PHONECLI: FROM APP INTERFACES TO CALLABLE COMMANDS FOR MOBILE AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Mobile GUI agents operate through a perception–action loop: at each step they screenshot the device, invoke a vision–language model (VLM), and emit an action. It is slow, costly, and brittle, yet most of what it does is navigation—and everyday navigation is static, ordered, and endlessly repeated. We present PhoneCLI, which compiles an app's GUI navigation into callable commands, without any app-internal API, runtime instrumentation, or model training. Offline, PhoneCLI explores a target app from the outside and distills its screens, interactive elements, and navigation edges into a semantically annotated map; each screen yields one deterministic command: a replay sequence that reaches it. Online, the agent selects a command, verifies it before execution, and then executes it deterministically in sub-second time at zero VLM cost; open-ended interaction and every failure of the compiled path fall back to the embedded VLM interpreter, exactly the pure VLM agent, so compilation can only help. On AndroidLab, PhoneCLI improves the task success rate while reducing steps and token consumption, and it transfers to AndroidWorld's official M3A agent with consistent efficiency gains. What PhoneCLI compiles is the app's navigation rather than one run, so it serves new tasks, not only repeated ones. We have made our PhoneCLI available anonymously at: https://anonymous.4open.science/r/PhoneCLI-A23D/.

1 INTRODUCTION

GUI agents for mobile devices largely operate through a perception–action loop: at each step, the agent captures the screen, invokes a vision–language model (VLM) to reason over the visual information, and emits an action at either the index level or in coordinate form. The agent's competence is acquired through training on GUI corpora and/or via fine-tuning and reinforcement learning (Liu et al., 2024; Xu et al., 2026; Li et al., 2025; Zhang et al., 2024). This paradigm is confronted with three compounding challenges.

(I) Closedness. Desktop computer-using agents can operate via rich programmatic surfaces—*e.g.*, CLI tools, APIs, or even executable code (Song et al., 2026; Yan et al., 2025; Abhyankar et al., 2026)—because the desktop ecosystem provides open interfaces. By contrast, mobile apps offer no comparable privilege. They function as black boxes with no client-side, third-party programmatic interface, and even modern mobile frameworks typically only allow composition with system-level commands (Li et al., 2026), leaving the internal state and logic of each app inaccessible. Consequently, the agent is restricted to a screenshot-based loop, which is slow (on the order of seconds per step), costly (requiring one API call per step), and brittle, often leading to hallucinated coordinates and navigation failures.

(II) Repetitiveness. Nevertheless, the navigation induced by the loop targets interfaces whose structure is largely known a priori. Everyday mobile app usage is highly repetitive and follows consistent, ordered patterns (Li et al., 2020; Peters et al., 2024): for instance, opening a settings page, selecting a tab, and locating a specific item. Such behavior typically traces a small set of static paths repeatedly. Unlike free-form interaction, the navigation structure can therefore be enumerated offline. This motivates the core idea of this work: **pre-build the repeated navigation once and replay it deterministically thereafter.**

(III) Generalization. The alternative route—training—stores capability in the model weights. However, each new app typically requires additional data collection and/or re-training, which exposes

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

a major bottleneck studied in recent work on RL generalization (Gu et al., 2026) and on adaptation to previously unseen apps (Guo et al., 2026). Re-training is both costly and inflexible: a model that has never encountered an app has no direct shortcut for navigating it. Given the scale of modern app stores, with millions of apps, per-app training is infeasible in principle.

Existing remedies either improve the loop from within—by pairing stronger VLMs with grounding and memory scaffolding (Wang et al., 2024; Jiang & Huang, 2026)—or incur the training cost to sharpen the agent (Xu et al., 2026). However, neither approach eliminates the repeated navigation bottleneck itself. We propose **PhoneCLI**, which takes a third route: **compile an app's GUI navigation into callable commands**. **Offline**, PhoneCLI explores each target app once from the outside, without any app-internal API and without instrumentation. It distills the app's screens, UI elements, and navigation edges into a structured map, and compiles each screen into a deterministic command—*i.e.*, a replayable action sequence that reliably reaches that screen. This offline procedure does not involve training or model updates, so integrating a new app takes only minutes. **Online**, the agent maps each task to a command, verifies the selection, and replays it deterministically in sub-second time with zero VLM calls. Only open-ended interactions not covered by the command table are handled by the VLM, which serves as a graceful fallback.

Recent work shares parts of this picture. PreAct (Li, 2026) compiles agent trajectories online to replay the same task. UI-KOBE (Chai et al., 2026) enables runtime following of an explored UI graph. Trajectory-mining approaches recycle explored paths back into the VLM as textual hints (Xie et al., 2025; Sun et al., 2025), meaning that reuse still incurs perception costs. PhoneCLI differs along the decisive axis: it compiles offline, at the granularity of navigation commands, and then executes them deterministically. As a result, the compiled knowledge supports unseen tasks rather than only repeated ones, and its reuse requires no perception at runtime. In short, PreAct compiles what an agent has done, whereas PhoneCLI compiles what the app is. The detailed discussion of related work is in Appendix A.1. We discuss our contributions are threefold:

- **GUI-to-CLI compilation.** We present a compilation pipeline that transforms a black-box app—lacking any app-internal API, instrumentation, and training—into a structured app map and a catalog of deterministic callable commands, one per screen. This pipeline provides the agent with a programmatic surface that the app itself does not expose, enabling new apps to be integrated within minutes without any model update.
- **Reliable command invocation.** We design an invocation mechanism based on two-phase routing that selects and verifies a command prior to execution. Deterministic replay places the agent in the target screen in sub-second time at zero VLM cost, while any failure falls back gracefully to the embedded VLM interpreter. By removing the most frequently repeated aspect of mobile app usage—navigation—from the VLM loop, PhoneCLI preserves the performance of its backbone without degradation.
- **Dual-benchmark evidence.** On AndroidLab (Xu et al., 2025b), PhoneCLI achieves state-of-the-art performance with zero training, while reducing the step and token cost of successful tasks. On AndroidWorld (Rawles et al., 2025), it transfers to the official agent with measurable efficiency gains. A controlled three-way ablation further attributes the improvements to the map information and deterministic replay, independently.

2 METHODOLOGY

2.1 DESIGN OVERVIEW

Motivation and design rationale. As detailed in Section 1, our design rests on three observations: **(I) closedness**, mobile apps expose no programmatic surface, unlike the CLI tools and APIs available on desktop, which confines mobile agents to a slow, costly, and brittle screenshot-perception loop; **(II) repetitiveness**, everyday app usage is highly repetitive and ordered, and the navigation skeleton that connects screens changes far more slowly than app content, so a path distilled once is replayed many times; and **(III) generalization**, training-based agents are prone to distribution shift on unseen apps and costly to re-train, whereas on-demand exploration integrates a new app in minutes.

This motivates our central design principle: **build once what is static, and handle on the fly what is not**. Navigation is static, frequent, and pre-enumerable, so we distill it offline into a table of callable commands and reuse it across tasks; open-ended interaction—form filling, dynamic content,

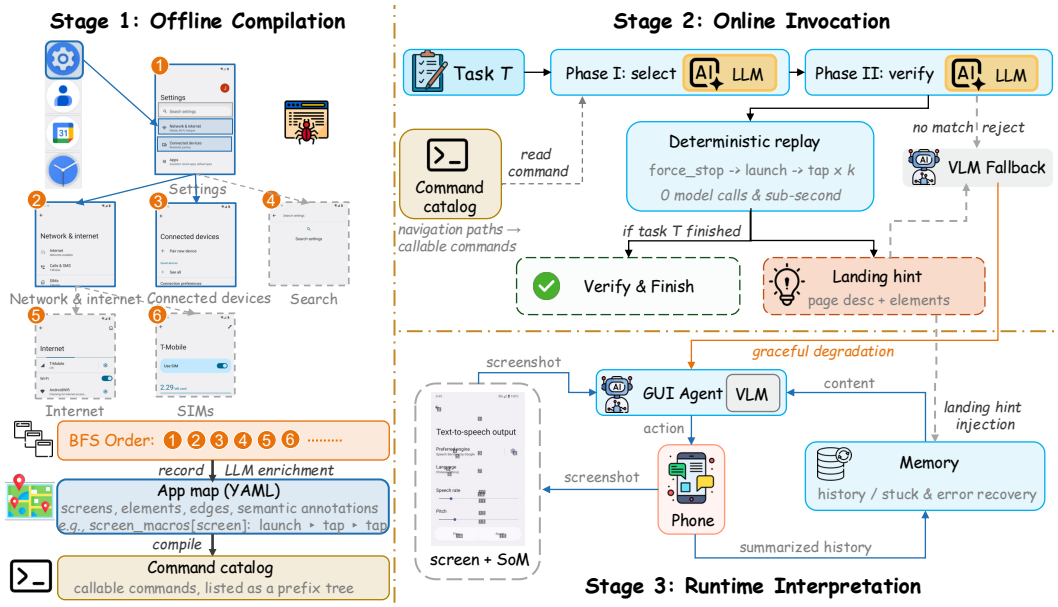

Figure 1: Overall framework of the proposed PhoneCLI.

unexpected dialogs—is variable and cannot be exhaustively enumerated, so it is handled by the VLM at runtime. Such content—feeds, recommendations, merchant lists—changes on every load; the navigation skeleton does not, and only the skeleton is compiled. We refer to the offline distillation as **compilation** and to the runtime handling as **interpretation**—a division of labor analogous to mixed-mode execution in programming languages, where frequently used code is compiled ahead of time and the remainder is interpreted on demand. Accordingly, PhoneCLI comprises three stages:

- **Offline compilation (Stage 1, Sec. 2.2).** An automated exploration traverses the app from a clean state and compiles its navigation skeleton into a semantically annotated map plus a catalog of commands, turning an interface that can be operated only manually into one that can be invoked programmatically.
- **Online invocation (Stage 2, Sec. 2.3).** A router maps a task to a compiled command, the match is confirmed, and the command is replayed deterministically—the analog of invoking a compiled function, with a guard on either side of the call.
- **Runtime interpretation (Stage 3, Sec. 2.4).** Everything outside the command table is handled by the VLM, which receives every failure of the compiled path as a graceful degradation.

Formalization. Formally, an app map is a structure $\mathcal{G} = (S, E, \lambda, M)$, where S is the set of discovered screens; E is the set of interactive elements, each annotated with normalized coordinates; $\lambda : E \rightharpoonup S \times \text{Alias}^*$ is a partial function that records navigation by mapping an element to the screen reached upon its activation together with its semantic aliases, so that a navigation edge is a triple (s, e, s') with e an element of s; and M is a set of *commands*, one per screen. Each command $c = [a_1, \ldots, a_k]$ is a deterministic action sequence that navigates to its target screen.

Stage 2 consumes a *catalog* $\mathcal{C}$ derived from $\mathcal{G}$: each entry pairs a stable identifier with the natural language description and target of a command, and carries neither coordinates nor action steps, so that routing operates on symbols rather than on plans. A router $r(T, \mathcal{C}) \mapsto (d, id, ans)$ maps a task description T to a decision d, the identifier of a selected command when one applies, and an answer string ans when $d = \texttt{FINISH}$; the identifier resolves to a command $c \in M$. The decision takes one of four values: $d = \texttt{OP}$, the command alone completes the task and is replayed with the answerability check; $d = \texttt{MACRO_VLM}$, the compiled macro performs the required navigation and then transfers control to the VLM, and we call a screen's replay sequence its macro; $d = \texttt{NEED_VLM}$, no applicable command exists and the VLM explores from the current state; and $d = \texttt{FINISH}$, the task can be answered without device interaction.

Design properties. (I) Compile once, reuse many. The offline cost is amortized across all subsequent tasks and runs, and navigation no longer consumes the VLM budget. **(II) Guaranteed floor.** Every failure path—no matching command, a rejected verification, or a landing mismatch—resolves to the

pure VLM agent, so the success rate is never below that of the VLM-only baseline; what the compiled path adds is a bounded number of routing calls, not a new failure mode. **(III) Model-agnostic knowledge acquisition.** The command table is constructed without training or model updates; a new app is integrated in minutes by re-running the offline stage.

2.2 Offline Compilation: Building App Maps

The first stage constructs the app map $\mathcal{G} = (S, E, \lambda, M)$ from a black-box app in two steps: structural exploration recovers the app's navigation skeleton, and semantic enrichment renders that skeleton interpretable to a language model. Our exploration inherits systematic UI traversal from Android testing crawlers (Su et al., 2017; Gu et al., 2019), but targets reusable navigation rather than test coverage. Unlike trajectory-based methods that inject mined knowledge back into the VLM as text (Xie et al., 2025; Sun et al., 2025), the explored structure is compiled into deterministic, executable commands.

Structural exploration. Starting from a clean launch state, the crawler traverses the app in breadth-first order. At each screen it reads the UI hierarchy, records the interactive elements it finds, activates them one by one, and observes the screens they reach, thereby assembling the screens, elements, and navigation edges of $\mathcal{G}$. Scrollable screens are explored over multiple scroll pages so that content below the fold is not omitted. The walk is bounded by depth and screen budgets chosen to cover the navigation core of typical apps. Screen deduplication is computed from raw elements before semantic annotation, so that persistent bottom navigation bars shared across tabs do not dominate the signature and collapse distinct tabs into a single screen.

Semantic enrichment. The raw graph captures syntax—text and coordinates—but not semantics: what an element is, or what a screen is for. Because both routing (Stage 2) and verification depend on semantic understanding, a language model annotates the graph. Each element is classified as `STABLE` (present across app versions) or `DYNAMIC`; assigned aliases with a semantic type (`button`, `input`, `label`, etc.); and each screen is given a natural language description.

Compilation to commands. From the enriched map, the compiler derives one command per screen: the action sequence that reaches it from a clean launch—typically `force_stop`, `launch`, followed by a sequence of taps and swipes with fixed coordinates and inter-step waits. A command is executable rather than advisory: replaying it reproduces the recorded trajectory exactly, with no model in the loop. Because the map is obtained purely by exploration and LLM annotation, it can also be rebuilt whenever an app changes. An excerpt of the resulting app map is shown in Figure 6 (Appendix A.3).

2.3 Online Invocation: Routing and Replaying Commands

Stage 2 is the online counterpart of the compiled command table: given a task, it decides whether a compiled command applies and, if so, executes it reliably. It is deliberately conservative—whenever the mapping is uncertain, the command is declined and the task proceeds to Stage 3.

Command routing. Routing proceeds in two phases, each implemented by a lightweight LLM call. **Phase I (selection)** presents the task together with a compact command catalog and asks the router to choose among the four decisions defined in Section 2.1. The catalog is formatted as a prefix tree-commands that share a navigation path are merged at their common prefix and expanded by indentation—so that it remains compact, and it is deliberately symbolic: it carries element paths, semantic tags, and stable operation identifiers, but neither coordinates nor action steps, which keeps it short and prevents the router from inventing device-level actions. A catalog excerpt, together with the router's output format and an example decision, is shown in Figure 7 (Appendix A.4). **Phase II (verification)** inspects the selected command before any execution: the router judges whether the command's target screen, described in natural language, is semantically consistent with the task. If it is not, the command is rejected and the task falls back to the VLM, since replaying a plausible but incorrect command would land the VLM on an irrelevant screen and waste the rounds spent recovering.

Deterministic replay and completion. A selected identifier resolves to the command compiled from the map—the model chooses *what* to do, while the program retains *how*—so replay requires no perception, incurs no model call, and cannot hallucinate coordinates. Execution is therefore deterministic and completes in sub-second time, in contrast to screen-by-screen navigation, where

every step costs seconds and one API call. This is where compilation pays off: navigation, the most repeated part of app use, is removed from the VLM loop entirely. A replayed command is then completed in one of two ways. If the command alone solves the task (`OP`), the agent performs an answerability check on the landed screen and finishes with that answer if it is available; this check differs in kind from the Phase II verification, which asks whether the command's destination fits the task, whereas this one asks whether the reached screen actually yields the answer. If the answer is not available, the command is demoted to the `MACRO_VLM` path, so that a command which navigated correctly but cannot answer on its own still saves the VLM the navigation. To avoid a cold start on an unfamiliar screen, the handoff injects a landing hint: the target screen's natural language description and the list of its interactive elements, together with an explicit note that the command has only navigated to this screen and that the interaction remains to be performed. The hint is cheap—it is drawn from the app map—and turns the apparent teleportation into a warm start.

Landing check. Replay is not trusted blindly. After execution, the current UI hierarchy is matched against the recorded screens; if the agent has demonstrably landed on a screen unrelated to the command's target, the app is restarted to a clean state before the VLM takes over, so that a mismatch never silently misleads it. The command is thus verified before replay and checked afterward, which makes the invocation path self-checking and routes every failure to Stage 3. The full invocation path is summarized in Algorithm 1.

Algorithm 1: PhoneCLI: executing one task

Input: Task T, app map $\mathcal{G} = (S, E, \lambda, M)$, VLM
Output: Task outcome (graded success or failure)

$\mathcal{C} \leftarrow$ catalog of $\mathcal{G}$; `// Stage 1: symbolic entries, built once per app`
$(d, id, ans) \leftarrow r(T, \mathcal{C})$; `// Stage 2, Phase I: selection`
switch d **do**
  case *FINISH* **do**
    return ans `// answerable without device interaction`
  end
  case *NEED_VLM* **do**
    return INTERPRET*(T)* `// no command applies; continue from the current state`
  end
  case *OP* / *MACRO_VLM* **do**
    $c \leftarrow$ RESOLVE(id); `// identifier → recorded action sequence`
    if $\neg$ VERIFY*(c, T)* **then**
      return INTERPRET*(T)* `// Phase II rejects; current state retained`
    RESTART(app); REPLAY(c); `// clean state; deterministic, zero VLM calls`
    if $\neg$ LANDINGCHECK*(c)* **then**
      RESTART(app); **return** INTERPRET*(T)* `// clean-state fallback`
    if $d =$ *OP* $\wedge$ *answerable on the landed screen* **then**
      return *the answer as ans* `// command alone completes the task`
    $h \leftarrow$ HINT(c); **return** INTERPRET*(T, h)* `// OP demoted, or MACRO_VLM; Stage 3`
  end
end

2.4 Runtime Interpretation: VLM Fallback and Graceful Degradation

Why interpretation is needed. Compilation is inherently partial: the map covers only what exploration could observe, and open-ended interaction cannot be enumerated in advance. Stage 3 therefore interprets what was not compiled. The VLM perceives the screen and acts step by step, following the standard perception—action loop of GUI agents (Wang et al., 2024; Jiang & Huang, 2026); we write INTERPRET(T, h) for this loop, where the optional h is the landing hint injected after a replayed navigation.

Memory and recovery. The interpreter maintains a structured state across rounds. Each VLM response ends with a `STATE_ASSESSMENT` field, and past assessments are injected into the next prompt in chronological order, giving the agent an explicit memory of what it has tried and where it is stuck. On top of this, the loop detects unproductive behavior—repeated scrolling or waiting without screen progress—and injects targeted hints (*e.g.*, try the search entry; close the ad overlay). When an action fails at execution time, the error is recorded and re-presented in the next round with a

corrective hint. These mechanisms turn the interpreter from a stateless screen actor into a small but self-correcting agent.

Graceful degradation. The degradation rule is uniform: whenever the compiled path is unavailable, the system runs the pure VLM agent, continuing from the current state, or from a restarted clean state when the landing check has failed. Stage 3 is thus a co-routine rather than a last resort—it is entered by design on every failure of the compiled path, and the interpreter it supplies is exactly the agent that would otherwise have run alone.

3 EVALUATION

3.1 EXPERIMENTAL SETUP

Benchmarks. We evaluate on two Android agent benchmarks: AndroidLab (Xu et al., 2025b) (138 tasks, 9 apps) and AndroidWorld (Rawles et al., 2025) (116 tasks, 20 apps, open-world with cross-app workflows), where we compare against its official M3A agent.

Backbone Models. We use three backbone models — Qwen3.7-Plus (Cloud, 2026), GLM-4.6V (AI, 2026), and Kimi-K3 (Team et al., 2026). AndroidWorld experiments use Qwen3.7-Plus for both the agent and the baseline, keeping the comparison model-controlled.

Baseline methods. Comparisons use two primary model groups: (1) *General-purpose vision-capable LLMs*: closed-source models (Qwen3.7-Plus (Cloud, 2026), Gemini-2.5-Pro (Comanici et al., 2025), GPT-4o (Hurst et al., 2024), Claude-Sonnet 4) and open-weight models (Kimi-K3 (Team et al., 2026), GLM-4.6V (AI, 2026)) (2) *GUI-specialized/fine-tuned models*: AutoGLM-Phone, AutoGLM-Mobile (Xu et al., 2026), MobileUse (Li et al., 2025), UI-Genie-Agent (Xiao et al., 2026), UI-Tars-1.5 (Qin et al., 2025), V-Droid (Dai et al., 2025), and AutoGLM-2024-10 (Liu et al., 2024). More details of the experimental setup are reported in Appendix A.2.

3.2 OVERALL PERFORMANCE

PhoneCLI is an enhancement layer attached to any GUI agent rather than a standalone backbone, so the first question is whether the attachment itself helps. To answer this, Table 1 compares, on three backbones, the full PhoneCLI system against its embedded pure VLM agent (**VLM-only**) — PhoneCLI with the compiled layer removed, all else identical.

Table 1: Success rate, average steps, and tokens per task on AndroidLab.

Model	Condition	Success	Δ	Avg. Steps	Avg. Tokens
Kimi-K3	VLM-only	68.8%	+0.8	6.34	53.3k
	PhoneCLI	**69.6**%		6.02$^{\downarrow 5\%}$	48.1$k^{\downarrow 10\%}$
GLM-4.6V	VLM-only	41.3%	+3.2	7.07	61.4k
	PhoneCLI	**44.5**%		6.52$^{\downarrow 8\%}$	54.4$k^{\downarrow 11\%}$
Qwen3.7-Plus	VLM-only	50.7%	+12.3	7.52	40.1k
	PhoneCLI	**63.0**%		6.72$^{\downarrow 11\%}$	34.4$k^{\downarrow 14\%}$

Three observations follow:

- **(i) PhoneCLI adapts across backbones, with the largest gains on weaker ones.** Attaching PhoneCLI improves the success rate by 12.3 points on Qwen3.7-Plus and by 0.8 points on Kimi-K3. Weaker backbones carry limited app-specific navigation knowledge, and the compiled map supplies exactly this missing knowledge.
- **(ii) Beyond the success-rate gains, tasks are also completed faster and more economically.** On Qwen3.7-Plus, PhoneCLI reduces the average number of steps by 11% and the token consumption by 14%, as deterministic replay replaces multi-round visual navigation with sub-second, zero-token commands.
- **(iii) As backbone capability grows, the success-rate gain shrinks to 0.8 points on Kimi-K3, while the cost advantage persists** — token consumption remains 10% lower and the step count remains 5% lower. An enhancement layer that keeps reducing cost even when it no longer adds capability is particularly valuable under the latency and cost constraints of mobile deployment.

Table 2 situates PhoneCLI against both general-purpose LLMs and GUI-specific models, and two points stand out. **First, PhoneCLI lifts a modest backbone toward frontier level:** Qwen3.7-Plus (35B) moves from 50.7 as a plain VLM agent to 63.0 with PhoneCLI, closing a 19.2 point gap against the 2.8T-A104B open-weight Kimi-K3 (68.8) to 5.8 points — and on the Kimi backbone itself, PhoneCLI still adds a further 0.8 points (69.6, Table 1). A 35B model enhanced by PhoneCLI is thus brought within reach of a 2.8T-class one, demonstrating the strong amplification PhoneCLI provides for weaker models. Notably, 63.0 is also above every GUI-specific model in Table 2, even though those models are trained for GUI control while PhoneCLI is not.

Table 2: Success rate (%) on AndroidLab.

Method	Size	SR
General-purpose LLMs		
GPT-4o	–	31.2
Claude-Sonnet-4	–	40.6
GLM-4.6V	107B-A12B	41.3
Qwen3.7-Plus	35B	50.7
Gemini-2.5-Pro	–	56.5
Kimi-K3	2.8T-A104B	68.8
GUI-specific models		
V-Droid	8B	38.4
UI-Tars-1.5	72B	38.4
UI-Genie-Agent	72B	41.2
MobileUse	72B	44.2
AutoGLM-Mobile	9B	46.8
AutoGLM-Phone	9B	47.7
AutoGLM-2024-10	–	36.2
PhoneCLI method *w Qwen3.7-Plus*		
PhoneCLI *w/o* Replay	35B	55.1
PhoneCLI	35B	63.0

Second, the two PhoneCLI rows preview where the gain comes from. PhoneCLI w/o replay keeps the app map but removes the replay mechanism: the map is only injected as reference text, and the VLM must navigate screens on its own. Information injection alone lifts Qwen3.7-Plus from 50.7 to 55.1, while adding deterministic replay raises it further to 63.0 — the replay step contributes the larger share of the gain. This shows that PhoneCLI's improvement is not merely from exposing map information to the VLM, but substantially from a dedicated design: executing that knowledge deterministically, without VLM involvement. Moreover, App. A.5 reports a generalization test in which our method is attached to the official AndroidWorld agent.

3.3 Ablation Studies

We conduct ablations at two granularities: (I) how much each of PhoneCLI 's two core components contributes—the compiled map used as information and deterministic replay—and (I) whether any individual mechanism within these components is dispensable. We study both questions on AndroidLab using Qwen3.7-Plus across nine apps.

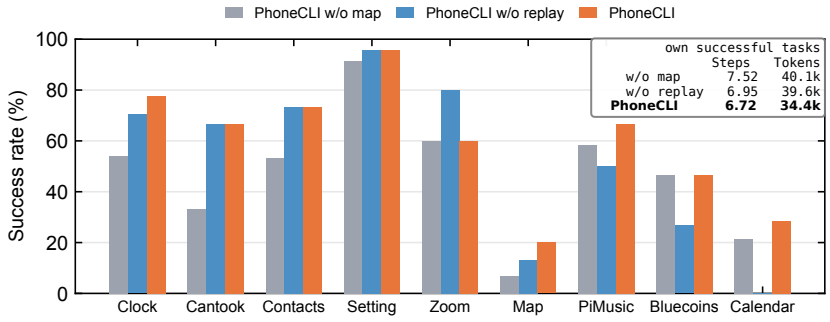

Figure 2: Success rate of the three conditions per app on AndroidLab (Qwen3.7-Plus), sorted by the information contribution (left: helpful, right: harmful). Inset: average steps and tokens on each condition's own successful tasks.

(I) Core decomposition. Three conditions decompose PhoneCLI's stack along its modules. *w/o map* (VLM only) removes the compiled map together with its replay machinery, leaving Stage 3, the pure VLM agent, at 50.7%. *w/o replay* keeps the map but only as information, injecting its navigation reference into the VLM's prompt, at 55.1% (+4.4). *PhoneCLI* additionally executes matched commands deterministically, at 63.0% (+7.9). Both components pay off, replay the more.

Cost and coupling. Injection shortens navigation but leaves the VLM in the loop: every remaining round still re-reads the injected reference and still pays a full VLM call, so rounds fall while tokens stay flat (7.52 → 6.95; 40.1k → 39.6k). Replay removes the call itself — compiled navigation consumes no VLM call — and collapses a multi-step traversal into a single recorded operation (6.95 → 6.72; 34.4k, −13%). The rounds that remain are the actual interaction rather than navigation. All efficiency numbers average over each condition's own successful tasks (Fig. 2).

Failure modes. Figure 2 orders the nine apps by how much the injected information contributes, and the ordering is uneven: injection helps on some apps and hurts on others. Its failure mode is that every screen must pass through the model's understanding, so it degrades exactly where the map is semantically sparse or noisy — which is what collapses Calendar to 0/14. Replay has no understanding step and recovers the same app to 4/14, but it executes the command as recorded, so a wrong landing is silent and only a post-replay check can catch it.

(II) Fine-grained mechanisms. Every variant in Table 3 removes one mechanism, and every removal costs at least 12 success-rate points. *On the map side*, element classification — the `STABLE`/`DYNAMIC` annotation the router uses to tell version-persistent elements from transient ones — is the costliest single removal (-15.9%), and semantic enrichment, the aliases and page descriptions that make the structural graph routable to both the router and the verifier, costs nearly as much (-15.2%). *On the replay side*, the landing check verifies that replay actually reached the target screen; without it, wrong landings silently mislead the VLM onto irrelevant screens (-12.3%). The landing hint — the description and element list injected at handoff — keeps a teleported VLM from starting cold (-13.0%). Command completion lets the `OP` path finish a replay-only task in one deterministic replay and one verification, with no VLM interaction at all; forcing such tasks through the VLM wastes that and adds risk (-14.4%).

Table 3: Ablation of PhoneCLI components on AndroidLab (Qwen3.7-Plus, 9 apps).

Variant	Success	Δ
Module 1: map annotation		
w/o element classification	47.1%	−15.9
w/o semantic enrichment	47.8%	−15.2
Module 2: routing and replay		
w/o landing check	50.7%	−12.3
w/o landing hint	50.0%	−13.0
w/o command completion	48.6%	−14.4
PhoneCLI	**63.0%**	–

Across both ablation resolutions, we reach the same conclusion: map information and deterministic replay contribute complementarily, and no single mechanism can be removed. Because each variant is evaluated against the full PhoneCLI system, the observed performance drops do not partition the overall gain; instead, multiple components independently account for improvements beyond the VLM-only baseline gap, indicating that each mechanism is load-bearing.

3.4 Deep Analysis of Map Scale

The ablation above removes components; another complementary question is *how much to build*: what budget should the offline crawler be given, and is a larger map always better? We vary the screen budget of map construction on two representative apps — Setting (native Android UI) and Clock (moderate complexity) — and report success rates in Figure 3.

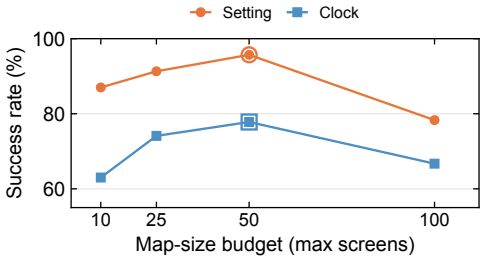

Figure 3: Success rate versus the map-size budget (max screens) on two apps (Qwen3.7-Plus).

Two patterns stand out. *First*, both apps exhibit an inverted-U: success rises as the budget grows from 10 to 50 screens, then drops at 100. Setting climbs from 87.0% to 95.7% and falls to 78.3%; Clock climbs from 63.0% to 77.8% and falls to 66.7%. *Second*, the two ends of the curve fail for different reasons. A small map under-covers the app: more tasks find no matching command and fall back to the VLM, forfeiting the compiled layer. An oversized map inflates the command catalog — from roughly 130 tokens at 10 screens to 3.5k at 100 — so Phase I must choose among a far larger, noisier candidate set, and mis-routing becomes more likely: the compiled layer then actively hurts. The default budget of 50 lands on the peak, which is also why the sweet spot is app-dependent rather than universal. Compression is thus not merely a cost measure but a prerequisite for routing accuracy.

3.5 Error Analysis

To see what the compiled layer actually fixes, we categorize failed trajectories of the two variants (VLM-only and PhoneCLI) into four failure modes, cross-checked by deterministic rules where possible (repeated action sequences, aborted trajectories). Figure 4 reports the per-mode totals.

The net drop of 17 failures (68 → 51) splits into 27 saved and 10 lost. The losses are mostly routing cost — two tasks mis-judged as command-only and four that landed on mismatched screens or failed the form interaction afterwards — plus four containing no macro at all, which are single-run VLM variance. The saved 27 span all four modes — looping 16, premature finish 1, execution 6, grounding 4 — which is the key finding: the compiled layer repairs no single failure mode, it *prevents the search for the entry point*. Once a command lands the VLM on the right screen, whole classes of failure lose their occasion. Grounding is the cleanest case: all four VLM-only grounding failures are saved, and the two that remain under PhoneCLI are single-run variance rather than mis-routing, since replay neither taps the wrong element nor is misled by semantics. Execution and other failures lose six, driven by fewer abnormal aborts — deterministic replay does not crash the way step-by-step interaction does — and fewer pattern-less exhaustions, as macros save rounds.

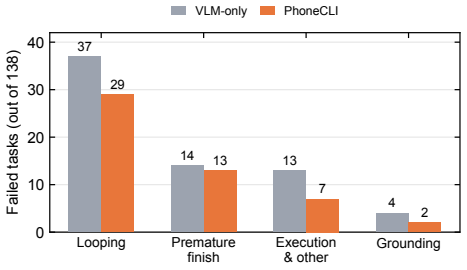

Figure 4: Failure-mode counts of the two variants (with Qwen3.7-Plus, on AndroidLab).

3.6 Case Study



Figure 5: Case study on Calendar ("add a 5PM event titled work").

Figure 5 shows the same task — add a 5PM event titled "work" in Calendar — under the two conditions. **PhoneCLI** (top): routing selects the `tasks_events_new_event` command, and its replay — force-stop, launch, tap the floating action button, tap "new event" — lands directly on the event form; the VLM then fills the title (type "work"), selects the time (`tap(1048,744)`), and saves (`tap(1174,2966)`), finishing the task in 17 rounds. **VLM-only** (bottom): at the same navigation stage that the command replay completes in one shot, the agent gets stuck in an error loop early on, and fails after 25 rounds. The comparison isolates the exact failure: task *understanding* was not the bottleneck — both agents know a 5PM "work" event must be created; *locating the entry point* was. Three recorded taps replace lots of exploratory ones, and the compilation pays off.

4 Conclusion

In this work, we propose PhoneCLI to address the lack of mobile app programmatic interfaces. We compile the static, repetitive portion of navigation offline into deterministic, zero-VLM commands and replay the matched command online in sub-second time; open-ended interaction and any compilation failures fall back to the embedded VLM interpreter. On AndroidLab, PhoneCLI improves task success rate while reducing step and token costs, yielding a stronger agent at lower cost.

AI USE STATEMENT

In this work, we used generative AI tools solely to edit the manuscript and improve its readability, including grammar, wording, and clarity of presentation. We have not used generative AI tools to generate synthetic datasets, develop theoretical models or conceptual frameworks, formulate mathematical claims, provide ingredients for proofs, assist in writing proofs, propose or refine hypotheses, design or provide feedback on research methodology or experiments, implement methods, translate the paper, clean or reformat datasets, support qualitative or thematic data analysis, or interpret results; these uses are not applicable to this work. We have reviewed all AI-assisted edits and verified that they do not alter the scientific claims, experimental results, or technical content of the paper. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Reyna Abhyankar, Qi Qi, and Yiying Zhang. Osworld-human: Benchmarking the efficiency of computer-use agents. *Proceedings of Machine Learning and Systems*, 8:482–494, 2026.

Zhipu AI. Glm-4.6v: Open source multimodal models with native tool use. `https://docs.z.ai/guides/vlm/glm-4.6v`, 2026.

Yuxiang Chai, Han Xiao, Xinyu Fu, Jinpeng Chen, Rui Liu, and Hongsheng Li. Ui-kobe: Knowledge-oriented behavior exploration for lightweight graph-guided gui agents. *arXiv preprint arXiv:2605.29534*, 2026.

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

Alibaba Cloud. Qwen3.7: The agent frontier. `https://www.alibabacloud.com/blog/qwen3-7-the-agent-frontier_603154`, 2026.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

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

Li Gu, Zihuan Jiang, Zhixiang Chi, Huan Liu, Ziqiang Wang, Yuanhao Yu, Glen Berseth, and Yang Wang. Generalization in online reinforcement learning for mobile agents. *arXiv preprint arXiv:2603.07432*, 2026.

Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. Practical gui testing of android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 269–280. IEEE, 2019.

Linqiang Guo, Li Gu, Zihuan Jiang, Zhixiang Chi, Siobhan Reid, Ziqiang Wang, Yuanhao Yu, Wei Liu, Yang Wang, et al. Coadapt-gui: Joint workflow context and policy adaptation for unseen gui applications. *arXiv preprint arXiv:2608.11588*, 2026.

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

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

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

Bojie Li. Preact: Computer-using agents that get faster on repeated tasks. *arXiv preprint arXiv:2606.17929*, 2026.

Chenxin Li, Zhengyao Fang, Zhengyang Tang, Pengyuan Lyu, Xingran Zhou, Xin Lai, Fei Tang, Liang Wu, Yiduo Guo, Weinong Wang, et al. Phoneharness: Harnessing phone-use agents through mixed gui, cli, and tool actions. *arXiv preprint arXiv:2606.14832*, 2026.

Ning Li, Xiangmou Qu, Jiamu Zhou, Jun Wang, Muning Wen, Kounianhua Du, Xingyu Lou, Qiuying Peng, and Weinan Zhang. Mobileuse: A gui agent with hierarchical reflection for autonomous mobile operation. *arXiv preprint arXiv:2507.16853*, 2025.

Tong Li, Yong Li, Mohammad Ashraful Hoque, Tong Xia, Sasu Tarkoma, and Pan Hui. To what extent we repeat ourselves? discovering daily activity patterns across mobile app usage. *IEEE Transactions on Mobile Computing*, 21(4):1492–1507, 2020.

Guangyi Liu, Pengxiang Zhao, Yaozhen Liang, Liang Liu, Yaxuan Guo, Han Xiao, Weifeng Lin, Yuxiang Chai, Yue Han, Shuai Ren, et al. Llm-powered gui agents in phone automation: Surveying progress and prospects. *arXiv preprint arXiv:2504.19838*, 2025.

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

Heinrich Peters, Joseph B Bayer, Sandra C Matz, Yikun Chi, Sumer S Vaid, and Gabriella M Harari. Social media use is predictable from app sequences: Using lstm and transformer neural networks to model habitual behavior. *Computers in Human Behavior*, 161:108381, 2024.

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

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

Linxin Song, Yutong Dai, Viraj Prabhu, Jieyu Zhang, Taiwei Shi, Li Li, Junnan Li, Zeyuan Chen, Jieyu Zhao, Ran Xu, et al. Coact-1: Computer-using multi-agent system with coding actions. In *International Conference on Learning Representations*, volume 2026, pp. 127091–127107, 2026.

Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pp. 245–256, 2017.

Yuchen Sun, Shanhui Zhao, Tao Yu, Hao Wen, Samith Va, Mengwei Xu, Yuanchun Li, and Chongyang Zhang. Gui-xplore: Empowering generalizable gui agents with one exploration. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 19477–19486. IEEE, 2025.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, Jianfeng Cai, Xinyuan Cai, Peizhou Cao, Yuxuan Cao, Ziwei Chai, Y Charles, et al. Kimi k3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653*, 2026.

Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. Mobile-agent: Autonomous multi-modal mobile device agent with visual perception. *arXiv preprint arXiv:2401.16158*, 2024.

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

Han Xiao, Guozhi Wang, Yuxiang Chai, Zimu Lu, Weifeng Lin, Hao He, Lue Fan, Liuyang Bian, Rui Hu, Liang Liu, et al. Ui-genie: A self-improving approach for iteratively boosting mllm-based mobile gui agents. *Advances in Neural Information Processing Systems*, 38:150376–150411, 2026.

Bin Xie, Rui Shao, Gongwei Chen, Kaiwen Zhou, Yinchuan Li, Jie Liu, Min Zhang, and Liqiang Nie. Gui-explorer: Autonomous exploration and mining of transition-aware knowledge for gui agent. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5650–5667, 2025.

Ran Xu, Kaixin Ma, Wenhao Yu, Hongming Zhang, Joyce C Ho, Carl Yang, and Dong Yu. Retrieval-augmented gui agents with generative guidelines. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 17877–17886, 2025a.

Yifan Xu, Xiao Liu, Xueqiao Sun, Siyi Cheng, Hao Yu, Hanyu Lai, Shudan Zhang, Dan Zhang, Jie Tang, and Yuxiao Dong. Androidlab: Training and systematic benchmarking of android autonomous agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2144–2166, 2025b.

Yifan Xu, Xiao Liu, Xinghan Liu, Jiaqi Fu, Jiayu Huang, Hanchen Zhang, Bohao Jing, Shudan Zhang, Yuting Wang, Yuxiao Dong, et al. Mobilerl: Online agentic reinforcement learning for mobile gui agents. In *International Conference on Learning Representations*, volume 2026, pp. 35282–35315, 2026.

Yunhe Yan, Shihe Wang, Jiajun Du, Yexuan Yang, Yuxuan Shan, Qichen Qiu, Xianqing Jia, Xinge Wang, Xin Yuan, Xu Han, et al. Mcpworld: A unified benchmarking testbed for api, gui, and hybrid computer use agents. *arXiv preprint arXiv:2506.07672*, 2025.

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

Chaoyun Zhang, Shilin He, Jiaxu Qian, Bowen Li, Liqun Li, Si Qin, Yu Kang, Minghua Ma, Guyue Liu, Qingwei Lin, et al. Large language model-brained gui agents: A survey. *arXiv preprint arXiv:2411.18279*, 2024.

Chi Zhang, Zhao Yang, Jiaxuan Liu, Yanda Li, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. Appagent: Multimodal agents as smartphone users. In *Proceedings of the 2025 CHI conference on human factors in computing systems*, pp. 1–20, 2025.

A Appendix

A.1 Related Work

Mobile GUI Agents and Programmatic Execution GUI agents for mobile devices largely adopt the perception-action paradigm: each step observes the screen and emits an indexed or coordinate-level action, with the underlying competence acquired through training. One line of work trains vision-language backbones for GUI *grounding* — perceiving screens and localizing elements (Hong et al., 2024; Cheng et al., 2024); another specializes models for *task completion* through supervised fine-tuning or reinforcement learning, including AutoGLM (Liu et al., 2024), MobileRL (Xu et al., 2026), MobileUse (Li et al., 2025), UI-Genie-Agent (Xiao et al., 2026), UI-Tars (Qin et al., 2025), and V-Droid (Dai et al., 2025). Surveys consolidate this paradigm (Zhang et al., 2024; Liu et al., 2025). Because capability lives in the weights, a new app demands new data or re-training — a bottleneck studied directly in recent work on RL generalization (Gu et al., 2026) and unseen-app adaptation (Guo et al., 2026). Orthogonal to this, a second line augments the GUI loop with programmatic execution: agents that write and execute code (Song et al., 2026), API-based computer use over instrumented applications (Yan et al., 2025), and mobile harnesses that mix GUI with device-side commands (Li et al., 2026). All of these assume an execution surface already exists — an OS that runs code, apps that expose APIs, or system-level CLI access. PhoneCLI takes the opposite stance: it *synthesizes* that surface by compiling an app's GUI navigation into callable commands, needing no API, no instrumentation, and no training, while keeping the perception-action loop only as a fallback.

App Exploration and Structure Mining The idea of systematically traversing an app to learn its structure goes back to Android GUI testing crawlers (Su et al., 2017; Gu et al., 2019), which optimize coverage and crash discovery rather than agent performance. Recent work reorients exploration toward agents, and diverges by what the explored structure becomes. In the first line, exploration yields *knowledge to read*: AppAgent (Zhang et al., 2025) writes natural-language operation documents; AutoDroid (Wen et al., 2024) turns UI transitions into executable scripts; GUI-explorer (Xie et al., 2025) mines function-aware trajectories; GUI-Xplore (Sun et al., 2025) packages exploration videos into a generalization dataset; and RAG-GUI (Xu et al., 2025a) retrieves web tutorials at inference time. In every case the model consumes the knowledge, yet still performs each action itself. In the second line, exploration yields *graph guidance*: UI-KOBE (Chai et al., 2026) builds an app knowledge graph offline and lets a lightweight agent navigate by matching its current screen to graph nodes and choosing among their transitions — the agent stays in the loop, steered but not replaced. In the third line, exploration yields *replay programs*: PreAct (Li, 2026) compiles a successful agent run into a state-machine program online, so that repeating the same task replays instead of re-reasoning.

PhoneCLI differs from all three lines on the same axes: it compiles offline, before any task arrives; at the granularity of one navigation command per screen rather than whole task runs; and executes deterministically rather than reading or following. PreAct compiles what an agent has done; PhoneCLI compiles what the app is. Taken together, the familiar elements PhoneCLI reuses — systematic traversal, exploration for knowledge, programmatic execution — are united by a single decision none of its predecessors makes: compiling the explored structure offline into deterministic commands.

A.2 Experimental Setup

App maps. Our maps are built offline, once per app, and reused across every task and every condition. Starting from the app's home screen, a breadth-first crawler explores up to 50 screens to a depth of 3, scrolling each list up to three pages; the only app-specific input is the package name, so no per-app engineering is involved. Each discovered screen is annotated with the text, normalized center, and semantic aliases of its elements, together with the navigation edges it exposes. Across the nine AndroidLab apps the resulting maps contain 320 screens, 2,822 elements, and 1,089 navigation edges, which compile into 2,017 operations (891 navigation and 1,126 action). Building one map takes on the order of ten minutes on a single emulator and runs entirely offline, consuming no rounds or tokens of any evaluated agent. Crucially, construction observes only the app itself: it never reads task instructions, target states, or evaluation code. All variants share the same maps, and the PhoneCLI w/o replay variant receives the same map serialized as a navigation catalog, without macro replay.

Evaluation. Operation tasks are judged by programmatic XML checkers and query tasks by an LLM judge; all conditions are re-judged under one unified judge model (Qwen3.7-Plus) to keep cross-model comparisons free of judge bias. Of the 138 tasks, 94 are graded by deterministic checkers over the final screen XML and the remaining 44 are query-style tasks graded by an LLM judge. Because different agents terminate in different states, we re-grade the logs of every condition with a single unified judge, so that the reported differences reflect the agents rather than the metric. All runs use an Android 13 emulator (Pixel 7 Pro AVD, 1440×3120); the emulator is cold-booted once and a clean snapshot is restored before every task, so tasks never observe each other's state. Each task is attempted once under a cap of 25 rounds. Success rate is computed over all 138 tasks, whereas step and token counts are averaged over each condition's successful tasks and therefore measure the cost of a successful trajectory. The gains of our methods over the strongest baseline in Table 1 and Table 2 are statistically significant ($p < 0.05$).

Implementation details. Observations follow the set-of-mark convention (Yang et al., 2023): interactable elements are drawn with numeric tags, and the model acts on those tags. All backbones are used off the shelf without fine-tuning, at temperature 0 and at most 512 new tokens per call; Qwen3.7-Plus is our default, and we additionally report runs with GLM-4.6V and Kimi-K3. Every condition shares the same observation format, prompt template, action space, and backbone, and conditions differ only in what is injected into the model's context, which isolates the effect of compiled commands. Executing a compiled operation issues ADB commands directly and invokes no model call. Token usage is accounted by call labels: `agent_vlm` for the interpreter loop of Section 2.4, `macro_map_task` for the Phase I routing call of Section 2.3, and `macro_verify` for its Phase II verification; judge calls are excluded.

A.3 App Map Data Structure

The app map is stored as a YAML file. Figure 6 shows a simplified excerpt from the Settings app map: each screen records its elements with normalized centers and navigation edges, and each screen is associated with a deterministic replay command that reaches it.

A.4 From App Maps to Callable Commands

Figure 6 shows what an app map stores; this section describes the second half of Stage 1, which turns those records into the command catalog $\mathcal{C}$ that the Stage 2 router reads, and illustrates the one-line interface the router answers with.

Compiling commands. The compiler walks the screen graph from the home screen and, for every interactive element, emits one *operation*: the element path that leads to it (for example `Connected devices` → `Pair new device`) paired with the action sequence that produces it, namely the macro of the parent screen, followed by the scroll steps needed to bring the element into view, followed by a tap at its recorded center. Operations whose element navigates to another screen are marked NAV; the remainder are terminal actions (ACT) such as toggles and text fields. Because several element paths may reach the same screen, an operation is keyed by its *path*, not by its destination: the catalog lists one entry per path, and only the path along which a screen was first expanded carries that screen's own children.

Rendering the catalog. Because the catalog is re-sent on every routing call, its size is a recurring cost, so we compress it aggressively. The router only has to decide *where* to go, so the catalog lists navigation operations only: for Settings this reduces 288 operations to 98 NAV entries in 9 groups, and the rendered catalog from roughly 6.2k to 1.9k tokens. Entries sharing a common path prefix are merged into a tree, so a shared navigation prefix is written once and children are indented. Each line carries the element text, its semantic type, one of its aliases, and a stable operation id — a slug of its element path, truncated at 60 characters — which is the only thing the router has to output.

From catalog to action. The router answers in a single line with one of four decisions (OP, MACRO_VLM, NEED_VLM, FINISH); the selected id resolves to the macro compiled above, which is replayed with fixed coordinates and no model call. For example, for the task *"pair a new Bluetooth device"* the router answers `MACRO_VLM: settings_connected_devices_pair_new_device` and the system replays `force_stop` | `launch` | `tap(548,1428)` | `tap(496,874)` before handing

App-map excerpt (Settings, simplified)

```yaml
screens:
- id: screen_0                          # Settings home screen
  description: "Main Settings screen with a profile header, search
  ↪  bar, and a comprehensive list o system configuration
  ↪  categories."
  elements:
  - text: Connected devices
    center: [0.3802, 0.4575]            # normalized (x, y);
    ↪  resolution-independent
    found_at_scroll: 0                  # 0 = visible on the first
    ↪  screenful
    leads_to: screen_5                  # navigation edge: tapping
    ↪  this reaches screen_5
    aliases: [devices, connections]     # LLM-generated aliases
    semantic_type: setting
- id: screen_5                          # Connected devices page
↪  (one tap from home)
  description: "Connected devices settings page where users can pair
  ↪  new devices, view saved devices, and manage connection
  ↪  preferences."
  elements:
  - text: Pair new device
    center: [0.3441, 0.2800]
    found_at_scroll: 0
    leads_to: screen_25
    aliases: [add device, connect device]
    semantic_type: button
screen_macros:                          # compiled command: clean
↪  start -> screen_25
  screen_25:                            # Bluetooth settings page
  - action: force_stop
    package: com.android.settings
    wait: 0.5
  - action: launch
    package: com.android.settings
    wait: 3.0
  - action: tap
    x: 548                              # taps "Connected devices"
    ↪  on screen_0
    y: 1428
    wait: 1.5
  - action: tap
    x: 496                              # taps "Pair new device" on
    ↪  screen_5
    y: 874
    wait: 1.5
```

Figure 6: App-map excerpt (Settings, simplified): elements, the navigation edges they define, and the deterministic replay command compiled from those edges. Tap coordinates equal `center`×screen size, and invoking a compiled command costs no VLM call.

control to the VLM on the Bluetooth page. The catalog is compiled once per app and cached, so this interface costs one prefix of a single routing call per task, and it is plain text: it can be inspected, edited, or reused across models.

A.5 Generalization to AndroidWorld

PhoneCLI is an enhancement layer, and the strongest test of that claim is whether it helps an agent it was not designed for. We therefore attach the compiled layer to M3A — the Multimodal Autonomous

(a) Command catalog: what the router reads

```
# Settings (com.android.settings) - 98 NAV operations in 9 groups
Apps: [SETTING] [applications]              # apps
  All apps: [MENU] [app list]               # apps_all_apps
  Default apps: [MENU] [default apps]       # apps_default_apps
  Screen time: [MENU] [screen usage]        # apps_screen_time
Connected devices: [SETTING] [connections] # connected_devices
Settings: [LABEL] [preferences]             # settings
  Connected devices: [SETTING] [connections]
    # settings_connected_devices
    Pair new device: [BUTTON] [add device]
      # settings_connected_devices_pair_new_device
Notifications: [SETTING] [alerts]           # notifications
```

(b) Router output: contract and one example decision

```
OP:         <op_id>    viewing the target page completes the task
MACRO_VLM: <op_id>    macro navigates, then the VLM interacts
NEED_VLM:  <reason>   no matching operation
FINISH:     <answer>   a query, no device action needed

Task: "Pair a new Bluetooth device"
-> MACRO_VLM: settings_connected_devices_pair_new_device
   replay: force_stop | launch | tap(548,1428) | tap(496,874)
   # replayed at zero model cost, then the VLM takes over
```

Figure 7: Command catalog compiled from the app map of Figure 6.

Agent and official baseline of AndroidWorld (Rawles et al., 2025), which acts step by step on annotated screenshots — by subclassing it and adding only the map-based routing of Section 2.3. M3A's prompts, action space, and perception remain untouched, and when routing declines, execution proceeds exactly as the original M3A. Both agents run on Qwen3.7-Plus.

Table 4: AndroidWorld results.

Agent	Success	Tokens	Steps
M3A (official)	61.2%	27.58M	8.4
M3A w PhoneCLI	**65.5%**	**25.04M**	**7.7**

Table 4 reports the results. The map-augmented agent scores higher than the original M3A agent. And the efficiency gains are consistent and substantial: 9% fewer tokens overall, and on the 70 tasks both agents solve, 7.7 steps versus 8.4. That these savings transfer to a different benchmark, a different agent framework, and an untouched M3A shows the compiled layer's efficiency does not depend on any AndroidLab-specific design. M3A is already a mature agent, and on this backbone the enhancement approaches the ceiling of what Qwen3.7-Plus can achieve — what remains to gain is efficiency rather than capability. Across environments, the enhancement-layer claim of our method therefore holds.