces mobile UI ambiguity and improves grounding, especially for *localization-critical* actions (*e.g.*, tapping specific buttons and filling designated fields). Overall, accurate parsing strengthens downstream planning and execution by providing a more reliable UI state.

7.5.2 Impact of Meta-Controller. We then add the meta-controller on top of the parser-augmented agent, improving success from 62.9% to 67.2% (+4.3%), the largest single-module gain. The meta-controller enables closed-loop coordination via pre-action feasibility checks and post-action outcome perception (including transient signals such

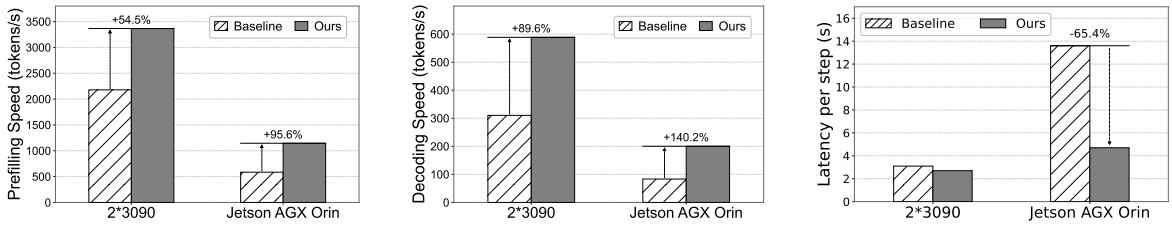


Fig. 18. System performance comparison on cloud and edge platforms.

as toast messages), and supports targeted recovery without restarting workflows. Together, these mechanisms mitigate compounding errors and are central to UIAnchor's reliability on multi-step GUI tasks.

7.5.3 Benefits of Multi-Stage Training. We evaluate our three-stage training pipeline (SFT-Stage1, SFT-Stage2, GRPO-Stage3) on Qwen3-VL models at three scales (2B/4B/8B). All variants use the same pipeline (UI parser + meta-controller) and are evaluated on AndroidWorld. Across scales, performance improves monotonically with each stage; we highlight the 4B model as it best balances capability and efficiency. Fig. 15 shows the results. *First*, **Action-only SFT (Stage 1)** grounds the base 4B model in mobile GUI semantics, improving success rate from 45.3% to 49.1%. Although Stage 1 achieves strong action-level metrics (92.41% action-type accuracy; 66.28% action accuracy), task success remains limited, suggesting that accurate low-level actions alone do not ensure end-to-end completion. *Second*, **Reasoning-augmented SFT (Stage 2)** further increases success to 51.8%, indicating that supervising intermediate reasoning better supports multi-step task execution, even with slightly lower action-level metrics (88.03% type; 66.15% action). *Third*, **GRPO with a failure-aware curriculum (Stage 3)** yields the largest gain, boosting success to 67.2% (+15.4% over SFT-variant), demonstrating the value of preference-based optimization for aligning behavior with task completion. **Summary**. The staged pipeline delivers compounding benefits. SFT provides essential grounding, while GRPO drives substantial end-to-end improvements. Notably, the final 4B model reaches 67.2% success, surpassing several larger baselines on AndroidWorld.

7.5.4 Trade-Offs between Model Size and Task Performance. We study the model size-performance trade-off in the edge-deployable regime (≤8B) to identify (i) the lower bound of usable capacity and (ii) the best performance achievable under a fixed parameter budget. *First*, varying the backbone size shows a clear capacity threshold: the 4B UIAnchor reaches 67.2% success with only 2.1% LoRA overhead, while shrinking to 2B causes an 15% absolute drop, indicating that *4B* is a practical inflection point for supporting the semantic understanding and multi-step reasoning required by GUI tasks. *Second*, under the ≤8B budget, UIAnchor is highly parameter-efficient: it exceeds GUI-Owl-7B (66.4%) by 0.8 points with *41.6%* fewer parameters, and outperforms V-Droid (59.5%) by 7.7 points with roughly half the parameter budget. *Third*, gains are driven more by specialization than scaling: the additional *83M* LoRA parameters (2.1% of total) enable task-specific planning/grounding/reflection behaviors that deliver disproportionate improvements. **Summary**. UIAnchor suggests *4B* as a competitive lower bound for edge GUI automation, and shows that architectural specialization can improve performance-per-parameter more effectively than increasing model size within edge constraints.

7.6 System Overhead

7.6.1 Latency. We evaluate UIAnchor against the M3A implementation under both *cloud* and *edge* deployments. On cloud GPUs, we compare three backbones: Qwen3-VL-4B-Instruct (ours), Qwen2.5-VL-7B-Instruct (a common base for many GUI agents such as GUI-Owl-7B), and Qwen3-VL-8B-Instruct (Fig. 17). Under the UIAnchor

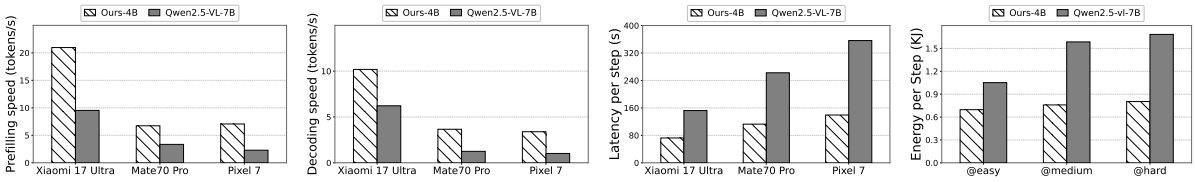


Fig. 19. On-device efficiency analysis of UIAnchor compared with baselines across diverse mobile platforms (*i.e.*, Xiaomi 17 Ultra, Huawei Mate70 Pro, Google Pixel 7).

framework, we further compare Qwen3-VL-4B-Instruct with the Qwen2.5-VL-7B-Instruct baseline across edge and cloud settings (Fig. 18). The edge server is an NVIDIA Jetson AGX Orin; the cloud server uses 2×RTX 3090.

First, as shown in Fig. 18c, UIAnchor achieves 4.7 s/step under weak edge assistance, reducing latency by 65.4% compared to the Qwen2.5-VL-7B-Instruct. With cloud assistance, UIAnchor further reduces per-step latency to 2.7 s. The two-stage UI Parser completes within 100 ms per step on average, which is negligible compared to the VLM inference time. Second, to attribute latency gains to model-side efficiency, we benchmark prefill and decoding using vLLM under identical runtime settings [24]. On the cloud server (2×RTX 3090, `–gpu-memory-utilization 0.85`), UIAnchor reaches 3365.5 tokens/s prefill (+54.5% over 2178.8) and 588.6 tokens/s decoding (+89.7% over 310.4) (Fig. 18a, Fig. 18b). On the edge server (Jetson AGX Orin, `–gpu-memory-utilization 0.7`), UIAnchor achieves 1145.5 tokens/s prefill (+95.6% over 585.5) and 200.3 tokens/s decoding (+140.2% over 83.4). These show that our 4B backbone materially accelerates both prompt processing and token generation, with the largest gains appearing on VRAM/bandwidth-constrained hardware. *Third*, as shown in Fig. 17b, removing A11y-tree inputs reduces per-step tokens from 4.6k to 1.4k (-69%), directly shrinking decoding workload and translating into lower step latency. Together with the throughput gains above, this explains UIAnchor's consistent end-to-end speedup. Fourth, on high-end cloud GPUs, end-to-end serving can become memory/overhead-bound, so 4B and 7B/8B models may exhibit similar wall-clock speed. In contrast, on resource-limited servers (*e.g.*, Jetson-class devices or GPUs with limited VRAM/bandwidth), 7B/8B models hit memory constraints earlier (tighter batch/context budgets and heavier quantization/offload), making parameter count a first-order bottleneck. Accordingly, our 4B design cuts end-to-end latency by 65% on such deployments, while enabling cheaper tiers and higher concurrency without hardware upgrades. Fifth, UIAnchor targets hands-busy and safety-/distraction-sensitive workflows, where verified reliability matters more than sub-second reactions. At 2.7 s/step, UIAnchor is human-comparable: humans spend ∼20 s of *active* interaction to finish an 8-12 step task, whereas UIAnchor reduces user effort to one spoken instruction and bears the remaining latency itself.

Summary. Across edge and cloud, UIAnchor achieves human-like speed (2.7 s/step) while substantially reducing latency versus the baseline. The gains come from (i) a compact 4B backbone that improves prefill/decoding throughput, especially on constrained devices, and (ii) token-efficient screenshot-only UI representations (-69% tokens without A11y trees), alongside a self-verifying meta-control loop that prioritizes reliable execution.

7.6.2 Reasoning Throughput. We evaluate reasoning efficiency by comparing token throughput and per-step latency of UIAnchor (Qwen3-VL-4B-Instruct) against prevalent vision-based agents built on Qwen2.5-VL-7B-Instruct (*e.g.*, GUI-Owl-7B and UI-Tars-7B) on identical multi-step tasks. We use three smartphones: Xiaomi 17 Ultra, Huawei Mate70 Pro, and Google Pixel 7. *First*, UIAnchor achieves higher prefilling throughput (20.96/6.73/7.06 token/s) than the 7B baseline (9.53/3.35/2.29 tokens/s), improving by 119.7% on average. *Second*, the advantage persists in decoding: UIAnchor reaches 10.19/3.66/3.39 tokens/s versus 6.22/1.25/1.02 tokens/s, a 161.5% average increase. *Third*, reduced token count further lowers end-to-end latency (*e.g.*, 72.39s vs. 152.26s on Xiaomi 17

(a) Automation preference (b) Tool limitations (c) Feedback burden (d) Primary benefit

Fig. 20. Survey results highlighting user demands for reliable mobile GUI automation.

(a) Navigation alignment (b) Reliability perception (c) Robustness to pop-ups (d) Need for self-recovery

Fig. 21. Post-demo user evaluation of UIAnchor.

Ultra), though real-time interaction remains future work. **Summary**. Higher throughput and fewer input tokens jointly reduce per-step cost and overall task time.

7.6.3 Energy Efficiency. We measure energy cost on AndroidWorld (Easy/Medium/Hard) using a controlled setup on Google Pixel 7: WiFi-connected, no charging interference, real-time current/voltage logging, and energy computed by integrating power over execution time. We compare against Qwen2.5-VL-7B-Instruct based agents (GUI-Owl-7B and UI-Tars-7B). *First*, UIAnchor consumes less energy across all difficulties: Easy 0.70 *vs.* 1.05 kJ (-33.3%). *Second*, energy savings increase with complexity: Medium 0.76 *vs.* 1.59 kJ (-52.2%), Hard 0.80 vs. 1.68 kJ (-52.4%). *Third*, reductions stem from the compact 4B backbone, token-efficient UI parsing, and targeted execution that avoids redundant computation. **Summary**. UIAnchor reduces energy by 33∼52%, enabling more practical sustained on-device automation.

7.7 User Study

We conduct a two-phase user study to assess the perceived utility and reliability of UIAnchor for mobile task automation. We recruited 71 participants via online and in-person channels to ensure diverse technical backgrounds and usage patterns. Participants included 48 males and 23 females, aged 19–38 (μ=26.7, σ=4.8), all being regular smartphone users. The study consists of: (i) an online *general survey* completed by all 71 participants, and (ii) an in-lab *post-demonstration evaluation* with 24 participants after observing live UIAnchor demos. Both phases used single-choice questions (one option per question).

7.7.1 User Demand Study. Results (Fig. 20) indicate strong demand for GUI automation but clear dissatisfaction with current tools. 45.1 % report high desirability for task automation (Fig. 20a), while 50.7 % identify fragility to app/UI updates as the primary limitation (Fig. 20b). Participants primarily value efficiency gains: 56.3 % select time saving and reduced cognitive load as the most significant benefit (Fig. 20c). Meanwhile, privacy/security is the top adoption concern (49.3 %, Fig. 20d), highlighting the importance of on-device execution.

7.7.2 Post-Demo User Evaluation. After observing UIAnchor (Fig. 21a-Fig. 21d), participants provide feedback on robustness and trust. 50.0 % feel UIAnchor's navigation behavior matched their expectations (Fig. 21a), and 62.5 % report improved perceived reliability due to staged parsing and verification (Fig. 21b). Participants further emphasize dynamic robustness: 58.3 % state that pop-up adaptation increased perceived intelligence (Fig. 21c), and 79.2 % consider self-recovery critical for trusting automation tools (Fig. 21d).

Fig. 22. Representative scenarios and devices used in our real-world case study.

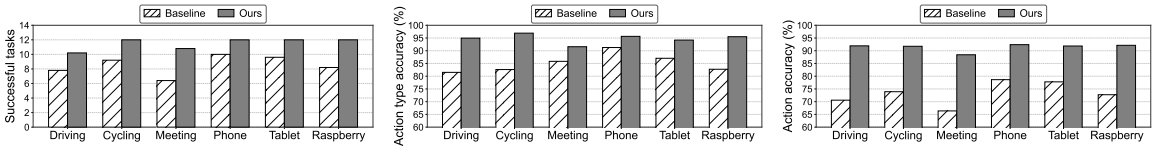


Fig. 23. Performance comparison between baseline and UIAnchor method across six scenarios.

Summary. The study suggests three practical demands for mobile GUI automation: 1) robustness to UI changes and disruptions is a dominant unmet need; 2) privacy is a primary barrier, motivating on-device solutions; 3) trust hinges on closed-loop execution, especially self-recovery. These findings support UIAnchor's design choice of GUI agent and meta-controller-based verification and recovery for reliable real-world GUI automation.

7.8 Real-World Case Study

We further validate UIAnchor in the kinds of moments where mobile automation is most valuable, when users are *in motion*, *hands-busy*, or *attention-limited*. As shown in Fig. 22, we conduct a real-world study spanning **six** practical conditions along two axes: *scenario* and *device*. For *scenarios*, we test **Driving** (in-vehicle use while driving), **Cycling** (delivery riders on the move), and **Meeting** (professional settings with social/attention constraints), each with **12** scenario-specific tasks reflecting realistic intents under interruptions and UI volatility. For *devices*, we evaluate a **Phone** (Pixel 7, 1080×2400), a **Tablet** (MatePad Pro, 2560×1600), and a **Raspberry Pi** setup (1080p monitor), running **12** cross-app common tasks per device to stress UI adaptation across screen sizes and interaction paradigms. We compare against a strong baseline, *i.e.*, the M3A agent with Qwen2.5-VL-7B [7]. In each scenario, we carry the device and system into real use and continuously evaluate for *2-3 hours*, reporting averaged results across five users.

Across all conditions, UIAnchor consistently outperforms the baseline across all metrics (Fig. 23). *First*, it completes more tasks in every scenario, with the largest gains in the most human-constrained contexts, such as **Cycling** (12 vs. 9.2) and **Meeting** (10.8 vs. 6.4) successful tasks (Fig. 23a). *Second*, it makes substantially better interaction decisions under real-world disturbances, improving action-type accuracy by over 13% in **Driving** and **Cycling**(Fig. 23b). *Third*, end-to-end action accuracy sees similarly large gains, notably in **Driving** (91.92% vs. 70.59%) (Fig. 23c). *Fourth*, device-wise, UIAnchor generalizes well across form factors, with consistent improvements on **Tablet** and **Raspberry Pi**, indicating robust cross-resolution UI understanding. **Summary**. This real-world case study shows that UIAnchor translates benchmark gains into practical robustness, with substantially fewer perception misses, less state drift, and more stable recovery when interaction is inconvenient, unsafe, or inaccessible. These results support practical promise in such scenarios, while stronger claims about driving, accessibility, or safety-critical deployment require dedicated task-based user studies and broader validation.

8 RELATED WORK

8.1 Mobile Task Automation

Mobile task automation translates high-level intents (often natural language) into executable multi-step interactions on smartphones. Prior work falls into three paradigms: **API-based** methods map user intents to privileged APIs [13, 47]. They are efficient and reliable, but fundamentally limited by API availability and permission scope. **GUI-to-API** hybrids first mine GUIs offline to infer reusable functions/scripts, then invoke them online (e.g., AXIS [30], AutoDroid-v2 [43]). They improve coverage but require costly offline exploration and remain brittle to UI/version changes. **GUI-based** agents act directly through the visual interface via taps/typing/swipes, offering maximal generality across system and third-party apps. However, open-loop execution and imperfect UI grounding often cause compounding errors under dynamic mobile UIs. UIAnchor follows the GUI-based paradigm, but strengthens reliability via (i) structured, high-recall UI parsing to anchor *what* to act on, and (ii) a meta-controller for state verification and targeted recovery to anchor *whether* actions succeed.

8.2 GUI-Based Mobile Task Automation

GUI agents automate tasks by operating directly on app interfaces. Prior work falls into two lines: **UI-tree–based LLM agents** reason over accessibility/UI trees to select actions (e.g., AutoDroid [42], MobileGPT [26], V-Droid [12], AndroidGen [25]). They are token-efficient and interpretable, and can be accurate when metadata is complete, but often break in real apps with *incomplete, noisy, or restricted* UI trees (*e.g.*, custom rendering, dynamic feeds, obfuscation), leading to missed targets, state loss, and brittle execution. **Screenshot-grounded VLM agents** operate on screenshots and generalize better across system and third-party apps [9, 19, 50, 51]. However, most are *monolithic*, entangling perception, planning, and execution, which makes grounding fragile and errors easy to compound under mobile constraints. Overall, both paradigms suffer from *error amplification* in dynamic UIs. UIAnchor adopts screenshot grounding for generality, but improves reliability by decomposing parsing/planning/execution and coordinating them with a meta-controller for step-wise verification and targeted recovery.

8.3 Multi-Agent Systems for Mobile Workflows

Multi-agent designs improve long-horizon mobile automation by assigning specialized roles. Prior work broadly falls into three lines: **Orchestrator-centric architectures** use a central planner to decompose tasks and dispatch subtasks to tool/GUI agents [26]. They improve modularity, but the orchestrator can become a reasoning bottleneck and often fails to track step-level state under dynamic UIs. **Reflective frameworks** add a monitor/reflector for failure detection and recovery [27, 41, 50]. They help correct major mistakes, but are usually coarse-grained, can be unreliable (hallucinated judgments), and add latency via repeated re-parsing/re-planning. **Experience-driven systems** introduce memory or self-evolution to reuse trajectories [41, 42, 51], yet stored experience is often rigid (coarse granularity, weak state abstraction), limiting reuse when UI layouts or contexts shift. In contrast, UIAnchor treats parsing, planning, and execution as peer agents coordinated by a *meta-controller* with *fine-grained*, *per-step* verification and targeted recovery, improving grounding and long-horizon robustness in dynamic mobile UIs.

9 CONCLUSION

This paper addresses the reliability bottleneck of mobile GUI automation under today's *service-composed* usage trends (*e.g.*, app switching, forms, pop-ups/permissions, copy-paste), which demand fine-grained actions on cross-app, mm-scale, dynamic, and heterogeneous UIs. While screenshot-grounded VLM agents enable API-free control, real apps expose parsing errors plus open-loop execution cascade into long failure chains. We present UIAnchor, which jointly anchors *what to act on* and *whether actions succeed* by integrating structured UI grounding with runtime execution verification. A two-stage parser produces confidence-aware, context-grounded UI states, while a lightweight meta-controller performs per-action verification, maintains runtime meta-state

memory (*e.g.*, toasts and activity transitions), and enables targeted recovery. Despite a compact 4B backbone with lightweight LoRA adapters, UIAnchor substantially improves end-to-end success on hard L4 tasks, significantly outperforming GPT-4o and a 7B GUI agent; with edge/cloud assistance, it runs at ~2 s per step (human-like), reduces latency/energy, and generalizes to unseen apps and evolving cross-platform GUIs. Future work will incorporate richer system signals beyond screenshots, improve UI-shift adaptation via continual learning, and further reduce latency while preserving closed-loop reliability.

Acknowledgments

This work was partially supported by the National Natural Science Foundation of China (62522215, 62532009, 62472354, 62572412). Zimu Zhou's research is supported by City University of Hong Kong Shenzhen Research Institute (CityUHKSRI).

References

[1] Saaket Agashe, Kyle Wong, Vincent Tu, Jiachen Yang, Ang Li, and Xin Eric Wang. 2025. Agent S2: A Compositional Generalist-Specialist Framework for Computer Use Agents. arXiv:2504.00906 [cs.AI] https://arxiv.org/abs/2504.00906

[2] Anthropic. 2024. Introducing Claude 3.5 Sonnet. https://www.anthropic.com/news/claude-3-5-sonnet. Accessed: 2026-01-23.

[3] Anthropic. 2025. Introducing Claude 4. https://www.anthropic.com/news/claude-4. Accessed: 2026-01-12.

[4] Anthropic. 2026. Introducing Claude 3.7 Sonnet. https://www.anthropic.com/news/claude-3-7-sonnet. Accessed: 2026-01-23.

[5] Apple Inc. 2022. Shortcuts User Guide (for iOS 16 and iPadOS 16). https://support.apple.com/zh-cn/guide/shortcuts/welcome/ios. Accessed: 2026-01-29.

[6] Jinze Bai, Shuai Bai, Shusheng Yang, Shijie Wang, Sinan Tan, Peng Wang, Junyang Lin, Chang Zhou, and Jingren Zhou. 2023. Qwen-VL: A Versatile Vision-Language Model for Understanding, Localization, Text Reading, and Beyond. *arXiv preprint arXiv:2308.12966* (2023).

[7] Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, Humen Zhong, Yuanzhi Zhu, Mingkun Yang, Zhaohai Li, Jianqiang Wan, Pengfei Wang, Wei Ding, Zheren Fu, Yiheng Xu, Jiabo Ye, Xi Zhang, Tianbao Xie, Zesen Cheng, Hang Zhang, Zhibo Yang, Haiyang Xu, and Junyang Lin. 2025. Qwen2.5-VL Technical Report. *arXiv preprint arXiv:2502.13923* (2025).

[8] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*. 41–48.

[9] Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Yantao Li, Jianbing Zhang, and Zhiyong Wu. 2024. Seeclick: Harnessing gui grounding for advanced visual gui agents. *arXiv preprint arXiv:2401.10935* (2024).

[10] Xiangxiang Chu, Limeng Qiao, Xinyu Zhang, Shuang Xu, Fei Wei, Yang Yang, Xiaofei Sun, Yiming Hu, Xinyang Lin, Bo Zhang, et al. 2024. Mobilevlm v2: Faster and stronger baseline for vision language model. *arXiv preprint arXiv:2402.03766* (2024).

[11] Cheng Cui, Ting Sun, Manhui Lin, Tingquan Gao, Yubo Zhang, Jiaxuan Liu, Xueqing Wang, Zelun Zhang, Changda Zhou, Hongen Liu, Yue Zhang, Wenyu Lv, Kui Huang, Yichao Zhang, Jing Zhang, Jun Zhang, Yi Liu, Dianhai Yu, and Yanjun Ma. 2025. PaddleOCR 3.0 Technical Report. arXiv:2507.05595 [cs.CV] https://arxiv.org/abs/2507.05595

[12] Gaole Dai, Shiqi Jiang, Ting Cao, Yuanchun Li, Yuqing Yang, Rui Tan, Mo Li, and Lili Qiu. 2025. Advancing mobile gui agents: A verifier-driven approach to practical deployment. *arXiv preprint arXiv:2503.15937* (2025).

[13] Lutfi Eren Erdogan, Nicholas Lee, Siddharth Jha, Sehoon Kim, Ryan Tabrizi, Suhong Moon, Coleman Hooper, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. 2024. Tinyagent: Function calling at the edge. *arXiv preprint arXiv:2409.00608* (2024).

[14] Georgi Gerganov and Contributors. 2023. *llama.cpp*. https://github.com/ggml-org/llama.cpp LLM inference in C/C++.

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

[16] Jihao Gu, Qihang Ai, Yingyao Wang, Pi Bu, Jingxuan Xing, Zekun Zhu, Wei Jiang, Ziming Wang, Yingxiu Zhao, Ming-Liang Zhang, et al. 2025. Mobile-R1: Towards Interactive Reinforcement Learning for VLM-Based Mobile Agent via Task-Level Rewards. *arXiv preprint arXiv:2506.20332* (2025).

[17] Zhangxuan Gu, Zhengwen Zeng, Zhenyu Xu, Xingran Zhou, Shuheng Shen, Yunfei Liu, Beitong Zhou, Changhua Meng, Tianyu Xia, Weizhi Chen, et al. 2025. Ui-venus technical report: Building high-performance ui agents with rft. *arXiv preprint arXiv:2508.10833* (2025).

[18] Dong Guo, Faming Wu, Feida Zhu, Fuxing Leng, Guang Shi, Haobin Chen, Haoqi Fan, Jian Wang, Jianyu Jiang, Jiawei Wang, et al. 2025. Seed1. 5-vl technical report. *arXiv preprint arXiv:2505.07062* (2025).

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

[20] Wenyi Hong, Wenmeng Yu, Xiaotao Gu, Guo Wang, Guobing Gan, Haomiao Tang, Jiale Cheng, Ji Qi, Junhui Ji, Lihang Pan, et al. 2025. GLM-4.1 V-Thinking: Towards Versatile Multimodal Reasoning with Scalable Reinforcement Learning. *arXiv preprint arXiv:2507.01006* (2025).

[21] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. *ICLR* 1, 2 (2022), 3.

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

[23] 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. 2025. MobileWorld: Benchmarking Autonomous Mobile Agents in Agent-User Interactive, and MCP-Augmented Environments. arXiv:2512.19432 [cs.AI] https://arxiv.org/abs/2512.19432

[24] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[25] Hanyu Lai, Junjie Gao, Xiao Liu, Yifan Xu, Shudan Zhang, Yuxiao Dong, and Jie Tang. 2025. AndroidGen: Building an Android Language Agent under Data Scarcity. *arXiv preprint arXiv:2504.19298* (2025).

[26] Sunjae Lee, Junyoung Choi, Jungjae Lee, Munim Hasan Wasi, Hojun Choi, Steve Ko, Sangeun Oh, and Insik Shin. 2024. Mobilegpt: Augmenting llm with human-like app memory for mobile task automation. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*. 1119–1133.

[27] Ning Li, Xiangmou Qu, Jiamu Zhou, Jun Wang, Muning Wen, Kounianhua Du, Xingyu Lou, Qiuying Peng, and Weinan Zhang. 2025. MobileUse: A GUI Agent with Hierarchical Reflection for Autonomous Mobile Operation. *arXiv preprint arXiv:2507.16853* (2025).

[28] Xiao Liu, Bo Qin, Dongzhu Liang, Guang Dong, Hanyu Lai, Hanchen Zhang, Hanlin Zhao, Iat Long Iong, Jiadai Sun, Jiaqi Wang, et al. 2024. Autoglm: Autonomous foundation agents for guis. *arXiv preprint arXiv:2411.00820* (2024).

[29] LSPosed Community. 2021. *LSPosed Framework: A Riru / Zygisk module trying to provide an ART hooking framework*. https://github.com/LSPosed/LSPosed [Online; accessed 2026-02-01]. Supported Android versions: 8.1 to 14..

[30] Junting Lu, Zhiyang Zhang, Fangkai Yang, Jue Zhang, Lu Wang, Chao Du, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. 2024. AXIS: Efficient Human-Agent-Computer Interaction with API-First LLM-Based Agents. *arXiv preprint arXiv:2409.17140* (2024).

[31] Yadong Lu, Jianwei Yang, Yelong Shen, and Ahmed Awadallah. 2024. Omniparser for pure vision based gui agent. *arXiv preprint arXiv:2408.00203* (2024).

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

[33] Nikhila Ravi, Valentin Gabeur, Yuan-Ting Hu, Ronghang Hu, Chaitanya Ryali, Tengyu Ma, Haitham Khedr, Roman Rädle, Chloe Rolland, Laura Gustafson, Eric Mintun, Junting Pan, Kalyan Vasudev Alwala, Nicolas Carion, Chao-Yuan Wu, Ross Girshick, Piotr Dollár, and Christoph Feichtenhofer. 2024. SAM 2: Segment Anything in Images and Videos. *arXiv preprint arXiv:2408.00714* (2024). https://arxiv.org/abs/2408.00714

[34] Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, et al. 2024. Androidworld: A dynamic benchmarking environment for autonomous agents. *arXiv preprint arXiv:2405.14573* (2024).

[35] Samsung Electronics America. 2024. Set up and use Bixby Routines on your Galaxy phone. https://www.samsung.com/us/support/answer/ANS10002624/. Accessed: 2026-01-29.

[36] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300* (2024).

[37] Qwen Team. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388

[38] Weihan Wang, Qingsong Lv, Wenmeng Yu, Wenyi Hong, Ji Qi, Yan Wang, Junhui Ji, Zhuoyi Yang, Lei Zhao, Song XiXuan, et al. 2024. Cogvlm: Visual expert for pretrained language models. *Advances in Neural Information Processing Systems* 37 (2024), 121475–121499.

[39] Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. 2020. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. *Advances in neural information processing systems* 33 (2020), 5776–5788.

[40] Xuehui Wang, Zhenyu Wu, JingJing Xie, Zichen Ding, Bowen Yang, Zehao Li, Zhaoyang Liu, Qingyun Li, Xuan Dong, Zhe Chen, Weiyun Wang, Xiangyu Zhao, Jixuan Chen, Haodong Duan, Tianbao Xie, Shiqian Su, Chenyu Yang, Yue Yu, Yuan Huang, Yiqian Liu, Xiao Zhang, Xiangyu Yue, Weijie Su, Xizhou Zhu, Wei Shen, Jifeng Dai, and Wenhai Wang. 2025. MMBench-GUI: Hierarchical Multi-Platform Evaluation Framework for GUI Agents. *arXiv preprint arXiv:2507.19478* (2025).

[41] Zhenhailong Wang, Haiyang Xu, Junyang Wang, Xi Zhang, Ming Yan, Ji Zhang, Fei Huang, and Heng Ji. 2025. Mobile-agent-e: Self-evolving mobile assistant for complex tasks. *arXiv preprint arXiv:2501.11733* (2025).

[42] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, et al. 2024. Autodroid: Llm-powered task automation in android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*. 543–557.

[43] Hao Wen, Shizuo Tian, Borislav Pavlov, Wenjie Du, Yixuan Li, Ge Chang, Shanhui Zhao, Jiacheng Liu, Yunxin Liu, Ya-Qin Zhang, et al. 2024. Autodroid-v2: Boosting slm-based gui agents via code generation. *arXiv preprint arXiv:2412.18116* (2024).

[44] Zhiyong Wu, Zhenyu Wu, Fangzhi Xu, Yian Wang, Qiushi Sun, Chengyou Jia, Kanzhi Cheng, Zichen Ding, Liheng Chen, Paul Pu Liang, et al. 2024. OS-ATLAS: A Foundation Action Model for Generalist GUI Agents. *arXiv preprint arXiv:2410.23218* (2024).

[45] Bin Xiao, Haiping Wu, Weijian Xu, Xiyang Dai, Houdong Hu, Yumao Lu, Michael Zeng, Ce Liu, and Lu Yuan. 2024. Florence-2: Advancing a unified representation for a variety of vision tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4818–4829.

[46] Bin Xie, Rui Shao, Gongwei Chen, Kaiwen Zhou, Yinchuan Li, Jie Liu, Min Zhang, and Liqiang Nie. 2025. GUI-explorer: Autonomous Exploration and Mining of Transition-aware Knowledge for GUI Agent. In *Annual Meeting of the Association for Computational Linguistics (ACL)*.

[47] Weikai Xie, Li Zhang, Shihe Wang, Rongjie Yi, and Mengwei Xu. 2024. Droidcall: A dataset for llm-powered android intent invocation. *arXiv preprint arXiv:2412.00402* (2024).

[48] Yifan Xu, Xiao Liu, Xinghan Liu, Jiaqi Fu, Hanchen Zhang, Bohao Jing, Shudan Zhang, Yuting Wang, Wenyi Zhao, and Yuxiao Dong. 2025. Mobilerl: Online agentic reinforcement learning for mobile gui agents. *arXiv preprint arXiv:2509.18119* (2025).

[49] Haolong Yan, Jia Wang, Xin Huang, Yeqing Shen, Ziyang Meng, Zhimin Fan, Kaijun Tan, Jin Gao, Lieyu Shi, Mi Yang, et al. 2025. Step-gui technical report. *arXiv preprint arXiv:2512.15431* (2025).

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

[51] Chi Zhang, Zhao Yang, Jiaxuan Liu, Yanda Li, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2025. Appagent: Multimodal agents as smartphone users. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 1–20.

[52] Yuze Zhao, Jintao Huang, Jinghan Hu, Xingjun Wang, Yunlin Mao, Daoze Zhang, Zeyinzi Jiang, Zhikai Wu, Baole Ai, Ang Wang, Wenmeng Zhou, and Yingda Chen. 2024. SWIFT:A Scalable lightWeight Infrastructure for Fine-Tuning. arXiv:2408.05517 [cs.CL] https://arxiv.org/abs/2408.05517

[53] Hanzhang Zhou, Xu Zhang, Panrong Tong, Jianan Zhang, Liangyu Chen, Quyu Kong, Chenglin Cai, Chen Liu, Yue Wang, Jingren Zhou, and Steven Hoi. 2025. MAI-UI Technical Report: Real-World Centric Foundation GUI Agents. arXiv:2512.22047 [cs.CV] https://arxiv.org/abs/2512.22047

[54] Jinguo Zhu, Weiyun Wang, Zhe Chen, Zhaoyang Liu, Shenglong Ye, Lixin Gu, Hao Tian, Yuchen Duan, Weijie Su, Jie Shao, et al. 2025. Internvl3: Exploring advanced training and test-time recipes for open-source multimodal models. *arXiv preprint arXiv:2504.10479* (2025).