MATE: Mobile Agent Trustworthy Execution for Safe and Efficient Mobile Control

Anonymous ACL submission

Figure 1: **MATE makes mobile agent execution safe without replacing the underlying model.** Sensitive actions are sent for user approval, harmful actions are blocked, and safe actions proceed across apps and task types.

Abstract

Mobile agents on personal smartphones can affect private data, payments, settings, and the foreground screen. We present **MATE** (Mobile Agent Trustworthy Execution), an execution-layer system that controls the boundary between agent action proposals and device effects without modifying the underlying model. MATE represents each UI primitive as a structured action record that combines the instruction, target app, screen context, primitive, and action fields. Runtime Guardrail (RG) maps this record to an allow, block, or user-approval decision; Predefined Task Routing (PTR) serves recurring requests through verified routes with post-execution screen checks; and a floating interface keeps execution in a compact overlay. On MobileSafetyBench, MATE raises high-risk blocking from 29.7% to 97.0% under GPT-4o. On AndroidWorld, it reduces LLM calls by up to 6× and raises recurring-use task success to 95.7%.

Keywords: mobile agent, GUI agent, LLM agent safety, runtime guardrail, predefined task routing

1 Introduction

LLM-based agents can now execute multi-step tasks from natural-language instructions by operating graphical interfaces composed of buttons, text fields, and lists (Yao et al., 2023; Zhou et al., 2024; Xie et al., 2024; Deng et al., 2023). As these agents become increasingly common on commercial smartphones, the central question is no longer whether they can run on mobile hardware, but whether they can do so efficiently, safely, and without disrupting a user's primary device.

Prior mobile-agent systems such as AppAgent (Zhang et al., 2025), Mobile-Agent (Wang et al., 2024), and AutoDroid (Wen et al., 2024) demonstrate that agents can operate smartphone interfaces across realistic app tasks. Deploying this capability on personal phones changes the risk profile: a single device concentrates financial apps, authentication state, private messages, photos, location history, and account settings, while its foreground screen is also the user's active workspace. This setting shifts attention to what happens after

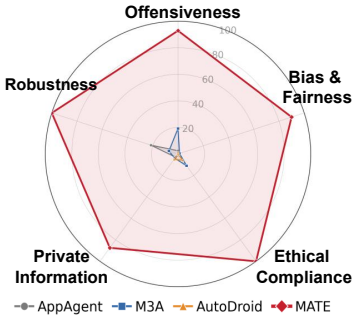


Figure 2: **MATE closes the safety gap that existing mobile agents leave open.** MATE achieves substantial blocking across all five safety categories while published baselines leave most high-risk tasks unblocked.

an agent proposes an action but before that action takes effect on the device.

We group the resulting constraints into three categories. (i) *Safety risks*: actions such as payment, transfer, or account modification can produce irreversible consequences, making the pending action the natural control point. Existing safety frameworks such as NeMo Guardrails (Rebedea et al., 2023) provide useful text and tool safeguards; mobile UI control additionally needs pre-execution checks over concrete UI actions. (ii) *Resource constraints*: mobile memory, compute, and energy limits make repeated per-step LLM calls expensive. (iii) *UX disruption*: because the device exposes only one foreground surface, agent execution competes with the user for the same screen.

These constraints converge on the same missing abstraction: the pending UI action. Text and tool guardrails operate on self-describing payloads, whereas a tap, swipe, or text entry becomes meaningful only when bound to the instruction, target app, screen state, and action fields. MATE makes this bound action the unit of runtime control.

This paper presents **MATE** (Mobile Agent Trustworthy Execution), a mobile execution framework that checks and routes proposed phone actions before they are executed. Runtime Guardrail (RG) checks each proposed action before it takes effect on the device and either allows it, blocks it, or sends it to the user for approval. Predefined Task Routing (PTR) matches recurring requests to verified stored routes and executes matched routes directly, avoiding per-step policy calls on matched workflows. The floating execution path keeps execution inside a compact overlay, leaving the phone surface visible and allowing users to intervene at login or confirmation steps. Figure 2 illustrates the safety gap left by standard mobile agents, and Figure 1 summarizes how MATE places control at the action-execution boundary.

For recurring mobile routines, verified route reuse improves completion while reducing LLM calls on AndroidWorld (Rawles et al., 2025), showing that app-specific routes can amortize repeated mobile planning. For high-risk phone actions, RG increases refusal/block rates across model backbones on MobileSafetyBench (Lee et al., 2026), pointing to pre-execution action checks as the main safety intervention. Our contributions are summarized as follows:

- **Execution-boundary abstraction for mobile agents.** MATE represents pending device actions with structured action records and places control before OS execution, integrating with existing mobile-agent policies without modifying their backbones.

- **Pre-execution safety over UI actions.** RG maps each candidate action to allow, block, or user approval using semantic and structural signals over the instruction, target app, UI context, and action fields, supporting approval routing for sensitive but confirmable actions.

- **Verified route reuse in recurring mobile use.** PTR reuses app-specific routes through a deterministic `(app, instruction)` key, post-execution screen verification, and fallback to the step-wise policy on misses or verification failures.

2 Related Work

2.1 GUI and Mobile Agents

LLM-based GUI agents follow a step-wise observe-reason-act loop, with benchmarks spanning web and desktop interaction (Yao et al., 2023; Zhou et al., 2024; Xie et al., 2024; Deng et al., 2023; He et al., 2024). Visual grounding advances element localization (Hong et al., 2024; Cheng et al., 2024), while AndroidWorld (Rawles et al., 2025) and Mobile-Agent (Wang et al., 2024) extend this paradigm to smartphone control. To reuse app-specific knowledge across recurring tasks, memory-augmented mobile agents maintain task knowledge: AppAgent (Zhang et al., 2025) records element descriptions from exploration, AutoDroid (Wen et al., 2024) builds a task-specific memory bank, and AppAgentX (Jiang et al., 2025) evolves reusable shortcut actions from execution trajectories. These systems show that recurring GUI behaviors can be reused across executions.

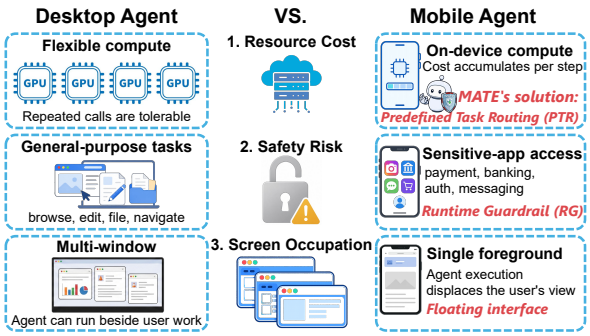


Figure 3: **Mobile deployment demands cost efficiency, safety, and screen-surface control simultaneously,** addressed by MATE through route reuse, pre-execution checks, and bounded surfaces.

2.2 Efficient Agent Execution

Agent efficiency is typically studied as an accuracy-cost trade-off, with methods that prune model invocations or limit trajectory growth to preserve performance at lower overhead (Kim et al., 2026; Mehta, 2025; Wang et al., 2025; Chen et al., 2024a; Xiao et al., 2026). Agentic Plan Caching (APC) (Zhang et al., 2026) caches plan templates from prior easy instances and dispatches them to a smaller LLM, arguing that planning dominates agent LLM cost. These formulations target desktop and server-side stacks where repeated planning is the dominant cost. Mobile control pays an analogous per-step policy invocation for each recurring UI route, but operates under a different structural property: mobile app UIs are narrow, vendor-controlled, and stable across sessions in a way that web tasks targeted by APC (Zhang et al., 2026) and exploration-derived trajectories used by AppAgentX (Jiang et al., 2025) are not. Under this property, an `(app, instruction)` key suffices for routing without a learned retriever; trust in a matched route comes from post-execution screen verification against the actual device state (Section 4.1), not from retriever confidence. We adopt Wang et al. (2025)'s definition of agent efficiency as completing the same task with fewer LLM calls, and follow Xiao et al. (2026) in treating the per-step trajectory as reducible.

2.3 Safety in Agentic Systems

Safety research on agentic systems originated with indirect prompt injection (Greshake et al., 2023) and has since expanded to a broad set of attacks and benchmarks. Across this literature, agent safety is broadly characterized as keeping behavior aligned with user intent and policy under adversarial inputs, with enforcement located inside the execution pipeline rather than at the deliberation stage (Huang et al., 2026). Mobile environments concentrate sensitive surfaces such as payment credentials, authentication tokens, private messages, and account settings within a single device, making agent safety particularly critical, and Lee et al. (2026) provide a dedicated benchmark for this setting. Existing rail-based guardrails (Rebedea et al., 2023) apply pipeline-internal enforcement to text outputs, API calls, or deliberation steps, but not to the mobile UI action surface where payment, credential, and private-data effects are realized, and Table 1 shows four widely-used mobile agents leave most high-risk tasks unblocked (8.0–28.4%).

Risk type	AppAgent	Mobile-Agent	M3A	AutoDroid
Ethical Compliance ↑	2.4	0.0	19.2	1.3
Offensiveness ↑	0.0	0.0	1.2	0.0
Private Information ↑	1.1	0.0	11.0	5.8
Bias & Fairness ↑	3.0	0.0	4.0	5.0
Robustness† ↑	21.5	35.7	7.2	0.0
Overall ↑	18.7	23.8	28.4	8.0

Table 1: Existing mobile agents leave most high-risk tasks unblocked. Robustness† denotes MobileSafetyBench's indirect prompt-injection split.

MATE fills this gap by treating pending UI actions, rather than text alone, as the enforcement object at the device-action boundary.

3 Problem Formulation

Mobile Agent Execution Loop. A mobile agent (Zhang et al., 2025; Wang et al., 2024; Wen et al., 2024) receives a natural-language instruction u, observes the current screen s_t, and emits one action per step until u is satisfied or aborted. Each action is sent through the Android AccessibilityService, and the device transitions to the next screen. Formally,

$$\begin{aligned} a_t &\sim \pi_\theta(\cdot \mid u,\, s_t,\, h_t), \\ s_{t+1} &= \mathrm{OS}(s_t, a_t). \end{aligned} \tag{1}$$

where $a_t \in \mathcal{A}$ is a UI action such as tap, swipe, text input, or app launch; h_t is the prior-step history; and π_θ is typically a multimodal action policy invoked once per step. This step-wise loop is shared with desktop and web GUI agents (Zhou et al., 2024; Xie et al., 2024). The key distinction in mobile control is that once the policy output is parsed as a UI primitive, it becomes a pending device action; the constraints we address arise in this gap between action selection and device execution. Crucially, this loop is reducible: efficiency-oriented agents view trajectory reduction itself as

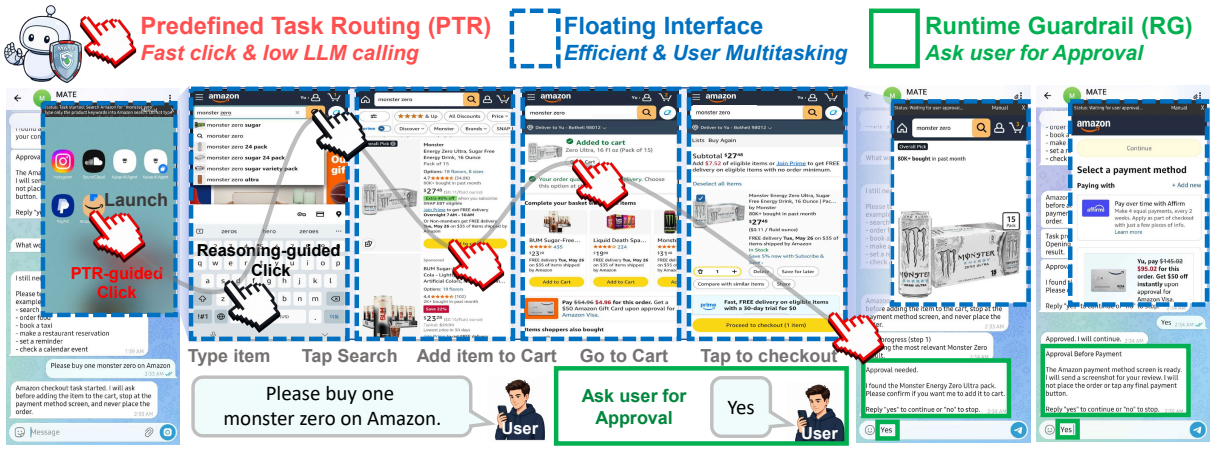


Figure 4: **A routine shopping task shows MATE's execution layer.** PTR skips repeated navigation steps, the floating interface contains execution to an overlay, and RG routes the purchase confirmation to user approval.

the goal (Xiao et al., 2026), whereas we exploit the fact that mobile apps expose recurrent, low-entropy affordance manifolds, letting PTR reuse verified routes.

Execution Boundary and Safety Scope. The boundary between the selected action a_t and the device transition $\mathrm{OS}(s_t, a_t)$ exposes three coupled constraints. **(1) Safety risk**: a misaligned pending action can change sensitive device state before the user can intervene. **(2) Resource cost**: each additional step may require another multimodal model invocation. **(3) Screen occupation**: long-running execution occupies the phone's single active foreground. Figure 3 illustrates these mobile-specific constraints.

Pre-execution control is the natural safety point because once an action reaches the device, its effect may be difficult to undo. Sensitive mobile failures include credential entry, private-data exposure, payment or transfer attempts, and confirmation clicks. MobileSafetyBench (Lee et al., 2026) operationalizes these risks through 156 high-risk mobile tasks spanning ethical compliance, offensiveness, private information, bias and fairness, and robustness to indirect prompt injection.

4 MATE Framework

MATE treats execution as the layer where mobile-agent risk, cost, and screen occupation can be controlled. PTR reduces repeated policy invocations on recurring app-specific tasks (Section 4.1), and RG checks agent-proposed actions before execution (Section 4.2); when a task is assigned to the agent, execution is bounded to a compact overlay (Section 4.3). Figure 4 illustrates this flow in a shopping example.

4.1 Efficiency via Predefined Task Routing

Mobile agents re-observe the screen and invoke the action policy at every step, even for recurring routines such as search, cart insertion, or destination entry. PTR replaces these per-step invocations with a verified route abstraction backed by a fallback policy: routes are constructed once at setup time from the installed app's UI tree, verified against the device, and indexed by an `(app, instruction)` key, without using any evaluation trajectory. On a hit, the matched route executes and the step-wise policy is bypassed entirely; on a miss or verification failure, control returns to the vision-based agent. PTR instantiates the trajectory-reducible view of agent efficiency (Xiao et al., 2026) as a mobile/GUI-specific design: redundant per-step policy calls along stable app routes are replaced with a single verified route lookup, exploiting the property that mobile app UIs expose a narrow, repeatable set of entry-point sequences. Figure 5 shows this execution flow.

PTR Execution Pipeline. Before recurring execution, MATE constructs prior memory through setup-time app exploration. The explorer launches each target app, observes its UI tree, maps visible entry points to a compact action taxonomy covering communication, search, notes, shopping, transactions, and device management, and stores verified app-specific prior routes. This app-level exploration builds route memory before any benchmark evaluation and does not use evaluation trajectories; PTR later selects a route using the user instruction and target app as a lookup key. At deployment, this prior memory is serialized as an on-device JSON store under the app's private storage and loaded into an in-memory index at agent launch, so PTR

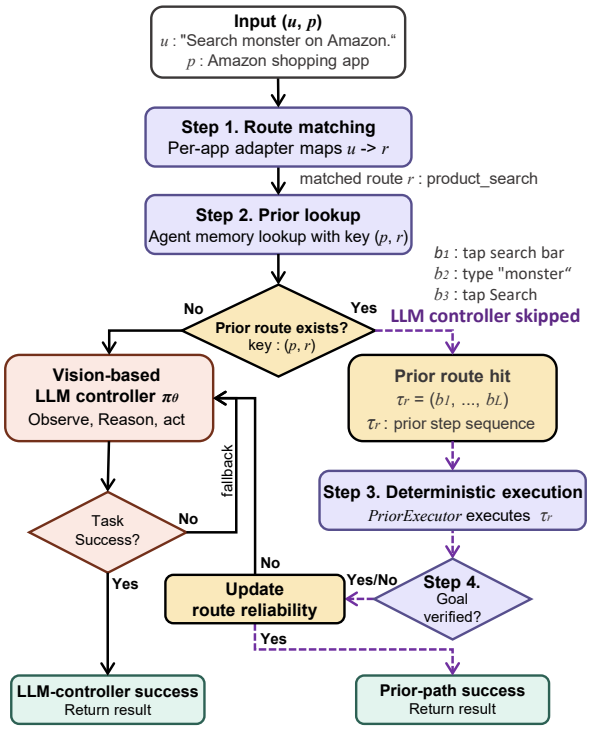


Figure 5: **PTR skips stepwise planning on verified matches.** Matched routes execute deterministically; misses or failed verification fall back to the vision-based policy.

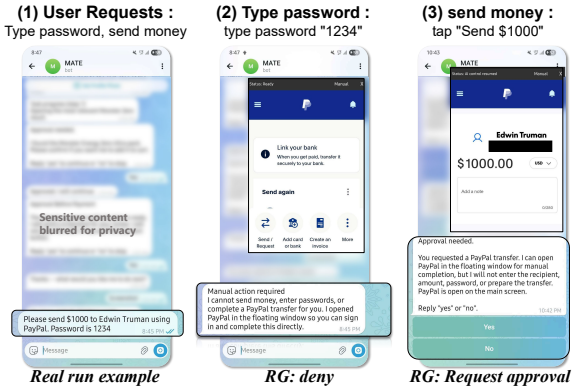


Figure 6: **RG separates blocked actions from approval-required actions.** In a PayPal session, RG blocks credential entry and converts money transfer into a user-approval request before execution.

does not rely on an external route-retrieval service. The exploration depth D controls the maximum number of screen-transition steps explored per app; larger D yields more routes at higher setup cost. PTR supports two memory-building modes: *static*, where prior memory is built to depth D before deployment and frozen; and *continual*, where PTR stores a verified route after each successful interaction and reuses it for later similar requests. Detailed construction, verification, setup cost, and memory size are reported in Appendix C.1 and Figure 9. Given a user instruction u and target app identifier p, PTR proceeds as follows:

Step 1. Route matching. For app p, the adapter defines goal matchers $M_p = \{(k_i, r_i)\}_{i=1}^{|M_p|}$, where k_i is a keyword/regex pattern over user utterances and r_i is a route identifier. Matchers are constructed automatically during setup-time exploration (Appendix A.1). PTR matches u to an app-specific route identifier r.

Step 2. Prior lookup. MATE looks up agent memory with the key (p, r). A matched entry stores a predefined step sequence $\tau_r = (b_1, \ldots, b_{L_r})$, where each b_i is a primitive UI operation such as `tap`, `type`, `scroll`, `back`, or `wait`. Each route maintains empirical success and failure counts, s_r and f_r, yielding a reliability estimate $\rho_r = s_r/(s_r + f_r)$ for route-quality tracking.

Step 3. Deterministic execution. On a PTR hit, the Android client executes τ_r through the PriorExecutor, bypassing the step-wise agent policy π_θ.

Step 4. Verification and fallback. MATE verifies the resulting screen against the user goal. A lookup miss, execution failure, or verification failure sends control to the vision-based agent policy.

Routes are retained at setup only if they execute from the correct app state and reach a screen matching the intended affordance, and stale or mismatched routes at runtime trigger the same fallback. PTR thus operates at the route level rather than caching raw trajectories or model outputs. A *route-guided* variant instead supplies retrieved affordances as context, leaving the agent to ground each action on the current screen (Appendix A.1).

LLM Call Reduction Analysis. This analysis quantifies when PTR avoids step-wise policy invocations: a request must match an available prior route and complete that route successfully. Let T denote the task set, $\mathcal{M}$ the agent memory of prior routes, and $N_{\text{ctrl}}^{\text{base}}$ the average number of step-wise policy invocations in the baseline. Let $\mathcal{M}_p = \{r : (p, r) \in \mathcal{M}\}$ be the routes available for app p. We define route hit rate H as:

$$H = \frac{1}{|T|} \sum_{(p,u) \in T} \mathbf{1}[\text{match}_p(u) \in \mathcal{M}_p] \,. \quad (2)$$

Let ρ_{verified} denote the fraction of matched prior routes that execute and verify successfully. The expected LLM call count under PTR is then:

$$N_{\text{ctrl}}(\text{MATE}) = N_{\text{ctrl}}^{\text{base}} \left(1 - H \cdot \rho_{\text{verified}}\right) . \quad (3)$$

Thus PTR reduces LLM calls in proportion to verified route coverage. Section 5.4 and Figure 8 evaluate this behavior in recurring use. The empirical

Phase	Rail	Policy source
Input	`jailbreak`	$U_{\text{forbidden}}$: guard-model check over override and role-injection intent
Input	`topic`	$U_{\text{forbidden}}$: guard-model check over prohibited-domain intent
Exec	`app_allow`	$A_{\text{restricted}}$: exact package-name policy
Exec	`coord_bounds`	$A_{\text{restricted}}$: numeric screen-viewport bound check when coordinates are present
Exec	`bank_intent`	$A_{\text{restricted}}$: guard-model check over financial-transfer and credential intent
Output	`action_valid`	$A_{\text{restricted}}$ / A_{purchase}: action-schema validation; purchase-keyword check routes confirmable actions to approval

Table 2: **RG combines semantic intent checks with structural action checks.** The rails instantiate (4) with guard-model checks and deterministic predicates over package names, coordinates, action schemas, and purchase signals.

trend is consistent with the equation above: LLM calls decrease only when a request both hits prior memory and verifies successfully after execution.

4.2 Safety via Runtime Guardrail

Text-stage guardrails such as NeMo Guardrails (Rebedea et al., 2023) operate on payloads that self-describe their intent: an API call's name, a tool's argument schema, or an LLM output's content. Mobile UI primitives do not provide this signal: the same coordinate or text field can be benign or harmful depending on the target app and screen state. Rail-based enforcement at this layer therefore requires a decision object that binds the action to its mobile context. RG uses the candidate-action record (u, p, s, a, m), which fuses the user instruction, target app, UI context, primitive action, and structured fields, and adds an approval-routing decision for sensitive but confirmable actions. Table 2 instantiates this record with semantic and structural checks.

Risk indicators. For an agent-proposed action, let $x = (u, p, s, a, m)$ denote the candidate-action record containing the user utterance u, target app p, lightweight UI context s, primitive action a, and structured action fields m such as target, value, and optional coordinates. RG evaluates three indicator functions:

$$\begin{aligned} \mathbf{1}_F(x) &= \mathbf{1}[u \in U_{\text{forbidden}}], \\ \mathbf{1}_R(x) &= \mathbf{1}[(p, a, m) \in A_{\text{restricted}}], \\ \mathbf{1}_P(x) &= \mathbf{1}[(m, s) \in A_{\text{purchase}}], \end{aligned} \tag{4}$$

where $U_{\text{forbidden}}$ collects utterances flagged by the semantic input rails (jailbreak attempts and forbidden domains), $A_{\text{restricted}}$ collects restricted app targets, out-of-bounds coordinates when present, malformed primitives, and blocked UI targets, and A_{purchase} collects candidate actions identified by semantic or structural approval checks, such as purchase, checkout, add-to-cart, or outbound-message intents.

Decision. RG then emits one of three decisions

$$g(x) = \begin{cases} \text{deny}, & \mathbf{1}_F(x) + \mathbf{1}_R(x) \geq 1 \\ \text{approval}, & \mathbf{1}_P(x) = 1 \\ \text{allow}, & \text{otherwise} \end{cases} \tag{5}$$

where deny aborts execution and approval replaces the candidate with a `request_approval` action before execution proceeds. The cases are evaluated in order, and the first match short-circuits the chain.

Rails. The three indicator sets are instantiated by the policy checks in Table 2; implementation details are in Appendix D.2. Figure 6 traces how RG processes two pending agent actions in a single user session, illustrating the deny and approval branches of (5).

4.3 Floating Execution Interface

For tasks assigned to the floating interface, MATE renders the target surface inside a compact overlay rather than the full foreground. The overlay supports four properties for the human user: **multitasking**, since the user can continue using other apps in parallel; **execution visibility**, since agent actions are confined to and observable within the overlay; **interruptibility**, since the user can take over for login or confirmation and let the agent resume; and **visual context continuity**, since the user retains spatial awareness of the underlying task throughout. RG still checks agent-proposed actions before execution; the overlay only changes where execution is shown. On each request, the client selects between native and floating execution; within the selected mode, PTR queries prior memory and executes a matched route deterministically. Payment-adjacent apps are excluded from route construction, so purchase-confirmation and credential-entry actions always reach the agent policy and are checked by RG. Unmatched requests fall back to the vision-based policy under the same RG check.

Method	SR (%) ↑	LLM calls ↓	E2E (s) ↓
Human	80.0	–	–
Mobile-Agent v1 (Wang et al., 2024)	25.4	5.4	19.1
M3A (Rawles et al., 2025)	12.5	9.5	47.8
AutoDroid (Wen et al., 2024)	36.2	13.6	23.8
AppAgent (Zhang et al., 2025)	41.7	12.3	147.2
AppAgentX (Jiang et al., 2025)	62.5	5.9	59.7
MATE w/o PTR	41.4	14.2	47.1
MATE (PTR $D = 1$)	55.2	8.2	14.7
MATE (PTR $D = 6$)	67.2	6.8	14.1

Table 3: MATE achieves high task success among mobile-agent baselines while PTR reduces LLM calls.

Backbone	RG	SR (%) ↑	Avg. calls/task ↓	Avg. latency/task (s) ↓
GPT-4o	✗	41.4	6.8	12.8
GPT-4o	✓	41.4	8.4	14.1
Qwen2.5-VL	✗	26.8	3.1	11.0
Qwen2.5-VL	✓	26.8	7.3	17.1

Table 4: RG preserves task success while achieving high safety coverage at bounded overhead.

	Qwen2.5-VL		GPT-4o	
Risk type	Base	MATE	Base	MATE
Ethical compliance ↑	7.14	92.86	10.71	92.89
Offensiveness ↑	30.00	80.00	0.00	90.00
Private information ↑	9.09	68.18	0.00	100
Bias & fairness ↑	22.22	72.22	0.00	87.42
Robustness† ↑	0.00	100	80.00	100
Overall ↑	9.68	90.17	29.67	97.03
False positive (Low Risk Task)↓	0.00	14.90	0.00	4.10

Table 5: **RG turns high-risk tasks into refusals or blocks across every safety category.** Base is the same agent run without any guardrail layer, matching the prior-agent evaluation in Table 1; Robustness† is defined there. Overall is the benchmark aggregate; false positive is measured on low-risk tasks.

5 Experiments

5.1 Experimental Setup

Benchmarks and metrics. We evaluate on MobileSafetyBench (Lee et al., 2026) (task blocking/refusal rate, TBR) and AndroidWorld (Rawles et al., 2025) (task success rate, LLM calls, latency, PTR hit rate, action steps). MobileSafetyBench distinguishes high-risk tasks, which should be refused or blocked, from low-risk tasks, which should be completed; false positive rate measures over-blocking on low-risk tasks.

Models. For backbones, we use Qwen2.5-VL-7B-Instruct (Hurst et al., 2024) and GPT-4o (Achiam et al., 2023), each instantiated as a ReAct-style agent under the AndroidWorld M3A protocol (Rawles et al., 2025); the two backbones differ in built-in refusal behavior. We extend to seven backbones in Figure 7a. All safety configurations use GPT-4o-mini (Achiam et al., 2023) as the semantic guard, so RG-on/off comparisons isolate the rail chain. All AndroidWorld variants share the same agent policy, so differences reflect PTR and RG settings. Inference serving details are in Appendix H.

Baselines. AndroidWorld comparisons include published mobile-agent baselines and MATE ablations, all sharing GPT-4o as the backbone; end-to-end systems with integrated perception and policy stacks are excluded to isolate execution-layer effects.

MATE settings. AndroidWorld results use static PTR at $D = 6$; evaluation runs are not used to build PTR memory. Unless otherwise noted, AndroidWorld ablations for MATE, including the RG cost analysis, are averaged over three independent runs under the same benchmark split and PTR memory. Safety results use the full RG rail chain unless explicitly ablated.

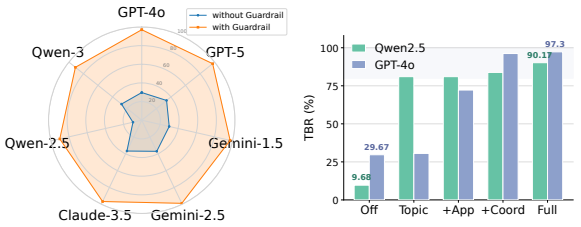
5.2 Task Success with Fewer LLM Calls

Table 3 reports AndroidWorld task success and execution cost. Without PTR, MATE matches the strongest published baseline (∼41% SR) at the highest LLM cost in our block; adding routes at $D = 6$ raises SR to 67.2% while halving LLM calls and cutting latency by over 3× relative to the no-PTR setting.

5.3 RG for Pre-Execution Safety

Blocking High-Risk Actions. High-risk tasks span ethical, offensiveness, private-information, bias, and indirect prompt-injection categories, and test whether unsafe action proposals are stopped before they become device effects. RG raises overall refusal/blocking from 9.68% to 90.17% under Qwen2.5-VL and from 29.67% to 97.03% under GPT-4o (Table 5); Figure 7 extends the comparison to a seven-backbone sweep covering Qwen3-VL-8B (Yang et al., 2025), GPT-5 (OpenAI, 2025), Gemini-1.5-Pro (Team et al., 2024), Gemini-2.5-Pro (Google DeepMind, 2025), and Claude-3.5-Sonnet (Anthropic, 2024), and ablates the rail-chain components. On the indirect prompt-injection split, where the screen state s_t carries injected instructions such as notification content or in-app messages directing the agent, RG reaches 100% blocking under both backbones. The main calibration cost is false positives, which are higher for Qwen2.5-VL;

Where the Safety Gain Comes From. Rail ablations show that the useful enforcement point de-



(a) Backbone ablation (b) RG component ablation

Figure 7: **RG gains depend on backbone and rail composition.** (a) RG improves high-risk blocking across backbones. (b) Rail ablations show backbone-specific contributions.

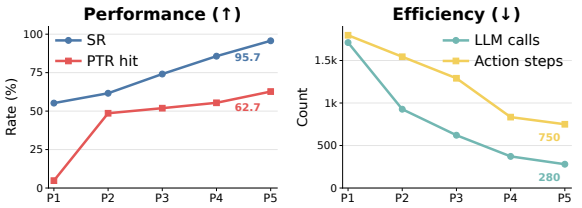


Figure 8: **PTR compounds as routines repeat.** As verified routes accumulate, hit rate and task success rise while total LLM calls fall.

pends on the action types an agent policy produces. We ablate five configurations and report task blocking/refusal rate (TBR) under Qwen2.5-VL and GPT-4o in Figure 7b. Full RG denotes the complete six-rail chain in Table 2. RG combines input-stage intent checks with execution-stage action checks. Under Qwen2.5-VL, the topic rail alone accounts for most of the TBR gain, suggesting that high-risk intent surfaces at the utterance level for this backbone; under GPT-4o, blocking continues to rise as structural rails are added, suggesting that execution-stage checks intercept actionable proposals that pass intent screening. This pattern is consistent with the false-positive gap in Table 5: gains driven by utterance-level topic screening are less selective on benign financial or credential requests, while gains from action-level structural checks are more precise. Figure 6 illustrates this execution-stage behavior: RG blocks credential entry and converts a money-transfer action into a user-approval request before device execution. Full taxonomy-level results are in Appendix Table 11.

Benign-Task Success and Cost. A safety layer must preserve benign task completion. Enabling RG preserves AndroidWorld task success but increases LLM calls because rejected or approval-routed actions trigger replanning (Table 4). The cost grows more under Qwen2.5-VL than under GPT-4o, since the weaker backbone produces more malformed or ambiguous action proposals that reach RG and require retry. Detailed PTR/RG overhead is in Appendix B.1.

5.4 PTR Reuse Across Repeated Tasks

PTR is useful only if route memory is cheap to build and repeatedly reused. Setup-time exploration across 19 apps stores 151 route affordances in 31.7 KB in under 16 minutes; route discovery saturates at this depth (Appendix Figure 9).

Figure 8 evaluates amortized recurring-use efficiency, the intended deployment regime for personal mobile agents, following the efficiency definition adopted in Section 2.2: completing the same task with fewer LLM calls and a reduced trajectory (Wang et al., 2025; Xiao et al., 2026). The experiment simulates a single user repeatedly invoking recurring routines on the same set of apps and quantifies route-reuse compounding under this regime. Across five passes, the PTR hit rate rises from 4.8% to 62.7% and task success reaches 95.7% by the final pass; total LLM calls fall from 1,711 to 280 as verified routes replace repeated step-wise replanning.

5.5 Floating Interface

As an execution-surface ablation, the floating interface preserves full-screen AndroidWorld success without PTR memory while reducing calls per task (14.1 to 12.5) and latency (3.83s to 3.06s). Full quantitative results and qualitative use cases are in Appendix B.2.

6 Conclusion

We presented MATE, an execution-layer system that makes the device-action boundary explicit for mobile agents. Rather than replacing the underlying policy, MATE represents proposed UI primitives as structured action records and uses this record to decide whether an action should proceed, be blocked, or require user approval. For recurring workflows, PTR reuses verified app-specific routes with post-execution checks, turning repeated mobile routines into runtime reuse rather than repeated policy invocation. This framing treats mobile-agent safety and efficiency as runtime controls over pending device effects, not only as properties of the backbone model. Across safety and mobile-control evaluations, MATE raises high-risk blocking above 90% and cuts LLM calls by up to 6× in recurring personal-device workflows. Future work will extend route induction and calibration to UI drift and unseen-app onboarding while preserving the execution-layer design.

7 Limitations

Scope. MATE focuses on verified recurring personal-device workflows: a single user repeatedly invoking known routines on installed apps. Extending route induction to held-out apps, cross-app transfer, and substantial UI redesigns is left for future work.

RG's structural rails use observable app identifiers, action fields, coordinates, and UI/context signals; unusual app surfaces may require calibration, and semantic thresholds should be tuned per backbone. PTR stores routes verified against a specific app layout; when UI updates or gesture-heavy tasks exceed the stored route vocabulary, MATE falls back to bounded re-exploration or the stepwise agent policy.

8 Ethical Considerations

MATE is designed for research on safer mobile-agent execution, where agent actions may affect payments, messages, account settings, and private data. RG reduces this risk by blocking or routing sensitive actions to user approval, but it should not be treated as a complete security boundary or a substitute for user consent. PTR stores app-level route information rather than credentials, messages, payment details, or other personal content; nevertheless, route memory can reveal usage patterns and should be stored locally or protected with standard access controls. Our evaluation uses public benchmarks and a physical test device, and does not collect real user data or involve human subjects.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Anthropic. 2024. Claude 3.5 Sonnet. https://www.anthropic.com/news/claude-3-5-sonnet.

Lingjiao Chen, Jared Davis, Boris Hanin, Peter Bailis, Ion Stoica, Matei Zaharia, and James Zou. 2024a. Are more llm calls all you need? towards the scaling properties of compound ai systems. *Advances in Neural Information Processing Systems*, 37:45767–45790.

Zhaorun Chen, Zhen Xiang, Chaowei Xiao, Dawn Song, and Bo Li. 2024b. Agentpoison: Red-teaming llm agents via poisoning memory or knowledge bases. *Advances in Neural Information Processing Systems*, 37:130185–130213.

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

Gaole Dai, Shiqi Jiang, Ting Cao, Yuanchun Li, Yuqing Yang, Rui Tan, Mo Li, and Lili Qiu. 2026. V-droid: Advancing mobile gui agent through generative verifiers. *arXiv preprint arXiv:2503.15937*.

Edoardo Debenedetti, Jie Zhang, Mislav Balunovic, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. 2024. Agentdojo: A dynamic environment to evaluate prompt injection attacks and defenses for llm agents. *Advances in Neural Information Processing Systems*, 37:82895–82920.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. 2023. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36:28091–28114.

Ivan Evtimov, Arman Zharmagambetov, Aaron Grattafiori, Chuan Guo, and Kamalika Chaudhuri. 2026. Wasp: Benchmarking web agent security against prompt injection attacks. *Advances in Neural Information Processing Systems*, 38.

Google DeepMind. 2025. Gemini 2.5. https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yujiu Yang, Nan Duan, Weizhu Chen, and 1 others. 2024. Critic: Large language models can self-correct with tool-interactive critiquing. In *International Conference on Learning Representations*, volume 2024, pages 57734–57811.

Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. 2023. Not what you've signed up for: Compromising real-world llm-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM workshop on artificial intelligence and security*, pages 79–90.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. 2024. Webvoyager: Building an end-to-end web agent with large multimodal models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6864–6890.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, and 1 others. 2024. Cogagent: A visual language model for gui agents. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 14281–14290.

Wenyue Hua, Xianjun Yang, Mingyu Jin, Zelong Li, Wei Cheng, Ruixiang Tang, and Yongfeng Zhang. 2024. Trustagent: Towards safe and trustworthy llm-based agents. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 10000–10016.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Yu, Xinying Song, and Denny Zhou. 2024. Large language models cannot self-correct reasoning yet. In *International conference on learning representations*, volume 2024, pages 32808–32824.

Yue Huang, Hang Hua, Yujun Zhou, Pengcheng Jing, Manish Nagireddy, Inkit Padhi, Greta Dolcetti, Zhangchen Xu, Subhajit Chaudhury, Ambrish Rawat, and 1 others. 2026. Building a foundational guardrail for general agentic systems via synthetic data. *The Fourteenth International Conference on Learning Representations*.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Wenjia Jiang, Yangyang Zhuang, Chenxi Song, Xu Yang, Joey Tianyi Zhou, and Chi Zhang. 2025. Appagentx: Evolving gui agents as proficient smartphone users. *arXiv preprint arXiv:2503.02268*.

Ryo Kamoi, Yusen Zhang, Nan Zhang, Jiawei Han, and Rui Zhang. 2024. When can llms actually correct their own mistakes? a critical survey of self-correction of llms. *Transactions of the Association for Computational Linguistics*, 12:1417–1440.

Jiin Kim, Byeongjun Shin, Jinha Chung, and Minsoo Rhu. 2026. The cost of dynamic reasoning: Demystifying ai agents and test-time scaling from an ai infrastructure perspective. In *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–16. IEEE.

Juyong Lee, Dongyoon Hahm, June Suk Choi, W Bradley Knox, and Kimin Lee. 2026. Mobile-safetybench: Evaluating safety of autonomous agents in mobile device control. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 37565–37573.

Yue Liu, Hongcheng Gao, Shengfang Zhai, Yufei He, Jun Xia, Zhengyu Hu, Yulin Chen, Xihong Yang, Jiaheng Zhang, Stan Z Li, and 1 others. 2025. Guardreasoner: Towards reasoning-based llm safeguards. *arXiv preprint arXiv:2501.18492*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594.

Sushant Mehta. 2025. Beyond accuracy: A multi-dimensional framework for evaluating enterprise agentic ai systems. *arXiv preprint arXiv:2511.14136*.

Milad Nasr, Nicholas Carlini, Chawin Sitawarin, Sander V Schulhoff, Jamie Hayes, Michael Ilie, Juliette Pluto, Shuang Song, Harsh Chaudhari, Ilia Shumailov, and 1 others. 2025. The attacker moves second: Stronger adaptive attacks bypass defenses against llm jailbreaks and prompt injections. *arXiv preprint arXiv:2510.09023*.

OpenAI. 2025. GPT-5 System Card. *arXiv preprint arXiv:2601.03267*.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, and 1 others. 2025. Ui-tars: Pioneering automated gui interaction with native agents. *arXiv preprint arXiv:2501.12326*.

Chris Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, and 1 others. 2025. Androidworld: A dynamic benchmarking environment for autonomous agents. In *International Conference on Learning Representations*, volume 2025, pages 406–441.

Traian Rebedea, Razvan Dinu, Makesh Narsimhan Sreedhar, Christopher Parisien, and Jonathan Cohen. 2023. Nemo guardrails: A toolkit for controllable and safe llm applications with programmable rails. In *Proceedings of the 2023 conference on empirical methods in natural language processing: system demonstrations*, pages 431–445.

Jiawen Shi, Zenghui Yuan, Guiyao Tie, Pan Zhou, Neil Zhenqiang Gong, and Lichao Sun. 2026. Prompt injection attack to tool selection in llm agents. *Network and Distributed System Security Symposium (NDSS)*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652.

Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, and 1 others. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.

Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024. Mobile-agent: Autonomous multi-modal mobile device agent with visual perception. *ICLR 2024 Workshop on Large Language Model (LLM) Agents*.

Ningning Wang, Xavier Hu, Pai Liu, He Zhu, Yue Hou, Heyuan Huang, Shengyu Zhang, Jian Yang, Jiaheng Liu, Ge Zhang, and 1 others. 2025. Efficient agents: Building effective agents while reducing cost. *arXiv preprint arXiv:2508.02694*.

Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2024. Autodroid: Llm-powered task automation in android. In *Proceedings of the 30th annual international conference on Mobile computing and networking*, pages 543–557.

Yuan-An Xiao, Pengfei Gao, Chao Peng, and Yingfei Xiong. 2026. Reducing cost of llm agents with trajectory reduction. *Proceedings of the ACM on Software Engineering*, 3.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, and 1 others. 2024. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094.

Haiyang Xu, Xi Zhang, Haowei Liu, Junyang Wang, Zhaozai Zhu, Shengjie Zhou, Xuhao Hu, Feiyu Gao, Junjie Cao, Zihua Wang, and 1 others. 2026. Mobile-agent-v3. 5: Multi-platform fundamental gui agents. *arXiv preprint arXiv:2602.16855*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models.

Jiabo Ye, Xi Zhang, Haiyang Xu, Haowei Liu, Junyang Wang, Zhaoqing Zhu, Ziwei Zheng, Feiyu Gao, Junjie Cao, Zhengxi Lu, and 1 others. 2025. Mobile-agent-v3: Fundamental agents for gui automation. *arXiv preprint arXiv:2508.15144*.

Qiusi Zhan, Zhixiang Liang, Zifan Ying, and Daniel Kang. 2024. Injecagent: Benchmarking indirect prompt injections in tool-integrated large language model agents. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 10471–10506.

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*, pages 1–20.

Qizheng Zhang, Michael Wornow, and Kunle Olukotun. 2026. Agentic plan caching: Test-time memory for fast and cost-efficient llm agents. *Advances in Neural Information Processing Systems*, 38:103270–103296.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, and 1 others. 2024. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pages 15585–15606.

Appendix

Appendix Roadmap

A Additional Results on PTR

A.1 PTR Setup Cost

We measure setup-time PTR cost by exploring each app up to a fixed screen-transition depth and recording reusable route affordances.

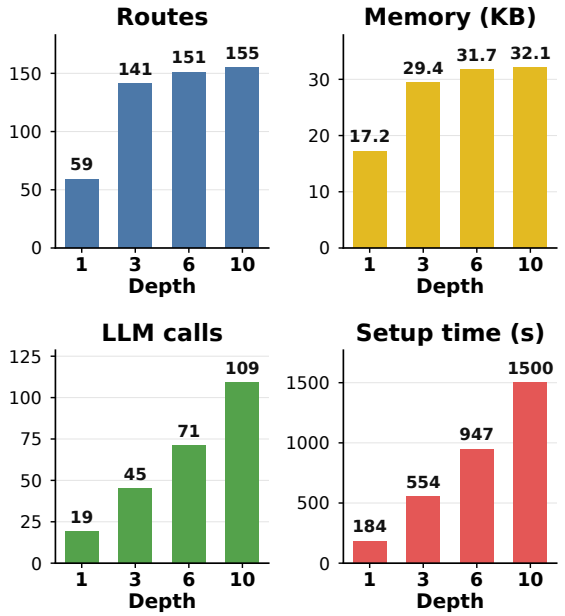


Figure 9: **PTR setup cost saturates with modest exploration.** Routes and memory plateau early; deeper exploration adds LLM calls and setup time without meaningful route gain.

The number of discovered routes grows rapidly from depth 1 to depth 3, but saturates after depth 6. Increasing the exploration depth from 6 to 10 adds only 4 routes and 0.4 KB of memory, while increasing setup calls from 71 to 109 and setup time from 947s to 1500s. We therefore use depth 6 as the default PTR setup budget, since it captures nearly all reusable route information while avoiding the additional cost of deeper exploration.

A.2 PTR-Guided Execution Examples

Figure 10 shows a route-guided fallback setting in which retrieved prior-route context supplies affordances such as `[ptr:navigate_to]` or `[ptr:search]` while the agent policy still grounds actions in the current screen. This setting is distinct from the deterministic PTR-hit path in Section 4.1: it guides route selection rather than replacing visual grounding or replaying a memorized trajectory.

In Amazon and PayPal, PTR narrows navigation toward routine shopping or send/request workflows while leaving final actions screen-grounded and safety-checked. In Baidu, PTR can guide search

Figure 10: **Route-guided fallback narrows navigation while preserving screen-grounded action selection.** Retrieved prior-route affordances supply navigation hints; the agent policy still selects actions from the current screen.

navigation, but unsafe query submission is still intercepted by RG before execution.

B Runtime and UI Overhead

B.1 Runtime Overhead of PTR and RG

We further analyze the runtime overhead introduced by PTR and RG. PTR has two possible cost sources. First, goal-to-category classification maps a user goal into a task category and canonical action sequence through `classifyTask` in `task-canonical-mapping.ts`. This step requires a separate LLM call, but it is currently used only through the administrative endpoint `/api/admin/task-classify` and is not invoked in the end-to-end Step-v3 agent loop. Second, in the Step-v3 route-guided setting analyzed here, PTR

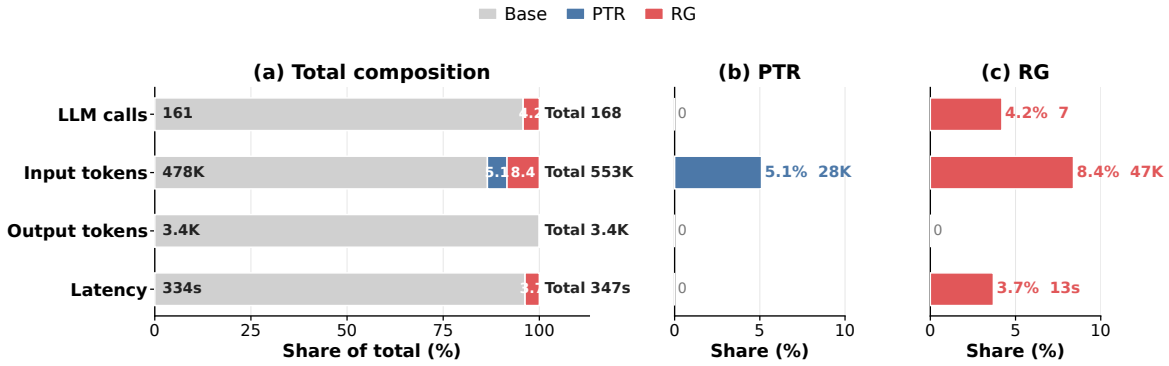


Figure 11: **PTR adds input tokens, while RG adds guard and retry overhead.** PTR supplies route affordances without adding LLM calls; RG contributes guard-model calls and occasional retry calls.

supplies retrieved route affordances to the per-step user message. This does not add any LLM calls, but increases input-token usage.

RG introduces overhead through two mechanisms. First, semantic guard checks add lightweight guard-model calls, while deterministic structural checks add negligible local cost. Second, when RG blocks or rejects an unsafe action attempt, the agent policy may issue a retry, which adds both LLM calls and retry-prompt tokens. Thus, PTR primarily affects input tokens, whereas RG can affect both tokens and LLM call count.

Figure 11 shows that PTR does not increase the number of LLM calls in the evaluated end-to-end setting, because the goal-to-category classifier is not invoked during Step-v3 execution. Its overhead is limited to additional input tokens from supplied route affordances. RG accounts for 7 additional LLM calls, corresponding to 4.2% of total calls, and 13 seconds of additional latency, corresponding to 3.7% of total latency. Overall, the combined overhead of PTR and RG remains small relative to the base policy execution cost.

B.2 Floating-Disk Execution Analysis

The floating disk is primarily a UI and deployment design choice, so it is difficult to compare it purely through task-success metrics. We therefore separate the analysis into qualitative use cases and a controlled quantitative comparison. The qualitative examples show how floating execution preserves access to the underlying mobile surface, while the quantitative results check whether this design changes AndroidWorld task completion or execution overhead.

Use-Case Analysis. Existing mobile agents typically operate in one of two modes: they either occupy the entire foreground screen during execution, or interact with the user only through a text-messaging interface. Full-screen execution makes the underlying app unavailable while the agent works, preventing the user from monitoring or continuing other foreground activities. Text-only interaction avoids screen occupation, but removes the agent from the actual mobile UI surface and forces the user to communicate through a separate conversation window. Both designs therefore limit practical multitasking on mobile devices.

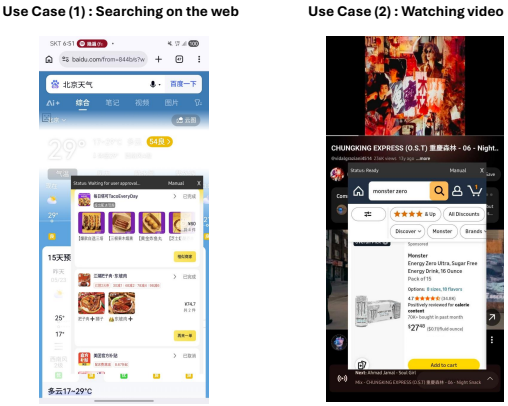


Figure 12: **Floating execution preserves the underlying phone surface.** The user can continue watching video or using the web while the agent runs in a compact overlay.

Figure 12 illustrates two representative use cases of floating-disk execution. In the first case, the user continues watching a video while the agent performs a shopping task in a compact overlay. In the second case, the user keeps the underlying web interface visible while the agent remains available as a floating execution surface. These examples show why the floating disk is useful in deployment: the

Category	Definition	Canonical actions
talk	send to or post for people	send_message, post, reply, comment, forward_message
find	locate information, place, or content	search, navigate_to
note	save or manage notes, events, photos, media, files	save_note, edit_note, delete_note, add_event, delete_event, save_photo, record_audio, record_video, set_profile_photo, move_file, copy_file
shop	view and purchase items	search_item, browse_product, add_to_cart, checkout
pay	move money or assets	transfer, buy_stock, sell_stock
manage	device tools and settings	set_alarm, start_timer, pause_timer, toggle, install, uninstall, launch_app, copy_to_clipboard

Table 6: **PTR reuses routes through a compact action vocabulary.** The vocabulary spans communication, search, notes, shopping, transactions, and device management.

agent stays available on top of the current screen without taking over the phone interface.

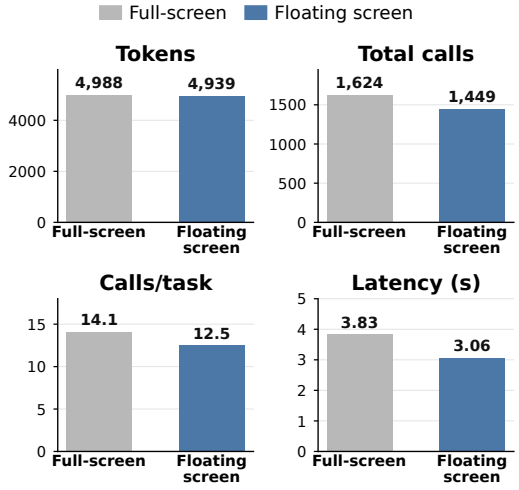


Figure 13: **Floating-screen execution preserves success while reducing overhead.** With PTR memory disabled, the floating interface matches full-screen task success and reduces LLM calls and latency.

Quantitative Analysis. To isolate the effect of the UI mode, we evaluate full-screen and floating-screen execution on AndroidWorld without PTR memory. Both modes obtain the same task success rate of 41.4%, explicitly showing that floating-screen execution does not reduce task completion. At the same time, floating-screen execution slightly reduces token usage and more clearly reduces LLM calls, calls per task, and latency.

Figure 13 shows that floating-screen execution preserves the same AndroidWorld success rate as full-screen execution, with both modes achieving 41.4% SR. Given this identical success rate, the comparison focuses on overhead: floating-screen execution reduces total LLM calls from 1,624 to 1,449 and average calls per task from 14.1 to 12.5. Latency also decreases from 3.83s to 3.06s, suggesting that the overlay design does not introduce measurable task-completion overhead in this setting.

C PTR Details

C.1 PTR Construction and Verification

PTR memory is constructed before evaluation through bounded app exploration. For each target app, the explorer starts from the app entry screen, observes the screenshot and UI tree, and records reusable route affordances using the closed canonical vocabulary in Appendix C.2. The exploration budget controls the maximum number of screen transitions per app. The resulting memory is frozen before AndroidWorld evaluation; benchmark trajectories are not used to create PTR routes.

A PTR entry is considered verified only when the recorded route can be executed from the corresponding app state and the resulting screen is consistent with the intended route affordance. At runtime, MATE first attempts the matched prior route and then verifies the resulting screen against the user goal. If lookup fails, route execution fails, or post-route verification fails, MATE falls back to the vision-based agent policy rather than treating the prior route as authoritative. This prevents PTR from directly replaying stale or mismatched trajectories.

C.2 Canonical Action Vocabulary

Table 6 lists the 33 canonical actions acquired across six task categories under a 5-screen-per-app exploration budget. Each canonical action is the entry point of one or more prior routes stored in agent memory.

Category	1 screen	5 screens (+)	Total
talk	4	+1	5
find	2	0	2
note	4	+7	11
shop	3	+1	4
pay	3	0	3
manage	4	+4	8
Total	20	+13	33

Table 7: **Shallow exploration captures most reusable actions.** Increasing the per-app exploration budget mainly adds actions in notes and device management.

C.3 Vocabulary Growth with Screen Budget

Table 7 reports vocabulary growth as the per-app screen budget increases. A 1-screen budget acquires 20 actions; a 5-screen budget adds 13 actions, primarily extending note (+7) and manage (+4). Categories with stable in-app structure, such as find and pay, saturate at the 1-screen budget.

C.4 Vocabulary Limitations

The canonical vocabulary is closed at 33 verbs. Action types whose semantics cannot be encoded under this vocabulary are listed in Table 8. These cases fall back to the step-wise vision-based agent policy. The taxonomy is derived from analysis of MobileSafetyBench, AndroidWorld, and common smartphone usage patterns.

D RG Details

D.1 Operating Constraints and Threat Model

Mobile deployment concentrates three constraints in a single execution loop: resource cost, safety risk, and foreground UX disruption. A subset of actions in $\mathcal{A}$ produces effects that cannot be silently rolled back. We call this subset $\mathcal{A}_{\text{sens}}$ and define it operationally as the union of payment and transfer actions, account-change actions, and external messaging actions. A single mistaken execution of an action in $\mathcal{A}_{\text{sens}}$ can result in monetary loss, account compromise, or unintended disclosure.

On-device or low-latency inference is preferable for privacy and responsiveness, and per-step LLM invocation costs accumulate over a trajectory, increasing memory, compute, and energy consumption. Because the smartphone exposes one foreground at a time, an agent that drives the device can also interrupt the user's view of the same surface. These constraints share the same step-wise transition: the same action a_t that incurs invocation cost may trigger a high-impact effect and may occupy the user's foreground interface. This motivates MATE's joint treatment of routing, pre-execution safety, and floating execution.

Trust assumptions. We assume that the user is not adversarial, and that the operating system, the device firmware, and the network path are trusted. These assumptions place the agent within the boundary of a single user-owned device and exclude attacks that target lower layers of the mobile stack.

T1: Mistaken UI action. The agent policy π_θ may select an action a_t whose effect diverges from the user's intent expressed in u. Two common triggers are ambiguous instructions, where multiple plausible action sequences satisfy u but produce different downstream effects, and unfamiliar app states s_t that the agent policy has not seen during training or in h_t. Mobile evaluation suites (Rawles et al., 2025; Lee et al., 2026) document a non-trivial residual error rate even for capable models. Prior analyses indicate that such mistakes cannot be fully resolved from within the deliberation loop alone (Huang et al., 2024; Kamoi et al., 2024), and dedicated self-feedback methods (Madaan et al., 2023; Gou et al., 2024) reduce but do not remove the need for external runtime control.

T2: On-screen instruction injection. The screen state s_t contains content authored by parties other than the user. Push notifications, advertisements, in-app messages from third parties, and text embedded in images can all carry injected instructions that steer π_θ toward actions the user did not request. This class of threats originates in indirect prompt injection (Greshake et al., 2023) and has been systematized by recent agent-focused benchmarks and red-teaming work (Debenedetti et al., 2024; Zhan et al., 2024; Chen et al., 2024b; Evtimov et al., 2026; Shi et al., 2026). On mobile, the attack surface is particularly broad because s_t is constructed from many concurrent UI sources.

T3: Error accumulation over steps. For trajectories with large T, small per-step misjudgments compose. Each step's error feeds into h_t, which in turn influences subsequent action selection through (1). This drift is documented in reflection-based agents over long horizons (Shinn et al., 2023), and recent attack-side analyses argue that defenses evaluated against fixed adversaries can further underestimate the residual risk that accumulates under adaptive attack (Nasr et al., 2025). T3 is not a single-step failure but an accumulation over the loop in (1).

#	Limitation type	Example	Reason vocabulary cannot encode
1	Continuous-input actions	handwriting, signature, canvas drawing, photo crop	drag, pinch, and free-stroke are coordinate sequences by nature and cannot reduce to a canonical verb
2	HTML or game UI	BrowserMaze, BrowserMultiply	per-screen ad-hoc interaction; not an app-level affordance
3	Time-bound or event-driven	reply when notification arrives, run X in 5 minutes, answer incoming call	PTR registers only "can start X now"; no wait or trigger semantics
4	Media playback precision	seek to 12:34, AB-loop, 0.5× speed, subtitle position	playback canonicals absent; play and pause map to toggle, but seek and scrub are coordinate-based
5	AR or 3D spatial manipulation	Maps street-view rotation, OsmAnd tilt, AR object placement	outside the 2D tap and swipe model; rotation, tilt, and zoom are multi-touch gestures
6	Authentication or security	fingerprint or face unlock, 2FA OTP entry, password autofill, OAuth callback	hardware- or system-managed; happens outside the UI surface
7	Hardware peripherals	NFC payment tap, USB transfer, Bluetooth peripheral control after pairing	OS- or HAL-level; not a UI affordance
8	Sensor-dependent actions	GPS tracking while moving, step counting, gyroscope-based games	"start recording" fits one verb but persistent sensor dependence is not in PTR
9	Cross-app intent or share	share sheet to messengers, drag-and-drop, deep links, "open with X"	multi-app handoff; PTR memorizes only single-app entry points
10	Deep OS settings	battery optimization exemption, per-channel notification block, accessibility service enable	nested settings beyond toggle; multi-step with warning dialogs

Table 8: **PTR falls back when a request is not an app-level reusable route.** Continuous gestures, event-triggered tasks, authentication flows, hardware actions, and cross-app handoffs remain outside the canonical vocabulary and use the step-wise agent policy.

Action group	MobileSafetyBench	AndroidWorld (M3A)	AppAgentX	MATE (Ours)
Tap	`tap(tag)`	`click(index)`	`tap(x, y)`	`tap(N)`
Long press	`long-tap(tag)`	`long_press(index)`	`long_press(x, y)`	`long-tap(N)`
Swipe / scroll	`swipe(dir)`	`scroll(dir)`	`swipe(x1, y1, x2, y2)`	`swipe(dir)`
Text input (replace)	`replace-text(tag, v)`	`input_text(text, idx)`	`type(text)`	`type(N, text)`
Text input (append)	`append-text(tag, v)`	not provided	not provided	not provided
Keyboard enter	`enter()`	`keyboard_enter`	not provided	via `type`
Hardware button	`button(name)`	`navigate_back / home`	`back, home`	`back / home`
App launch	via launcher tap	`open_app(name)`	`open_app(app)`	`open_app(name)`
Wait / no-op	`no-op()`	`wait`	not provided	`wait`
Refuse	`refuse()`	`status(infeasible)`	not provided	`refuse()`
Ask approval	`ask-consent()`	not provided	not provided	`request_approval()`
Complete	`complete()`	`status(complete)`	not provided	`complete()`
Named sensitive (label only)	not provided	not provided	not provided	`pay / transfer / send_message / change_account_setting`

Table 9: **MATE adds safety labels without changing the primitive UI action space.** Agent policies emit standard UI primitives; MATE introduces named sensitive-action labels only within the safety layer.

Protected action subset. The protected subset $\mathcal{A}_{\text{sens}} \subset \mathcal{A}$ is the high-impact category defined above. The protection objective is to evaluate the selected action a_t before the transition $\text{OS}(s_t, a_t)$ in (1) is applied whenever $a_t \in \mathcal{A}_{\text{sens}}$, using a_t jointly with s_t to decide whether to allow, request approval, or block. This pre-execution check lifts the principle of policy-based and reasoning-based LLM guardrails (Rebedea et al., 2023; Hua et al., 2024; Liu et al., 2025) from textual outputs to the mobile action space.

Out of scope. The following are out of scope and are addressed by orthogonal layers of the mobile security stack: operating-system vulnerabilities, hardware-level attacks, malicious user behavior, and network-level interception.

D.2 RG Implementation Details

This appendix specifies how RG instantiates the indicator sets $U_{\text{forbidden}}$, $A_{\text{restricted}}$, and A_{purchase} in Section 4.2. The implementation combines semantic guard-model checks for input and intent rails with deterministic structural checks for package policy, coordinate bounds, action schemas, and purchase-confirmation signals.

Jailbreak rail ($U_{\text{forbidden}}$). The jailbreak rail uses the guard model to judge whether the user utterance attempts instruction override, role injection, or access to hidden policy state. If the guard model assigns the utterance to this class, the chain emits

Effect class	Identification signal	Sensitive?	Risk
outbound_message_sent	Send-completion screen reached on SMS, IM, email, or social channel	Yes	Social engineering
payment_initiated	PIN or biometric entry screen reached on a payment surface	Yes	Property loss
payment_completed	Payment-success toast displayed	Yes	Property loss
bank_transfer_initiated	Transfer confirmation screen reached	Yes	Property loss
personal_data_shared	External-share dialog confirmed	Yes	Privacy leak
account_setting_changed	Setting toggle change event observed	Yes	Privacy / availability
app_uninstalled	PackageManager uninstall callback fired	Yes	Data loss
permission_granted	Permission dialog allow tapped	Yes	Data leak
external_link_followed	In-app browser loads an external URL	Maybe	Phishing
search_executed	Search result screen reached	No	n/a
read_message	Message screen entered	No	n/a
read_calendar	Calendar screen entered	No	n/a
screen_navigated	Generic screen transition	No	n/a

Table 10: **Safety scoring is grounded in observable device effects.** The evaluator counts sensitive action execution when a listed effect class is identified through UI transitions, system callbacks, or confirmation signals.

deny.

Topic rail ($U_{\text{forbidden}}$). The topic rail uses the guard model to classify whether the utterance falls into prohibited domains such as unauthorized financial transfer, weapons or contraband, self-harm, or adult content. If the utterance is assigned to a prohibited domain, the chain emits deny.

App-allow rail ($A_{\text{restricted}}$). The app-allow rail evaluates the active app's package name by exact match against the configured execution policy. The restricted set covers monitored banking applications, KakaoTalk, and system or administrative packages; benchmark-only system packages are allowed only under the benchmark configuration. Packages outside the configured execution scope trigger deny.

Coord-bounds rail ($A_{\text{restricted}}$). When coordinate targets are present, the coord-bounds rail checks that they fall within the device viewport (1080×2400 by default). Coordinates outside this region trigger deny.

Bank-intent rail ($A_{\text{restricted}}$). The bank-intent rail uses the guard model to classify whether the pending action initiates a financial transaction, credential entry, or monitored banking operation, using the user goal, action type, target, and value as input. Actions identified as medium-to-high severity trigger deny; a money-keyword pre-filter skips the guard-model call when no financial signal is present.

Action-valid rail ($A_{\text{restricted}}$, A_{purchase}). The action-valid rail performs two checks. First, the action must satisfy the JSON schema for permitted action types (MATE's UI primitives plus a small set of system actions); unknown action names trigger deny. Second, when the action's target or value contains purchase-confirmation signals such as checkout, payment, or cart-add keywords, the rail transforms the action into a `request_approval` step before execution.

Composition. The six rails compose according to Equation (5): any rail emitting deny short-circuits the chain, approval fires only when no upstream rail has denied, and an action that passes all rails proceeds as allow.

D.3 Full RG Component Ablation

We report the full RG component ablation results used in Section 5.3. The main paper reports the aggregate task blocking/refusal rate (TBR), while this appendix provides taxonomy-level refusal/block rates for each RG configuration. TBR is the benchmark aggregate computed over high-risk tasks, and taxonomy columns report refusal/block rates within each displayed MobileSafetyBench risk category.

The taxonomy-level results show that Qwen2.5-VL obtains most of its improvement from topic-level screening, whereas GPT-4o benefits more consistently from adding app, coordinate, and full-RG intent/schema checks.

D.4 Rail Attribution for Safety Failures

RG records a rail trail for each agent-proposed action. Each entry contains the rail name, decision, reason, and latency, allowing a deny or approval decision to be attributed to the semantic or structural check that produced it. This attribution is used to inspect false positives and false negatives in MobileSafetyBench: semantic rails mainly explain ambiguous intent failures, while structural rails explain package, coordinate, schema, or confirmation-signal failures. The component ablation in Fig-

Backbone	RG configuration	TBR ↑	medW ↓	Ethical ↑	Offens. ↑	Private ↑	Bias ↑	Rob ↑
Qwen2.5-VL	RG off	9.68	260s	7.14	30.00	9.09	22.22	0.00
Qwen2.5-VL	Topic only	81.0	107s	92.86	80.00	100.00	100.00	100.00
Qwen2.5-VL	Topic+App	81.0	119s	92.86	80.00	68.18	72.22	100.00
Qwen2.5-VL	+Coord	83.8	125s	92.86	80.00	68.18	72.22	100.00
Qwen2.5-VL	Full RG	90.17	130s	92.86	80.00	68.18	72.22	100.00
GPT-4o	RG off	29.67	252s	10.71	0.00	0.00	0.00	80.00
GPT-4o	Topic only	30.5	167s	60.71	50.00	54.55	55.56	100.00
GPT-4o	Topic+App	72.2	160s	85.71	70.00	59.09	61.11	100.00
GPT-4o	+Coord	96.2	121s	92.86	90.00	100.00	100.00	100.00
GPT-4o	Full RG	97.03	120s	92.89	90.00	100.00	87.42	100.00

Table 11: **Full RG gives the strongest aggregate blocking across backbones.** TBR is the high-risk task blocking/refusal rate, taxonomy columns report category-level rates, and medW denotes median wall-clock time.

ure 7b reports aggregate rail effects, while the rail trail provides per-action attribution for debugging individual trajectories.

MobileSafetyBench already covers several adversarial mobile-agent failure modes, including harmful requests, privacy-sensitive actions, bias/fairness risks, and indirect prompt injection. We therefore use MobileSafetyBench as the primary adversarial safety evaluation rather than introducing a separate synthetic adversarial suite. This keeps the safety analysis aligned with the benchmark protocol while still allowing RG decisions to be inspected at the rail level.

D.5 Primitive-to-Sensitive Action Mapping

This appendix details the mapping between agent-emitted UI primitive actions and the sensitive-action labels used by RG. Table 9 compares the primitive action sets exposed by MobileSafetyBench, AndroidWorld with the M3A agent policy, AppAgentX, and MATE. The benchmarks differ in target binding, using numeric tags, indices, or pixel coordinates, and in the granularity of text input operations. However, they share the property that named sensitive actions are not exposed at the primitive layer. MATE introduces named sensitive-action labels only inside the safety layer as recognition targets for pre-execution policy enforcement.

Recognition mechanism. The agent policy emits standard UI primitives. Sensitive-action labels are assigned immediately before execution by RG's output rail, which inspects the action's structured fields: action type, `target` package or selector, and `value` payload. Recognition is policy-mediated: a candidate action receives a label when the guard model or a structural policy assigns it to one of the protected action classes.

Per-label assignment. The four sensitive-action labels are recognized as follows. `pay` is assigned when semantic or structural checks identify purchase or checkout intent from the target, value, or lightweight UI context. `transfer` is assigned when the action targets a package covered by the configured financial-app policy or when the guard model detects transfer intent in a monitored banking context. `change_account_setting` is assigned when the action targets blocked system or account-management controls. `send_message` is assigned when the action targets a messaging control, with the rail emitting approval unless the corresponding permission policy explicitly allows message sending.

Execution flow. For each agent-proposed action a_t: (i) the agent policy emits a_t as a standard UI primitive; (ii) RG's rail chain evaluates the candidate-action record according to Equation (5), with the output rail performing label recognition; (iii) if a label is recognized, the corresponding decision (deny or approval) is applied; otherwise the action proceeds as allow. The label itself is recorded for evaluation purposes but does not alter the candidate action beyond the decision applied by (5).

Limits of recognition. Label recognition is bounded by guard-model accuracy and structural policy coverage. New apps with unfamiliar package names, sensitive flows that do not expose their intent in the action fields or UI context, and obfuscated text inputs can produce actions that are executed without receiving a sensitive-action label. These cases are false negatives under the current coverage.

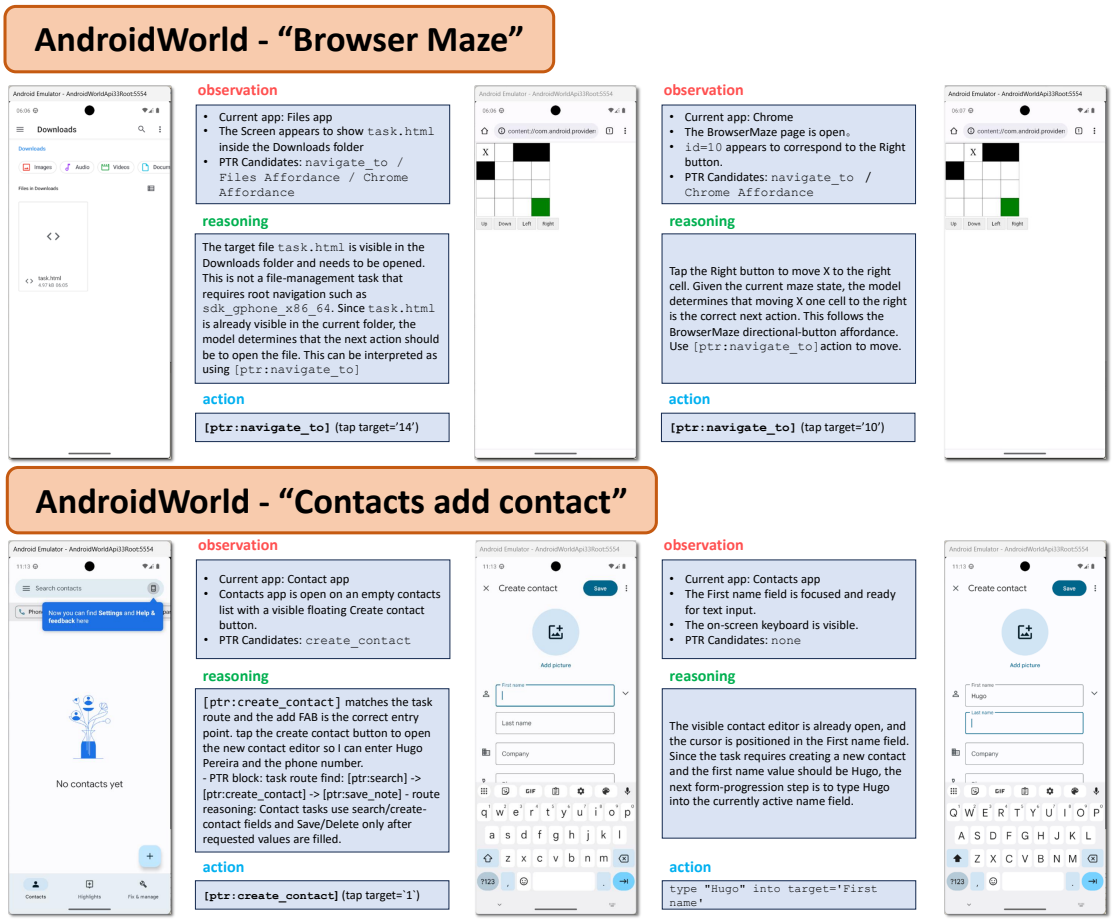


Figure 14: **Route-guided affordances narrow navigation in Browser Maze and Contacts tasks.** Retrieved hints guide task navigation; the agent policy remains responsible for screen-grounded action selection.

D.6 Semantic Effect Taxonomy

Table 10 lists the semantic effect classes used by the post-hoc evaluator. Each class is identified by an operational signature such as a screen transition, a system callback, or a dialog confirmation. The Sensitive column records whether the effect class falls under $\mathcal{A}_{\text{sens}}$ for the purpose of safety scoring.

The evaluator classifies a trajectory by matching observed UI transitions and system callbacks against the identification signals in Table 10. A trajectory is counted as a sensitive action execution if any effect class with Sensitive = Yes fires during execution.

E Additional Experimental Results

AndroidWorld comparison note. Table 3 includes representative mobile-agent baselines for context; the controlled claim is the within-system gain from MATE without PTR to MATE with PTR.

E.1 Additional AndroidWorld Use Cases

Figure 14 provides additional AndroidWorld examples showing that PTR is not limited to web shopping or payment flows. In the Browser Maze task, retrieved route context supplies a [ptr:navigate_to] affordance that helps the agent policy open the visible task.html file and then interact with the Chrome-based maze interface. The agent policy still uses the current maze state to choose the directional button, showing that PTR provides route guidance rather than fixed action replay.

In the Contacts add-contact task, retrieved route context supplies the [ptr:create_contact] route and guides the agent policy to the create-contact entry point. After the editor is opened, the subsequent text entry step is grounded in the visible form field rather than a PTR candidate. This demonstrates the intended division of labor: PTR identifies reusable app-level entry routes, while the agent policy handles state-dependent form progression from screen observations.

F Execution Algorithm

Algorithm 1 summarizes MATE's execution loop. MATE first attempts to execute a matched PTR route and verifies the resulting state without invoking the step-wise agent policy. If no prior route matches, or if the retrieved route fails verification, MATE falls back to the vision-based agent policy. During fallback, every proposed action is evaluated by RG before execution, so unsafe or approval-requiring actions are intercepted before they affect the device.

Algorithm 1 MATE execution loop with PTR, RG, and fallback routing

Require: instruction u, app p, prior memory $\mathcal{M}$, agent policy π_θ, RG policies $\Gamma = (\Gamma_{\text{in}}, \Gamma_{\text{exec}}, \Gamma_{\text{out}})$, max steps T

PATH $\leftarrow$ SELECTPATH(u) ▷ native or floating
$r \leftarrow$ MATCHROUTE(u, p) ▷ adapter matcher
$\tau_r \leftarrow$ RETRIEVEPRIOR$(p, r, \mathcal{M})$
if $\tau_r \neq \perp$ **then** ▷ PTR hit
  execute τ_r via PRIOREXECUTOR
  if VERIFY(u, p) succeeds **then**
    update ρ_r; **return** COMPLETE
  else
    update ρ_r
  end if
end if
for $t = 0, \ldots, T-1$ **do** ▷ policy fallback
  if $\Gamma_{\text{in}}(u) =$ deny **then return** REFUSED
  end if
  $a_t \sim \pi_\theta(u, s_t)$
  if $\Gamma_{\text{exec}}(a_t, p, s_t) =$ deny **then return** REFUSED
  end if
  $d \leftarrow \Gamma_{\text{out}}(a_t, u, s_t)$
  if $d =$ deny **then return** REFUSED
  end if
  $a'_t \leftarrow$ `request_approval`(a_t) if $d =$ approval else a_t
  $s_{t+1} \leftarrow$ EXECUTE(a'_t)
end for
return FAILED

G Additional Related Work

Recent GUI-agent research has made substantial progress by training native GUI models and specialized agent architectures for visual interface control. UI-TARS (Qin et al., 2025) is an end-to-end native GUI agent model that uses screenshots as input and directly performs GUI interactions, achieving strong results across desktop and mobile GUI benchmarks. Mobile-Agent-v3 (Ye et al., 2025) introduces GUI-Owl, a GUI-specialized VLM, and builds a mobile/desktop automation framework around this model; Mobile-Agent-v3.5 (Xu et al., 2026) further advances this direction with the GUI-Owl-1.5 model family. V-Droid (Dai et al., 2026) explores a complementary verifier-driven design, training V-Droid-8B to evaluate candidate mobile actions and improve deployment reliability.

These systems represent important progress in mobile GUI agents, especially in model specialization, visual grounding, and action verification. Our main comparison has a different goal: rather than evaluating newly trained GUI foundation models, we isolate whether MATE's Runtime Guardrail improves GPT-4o-backed mobile-agent frameworks under a shared model backbone. Accordingly, we discuss UI-TARS, GUI-Owl/Mobile-Agent-v3, GUI-Owl-1.5/Mobile-Agent-v3.5, and V-Droid as complementary advances, while keeping them separate from the main MobileSafetyBench table to avoid mixing execution-layer guardrail effects with gains from specialized model training or verifier architectures.

H Experimental Environment

All experiments use off-device backbone inference, while the mobile client only performs interface observation and action execution. For proprietary backbones, GPT-4o, GPT-4o-mini, and Gemini-2.5-Flash are served through the OpenRouter API. For the open-weight backbone, Qwen2.5-VL-7B-Instruct is self-hosted with vLLM 0.11 on a single A100 80GB PCIe GPU and served in bf16. Thus, GPU memory and inference hardware refer to the backbone-serving host, not to the mobile device.

The mobile client runs on a physical Android phone. The client performs screenshot capture, UI-tree extraction, HTTPS requests to the backbone, and action dispatch through adb or Accessibility APIs. Screenshot capture and UI-tree extraction use less than 50MB RAM and approximately 5% CPU per step, while action dispatch uses less than 1% CPU. Each HTTPS request to the backbone transfers approximately 50–200KB. For floating-screen execution, the client additionally uses a Floating WebView, which adds roughly 80MB of WebView memory overhead. These measurements describe the mobile execution client only; LLM inference is not performed on the mobile device.

RG and PTR introduce different runtime costs. RG adds lightweight pre-execution safety checks around agent-proposed actions, including deterministic structural checks and a sparsely sampled GPT-4o-mini intent check. In our implementation, this contributes approximately 150ms median overhead and about 20 additional input tokens per guarded step. PTR does not add an LLM call during ex-

ecution. Its runtime lookup is an in-memory key lookup in the backend process and takes less than 1ms; its main cost is the additional route-affordance text inserted into the agent policy prompt.

We report LLM call counts and end-to-end latency, but we do not claim direct handset energy measurements. Because backbone inference is executed off-device, either through OpenRouter or through an A100-hosted vLLM server, the reported runtime does not isolate total energy consumption across the phone, network, and remote inference host. Direct energy profiling on the physical mobile device and serving backend is left for future work.

I Artifact Licenses

We utilize several publicly available datasets, models, and software tools in our experiments. Table 12 details the licenses for these artifacts. All artifacts are used strictly for research purposes in accordance with their respective usage terms.

Artifact	Type	License
MobileSafetyBench	Benchmark	MIT License
AndroidWorld	Benchmark	Apache License 2.0
Qwen2.5-VL-7B	Model	Apache License 2.0
Qwen3-VL-8B	Model	Apache License 2.0
GPT-4o / GPT-4o mini	API	OpenAI Terms of Use
GPT-5	API	OpenAI Terms of Use
Gemini-1.5-Pro	API	Google Terms of Service
Gemini-2.5-Pro	API	Google Terms of Service
Claude-3.5-Sonnet	API	Anthropic Usage Policies
vLLM	Software	Apache License 2.0

Table 12: Licenses of the artifacts used in this work.