Beyond the GUI Paradigm: Do Mobile Agents Need the Phone Screen?

Anonymous ACL submission

Abstract

Recent advances in mobile agents are dominated by the GUI paradigm, in which agents perceive UI information and emit screen interactions. However, mobile platforms also expose a command-line interface (CLI) that provides direct access to device services and data. We argue CLI deserves first-class consideration alongside GUI. We evaluate three coding agents (Claude Code, Terminus-2, mini-swe-agent) across four model APIs on AndroidWorld and MobileWorld without any mobile-specific post-training, comparing against three reproducible GUI baselines (GUI-Owl-1.5-32B, MAI-UI, Qwen3-VL-32B). Claude Code (Opus 4.7) reaches 71.8% and 51.9%, outperforming every reproducible GUI baseline (69.3/68.1/57.8% on AndroidWorld; 43.2/26.3/13.3% on MobileWorld), while every other CLI configuration remains competitive. To establish the paradigm's ceiling, we provide oracle CLI solutions that reach 88.8% on AndroidWorld (103/116 tasks CLI-solvable) and 86.3% on MobileWorld (101/117 tasks CLI-solvable), indicating substantial room for future improvement. To cover everyday user intents beyond the GUI scope, we introduce the **CLI-Advantage Task Suite**, comprising 45 templates across five categories: bulk operations, multi-condition filtering, aggregation, cross-app workflows, and hidden device state. Every CLI agent outperforms every GUI baseline in all five categories, with substantially fewer steps per task (10.7 vs. 18.6). To support future research on mobile CLI agents, we open-source agent implementations, oracle solutions, the CLI-Advantage suite, and evaluation infrastructure (link).

1 Introduction

Mobile agents are autonomous systems that interact with smartphones via natural language to complete user tasks (Wu et al., 2024; Hu et al., 2025; Liu et al., 2025). The dominant paradigm treats this as a vision problem: agents observe the phone screen and produce GUI actions, including tapping, swiping, and typing, to navigate applications, mimicking human behavior. Substantial research effort has advanced this paradigm across benchmarks, models, training algorithms, and infrastructure (Bai et al., 2024; Zhou et al., 2025; Wang et al., 2025a; Xu et al., 2025a; Gu et al., 2026; Xu et al., 2026; Team et al., 2026).

However, this paradigm is not the only way mobile agents can operate on mobile platforms. Android is a Linux-based platform that exposes a full command-line interface (CLI) through the Android Debug Bridge (ADB), allowing programmatic access to device services and data without ever rendering a pixel (Google, 2025a). In addition, ADB can expose device states, including battery history, notification logs, and background processes, that never appear on any screen (Google, 2025b), enabling the CLI paradigm to reach beyond what GUI agents can observe. Meanwhile, frontier coding agents (Anthropic, 2025; Singh et al., 2025; MiniMax, 2026) have demonstrated strong capabilities on challenging long-horizon terminal tasks in Linux environments (Merrill et al., 2026), operating in a text-only modality native to large language models. Motivated by these observations, we argue that this CLI paradigm deserves first-class consideration alongside GUI for mobile agents, and we test the claim by answering two questions. **Q1 (viability):** Are general coding agents using ADB alone competitive with GUI agents that are post-trained on mobile datasets via Supervised fine-tuning and Reinforcement earning? **Q2 (superiority):** Are there everyday user intents better served by CLI but ignored by existing GUI-designed benchmarks?

To answer Q1, we evaluate three frontier coding agents on two standard mobile-agent benchmarks, AndroidWorld (Rawles et al., 2025) and MobileWorld (Kong et al., 2025), against three reproducible GUI baselines: GUI-Owl-1.5-32B (Xu

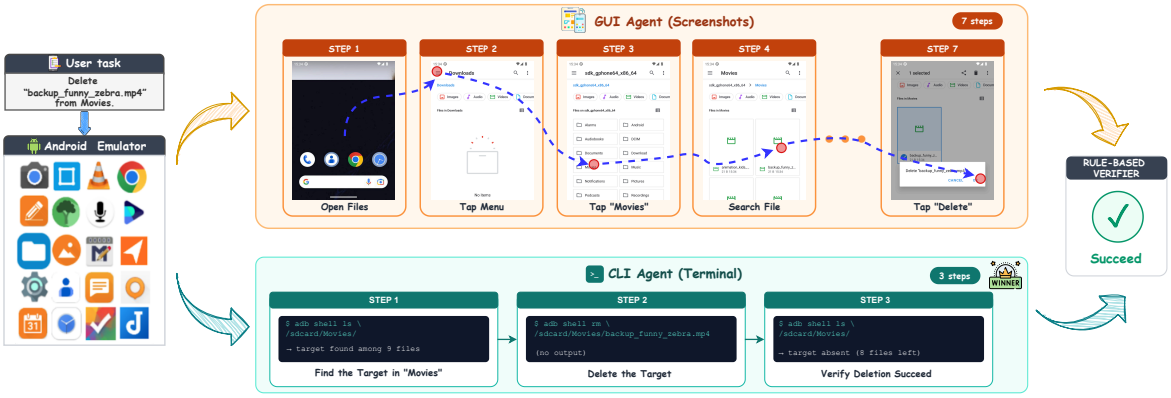

Figure 1: **GUI vs. CLI execution traces on a representative mobile task.** Both agents are asked to delete `backup_funny_zebra.mp4` from the Movies folder and proceed through three phases—*find the target*, *delete the target*, *verify deletion*. The GUI agent (top) consumes screenshots and emits taps, unfolding the phases over 7 UI steps (open Files → navigate menu → enter `Movies` → search → confirm delete). The CLI agent (bottom) consumes terminal `stdout` and emits ADB shell commands, collapsing the same phases into 3 steps (`ls`, `rm`, `ls`). A rule-based verifier scores both trajectories on their final state, isolating the interface paradigm as the only variable.

et al., 2026), MAI-UI-8B (Zhou et al., 2025), and Qwen3-VL-32B (Bai et al., 2025). All experiments are conducted under the same evaluation protocol, with identical tasks, rule-based verifiers, and metrics, and the primary intended variable is the interface paradigm. The coding agents are Claude Code (Anthropic, 2025), Terminus-2 (Merrill et al., 2026), and mini-swe-agent (Yang et al., 2024), all previously validated on Terminal-Bench, and each receives only terminal stdout as input (no screenshots or accessibility trees) and outputs bash commands, with no mobile-specific training. To bridge the Android platform gap, we manually construct a system prompt containing domain knowledge and strategies derived from public developer documentation, without leakage of task-specific or app-specific information. Motivated by recent advances in harness engineering (Tian et al., 2026; Lopopolo, 2026; Lee et al., 2026), we further design general-purpose tools that wrap raw ADB bash commands to boost agent performance. Finally, to establish an upper bound on the CLI paradigm's capability, we construct oracle bash command solutions through human-LLM collaboration, manually reviewed by three external professionals, showing which tasks are CLI-solvable in AndroidWorld and MobileWorld.

To answer Q2, we evaluate both CLI and GUI agents on tasks that the GUI paradigm was not designed to accommodate. Since existing benchmarks were designed around the GUI paradigm and consequently exclude user intents that no tap sequence can serve, we introduce the **CLI-Advantage Task Suite**: 45 LLM-generated, human-verified task templates spanning five categories. Three categories have operations the app UI does not expose at scale: *bulk operations* ("delete every `.apk` file in Downloads"), *multi-condition filtering* ("list expenses above my average spending"), and *aggregation and top-K* ("top-3 expense categories this month"). The remaining two categories address state the app layer does not surface: *cross-app workflows* ("did anyone text me during my meetings yesterday?") and *hidden device state* ("which apps have been granted location permission"). Each task corresponds to an AndroidWorld-compatible rule-based verifier and a hand-verified oracle CLI solution for reliable assessment.

Our experiments yield strong evidence that the CLI paradigm deserves first-class status alongside GUI for mobile agents. On AndroidWorld (AW) and MobileWorld (MW), Claude Code (Opus 4.7) achieves 71.8% and 51.9%, outperforming every reproducible GUI baseline (GUI-Owl-1.5-32B, MAI-UI-8B, and Qwen3-VL-32B at 69.3/68.1/57.8% on AW and 43.2/26.3/13.3% on MW). Moreover, mini-swe-agent with Sonnet 4.6 reaches 70.7% on AndroidWorld, and every other CLI configuration we evaluated (three harnesses and three additional model APIs) is competitive with reproducible GUI baselines, showing a paradigm-level property rather than an artifact of a single agent. Generic ADB tools substantially improve CLI agents with weaker models in both success rate and cost efficiency, but provide no consistent benefit to stronger ones, indicating that tools should be gated

on model strength. A trajectory-level error analysis reveals that GUI and CLI agents exhibit distinct failure modes, indicating that the two paradigms require different improvement strategies. On the CLI-Advantage Suite, the gap widens substantially: every CLI agent we evaluated outperforms every GUI baseline (60.7–68.9% vs. 22.2–33.3%), leading across all five categories, particularly on cross-app workflows where GUI agents cap at $\leq$11% regardless of model scale. Beyond success rates, CLI agents also use substantially fewer steps per task than GUI methods (10.7 vs. 18.6 mean steps). Finally, oracle CLI agents reach 88.8% on AndroidWorld (103/116 tasks CLI-solvable) and 86.3% on MobileWorld (101/117 tasks CLI-solvable), showing that current agents still fall short of the CLI paradigm's ceiling.

2 Related Work

Mobile GUI agents. Mobile GUI agents aim to complete natural-language tasks on smartphones by perceiving screen state and emitting touch, swipe, and type actions, a setting that couples long-horizon planning with fine-grained visual grounding under variable layouts and dynamic app state (Wu et al., 2024; Hu et al., 2025; Liu et al., 2025). Prior work falls into two categories. Multi-agent approaches orchestrate proprietary VLMs into pipelines that plan, act, reflect, and remember around a screenshot loop (Zhang et al., 2025; Wang et al., 2024b,a; Ye et al., 2025; Wen et al., 2024; Guo et al., 2026). End-to-end approaches instead post-train an open VLM on UI trajectories, first with supervised fine-tuning (Qin et al., 2025; Wang et al., 2025a; Gu et al., 2025; Team et al., 2026; Xu et al., 2026; Zhou et al., 2025; Yan et al., 2025; Wu et al., 2025; Xu et al., 2024; Hong et al., 2024; Cheng et al., 2024; Lin et al., 2025; You et al., 2024; Yang et al., 2025) and, more recently, with online reinforcement learning in live Android environments (Bai et al., 2024; Shi et al., 2025; Dai et al., 2026; Xu et al., 2025a; Gu et al., 2026). However, both categories share the same interface: the observation is the rendered UI (screenshot pixels or accessibility tree) and the action space is pixel-level coordinates, which makes performance bounded by grounding accuracy, blind to off-screen system state (battery history, background processes, notification logs), brittle to layout shifts, and unable to express bulk or cross-app operations that the rendered surface does not expose. Different from these works, we drive Android through adb alone — terminal text as the only observation, shell commands as the action space — sidestepping these limitations.

Mobile agent benchmarks. Mobile agent benchmarks fall into two categories. Offline datasets compile human-annotated Android trajectories and grade agents by per-step action matching against the human references (Rawles et al., 2023; Li et al., 2024), but static demonstrations restrict evaluation to step-level accuracy and cannot capture trajectory-level task success. To address this, online interactive benchmarks boot a live Android emulator and grade trajectory-level success through rule-based or LLM-as-judge verifiers (Rawles et al., 2025; Xu et al., 2025b; Wang et al., 2024c; Chen et al., 2024; Chai et al., 2025); MobileWorld (Kong et al., 2025) further extends this setting with broader app coverage, networked client-server applications, longer-horizon and cross-application tasks, agent-user interaction, and MCP-augmented environments. However, tasks in these benchmarks operationalize only the user intents the screen can surface, introducing a sampling bias that leaves a broad class of real user needs unmeasured. AppWorld (Trivedi et al., 2024) similarly evaluates daily multi-app tasks through 457 hand-crafted APIs, but operates in a simulated APIs rather than on Android's terminal and is not compared against GUI agents.

Coding and terminal agents. Coding agents issue text-based actions through programmatic interfaces such as shell commands and tool calls, evaluated on software-engineering and OS tasks (Wang et al., 2024d; Yang et al., 2024; Wang et al., 2025b); frontier coding agents (Anthropic, 2025; Singh et al., 2025; MiniMax, 2026) achieve strong performance on these benchmarks through text-only interaction. Harness engineering — the design of tools, system prompts, context management, and feedback loops around an off-the-shelf agent — has emerged as the principal mechanism for adapting such agents to new domains without retraining (Lopopolo, 2026; Lee et al., 2026). Terminal-Bench (Merrill et al., 2026) demonstrates that coding agents can solve realistic system and developer tasks on Linux through the shell alone, but no prior work evaluates coding agents at operating real Android devices through the OS interface. The closest related work is CoAct-1 (Song et al., 2025), a hybrid desktop agent on OSWorld (Xie et al., 2024) whose Orchestrator routes each sub-task to a

GUI Operator or a Python/Bash Programmer. However, CoAct-1 neither isolates the two paradigms on identical tasks nor extends to mobile. Different from these works, we compare both paradigms on identical AndroidWorld and MobileWorld tasks — coding agents through ADB alone, GUI specialists through screenshots — isolating the interface paradigm as the sole variable.

3 Experimental Setup

Benchmarks and Baselines. We evaluate on two interactive mobile-agent benchmarks whose rule-based verifiers check device state directly, ensuring compatibility with terminal-only execution. AndroidWorld (Rawles et al., 2025) provides 116 task templates across 20 Android applications, parameterized by random seeds. MobileWorld (Kong et al., 2025) extends this setting with broader app coverage spanning networked client-server applications in e-commerce, enterprise communication, and social media, and with longer-horizon, cross-application tasks (27.8 vs. 14.3 average steps per task; 62.2% vs. 9.5% multi-app tasks). We evaluate on its 117 device-state tasks, excluding the user-interaction and MCP (Hou et al., 2026) subsets that fall outside the scope of a GUI-vs-CLI comparison. We exclude AndroidLab (Xu et al., 2025b) because its XML-based verifier requiring UI state is incompatible with the CLI paradigm.

Following CoAct-1 (Song et al., 2025), we report success rate (SR) and average steps per task across three random seeds, with mean and standard deviation reported. For GUI baselines, we evaluate three reproducible GUI agents: two end-to-end specialists post-trained on mobile datasets (GUI-Owl-1.5-32B (Xu et al., 2026) and MAI-UI-8B (Zhou et al., 2025)) and a strong general-purpose vision-language model (Qwen3-VL-32B (Bai et al., 2025)). We distinguish these *reproducible* baselines, re-run end-to-end under our unified evaluation infrastructure, from leaderboard-reported results—UI-Venus-1.5 (Team et al., 2026), Seed-1.8 (Seed, 2026), and Step-GUI (Yan et al., 2025)—included in Table 1 for context but not reproducible as their evaluation pipelines are not released.

CLI Agents. We use three off-the-shelf coding agents spanning two design categories: a frontier agent (Claude Code (Anthropic, 2025)) representing state-of-the-art coding capability, and two minimal-scaffold agents (Terminus-2 (Merrill et al., 2026) and mini-swe-agent (Yang et al., 2024)) designed for Terminal-Bench command-line tasks with simple harness architectures, chosen to evaluate how effectively such designs transfer to mobile platforms. Specifically, each agent operates in a text-only Android terminal: it receives the task instruction and terminal stdout at each step (e.g., system state, database query responses, or error messages) as input, and emits ADB commands to interact with the device. No visual or structural UI information such as screenshots or accessibility trees is used at any step. Figure 1 contrasts CLI and GUI execution on a representative task.

System Prompt and Tools. To adapt general coding agents to the Android platform, we construct a system prompt derived from public Android developer documentation and engineering practices. The prompt provides four guidance categories: a structured four-phase interaction cycle (discover relevant data, inspect existing state, act through the terminal interface, and verify the result); a prioritized hierarchy of mechanisms for modifying device state; efficiency strategies including batching commands and limiting exploration budgets; and platform-specific patterns for file synchronization and data persistence. Crucially, the prompt contains no task-specific or app-specific information that could leak to CLI agents. Full system prompts for each CLI agent are provided in Appendix A.1.

Motivated by recent advances in harness engineering (Tian et al., 2026; Lopopolo, 2026; Lee et al., 2026), we provide general-purpose tools that wrap raw ADB shell commands into higher-level operations. For AndroidWorld, we provide four tools: `sql` (database queries), `read-file` and `write-file` (file I/O), and `find-files` (file search). For MobileWorld, with more challenging backend services, structured data, and networked APIs, we extend to twelve tools by adding `content` and `intent` (Android content providers and intents), `json-read` and `json-write` (structured-data parsing), `http` (HTTP requests), `pg` (backend PostgreSQL queries), `backend-exec` (backend container shell), and `service-status` (backend service listing). Appendix A.2 provides complete tool specifications.

Oracle Construction. To establish the empirical CLI ceiling on each benchmark, we construct oracle solutions through human–LLM collaboration. Each solution must satisfy two criteria: *solvability*, meaning the command sequence passes the rule-based verifier; and *integrity*, meaning the solution

arrives at the answer through principled exploration rather than extracting it from verifier's source code. Each task is attempted up to five times with verifier feedback; every solution is independently reviewed by all three external professionals. This process yields 88.8% (103/116) CLI-solvable tasks in AndroidWorld and 86.3% (101/117) in MobileWorld. The remaining 13 tasks on AndroidWorld and 16 on MobileWorld involve interactions outside the CLI paradigm's reach: interacting with visual content (e.g., transcribing receipts from photos, free-hand drawing) or capturing multimodal data (e.g., camera photos or audio recording); on MobileWorld this set additionally contains a handful of in-app panel-only fields (Google Maps detail labels, shopping-cart totals) that no public API exposes. Details can be found in Appendix A.3.

4 CLI-Advantage Task Suite

Real mobile users have everyday needs the touchscreen interface was not designed for: cleaning up leftover files, asking which expense category dominated this month, or finding out which apps have been granted background-location access. Existing mobile-agent benchmarks were designed around the GUI paradigm, so user intents that no tap sequence can serve fall outside their scope. This sampling gap leaves a measurable class of user intent invisible to benchmark scores. We introduce the CLI-Advantage Suite, a set of **45 task templates across five categories** that target this gap directly, each with an AndroidWorld-compatible rule-based verifier and a hand-verified oracle CLI solution.

Taxonomy. The 45 task templates fall into five categories of mobile-user intent that current touchscreen UIs serve poorly. Three require operations the app UI does not expose at scale: **Bulk operations** (10 tasks) apply one change to many items at once (e.g., *"delete every .apk file in Downloads"*); **Multi-condition filtering** (10) selects items satisfying a conjunction of constraints app filter UIs rarely compose (e.g., *"list expenses above my average spending"*); and **Aggregation and top-K** (10) computes summaries over many rows no single screen displays (e.g., *"top-3 expense categories by total this month"*). Two require state the app layer does not surface: **Cross-app workflows** (9) combine state from multiple apps (e.g., *"did anyone text me during my meetings yesterday?"*), and **Hidden device state** (6) queries system-level facts no app exposes (e.g., *"which apps have been granted location permission"*).

Construction and Verification. The 45 task templates are LLM-generated and human-verified by three mobile users. Following AndroidWorld's automatic task-parameterization mechanism, each template yields a distinct task instance per seed. With three seeds $\{7, 30, 1234\}$, we construct 135 task instances. Success is checked by a binary rule-based script that reads the final device state, matches the agent's reported answer against ground truth, or both—with no LLM-judge in the loop. We verify each task along three axes. *Solvability*: applying the Section 3 oracle procedure, we construct a hand-verified CLI command sequence per task; all 45 pass the rule-based verifier. *Realism*: three annotators individually rate each task template on a three-level rubric (highly realistic / stretched / contrived); inter-annotator Cohen's $\kappa = 0.91$. *Additivity*: for each task, we compute the cosine similarity to its nearest AndroidWorld neighbour using `text-embedding-3-large`; the resulting distribution sits to the left of AndroidWorld's within-suite reference, with 0% exceeding similarity 0.7, as in Figure 3. Appendix B lists details and the full task.

5 Results

5.1 Benchmark Performance

CLI agents are competitive with leading GUI agents . Claude Code with Opus 4.7 reaches 71.8% on AndroidWorld and 51.9% on MobileWorld (Table 1), outperforming every reproduced GUI baseline (GUI-Owl-1.5-32B, MAI-UI-8B, and Qwen3-VL-32B) on both benchmarks. On MobileWorld, Opus 4.7 also matches the strongest leaderboard-reported agent, Seed-1.8 (52.1%). On AndroidWorld, the leaderboard-reported UI-Venus-1.5-30B-A3B stands at 77.6%, but its evaluation pipeline is not released and we cannot verify this number under our protocol. The CLI lead is sharpest on the CLI-solvable subset (103 tasks on AW, 101 on MW): Opus 4.7 leads GUI-Owl-1.5-32B by roughly 9 percentage points on AndroidWorld (80.9% vs. 71.6%) and roughly 20 percentage points on MobileWorld (60.1% vs. 38.1%).

Competitiveness is a paradigm-level property. On AndroidWorld, every CLI configuration in Table 1 lands between 55% and 72% success rate, with mini-swe-agent and Sonnet 4.6 reaching 70.7% and matching GUI-Owl-1.5-32B (69.3%) at more step efficiency. The complete harness $\times$

Method	Model API	AndroidWorld			MobileWorld		
		SR↑	SR-CLI↑	Avg Step↓	SR↑	SR-CLI↑	Avg Step↓
GUI Agents (Leaderboard-reported)⋆							
UI-Venus-1.5-30B-A3B	-	77.6	-	-	17.1	-	-
Seed-1.8	-	70.7	-	-	52.1	-	-
Step-GUI-8B	-	67.7	-	-	16.1	-	-
GUI Agents (Reproduced)							
GUI-Owl-1.5-32B	-	69.3 (0.5)	71.6 (1.5)	15.5 (2.5)	43.2 (2.6)	38.1 (2.0)	34.7 (0.7)
MAI-UI-8B	-	68.1 (0.6)	69.3 (0.3)	18.4 (1.2)	26.3 (1.3)	26.0 (2.0)	39.7 (0.3)
Qwen3-VL-32B	-	57.8 (4.9)	62.4 (5.6)	17.4 (1.8)	13.3 (0.7)	11.7 (0.3)	25.6 (0.6)
CLI Agents (Ours)							
Claude Code	Claude Opus 4.7	71.8 (1.8)	80.9 (2.0)	15.2 (0.6)	51.9 (1.0)	60.1 (1.1)	30.2 (0.6)
mini-swe-agent	Claude Sonnet 4.6	70.7 (2.3)	79.6 (2.6)	12.1 (0.3)	43.6 (2.6)	50.5 (3.0)	33.5 (0.4)
Terminus-2	GPT-5.3 Codex	63.2 (0.5)	71.2 (0.6)	7.1 (0.1)	36.2 (1.0)	41.9 (1.1)	24.5 (1.9)
Terminus-2	MiniMax M2.7	55.2 (3.8)	62.1 (4.2)	14.5 (0.5)	16.8 (1.3)	19.5 (1.5))	32.9 (0.8)
Oracle (CLI-solvable ceiling)	-	88.8	100.0	3.7	86.3	100.0	5.6

Table 1: **Main results on AndroidWorld (AW) and MobileWorld (MW). SR** is success rate (%) over the full task set, mean (std) across 3 seeds. **SR-CLI** restricts the denominator to the CLI-solvable subset of each benchmark (103/116 on AW; 101/117 on MW); the remaining tasks require GUI interaction. **Avg Step** is mean agent steps per task. All CLI agents run under `bash` plus tools that wrap raw `adb shell` commands. **Oracle** is the CLI ceiling, $103/116 = 88.8\%$ on AW and $101/117 = 86.3\%$ on MW, derived from ground-truth reference solutions of each task using CLI; Oracle's Avg Step is averaged only over CLI-solvable tasks. ⋆ Reported in the original paper; evaluation code is unavailable, precluding reproduction. The full leaderboard-extended version is reported in Table 8 (Appendix C.1).

model-API matrix is reported in Table 2. On MobileWorld, the picture is more model-dependent: Claude Code with Opus 4.7 and mini-swe-agent with Sonnet 4.6 clear every reproduced GUI baseline, while the two Terminus-2 configurations (with Codex and with MiniMax) trail GUI-Owl, with the gap widening sharply for weaker model APIs on MobileWorld's more challenging, longer-horizon, cross-application tasks. The Oracle CLI ceiling sits at 88.8% on AndroidWorld and 86.3% on MobileWorld, leaving 17 percentage points of headroom on AndroidWorld and 34 on MobileWorld.

Tool gains are conditional on model strength. Table 2 compares bash-only and `+tools` modes across three CLI harnesses (Claude Code, mini-swe-agent, Terminus-2) and four model APIs on AndroidWorld. In bash-only mode, GPT-5.3 Codex is the weakest model API (48.9–49.7% SR), well below Claude Sonnet 4.6 (58.9–69.5%). Consistent with recent findings that tool gains are conditional on model strength (Lou et al., 2026), only GPT-5.3 Codex benefits substantially from the four CLI tools, with success rate rising by +11.8 pp on mini-swe-agent and +14.3 pp on Terminus-2; other configurations change by at most 1.4 pp. Inference cost mirrors SR: Codex sees the biggest cost reductions ($\sim$ 23% on mini-swe-agent and $\sim$ 28% on Terminus-2), with mini-swe-agent + Sonnet 4.6 next at $\sim$ 13%, while Opus 4.7 costs $\sim$ 3% more. One mechanism behind these effects is shell quoting: the four CLI tools handle escaping that bash invocations frequently mishandle. Two paired Codex trajectories in Appendix C.2 illustrate this: `+tools` converts a bash-only failure into a success (the SR gain) and eliminates retry steps on a single-quote edit (the cost reduction).

5.2 Trajectory-Level Error Analysis

Since average success rates do not reveal failure patterns, we conduct a trajectory-level error analysis along three comparison axes: (i) GUI vs. CLI paradigms; (ii) `bash-only` vs. `+tools` for GPT-5.3 Codex, selected for its largest tool-boosted SR gain in Table 2; and (iii) model APIs and harnesses jointly. We adapt the Multi-Agent System Taxonomy (MAST) (Cemri et al., 2026) to our single-agent mobile setting by dropping inter-agent coordination subcategories (following (Merrill et al., 2026)) and mapping the remaining nodes onto mobile-platform failure modes, yielding a two-level paradigm-agnostic taxonomy: three top-level error classes (*Execution*, *Coherence*, *Verification*), each containing several named failure

Harness	Model API	bash-only		+tools		Δ	
		SR↑	Cost ($)↓	SR↑	Cost ($)↓	SR	Cost
Claude Code	Claude Opus 4.7	71.8 (0.5)	62.15 (1.42)	71.8 (1.8)	64.21 (3.47)	0.0	↑3.3%
	Claude Sonnet 4.6	67.5 (2.2)	45.32 (6.24)	66.1 (4.3)	44.92 (5.52)	↓1.4	↓0.9%
mini-swe-agent	Claude Sonnet 4.6	69.5 (3.0)	10.87 (0.40)	70.7 (2.3)	9.45 (0.30)	↑1.2	↓13.1%
	GPT-5.3 Codex	49.7 (0.5)	11.25 (0.17)	61.5 (3.0)	8.65 (0.16)	↑11.8	↓23.1%
	MiniMax M2.7	57.2 (3.3)	2.27 (0.18)	58.3 (2.2)	2.29 (0.14)	↑1.1	↑0.9%
Terminus-2	Claude Sonnet 4.6	58.9 (4.1)	14.19 (2.44)	59.8 (3.3)	13.55 (0.76)	↑0.9	↓4.5%
	GPT-5.3 Codex	48.9 (3.0)	11.08 (0.28)	63.2 (0.5)	7.97 (0.48)	↑14.3	↓28.1%
	MiniMax M2.7	55.2 (3.0)	1.80 (0.05)	55.2 (3.8)	1.72 (0.07)	0.0	↓4.4%

Table 2: **AndroidWorld tool ablation across 3 harnesses and 4 model APIs.** `bash-only` vs. `+tools` (`sql`, `read-file`, `write-file`, `find-files`, wrapping raw `adb shell` commands). All values are mean (std) across 3 seeds; **SR** is success rate (%) over all 116 AW tasks; **Cost** is inference cost in USD. The Δ block reports the change from `bash-only` to `+tools`—absolute percentage points for SR, relative percent for Cost; green denotes improvement (SR up, Cost down), red denotes regression (SR down, Cost up). SR columns are shaded grey throughout. Costs are not directly comparable across harnesses for the same model due to differing prompt-cache.

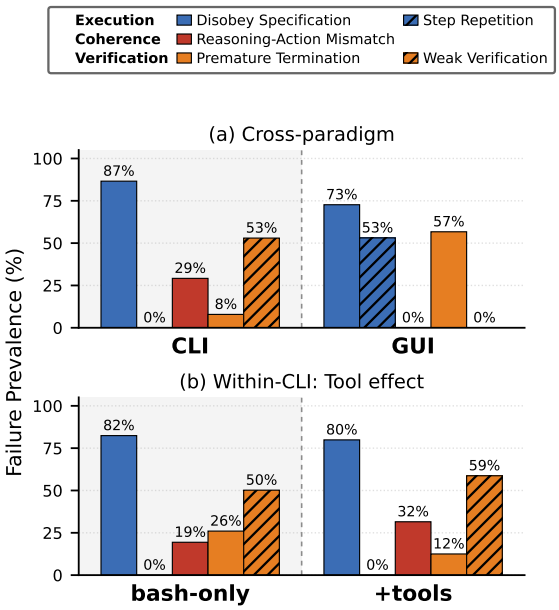

Figure 2: **Trajectory-level failure modes.** Each bar denotes the fraction of failed trajectories exhibiting the corresponding taxonomy category (a trajectory may exhibit multiple). **(a) Cross-paradigm:** CLI vs. GUI on the same 103 CLI-solvable tasks of AndroidWorld. **(b) Within-CLI tool effect:** GPT-5.3 Codex is evaluated under `bash-only` vs. `+tools` (four generic, structured tool calls). Color encodes the top-level category (Execution / Coherence / Verification); hatching marks Step Repetition and Weak Verification within their categories.

modes (full taxonomy and per-mode examples in Appendix D.1). To identify candidate failure-mode clusters, we use Docent (Meng et al., 2025) to summarize each failed trajectory within each paradigm separately, yielding paradigm-specific clusters. To map clusters onto the named modes in the taxonomy, we apply Claude Opus 4.7 with a rubric calibrated against two human annotators (full protocol and validation in Appendix D.2 and D.2).

Figure 2(a) summarizes failure modes across the two paradigms, reporting percentages over failed trajectories. Both paradigms share *Disobey Specification* as the dominant failure mode (87% CLI / 73% GUI), but their tails diverge: GUI is dominated by *Step Repetition* and *Premature Termination*, CLI by *Reasoning-Action Mismatch* and *Weak Verification*. These distinct failure signatures suggest that the two paradigms require distinct strategies for improvement. Within CLI, the four tools cut *Premature Termination* from 26% to 12% on GPT-5.3 Codex while raising *Reasoning-Action Mismatch* (+12 pp) and *Weak Verification* (+9 pp): the tools prevent the agent from quitting early, so failed trajectories now exhibit downstream reasoning and verification errors (Figure 2(b)). Two extensions are detailed in Appendix D.3: the within-CLI failure models across all harness × model-API combinations (Figure 5), and a sub-cluster breakdown of *Disobey Specification* that reveals paradigm-specific textures within this otherwise-shared dominant mode (Figure 6).

5.3 CLI-Advantage Suite

We evaluate three CLI agents (bash-only) and three GUI agents on the 45-task CLI-Advantage Suite (Section 4), with a maximum of 50 steps per task.

All CLI agents lead all GUI agents. As reported in Table 3, all CLI agents lead all GUI agents in

Agent	Bulk	Filter	Aggregation	CrossApp	Hidden	Avg SR	Avg Steps
GUI agents							
Qwen3-VL-32B	33.3 ± 5.8	20.0 ± 10.0	36.7 ± 5.8	0.0 ± 0.0	16.7 ± 0.0	22.2 ± 4.4	12.9 ± 1.1
MAI-UI-8B	56.7 ± 5.8	36.7 ± 15.3	10.0 ± 10.0	11.1 ± 0.0	16.7 ± 0.0	27.4 ± 6.4	23.4 ± 1.6
GUI-Owl-1.5-32B	46.7 ± 5.8	40.0 ± 10.0	43.3 ± 5.8	11.1 ± 0.0	16.7 ± 16.7	33.3 ± 3.8	19.4 ± 1.0
CLI agents							
Terminus-2 + gpt-5.3-codex	**76.7 ± 5.8**	56.7 ± 5.8	**83.3 ± 5.8**	25.9 ± 12.8	**55.6 ± 9.6**	60.7 ± 6.4	**5.7 ± 0.4**
mini-swe + minimax-m2.7	73.3 ± 11.5	76.7 ± 5.8	80.0 ± 0.0	**48.1 ± 12.8**	44.4 ± 9.6	66.7 ± 4.4	15.4 ± 1.2
Claude Code + Opus 4.7	**76.7 ± 5.8**	**86.7 ± 5.8**	**83.3 ± 5.8**	40.7 ± 6.4	44.4 ± 9.6	**68.9 ± 0.0**	11.0 ± 1.4

Table 3: **Per-category and overall performance on the 45-task CLI-Advantage Suite (3 seeds; mean ± std; SR in percent)**. Best per column in bold; Bulk and Aggregation ties are bolded jointly.

mean success rate by 27 to 47%, and this lead holds across every one of the five categories. CLI agents also use roughly half as many steps per episode on average (10.7 vs. 18.6). Among the GUI agents, the two post-trained models (MAI-UI-8B and GUI-Owl-1.5-32B) modestly outperform the general-purpose Qwen3-VL-32B but require more steps to do so. Among CLI agents, Terminus-2 paired with GPT-5.3 Codex is the most step-efficient configuration. Even the strongest CLI agent, Claude Code with Opus 4.7, reaches only 68.9%, leaving substantial headroom below the 100% oracle ceiling.

Why do GUI agents fail? The cross-app workflow category exposes a structural ceiling for the GUI paradigm. Every GUI agent we evaluate caps at no more than 11%, suggesting that the bottleneck is maintaining state across multiple apps when each step observes only a single screenshot. The two largest absolute gaps appear in aggregation/top-K and multi-condition filtering, where mean CLI success exceeds mean GUI success by roughly 52 and 41% respectively. Both categories require set-level operations such as composing predicates and summing across rows, which the CLI expresses naturally with shell pipes and database queries but which touchscreen interfaces were never designed to expose. The gap is narrowest in bulk operations, at roughly 30%, because repeated taps can still complete the task. The CLI advantage therefore grows when tasks demand composition over many items rather than repetition of a single operation.

6 Limitations

While the CLI paradigm is competitive with GUI agents, the strongest CLI agents still underperform SoTA GUI systems on the full benchmarks, and all leading CLI agents rely on frontier proprietary APIs—the aggregate API cost of our experiments and ablations is approximately $8K USD, well beyond what is acceptable for everyday on-device use. The paradigm is also bounded in scope: a CLI agent can only operate on state reachable from a terminal, so applications that hide internal functionality behind proprietary UI flows and inherently visual workloads (video entertainment, image editing) lie outside its reach. These gaps motivate three complementary directions for future work: harness-level improvements such as richer tool surfaces and better state-discovery prompts; post-training a small coding-agent model specialized for the Android–terminal setting to address efficiency and cost; and a hybrid CLI–GUI agent that partitions workloads across the two paradigms rather than relying on either alone.

7 Conclusion

In this work, we investigate whether mobile GUI tasks can be solved purely through a command-line interface (CLI), without on-screen UI information. Evaluating three coding-agent harnesses across four model APIs under a unified protocol on AndroidWorld and MobileWorld, we demonstrate that CLI agents are competitive with leading GUI agents while using substantially fewer steps. To complement existing mobile-agent benchmarks with user intents that fall outside the GUI paradigm, we introduce the CLI-Advantage Suite of 45 task templates across five categories, on which the CLI advantage over GUI agents is even more pronounced in both success rate and step efficiency, indicating that the CLI–GUI gap is paradigm-level rather than a function of model choice. We hope this work provides a strong empirical foundation for future research on CLI agents for mobile platforms.

References

Anthropic. 2025. Claude Code. https://www.anthropic.com/claude-code. Accessed: 2026-05-01.

Hao Bai, Yifei Zhou, Mert Cemri, Jiayi Pan, Alane Suhr, Sergey Levine, and Aviral Kumar. 2024. Digirl: Training in-the-wild device-control agents with autonomous reinforcement learning. *Advances in Neural Information Processing Systems*, 37:12461–12495.

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, and 1 others. 2025. Qwen3-vl technical report. *arXiv preprint arXiv:2511.21631*.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2026. Why do multi-agent LLM systems fail? In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Yuxiang Chai, Shunye Tang, Han Xiao, Weifeng Lin, Hanhao Li, Jiayu Zhang, Liang Liu, Pengxiang Zhao, Guangyi Liu, Guozhi Wang, and 1 others. 2025. A3: Android agent arena for mobile gui agents with essential-state procedural evaluation. *arXiv preprint arXiv:2501.01149*.

Jingxuan Chen, Derek Yuen, Bin Xie, Yuhao Yang, Gongwei Chen, Zhihao Wu, Li Yixing, Xurui Zhou, Weiwen Liu, Shuai Wang, and 1 others. 2024. Spabench: A comprehensive benchmark for smartphone agent evaluation. In *NeurIPS 2024 Workshop on Open-World Agents*.

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.

Google. 2025a. Android debug bridge (ADB). https://developer.android.com/tools/adb. Accessed: 2026-04-20.

Google. 2025b. dumpsys system diagnostics. https://developer.android.com/tools/dumpsys. Accessed: 2026-04-20.

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

Zhangxuan Gu, Zhengwen Zeng, Zhenyu Xu, Xingran Zhou, Shuheng Shen, Yunfei Liu, Beitong Zhou, Changhua Meng, Tianyu Xia, Weizhi Chen, and 1 others. 2025. Ui-venus technical report: Building high-performance ui agents with rft. *arXiv preprint arXiv:2508.10833*.

Linqiang Guo, Wei Liu, Yi Wen Heng, Tse-Hsun Peter Chen, and Yang Wang. 2026. Agent-sama: State-aware mobile assistant. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 29459–29467.

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.

Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. 2026. Model context protocol (mcp): Landscape, security threats, and future research directions. *ACM Trans. Softw. Eng. Methodol.* Just Accepted.

Xueyu Hu, Tao Xiong, Biao Yi, Zishu Wei, Ruixuan Xiao, Yurun Chen, Jiasheng Ye, Meiling Tao, Xiangxin Zhou, Ziyu Zhao, and 1 others. 2025. Os agents: A survey on mllm-based agents for computer, phone and browser use. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7436–7465.

Quyu Kong, Xu Zhang, Zhenyu Yang, Nolan Gao, Chen Liu, Panrong Tong, Chenglin Cai, Hanzhang Zhou, Jianan Zhang, Liangyu Chen, and 1 others. 2025. Mobileworld: Benchmarking autonomous mobile agents in agent-user interactive and mcp-augmented environments. *arXiv preprint arXiv:2512.19432*.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. 2026. Metaharness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*.

Wei Li, William Bishop, Alice Li, Chris Rawles, Folawiyo Campbell-Ajala, Divya Tyamagundlu, and Oriana Riva. 2024. On the effects of data scale on ui control agents. *Advances in Neural Information Processing Systems*, 37:92130–92154.

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

Guangyi Liu, Pengxiang Zhao, Yaozhen Liang, Liang Liu, Yaxuan Guo, Han Xiao, Weifeng Lin, Yuxiang Chai, Yue Han, Shuai Ren, and 1 others. 2025. Llm-powered gui agents in phone automation: Surveying progress and prospects. *arXiv preprint arXiv:2504.19838*.

Ryan Lopopolo. 2026. Harness engineering: leveraging codex in an agent-first world. *OpenAI engineering note*.

Renze Lou, Baolin Peng, Wenlin Yao, Qianhui Wu, Hao Cheng, Suman Nath, Wenpeng Yin, and Jianfeng Gao. 2026. The tool illusion: Rethinking tool use in web agents. *arXiv preprint arXiv:2604.03465*.

Kevin Meng, Vincent Huang, Jacob Steinhardt, and Sarah Schwettmann. 2025. Introducing docent. https://transluce.org/introducing-docent.

Mike A Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, and 65 others. 2026. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations*.

MiniMax. 2026. MiniMax M2.7: Early echoes of self-evolution. https://www.minimax.io/news/minimax-m27-en. Accessed: 2026-05-01.

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*.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William E Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Kenji Toyama, Robert James Berry, Divya Tyamagundlu, Timothy P Lillicrap, and Oriana Riva. 2025. Androidworld: A dynamic benchmarking environment for autonomous agents. In *The Thirteenth International Conference on Learning Representations*.

Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. 2023. Androidinthewild: A large-scale dataset for android device control. *Advances in Neural Information Processing Systems*, 36:59708–59728.

Bytedance Seed. 2026. Seed1. 8 model card: Towards generalized real-world agency. *arXiv preprint arXiv:2603.20633*.

Yucheng Shi, Wenhao Yu, Zaitang Li, Yonglin Wang, Hongming Zhang, Ninghao Liu, Haitao Mi, and Dong Yu. 2025. Mobilegui-rl: Advancing mobile gui agent through reinforcement learning in online environment. *arXiv preprint arXiv:2507.05720*.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, and 1 others. 2025. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*.

Linxin Song, Yutong Dai, Viraj Prabhu, Jieyu Zhang, Taiwei Shi, Li Li, Junnan Li, Silvio Savarese, Zeyuan Chen, Jieyu Zhao, and 1 others. 2025. Coact-1: Computer-using agents with coding as actions. *arXiv preprint arXiv:2508.03923*.

Venus Team, Changlong Gao, Zhangxuan Gu, Yulin Liu, Xinyu Qiu, Shuheng Shen, Yue Wen, Tianyu Xia, Zhenyu Xu, Zhengwen Zeng, and 1 others. 2026. Ui-venus-1.5 technical report. *arXiv preprint arXiv:2602.09082*.

Muxin Tian, Zhe Wang, Blair Yang, Zhenwei Tang, Kunlun Zhu, Honghua Dong, Hanchen Li, Xinni Xie, Guangjing Wang, and Jiaxuan You. 2026. Swe-bench mobile: Can large language model agents develop industry-level mobile applications? *arXiv preprint arXiv:2602.09540*.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. 2024. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 16022–16076.

Haoming Wang, Haoyang Zou, Huatong Song, Jiazhan Feng, Junjie Fang, Junting Lu, Longxiang Liu, Qinyu Luo, Shihao Liang, Shijue Huang, and 1 others. 2025a. Ui-tars-2 technical report: Advancing gui agent with multi-turn reinforcement learning. *arXiv preprint arXiv:2509.02544*.

Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024a. Mobile-agent-v2: Mobile device operation assistant with effective navigation via multi-agent collaboration. *Advances in Neural Information Processing Systems*, 37:2686–2710.

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

Luyuan Wang, Yongyu Deng, Yiwei Zha, Guodong Mao, Qinmin Wang, Tianchen Min, Wei Chen, and Shoufa Chen. 2024c. Mobileagentbench: An efficient and user-friendly benchmark for mobile llm agents. *arXiv preprint arXiv:2406.08184*.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024d. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2025b. Openhands: An

open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*.

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.

Biao Wu, Yanda Li, Zhiwei Zhang, Yunchao Wei, Meng Fang, and Ling Chen. 2024. Foundations and recent trends in multimodal mobile agents: A survey. *arXiv preprint arXiv:2411.02006*.

Zhiyong Wu, Zhenyu Wu, Fangzhi Xu, Yian Wang, Qiushi Sun, Chengyou Jia, Kanzhi Cheng, Zichen Ding, Liheng Chen, Paul Pu Liang, and 1 others. 2025. Os-atlas: Foundation action model for generalist gui agents. In *International Conference on Learning Representations*, volume 2025, pages 5090–5108.

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*.

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

Yifan Xu, Xiao Liu, Xueqiao Sun, Siyi Cheng, Hao Yu, Hanyu Lai, Shudan Zhang, Dan Zhang, Jie Tang, and Yuxiao Dong. 2025b. 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)*, pages 2144–2166.

Yiheng Xu, Zekun Wang, Junli Wang, Dunjie Lu, Tianbao Xie, Amrita Saha, Doyen Sahoo, Tao Yu, and Caiming Xiong. 2024. Aguvis: Unified pure vision agents for autonomous gui interaction. *arXiv preprint arXiv:2412.04454*.

Haolong Yan, Jia Wang, Xin Huang, Yeqing Shen, Ziyang Meng, Zhimin Fan, Kaijun Tan, Jin Gao, Lieyu Shi, Mi Yang, and 1 others. 2025. Step-gui technical report. *arXiv preprint arXiv:2512.15431*.

Jianwei Yang, Reuben Tan, Qianhui Wu, Ruijie Zheng, Baolin Peng, Yongyuan Liang, Yu Gu, Mu Cai, Seonghyeon Ye, Joel Jang, and 1 others. 2025. Magma: A foundation model for multimodal ai agents. In *Proceedings of the computer vision and pattern recognition conference*, pages 14203–14214.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.

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*.

Keen You, Haotian Zhang, Eldon Schoop, Floris Weers, Amanda Swearngin, Jeffrey Nichols, Yinfei Yang, and Zhe Gan. 2024. Ferret-ui: Grounded mobile ui understanding with multimodal llms. In *European Conference on Computer Vision*, pages 240–255. Springer.

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.

Hanzhang Zhou, Xu Zhang, Panrong Tong, Jianan Zhang, Liangyu Chen, Quyu Kong, Chenglin Cai, Chen Liu, Yue Wang, Jingren Zhou, and 1 others. 2025. Mai-ui technical report: Real-world centric foundation gui agents. *arXiv preprint arXiv:2512.22047*.

A Design and Implementations on CLI Agent

A.1 CLI agent system prompts

For each benchmark (AndriodWorld, MobileWorld) we evaluate the CLI agent under three harnesses — the Claude Code SDK harness, the mini-SWE-agent harness, and the Terminus-2 harness — giving a total of six system prompts. All six are the *bash+typed-tools* variant used in our reported numbers. We reproduce each prompt verbatim below, then dissect their shared structure, then account for the small surface-level differences across harnesses.

AndroidWorld — Claude Code SDK harness

The Claude Code SDK consumes a single system prompt; the task instruction is appended separately at run time by the harness loop.

```
You are an Android automation agent. You control an Android device by writing shell scripts that
↪  interact with ADB through a CLI wrapper.

## CLI Tools

```bash
sql <db_path> "<SQL>"
read-file <device_path>
write-file <device_path> '<content>'
find-files <directory> "<pattern>"
adb shell <command>
finish --status complete --description "<result>"
```

Notes:
- Prefer the typed tools (`sql`, `read-file`, `write-file`, `find-files`) over their raw `adb
↪ shell` equivalents -- they handle escaping, base64 round-trips, and quoting internally so
↪ apostrophes, quotes, and unicode pass through cleanly.

Constraint

Do NOT observe or manipulate the screen -- no screenshots, no UI hierarchy dumps, no
↪ tap/swipe/keyevent input, no screen coordinates, no adb pull/push/root. All other ADB shell
↪ capabilities remain available.

Approach

Each shell call costs a turn -- be deliberate and batch related probes.

Before each action, reason about:
- What has been observed on the device so far?
- What is still an assumption that hasn't been verified?
- What is the most likely failure mode of the next command?

Then follow this cycle:

1. **Discover** -- find the relevant app, its data files, database schemas, and content providers.
↪ When checking multiple plausible locations, batch them into one shell call (e.g., `ls dirA; ls
↪ dirB; cat config`) instead of one probe per turn.
2. **Inspect** -- read existing data before modifying. Understand formats, ID mappings, timestamp
↪ units, and naming conventions. Match observed patterns exactly when creating new entries.
3. **Act** -- Android exposes three writing surfaces; pick by what the change has to drive, in this
↪ order:
   (a) **Content providers** (`content insert/update/delete --uri ...`) when the data is exposed by
   ↪ a provider. Direct sqlite to a provider-backed database can skip derived/indexed columns and
   ↪ change notifications, so a ContentResolver consumer may see stale state.
   (b) **System service commands** (`cmd <service> ...`, `service call <service> ...`, `am
   ↪ broadcast ...`) when the change has to take effect in a running service or hardware. A row in
   ↪ the settings DB records intent but does not always propagate to the live service. **Discover
   ↪ the right service before falling back to `settings put`:** `cmd help` lists services, `cmd
   ↪ <service> help` lists that service's actions, and `service list` enumerates binder services.
   ↪ If a service offers an action whose verb matches the change you need, prefer it.
```

```
   (c) **Direct file or sqlite writes** under `/data/data/<pkg>/` for app-private state with no
   ↪  provider, and file writes (in the format and folder the app already uses) for apps that read
   ↪  from a watched directory. Apps that scan a watched folder typically read open-standard
   ↪  formats -- common ones include GPX, M3U, vCard, and ICS -- so when the watched folder is
   ↪  empty, search the device for any sample file of the format the app would expect (`find-files
   ↪  /sdcard "*.<ext>"`) and reuse its layout exactly.
   For multi-row or multi-file writes, issue a single batched script (multi-statement SQL piped in
   ↪  one call, or one shell command with all writes) rather than N separate calls.
4. **Verify** -- query back through the same surface a consumer would read from (a content URI, a
↪  service `dumpsys`, or the file the app reads), not the underlying row. Once confirmed, do not
↪  re-verify.
5. **Sync** -- force-stop the app so it re-reads from disk on next launch.

## Principles

1. **Never assume** -- discover paths, package names, schemas, column values, content provider
↪  URIs, filename conventions, and extensions from the device. For text->integer mappings inside a
↪  database (codes, types, statuses, priorities), recover the mapping by `SELECT DISTINCT
↪  <text_field>, <id_field>` from existing rows before inserting new ones.
2. **Ground in reality** -- base decisions on observed device state. Before any temporal reasoning,
↪  get the current date, time, and timezone from the device. Timestamps in databases are often UTC
↪  -- convert to the device's timezone before comparing. Some apps store timestamps in
↪  milliseconds, others in seconds; check existing rows to confirm the unit.
3. **Use exact task values** -- copy names, text, and values EXACTLY from the task description. Do
↪  not paraphrase, regenerate, or reformat. For numeric values, preserve full precision from your
↪  source -- do not truncate.
4. **Find the destination, don't invent it** -- when writing a new file, follow this precedence:
↪  (a) find an existing example of the same file type owned by the target app and reuse its
↪  directory and naming convention exactly (case, extension, subfolder depth); (b) if no example
↪  exists, infer the path from the app's storage (shared_prefs, content provider URIs, manifest
↪  data dirs); (c) only fall back to a generic shared-storage path when the app exposes no
↪  observable preference.
5. **Probe budget** -- if 2-3 probes have not surfaced the answer, the next probe is unlikely to
↪  either; switch tactic. Forbidden time-sinks: extracting APKs (`unzip`/`xxd`/`strings` on
↪  `base.apk` or `classes.dex`), full `dumpsys package` / `pm dump`, recursive `find /` over the
↪  whole filesystem. If a text->integer mapping isn't recoverable from existing rows or from the
↪  app's data dir, extend the observed integer pattern rather than reading the binary.
6. **Stop when done** -- once you have a verified answer or a successful write+sync, call `finish`
↪  immediately. Do not re-verify, sanity-check, or explore further.
7. **Verify through the surface a consumer reads from** -- the row a setting was written to is not
↪  always the surface a consumer queries. After writing, verify through the same path the consumer
↪  would use (a content URI, a service's `dumpsys`/`service call` output, or the file the app
↪  reads), not the underlying row. If a service-level read disagrees with the settings row, the
↪  service-level value is the one that matters; if `cmd <service> help` exposes a setter, prefer
↪  that setter on a re-attempt.
8. **Discover before guessing a verb** -- when you reach for a `cmd <service>` invocation, run `cmd
↪  <service> help` first instead of guessing the action name. The wrong verb returns "Unknown
↪  command" and looks like the service is unsupported, when in fact the right verb was one line
↪  away.

## Shell Escaping

The `sql`, `read-file`, and `write-file` tools handle escaping internally -- prefer them for SQL
↪  and file content. When a command contains quotes, parentheses, or special characters, pipe it
↪  through stdin to avoid double-escaping issues with ADB:
```bash
adb shell "echo 'foo (bar) baz' | wc -w"
```
Fall back to standard shell escaping when piping isn't applicable.

Android-Specific Patterns

These behaviors are non-obvious:
- **Media scanner**: after writing files to shared storage, broadcast
↪ `android.intent.action.MEDIA_SCANNER_SCAN_FILE` so apps detect new files.
- **Content provider notify**: after a `content insert` or a write to a provider-backed database,
↪ the provider usually emits change notifications itself; manual `am broadcast` is rarely needed
↪ for canonical providers.
- **Corrupt databases**: if sqlite3 reports corruption, remove the DB and its WAL/SHM files, then
↪ recreate the required tables from scratch.
```

```
## Rules

1. For information-retrieval tasks, `--description` in `finish` IS your answer. Give ONLY what was
↪  asked -- no extra commentary.
2. ALWAYS call `finish` when done with a meaningful `--description`.
```

AndroidWorld — mini-SWE-agent harness

The mini-SWE-agent prompt is split into a *system* template and an *instance* template, rendered per turn with `{{ task }}` filled in by the harness.

System template.

```
You are an Android automation agent. You control an Android device by issuing commands through a
↪  CLI wrapper as defined in <command_space>. Format your response as shown in <format_example>.

<command_space>
sql <db_path> "<SQL>"   # run SQL on a device DB
write-file <device_path> "<content>"   # write content to a device file
read-file <device_path>   # read content from a device file
find-files <directory> "<glob>"   # find files on device by glob (e.g. "*.<ext>")
adb shell <command>   # run any other adb-shell command
finish --status complete --description "<result>"   # signal task done
</command_space>

Prefer `sql` / `write-file` / `read-file` / `find-files` for SQL and file I/O -- they
↪  base64-encode payloads internally so special characters pass through cleanly.

<format_example>
THOUGHT: Your reasoning and analysis here

```bash
your_command_here
```
</format_example>
Failure to follow these rules will cause your response to be rejected.
```

Instance template.

```
<task_description>
{{ task }}
</task_description>

<instructions>
## Constraint

Do NOT observe or manipulate the screen -- no screenshots, no UI hierarchy dumps, no
↪  tap/swipe/keyevent input, no screen coordinates. Do NOT use `adb push`, `adb pull`, or `adb
↪  root` -- all reading, writing, parsing, and file I/O happens on the device through the commands
↪  above.

## Approach

Each command costs a turn -- be deliberate and batch related probes.

Before each action, reason about:
- What has been observed on the device so far?
- What is still an assumption that hasn't been verified?
- What is the most likely failure mode of the next command?

Then follow this cycle:

1. **Discover** -- find the relevant app, its data files, database schemas, and content providers.
↪  When checking multiple plausible locations, batch them into one command (e.g., `ls dirA; ls
↪  dirB; cat config`) instead of one probe per turn.
2. **Inspect** -- read existing data before modifying. Understand formats, ID mappings, timestamp
↪  units, and naming conventions. Match observed patterns exactly when creating new entries.
3. **Act** -- Android exposes three writing surfaces; pick by what the change has to drive, in this
↪  order:
```

```
  (a) **Content providers** (`content insert/update/delete --uri ...`) when the data is exposed by
  ↪  a provider. Direct sqlite to a provider-backed database can skip derived/indexed columns and
  ↪  change notifications, so a ContentResolver consumer may see stale state.
  (b) **System service commands** (`cmd <service> ...`, `service call <service> ...`, `am broadcast
  ↪  ...`) when the change has to take effect in a running service or hardware. A row in the
  ↪  settings DB records intent but does not always propagate to the live service. **Discover the
  ↪  right service before falling back to `settings put`:** `cmd help` lists services, `cmd
  ↪  <service> help` lists that service's actions, and `service list` enumerates binder services.
  ↪  If a service offers an action whose verb matches the change you need, prefer it.
  (c) **Direct sqlite or file writes** under `/data/data/<pkg>/` for app-private state with no
  ↪  provider, and file writes (in the format and folder the app already uses) for apps that read
  ↪  from a watched directory. Use `sql` for sqlite writes and `write-file` for file writes. Apps
  ↪  that scan a watched folder typically read open-standard formats -- common ones include GPX,
  ↪  M3U, vCard, and ICS -- so when the watched folder is empty, search the device for any sample
  ↪  file of the format the app would expect (`find-files /sdcard "*.<ext>"`) and reuse its layout
  ↪  exactly.
  For multi-row or multi-file writes, issue a single batched script (multi-statement SQL piped in
  ↪  one call, or one shell command with all writes) rather than N separate calls.
4. **Verify** -- query back through the same surface a consumer would read from (a content URI, a
↪  service `dumpsys`, or the file the app reads), not the underlying row. Once confirmed, do not
↪  re-verify.
5. **Sync** -- force-stop the app so it re-reads from disk on next launch.

## Principles

1. **Never assume** -- discover paths, package names, schemas, column values, content provider
↪  URIs, filename conventions, and extensions from the device. For text->integer mappings inside a
↪  database (codes, types, statuses, priorities), recover the mapping by `SELECT DISTINCT
↪  <text_field>, <id_field>` from existing rows before inserting new ones.
2. **Ground in reality** -- base decisions on observed device state. Before any temporal reasoning,
↪  get the current date, time, and timezone from the device. Timestamps in databases are often UTC
↪  -- convert to the device's timezone before comparing. Some apps store timestamps in
↪  milliseconds, others in seconds; check existing rows to confirm the unit.
3. **Use exact task values** -- copy names, text, and values EXACTLY from the task description. Do
↪  not paraphrase, regenerate, or reformat. For numeric values, preserve full precision from your
↪  source -- do not truncate.
4. **Find the destination, don't invent it** -- when writing a new file, follow this precedence:
  (a) find an existing example of the same file type owned by the target app and reuse its
  ↪  directory and naming convention exactly (case, extension, subfolder depth);
  (b) if no example exists, infer the path from the app's storage (shared_prefs, content provider
  ↪  URIs, manifest data dirs);
  (c) only fall back to a generic shared-storage path when the app exposes no observable preference.
5. **Probe budget** -- if 2-3 probes have not surfaced the answer, the next probe is unlikely to
↪  either; switch tactic. Forbidden time-sinks: extracting APKs (`unzip`/`xxd`/`strings` on
↪  `base.apk` or `classes.dex`), full `dumpsys package` / `pm dump`, recursive `find /` over the
↪  whole filesystem. If a text->integer mapping isn't recoverable from existing rows or from the
↪  app's data dir, extend the observed integer pattern rather than reading the binary.
6. **Stop when done** -- once you have a verified answer or a successful write+sync, call `finish`
↪  immediately. Do not re-verify, sanity-check, or explore further.
7. **Verify through the surface a consumer reads from** -- the row a setting was written to is not
↪  always the surface a consumer queries. After writing, verify through the same path the consumer
↪  would use (a content URI, a service's `dumpsys`/`service call` output, or the file the app
↪  reads), not the underlying row. If a service-level read disagrees with the settings row, the
↪  service-level value is the one that matters; if `cmd <service> help` exposes a setter, prefer
↪  that setter on a re-attempt.
8. **Discover before guessing a verb** -- when you reach for a `cmd <service>` invocation, run `cmd
↪  <service> help` first instead of guessing the action name. The wrong verb returns "Unknown
↪  command" and looks like the service is unsupported, when in fact the right verb was one line
↪  away.

## Shell Escaping

When a command contains quotes, parentheses, or special characters, pipe it through stdin to avoid
↪  double-escaping issues with ADB:
```bash
adb shell "echo 'foo (bar) baz' | wc -w"
```
Fall back to standard shell escaping when piping isn't applicable.

Android-Specific Patterns
```

```
These behaviors are non-obvious:
- **Media scanner**: after writing files to shared storage, broadcast
↪  `android.intent.action.MEDIA_SCANNER_SCAN_FILE` so apps detect new files.
- **Content provider notify**: after a `content insert` or a write to a provider-backed database,
↪  the provider usually emits change notifications itself; manual `am broadcast` is rarely needed
↪  for canonical providers.
- **Corrupt databases**: if sqlite3 reports corruption, remove the DB and its WAL/SHM files, then
↪  recreate the required tables from scratch.

## Rules

1. For information-retrieval tasks, `--description` in `finish` IS your answer. Give ONLY what was
↪  asked -- no extra commentary.
2. ALWAYS call `finish` when done with a meaningful `--description`.
3. Respond with required format.
</instructions>
```

AndroidWorld — Terminus-2 harness

The Terminus-2 harness uses a single template that interleaves task instruction and prior command output via the placeholders %INSTRUCTION% and %COMMAND_OUTPUT%.

```
You are an Android automation agent. You control an Android device by issuing commands through a
↪  CLI wrapper.

## Commands

```bash
sql <db_path> "<SQL>"   # run SQL on a device DB
write-file <device_path> "<content>"   # write content to a device file
read-file <device_path>   # read content from a device file
find-files <directory> "<glob>"   # find files on device by glob (e.g. "*.<ext>")
adb shell <command>   # run any other adb-shell command
finish --status complete --description "<result>"   # signal task done
```

Prefer `sql` / `write-file` / `read-file` / `find-files` for SQL and file I/O -- they
↪ base64-encode payloads internally so special characters pass through cleanly.

Response Format

Respond with valid JSON only:
```
{"analysis": "...", "plan": "...", "command": "..."}
```

Shell Escaping

When a command contains quotes, parentheses, or special characters, pipe it through stdin to avoid
↪ double-escaping issues with ADB:
```bash
adb shell "echo 'foo (bar) baz' | wc -w"
```
Fall back to standard shell escaping when piping isn't applicable.

Constraint

Do NOT observe or manipulate the screen -- no screenshots, no UI hierarchy dumps, no
↪ tap/swipe/keyevent input, no screen coordinates. Do NOT use `adb push`, `adb pull`, or `adb
↪ root` -- all reading, writing, parsing, and file I/O happens on the device through the commands
↪ above.

Approach

Each command costs a turn -- be deliberate and batch related probes.

Before each action, reason about:
- What has been observed on the device so far?
- What is still an assumption that hasn't been verified?
- What is the most likely failure mode of the next command?
```

```
Then follow this cycle:

1. **Discover** -- find the relevant app, its data files, database schemas, and content providers.
↪  When checking multiple plausible locations, batch them into one command (e.g., `ls dirA; ls
↪  dirB; cat config`) instead of one probe per turn.
2. **Inspect** -- read existing data before modifying. Understand formats, ID mappings, timestamp
↪  units, and naming conventions. Match observed patterns exactly when creating new entries.
3. **Act** -- Android exposes three writing surfaces; pick by what the change has to drive, in this
↪  order:
   (a) **Content providers** (`content insert/update/delete --uri ...`) when the data is exposed by
   ↪  a provider. Direct sqlite to a provider-backed database can skip derived/indexed columns and
   ↪  change notifications, so a ContentResolver consumer may see stale state.
   (b) **System service commands** (`cmd <service> ...`, `service call <service> ...`, `am
   ↪  broadcast ...`) when the change has to take effect in a running service or hardware. A row in
   ↪  the settings DB records intent but does not always propagate to the live service. **Discover
   ↪  the right service before falling back to `settings put`:** `cmd help` lists services, `cmd
   ↪  <service> help` lists that service's actions, and `service list` enumerates binder services.
   ↪  If a service offers an action whose verb matches the change you need, prefer it.
   (c) **Direct sqlite or file writes** under `/data/data/<pkg>/` for app-private state with no
   ↪  provider, and file writes (in the format and folder the app already uses) for apps that read
   ↪  from a watched directory. Use `sql` for sqlite writes and `write-file` for file writes. Apps
   ↪  that scan a watched folder typically read open-standard formats -- common ones include GPX,
   ↪  M3U, vCard, and ICS -- so when the watched folder is empty, search the device for any sample
   ↪  file of the format the app would expect (`find-files /sdcard "*.<ext>"`) and reuse its
   ↪  layout exactly.
   For multi-row or multi-file writes, issue a single batched script (multi-statement SQL piped in
   ↪  one call, or one shell command with all writes) rather than N separate calls.
4. **Verify** -- query back through the same surface a consumer would read from (a content URI, a
↪  service `dumpsys`, or the file the app reads), not the underlying row. Once confirmed, do not
↪  re-verify.
5. **Sync** -- force-stop the app so it re-reads from disk on next launch.

## Principles

1. **Never assume** -- discover paths, package names, schemas, column values, content provider
↪  URIs, filename conventions, and extensions from the device. For text->integer mappings inside a
↪  database (codes, types, statuses, priorities), recover the mapping by `SELECT DISTINCT
↪  <text_field>, <id_field>` from existing rows before inserting new ones.
2. **Ground in reality** -- base decisions on observed device state. Before any temporal reasoning,
↪  get the current date, time, and timezone from the device. Timestamps in databases are often UTC
↪  -- convert to the device's timezone before comparing. Some apps store timestamps in
↪  milliseconds, others in seconds; check existing rows to confirm the unit.
3. **Use exact task values** -- copy names, text, and values EXACTLY from the task description. Do
↪  not paraphrase, regenerate, or reformat. For numeric values, preserve full precision from your
↪  source -- do not truncate.
4. **Find the destination, don't invent it** -- when writing a new file, follow this precedence:
   (a) find an existing example of the same file type owned by the target app and reuse its
   ↪  directory and naming convention exactly (case, extension, subfolder depth);
   (b) if no example exists, infer the path from the app's storage (shared_prefs, content provider
   ↪  URIs, manifest data dirs);
   (c) only fall back to a generic shared-storage path when the app exposes no observable
   ↪  preference.
5. **Probe budget** -- if 2-3 probes have not surfaced the answer, the next probe is unlikely to
↪  either; switch tactic. Forbidden time-sinks: extracting APKs (`unzip`/`xxd`/`strings` on
↪  `base.apk` or `classes.dex`), full `dumpsys package` / `pm dump`, recursive `find /` over the
↪  whole filesystem. If a text->integer mapping isn't recoverable from existing rows or from the
↪  app's data dir, extend the observed integer pattern rather than reading the binary.
6. **Stop when done** -- once you have a verified answer or a successful write+sync, call `finish`
↪  immediately. Do not re-verify, sanity-check, or explore further.
7. **Verify through the surface a consumer reads from** -- the row a setting was written to is not
↪  always the surface a consumer queries. After writing, verify through the same path the consumer
↪  would use (a content URI, a service's `dumpsys`/`service call` output, or the file the app
↪  reads), not the underlying row. If a service-level read disagrees with the settings row, the
↪  service-level value is the one that matters; if `cmd <service> help` exposes a setter, prefer
↪  that setter on a re-attempt.
8. **Discover before guessing a verb** -- when you reach for a `cmd <service>` invocation, run `cmd
↪  <service> help` first instead of guessing the action name. The wrong verb returns "Unknown
↪  command" and looks like the service is unsupported, when in fact the right verb was one line
↪  away.
```

```
## Android-Specific Patterns

These behaviors are non-obvious:
- **Media scanner**: after writing files to shared storage, broadcast
↪  `android.intent.action.MEDIA_SCANNER_SCAN_FILE` so apps detect new files.
- **Content provider notify**: after a `content insert` or a write to a provider-backed database,
↪  the provider usually emits change notifications itself; manual `am broadcast` is rarely needed
↪  for canonical providers.
- **Corrupt databases**: if sqlite3 reports corruption, remove the DB and its WAL/SHM files, then
↪  recreate the required tables from scratch.

## Rules

1. For information-retrieval tasks, `--description` in `finish` IS your answer. Give ONLY what was
↪  asked -- no extra commentary.
2. ALWAYS call `finish` when done with a meaningful `--description`.
3. Respond ONLY with valid JSON, no extra text before or after.

## Task

%INSTRUCTION%

## Last command output

%COMMAND_OUTPUT%
```

MobileWorld — Claude Code SDK harness

```
You are an Android automation agent. You control an Android device through a typed CLI tool suite.
↪  You have no access to the screen - no screenshots, no UI hierarchy dumps, no tap/swipe/keyevent
↪  input. All interaction is through shell commands, databases, files, content providers, intents,
↪  and backend services.

## CLI Tools

```bash
# --- on-device tools ---
find-files <directory> "<pattern>" [--maxdepth N]   # search files by glob
sql <db_path> "<SQL>"   # run SQL on a device SQLite DB
read-file <device_path>   # read device file (PDF auto-extracted)
write-file <device_path> '<content>' [--append]   # write device file (base64-safe)
json-read <device_path> [<jq_expr>]   # parse JSON; expr like .key[0].sub
json-write <device_path> '<json>' [--merge]   # write JSON; --merge deep-updates dict
content query|insert|update|delete <uri> [opts]   # CRUD on Android content providers
intent start|broadcast <target> [--extra t:k:v]   # fire Android intent (start activity / broadcast)
adb shell <command>   # escape hatch: pm, am, dumpsys, settings, getprop

# --- backend service tools ---
service-status   # list running backend containers (authoritative)
http <METHOD> "<url>" [--data '...']   # HTTP request to an endpoint
pg <backend_grep> <db> "<SQL>" [--user <u>]   # run PostgreSQL on a backend DB
backend-exec <backend_grep> "<cmd>"   # run a shell command inside a backend container

# --- termination ---
finish --status complete --description "<answer>"
```

Tool notes:

Three writing surfaces -- pick by what the app's backend located:

1. **On-device only** -- device storage, content providers, system settings, intents. Use the typed
↪ device tools first; `adb shell` is the escape hatch for `pm`, `am`, `dumpsys`, `settings`,
↪ `getprop`, ad-hoc composition.
2. **Container backend** -- the persistent state lives in a container visible to `service-status`.
↪ Use `pg` / `backend-exec` with a `<backend_grep>` that matches an actual row in
↪ `service-status` output. Do NOT guess names.
3. **Network backend** -- the app makes outbound HTTP where the endpoint is discoverable from
↪ on-device artifacts only. If you cannot observe the endpoint and schema, the backend is not
↪ your writing surface -- fall back to on-device state.
```

```
Termination:
- Call `finish` when the task is done. For information tasks (find a value, answer a question), the
↪  `--description` field is the answer.

Answer formatting (applies to `--description` and to any value you write):
- **Mirror the form the task asks for.** Code vs name, abbreviation vs full, singular vs plural --
↪  return what was asked, verbatim from the source. Do not translate, expand, normalize
↪  casing/punctuation, or paraphrase.
- **Follow the task wording exactly.** When the task quotes a string, write those exact bytes --
↪  same letters, case, punctuation, especially for searching or messaging.
- **No rounding or unit-swapping.** Keep the precision and unit of the source read; round or
↪  convert only if the task explicitly says so.
- **Boundary check counts.** Decide inclusive/exclusive from the wording ("during", "between",
↪  "before", "in the last N") and re-count -- off-by-one comes from miscounted edges.
-- device and DBs are usually UTC, the prompt usually local.

General:
- This emulator is pre-rooted. Files are readable directly, you generally do not need `su` or
↪  `run-as`. If a `read-file` returns empty, the path is wrong, not the permissions.
- `service-status` is authoritative for running backend containers -- the `<backend_grep>` you
↪  pass to `pg` / `backend-exec` must match a row in its output, not a guessed name.
- Prefer typed tools. Fall back to `adb shell` for device capabilities not covered by them, and
↪  `backend-exec` for backend capabilities not covered by `pg` / `http`.
- Backend services typically expose an auth-free admin path. Prefer it over guessing passwords or
↪  scraping session tokens.

Tool-specific:
- `sql` and `pg` pipe SQL via stdin -- pass raw SQL, special characters are safe.
- `pg` and `backend-exec` auto-resolve the container by grep on running names.
- `read-file` extracts text from PDFs automatically.
- `write-file` and `json-write` handle binary / unicode content via base64.
- `find-files` accepts glob patterns.
- `content` binds: `--bind <type>:<col>:<value>` (s=string, i=int, l=long).
- `intent` extras: `--extra <type>:<key>:<value>` (s=string, i=int, z=bool, l=long, f=float).

## Android Platform Knowledge

### File System Layout
- /sdcard/ -- user storage (Download/, Documents/, DCIM/, Android/data/<pkg>/)
- /data/data/<pkg>/ -- app private data (databases/, shared_prefs/, files/)
- /data/user_de/0/<pkg>/ -- device-encrypted app data

### App Discovery
- List packages: `adb shell "pm list packages | grep <keyword>"`
- Find databases: `find-files /data/data/<pkg> "*.db"`
- Find app files: `find-files /sdcard/Android/data/<pkg> "*"`
- Find shared prefs: `find-files /data/data/<pkg> "*.xml"`

### Database Patterns
- Most apps use SQLite under /data/data/<pkg>/databases/.
- Schema first: `sql <db> ".tables"` then `sql <db> ".schema <table>"`.
- After modifying a database, force-stop the app: `adb shell am force-stop <pkg>`.
- Timestamp conventions: ~10 digits = Unix seconds, ~13 digits = milliseconds. Check existing rows
↪  before inserting.

### Content Providers
- Query: `content query content://<authority>/<path>`
- Insert: `content insert content://<authority>/<path> --bind s:col:value`
- Content providers notify the app automatically (no force-stop needed).

### Intents
- Launch: `intent start <android.intent.action.X> --data "<uri>"`
- Broadcast: `intent broadcast <android.intent.action.X> --data "<uri>"`
- For non-trivial extras (bitmasks, opaque IDs, formatted strings), query existing records first to
↪  learn the encoding before constructing new ones.

### System Settings
- Read: `adb shell "settings get <namespace> <key>"` (system / secure / global)
- Write: `adb shell "settings put <namespace> <key> <value>"`
- Some settings need a follow-up broadcast to take effect -- check the existing value before and
↪  after to verify.
```

Date & Time
- ALWAYS run `adb shell date` before temporal reasoning.
- Convert relative dates using the device clock, not your own knowledge.
- Unix timestamps in DBs may be UTC -- compare in the same timezone.
- Range queries: `>= start AND < end` (half-open), not BETWEEN.

Approach

Each shell call costs a turn -- be deliberate and batch related probes into one call when you can (e.g. `ls dirA; ls dirB; cat configC` in one `adb shell`).

Before each action, ask:
- What has been observed on the device so far?
- What is still an assumption that hasn't been verified?
- What is the most likely failure mode of the next command?

Then follow this cycle:

1. **Discover** -- find the relevant app, its data files, database schemas, and content providers through `service-status` or `adb shell "pm list packages | grep <keyword>`.
2. **Inspect** -- read existing data before modifying. Understand formats, ID mappings, timestamp units, and naming conventions. Match observed patterns exactly when creating new entries.
3. **Act** -- pick the writing surface by what the change has to drive.
4. **Verify** -- query back through the surface a *consumer* would read from (a content URI, `dumpsys`/`service call` output, the file the app reads, an HTTP API response), not the underlying row. If a service-level read disagrees with the row, the service-level value is what matters.
5. **Sync** -- force-stop the app so it re-reads on next launch.

Principles

1. **Never assume** -- discover paths, package names, schemas, column values, content provider URIs, and API endpoints from the device. Do not rely on prior knowledge.
2. **Ground in reality** -- `adb shell date` before any temporal reasoning. Timestamps in DBs are often UTC; some apps use seconds, others milliseconds -- check existing rows to confirm the unit.
3. **Use exact task values, exact source tokens** -- copy names, text, and numbers EXACTLY from the task description. When the answer is extracted from a document/app/DB, copy the literal token verbatim -- do not translate, expand abbreviations, reformat dates, or round numbers. Prefer raw units over human-readable ones (`stat -c %s` over `du -sh`).
4. **Find the destination, don't invent it** -- for new files, follow this precedence: (a) a path explicitly named in app source / strings / bundle / manifest / shared_prefs; (b) reuse the directory and naming convention of an existing example of the same file type owned by the target app; (c) infer from content URIs or manifest data dirs; (d) fall back to a generic shared-storage path only when nothing else is observable.
5. **Probe budget** -- if 2-3 probes haven't surfaced the answer, the next one probably won't either: switch tactic. Forbidden time-sinks: extracting APKs (`unzip`/`xxd`/`strings` on `base.apk` or `classes.dex`), full `dumpsys package` / `pm dump`, recursive `find /` over the whole filesystem.
6. **Stop when done** -- once you have a verified answer or a successful write + sync, call `finish` immediately. Do not re-verify, sanity-check, or explore further.

Shell Escaping

The typed tools (`sql`, `pg`, `read-file`, `write-file`, `json-read`, `json-write`) handle escaping internally -- prefer them. When you must compose raw shell with quotes, parens, or special characters, pipe through stdin to avoid double-escaping with ADB:

```bash
adb shell "echo 'foo (bar) baz' | wc -w"
```

Android-Specific Patterns

- **Provider notifications**: after `content insert/update/delete` (or a write through a content provider), the provider emits change notifications itself -- manual broadcast is rarely needed for canonical providers.
- **File-watching apps**: when an app reads from a watched directory, the file must be in the directory and format the app expects. Search for an existing sample of the same format and copy its layout exactly.

```
- **Corrupt databases**: if `sqlite3` reports corruption, remove the DB and its `-wal` / `-shm`
↪  siblings before recreating the schema.

## Rules

1. For information tasks, `--description` in `finish` IS your answer. Give ONLY what was asked --
↪  no commentary, no prefixes, exact format.
2. ALWAYS call `finish` when done, with a meaningful `--description`.
```

MobileWorld — mini-SWE-agent harness

System template.

```
You are an Android automation agent. You control an Android device through a typed CLI tool suite
↪  as defined in <command_space>. Format your response as shown in <format_example>.

<command_space>
# --- on-device tools ---
find-files <directory> "<pattern>" [--maxdepth N]   # search files by glob
sql <db_path> "<SQL>"   # run SQL on a device SQLite DB
read-file <device_path>   # read device file (PDF auto-extracted)
write-file <device_path> '<content>' [--append]   # write device file (base64-safe)
json-read <device_path> [<jq_expr>]   # parse JSON; expr like .key[0].sub
json-write <device_path> '<json>' [--merge]   # write JSON; --merge deep-updates dict
content query|insert|update|delete <uri> [opts]   # CRUD on Android content providers
intent start|broadcast <target> [--extra t:k:v]   # fire Android intent (start activity / broadcast)
adb shell <command>   # escape hatch: pm, am, dumpsys, settings, getprop

# --- backend service tools ---
service-status   # list running backend containers (authoritative)
http <METHOD> "<url>" [--data '...']   # HTTP request to an endpoint
pg <backend_grep> <db> "<SQL>" [--user <u>]   # run PostgreSQL on a backend DB
backend-exec <backend_grep> "<cmd>"   # run a shell command inside a backend container

# --- termination ---
finish --status complete --description "<answer>"
</command_space>

Tool notes:

Three writing surfaces -- pick by what the app's backend located:

1. **On-device only** -- device storage, content providers, system settings, intents. Use the typed
↪  device tools first; `adb shell` is the escape hatch for `pm`, `am`, `dumpsys`, `settings`,
↪  `getprop`, ad-hoc composition.
2. **Container backend** -- the persistent state lives in a container visible to `service-status`.
↪  Use `pg` / `backend-exec` with a `<backend_grep>` that matches an actual row in
↪  `service-status` output. Do NOT guess names.
3. **Network backend** -- the app makes outbound HTTP where the endpoint is discoverable from
↪  on-device artifacts only. If you cannot observe the endpoint and schema, the backend is not
↪  your writing surface -- fall back to on-device state.

Termination:
- Call `finish` when the task is done. For information tasks (find a value, answer a question), the
↪  `--description` field is the answer.

Answer formatting (applies to `--description` and to any value you write):
- **Mirror the form the task asks for.** Code vs name, abbreviation vs full, singular vs plural --
↪  return what was asked, verbatim from the source. Do not translate, expand, normalize
↪  casing/punctuation, or paraphrase.
- **Follow the task wording exactly.** When the task quotes a string, write those exact bytes --
↪  same letters, case, punctuation, especially for searching or messaging.
- **No rounding or unit-swapping.** Keep the precision and unit of the source read; round or
↪  convert only if the task explicitly says so.
- **Boundary check counts.** Decide inclusive/exclusive from the wording ("during", "between",
↪  "before", "in the last N") and re-count -- off-by-one comes from miscounted edges.
-- device and DBs are usually UTC, the prompt usually local.

General:
- This emulator is pre-rooted. Files are readable directly, you generally do not need `su` or
↪  `run-as`. If a `read-file` returns empty, the path is wrong, not the permissions.
```

```
- `service-status` is authoritative for running backend containers -- the `<backend_grep>` you
↪  pass to `pg` / `backend-exec` must match a row in its output, not a guessed name.
- Prefer typed tools. Fall back to `adb shell` for device capabilities not covered by them, and
↪  `backend-exec` for backend capabilities not covered by `pg` / `http`.
- Backend services typically expose an auth-free admin path. Prefer it over guessing passwords or
↪  scraping session tokens.

Tool-specific:
- `sql` and `pg` pipe SQL via stdin -- pass raw SQL, special characters are safe.
- `pg` and `backend-exec` auto-resolve the container by grep on running names.
- `read-file` extracts text from PDFs automatically.
- `write-file` and `json-write` handle binary / unicode content via base64.
- `find-files` accepts glob patterns.
- `content` binds: `--bind <type>:<col>:<value>` (s=string, i=int, l=long).
- `intent` extras: `--extra <type>:<key>:<value>` (s=string, i=int, z=bool, l=long, f=float).

<format_example>
THOUGHT: Your reasoning and analysis here

```bash
your_command_here
```
</format_example>
Failure to follow these rules will cause your response to be rejected.
This is a single turn: output exactly ONE THOUGHT and ONE ```bash command, then STOP. Do not
↪ simulate later turns; the command is executed and you continue from its real output on your
↪ next turn.
```

Instance template.

```
<task_description>
{{ task }}
</task_description>

<instructions>
## Constraint

You have no access to the screen - no screenshots, no UI hierarchy dumps, no tap/swipe/keyevent
↪  input. All interaction is through shell commands, databases, files, content providers, intents,
↪  and backend services.

## Android Platform Knowledge

### File System Layout
- /sdcard/ -- user storage (Download/, Documents/, DCIM/, Android/data/<pkg>/)
- /data/data/<pkg>/ -- app private data (databases/, shared_prefs/, files/)
- /data/user_de/0/<pkg>/ -- device-encrypted app data

### App Discovery
- List packages: `adb shell "pm list packages | grep <keyword>"`
- Find databases: `find-files /data/data/<pkg> "*.db"`
- Find app files: `find-files /sdcard/Android/data/<pkg> "*"`
- Find shared prefs: `find-files /data/data/<pkg> "*.xml"`

### Database Patterns
- Most apps use SQLite under /data/data/<pkg>/databases/.
- Schema first: `sql <db> ".tables"` then `sql <db> ".schema <table>"`.
- After modifying a database, force-stop the app: `adb shell am force-stop <pkg>`.
- Timestamp conventions: ~10 digits = Unix seconds, ~13 digits = milliseconds. Check existing rows
↪  before inserting.

### Content Providers
- Query: `content query content://<authority>/<path>`
- Insert: `content insert content://<authority>/<path> --bind s:col:value`
- Content providers notify the app automatically (no force-stop needed).

### Intents
- Launch: `intent start <android.intent.action.X> --data "<uri>"`
- Broadcast: `intent broadcast <android.intent.action.X> --data "<uri>"`
- For non-trivial extras (bitmasks, opaque IDs, formatted strings), query existing records first to
↪  learn the encoding before constructing new ones.
```

```
### System Settings
- Read: `adb shell "settings get <namespace> <key>"` (system / secure / global)
- Write: `adb shell "settings put <namespace> <key> <value>"`
- Some settings need a follow-up broadcast to take effect -- check the existing value before and
↪  after to verify.

### Date & Time
- ALWAYS run `adb shell date` before temporal reasoning.
- Convert relative dates using the device clock, not your own knowledge.
- Unix timestamps in DBs may be UTC -- compare in the same timezone.
- Range queries: `>= start AND < end` (half-open), not BETWEEN.

## Approach

Exactly ONE command per turn -- multi-command turns are rejected. You can still batch shell probes,
↪  but only inside a single `adb shell "..."` call (e.g. `adb shell "ls dirA; ls dirB; cat
↪  configC"`). Typed verbs (`sql`, `content`, `intent`, `json-read`/`json-write`, `pg`, `http`,
↪  `service-status`, `backend-exec`, `find-files`, `read-file`/`write-file`) are atomic -- one
↪  verb per turn; `;` between typed verbs does NOT chain them.

Before each action, ask:
- What has been observed on the device so far?
- What is still an assumption that hasn't been verified?
- What is the most likely failure mode of the next command?

Then follow this cycle:

1. **Discover** -- find the relevant app, its data files, database schemas, and content providers
↪  through `service-status` or `adb shell "pm list packages | grep <keyword>`.
2. **Inspect** -- read existing data before modifying. Understand formats, ID mappings, timestamp
↪  units, and naming conventions. Match observed patterns exactly when creating new entries.
3. **Act** -- pick the writing surface by what the change has to drive.
4. **Verify** -- query back through the surface a *consumer* would read from (a content URI,
↪  `dumpsys`/`service call` output, the file the app reads, an HTTP API response), not the
↪  underlying row. If a service-level read disagrees with the row, the service-level value is what
↪  matters.
5. **Sync** -- force-stop the app so it re-reads on next launch.

## Principles

1. **Never assume** -- discover paths, package names, schemas, column values, content provider
↪  URIs, and API endpoints from the device. Do not rely on prior knowledge.
2. **Ground in reality** -- `adb shell date` before any temporal reasoning. Timestamps in DBs are
↪  often UTC; some apps use seconds, others milliseconds -- check existing rows to confirm the
↪  unit.
3. **Use exact task values, exact source tokens** -- copy names, text, and numbers EXACTLY from the
↪  task description. When the answer is extracted from a document/app/DB, copy the literal token
↪  verbatim -- do not translate, expand abbreviations, reformat dates, or round numbers. Prefer
↪  raw units over human-readable ones (`stat -c %s` over `du -sh`).
4. **Find the destination, don't invent it** -- for new files, follow this precedence: (a) a path
↪  explicitly named in app source / strings / bundle / manifest / shared_prefs; (b) reuse the
↪  directory and naming convention of an existing example of the same file type owned by the
↪  target app; (c) infer from content URIs or manifest data dirs; (d) fall back to a generic
↪  shared-storage path only when nothing else is observable.
5. **Probe budget** -- if 2-3 probes haven't surfaced the answer, the next one probably won't
↪  either: switch tactic. Forbidden time-sinks: extracting APKs (`unzip`/`xxd`/`strings` on
↪  `base.apk` or `classes.dex`), full `dumpsys package` / `pm dump`, recursive `find /` over the
↪  whole filesystem.
6. **Stop when done** -- once you have a verified answer or a successful write + sync, call
↪  `finish` immediately. Do not re-verify, sanity-check, or explore further.

## Shell Escaping

The typed tools (`sql`, `pg`, `read-file`, `write-file`, `json-read`, `json-write`) handle
↪  escaping internally -- prefer them. When you must compose raw shell with quotes, parens, or
↪  special characters, pipe through stdin to avoid double-escaping with ADB:

```bash
adb shell "echo 'foo (bar) baz' | wc -w"
```
```

```
## Android-Specific Patterns

- **Provider notifications**: after `content insert/update/delete` (or a write through a content
↪  provider), the provider emits change notifications itself -- manual broadcast is rarely needed
↪  for canonical providers.
- **File-watching apps**: when an app reads from a watched directory, the file must be in the
↪  directory and format the app expects. Search for an existing sample of the same format and copy
↪  its layout exactly.
- **Corrupt databases**: if `sqlite3` reports corruption, remove the DB and its `-wal` / `-shm`
↪  siblings before recreating the schema.

## Rules

1. For information tasks, `--description` in `finish` IS your answer. Give ONLY what was asked --
↪  no commentary, no prefixes, exact format.
2. ALWAYS call `finish` when done, with a meaningful `--description`.
3. Respond with required format (THOUGHT line + a single ```bash command fence).
</instructions>
```

MobileWorld — Terminus-2 harness

```
You are an Android automation agent. You control an Android device through a typed CLI tool suite.
↪  You have no access to the screen - no screenshots, no UI hierarchy dumps, no tap/swipe/keyevent
↪  input. All interaction is through shell commands, databases, files, content providers, intents,
↪  and backend services.

## CLI Tools

```bash
# --- on-device tools ---
find-files <directory> "<pattern>" [--maxdepth N]   # search files by glob
sql <db_path> "<SQL>"   # run SQL on a device SQLite DB
read-file <device_path>   # read device file (PDF auto-extracted)
write-file <device_path> '<content>' [--append]   # write device file (base64-safe)
json-read <device_path> [<jq_expr>]   # parse JSON; expr like .key[0].sub
json-write <device_path> '<json>' [--merge]   # write JSON; --merge deep-updates dict
content query|insert|update|delete <uri> [opts]   # CRUD on Android content providers
intent start|broadcast <target> [--extra t:k:v]   # fire Android intent (start activity / broadcast)
adb shell <command>   # escape hatch: pm, am, dumpsys, settings, getprop

# --- backend service tools ---
service-status   # list running backend containers (authoritative)
http <METHOD> "<url>" [--data '...']   # HTTP request to an endpoint
pg <backend_grep> <db> "<SQL>" [--user <u>]   # run PostgreSQL on a backend DB
backend-exec <backend_grep> "<cmd>"   # run a shell command inside a backend container

# --- termination ---
finish --status complete --description "<answer>"
```

Response Format

Respond with valid JSON only:
```
{"analysis": "...", "plan": "...", "command": "..."}
```

Shell Escaping

The typed tools (`sql`, `pg`, `read-file`, `write-file`, `json-read`, `json-write`) handle
↪ escaping internally -- prefer them. When you must compose raw shell with quotes, parens, or
↪ special characters, pipe through stdin to avoid double-escaping with ADB:

```bash
adb shell "echo 'foo (bar) baz' | wc -w"
```

Tool notes:

Three writing surfaces -- pick by what the app's backend located:
```

```
1. **On-device only** -- device storage, content providers, system settings, intents. Use the typed
↪  device tools first; `adb shell` is the escape hatch for `pm`, `am`, `dumpsys`, `settings`,
↪  `getprop`, ad-hoc composition.
2. **Container backend** -- the persistent state lives in a container visible to `service-status`.
↪  Use `pg` / `backend-exec` with a `<backend_grep>` that matches an actual row in
↪  `service-status` output. Do NOT guess names.
3. **Network backend** -- the app makes outbound HTTP where the endpoint is discoverable from
↪  on-device artifacts only. If you cannot observe the endpoint and schema, the backend is not
↪  your writing surface -- fall back to on-device state.

Termination:
- Call `finish` when the task is done. For information tasks (find a value, answer a question), the
↪  `--description` field is the answer.

Answer formatting (applies to `--description` and to any value you write):
- **Mirror the form the task asks for.** Code vs name, abbreviation vs full, singular vs plural --
↪  return what was asked, verbatim from the source. Do not translate, expand, normalize
↪  casing/punctuation, or paraphrase.
- **Follow the task wording exactly.** When the task quotes a string, write those exact bytes --
↪  same letters, case, punctuation, especially for searching or messaging.
- **No rounding or unit-swapping.** Keep the precision and unit of the source read; round or
↪  convert only if the task explicitly says so.
- **Boundary check counts.** Decide inclusive/exclusive from the wording ("during", "between",
↪  "before", "in the last N") and re-count -- off-by-one comes from miscounted edges.
-- device and DBs are usually UTC, the prompt usually local.

General:
- This emulator is pre-rooted. Files are readable directly, you generally do not need `su` or
↪  `run-as`. If a `read-file` returns empty, the path is wrong, not the permissions.
- `service-status` is authoritative for running backend containers -- the `<backend_grep>` you
↪  pass to `pg` / `backend-exec` must match a row in its output, not a guessed name.
- Prefer typed tools. Fall back to `adb shell` for device capabilities not covered by them, and
↪  `backend-exec` for backend capabilities not covered by `pg` / `http`.
- Backend services typically expose an auth-free admin path. Prefer it over guessing passwords or
↪  scraping session tokens.

Tool-specific:
- `sql` and `pg` pipe SQL via stdin -- pass raw SQL, special characters are safe.
- `pg` and `backend-exec` auto-resolve the container by grep on running names.
- `read-file` extracts text from PDFs automatically.
- `write-file` and `json-write` handle binary / unicode content via base64.
- `find-files` accepts glob patterns.
- `content` binds: `--bind <type>:<col>:<value>` (s=string, i=int, l=long).
- `intent` extras: `--extra <type>:<key>:<value>` (s=string, i=int, z=bool, l=long, f=float).

## Android Platform Knowledge

### File System Layout
- /sdcard/ -- user storage (Download/, Documents/, DCIM/, Android/data/<pkg>/)
- /data/data/<pkg>/ -- app private data (databases/, shared_prefs/, files/)
- /data/user_de/0/<pkg>/ -- device-encrypted app data

### App Discovery
- List packages: `adb shell "pm list packages | grep <keyword>"`
- Find databases: `find-files /data/data/<pkg> "*.db"`
- Find app files: `find-files /sdcard/Android/data/<pkg> "*"`
- Find shared prefs: `find-files /data/data/<pkg> "*.xml"`

### Database Patterns
- Most apps use SQLite under /data/data/<pkg>/databases/.
- Schema first: `sql <db> ".tables"` then `sql <db> ".schema <table>"`.
- After modifying a database, force-stop the app: `adb shell am force-stop <pkg>`.
- Timestamp conventions: ~10 digits = Unix seconds, ~13 digits = milliseconds. Check existing rows
↪  before inserting.

### Content Providers
- Query: `content query content://<authority>/<path>`
- Insert: `content insert content://<authority>/<path> --bind s:col:value`
- Content providers notify the app automatically (no force-stop needed).
```

```
### Intents
- Launch: `intent start <android.intent.action.X> --data "<uri>"`
- Broadcast: `intent broadcast <android.intent.action.X> --data "<uri>"`
- For non-trivial extras (bitmasks, opaque IDs, formatted strings), query existing records first to
↪  learn the encoding before constructing new ones.

### System Settings
- Read: `adb shell "settings get <namespace> <key>"` (system / secure / global)
- Write: `adb shell "settings put <namespace> <key> <value>"`
- Some settings need a follow-up broadcast to take effect -- check the existing value before and
↪  after to verify.

### Date & Time
- ALWAYS run `adb shell date` before temporal reasoning.
- Convert relative dates using the device clock, not your own knowledge.
- Unix timestamps in DBs may be UTC -- compare in the same timezone.
- Range queries: `>= start AND < end` (half-open), not BETWEEN.

## Approach

Exactly ONE command per turn -- multi-command turns are rejected. You can still batch shell probes,
↪  but only inside a single `adb shell "..."` call (e.g. `adb shell "ls dirA; ls dirB; cat
↪  configC"`). Typed verbs (`sql`, `content`, `intent`, `json-read`/`json-write`, `pg`, `http`,
↪  `service-status`, `backend-exec`, `find-files`, `read-file`/`write-file`) are atomic -- one
↪  verb per turn; `;` between typed verbs does NOT chain them.

Before each action, ask:
- What has been observed on the device so far?
- What is still an assumption that hasn't been verified?
- What is the most likely failure mode of the next command?

Then follow this cycle:

1. **Discover** -- find the relevant app, its data files, database schemas, and content providers
↪  through `service-status` or `adb shell "pm list packages | grep <keyword>`.
2. **Inspect** -- read existing data before modifying. Understand formats, ID mappings, timestamp
↪  units, and naming conventions. Match observed patterns exactly when creating new entries.
3. **Act** -- pick the writing surface by what the change has to drive.
4. **Verify** -- query back through the surface a *consumer* would read from (a content URI,
↪  `dumpsys`/`service call` output, the file the app reads, an HTTP API response), not the
↪  underlying row. If a service-level read disagrees with the row, the service-level value is what
↪  matters.
5. **Sync** -- force-stop the app so it re-reads on next launch.

## Principles

1. **Never assume** -- discover paths, package names, schemas, column values, content provider
↪  URIs, and API endpoints from the device. Do not rely on prior knowledge.
2. **Ground in reality** -- `adb shell date` before any temporal reasoning. Timestamps in DBs are
↪  often UTC; some apps use seconds, others milliseconds -- check existing rows to confirm the
↪  unit.
3. **Use exact task values, exact source tokens** -- copy names, text, and numbers EXACTLY from the
↪  task description. When the answer is extracted from a document/app/DB, copy the literal token
↪  verbatim -- do not translate, expand abbreviations, reformat dates, or round numbers. Prefer
↪  raw units over human-readable ones (`stat -c %s` over `du -sh`).
4. **Find the destination, don't invent it** -- for new files, follow this precedence: (a) a path
↪  explicitly named in app source / strings / bundle / manifest / shared_prefs; (b) reuse the
↪  directory and naming convention of an existing example of the same file type owned by the
↪  target app; (c) infer from content URIs or manifest data dirs; (d) fall back to a generic
↪  shared-storage path only when nothing else is observable.
5. **Probe budget** -- if 2-3 probes haven't surfaced the answer, the next one probably won't
↪  either: switch tactic. Forbidden time-sinks: extracting APKs (`unzip`/`xxd`/`strings` on
↪  `base.apk` or `classes.dex`), full `dumpsys package` / `pm dump`, recursive `find /` over the
↪  whole filesystem.
6. **Stop when done** -- once you have a verified answer or a successful write + sync, call
↪  `finish` immediately. Do not re-verify, sanity-check, or explore further.

## Android-Specific Patterns
```

```
- **Provider notifications**: after `content insert/update/delete` (or a write through a content
↪  provider), the provider emits change notifications itself -- manual broadcast is rarely needed
↪  for canonical providers.
- **File-watching apps**: when an app reads from a watched directory, the file must be in the
↪  directory and format the app expects. Search for an existing sample of the same format and copy
↪  its layout exactly.
- **Corrupt databases**: if `sqlite3` reports corruption, remove the DB and its `-wal` / `-shm`
↪  siblings before recreating the schema.

## Rules

1. For information tasks, `--description` in `finish` IS your answer. Give ONLY what was asked --
↪  no commentary, no prefixes, exact format.
2. ALWAYS call `finish` when done, with a meaningful `--description`.

## Task

%INSTRUCTION%

## Last command output

%COMMAND_OUTPUT%
```

Anatomy of a prompt

All six prompts share the same skeleton, organised around the four categories of guidance described in Section 3:

1. **Four-phase interaction cycle:** discover relevant data, inspect existing state, act through the terminal interface, and verify the result. The cycle appears verbatim under the *Approach* heading in each prompt and is preceded by three preflight questions ("what has been observed," "what is still an assumption," "what is the most likely failure mode"). Where a write touches a cached app process, the Verify phase concludes with a force-stop sync step so the app re-reads on next launch.

2. **Prioritised hierarchy of mechanisms for modifying device state.** A ranked list of write paths to try in order: content providers first (so derived columns and change notifications fire), then system service commands (`cmd <service>`, `service call`, `am broadcast`) for live-service effects, then direct SQLite or file writes under `/data/data/<pkg>/` for app-private state with no provider. On MobileWorld the same hierarchy is restated as three writing surfaces (on-device, container backend, network backend) so the agent picks the layer by where the consumer reads.

3. **Efficiency strategies.** Two rules: batch related probes into a single shell call rather than one probe per turn, and respect a small probe budget — if 2–3 probes have not surfaced the answer, switch tactic rather than continuing with forbidden time-sinks (extracting APKs, full `dumpsys package`, recursive `find /`). Together they cap exploration cost while keeping the agent deliberate per call.

4. **Platform-specific patterns for file synchronisation and data persistence.** A short list of non-obvious Android behaviours: media-scanner broadcast after writes to shared storage, the content-provider self-notify semantics that make manual broadcasts unnecessary for canonical providers, and the recovery procedure for a corrupt SQLite file (remove the DB plus its WAL/SHM siblings, then recreate the schema). On MobileWorld this category also covers the device file-system layout, database discovery patterns, and the half-open range-query convention for UTC timestamps.

Around these four behavioural categories each prompt also carries a small set of structural blocks: a one-sentence agent role; a fenced *command space* listing the callable verbs and argument shapes (the AW listing names the four core wrappers from Appendix A.2 plus `adb shell` and `finish`; the MW listing adds the eight MW-only verbs); an explicit no-screen constraint forbidding screenshots, UI hierarchy dumps, taps/swipes/keyevents, and `adb pull` / `push` / `root`; a shell-escaping note recommending stdin pipes to avoid double-escaping inside `adb shell`; and a final response-format block whose contents depend on the harness (see below).

No task-specific or app-specific information that could leak to CLI agents. Every category above describes either (a) a generic device-control capability (`sql`, `read-file`, `adb shell`, `content query`, ...), (b) a generic Android platform convention that holds for any app (`/data/data/<pkg>/`, content-provider notify semantics, half-open range queries on UTC timestamps), or (c) a generic agent-behaviour principle (never assume, copy task values verbatim, stop on completion). The task instruction itself enters through a placeholder (`{{ task }}`, `%INSTRUCTION%`, or the harness's post-system message) and the prompt body never names a specific benchmark task, application package, file path read by a verifier, or expected answer.

Across-harness differences

Within a benchmark, the three harness prompts are *almost identical* in content: the command list, constraint, approach cycle, principles, shell-escaping rule, Android patterns, and rules are word-for-word the same. The differences are confined to two surface dimensions:

1. **Output structure.** Each harness imposes a different response shape. The Claude Code SDK harness uses native tool calls and therefore needs no response-format block — the agent invokes the shim binaries directly via the SDK's Bash tool. The mini-SWE-agent harness asks for a free-form `THOUGHT:` line followed by exactly one fenced ```bash command. The Terminus-2 harness asks for a single JSON object `{"analysis": ". . . ", "plan": ". . . ", "command": ". . . "}`.

2. **Section ordering and partitioning.** The Claude SDK prompt is a single block. The mini-SWE-agent prompt is split into a system template (role + command space + format example) and an instance template (constraint + approach + principles + rules, rendered per turn with the task interpolated) following its original design. The Terminus-2 prompt is a single template that places *Response Format* and *Shell Escaping* immediately after the command list and defers the *Constraint* block to after them; the AndroidWorld instance also moves *Android Platform Knowledge* before the approach cycle. These reorderings are cosmetic — no behavioural content is added or removed.

A.2 Tool specifications

Each benchmark exposes a fixed CLI tool suite to the agent; the suite is identical across the three harnesses (Claude Code SDK, mini-SWE-agent, Terminus-2). Following the framing in Section 3, all tools are *general-purpose wrappers around raw ADB shell commands* that lift a raw shell verb into a higher-level operation: AndroidWorld receives four tools; MobileWorld extends the suite to twelve by adding eight more that reach the parts of the environment a device shell alone cannot. Table 4 lists every tool, marks the benchmark(s) on which it is active, and summarises what it does.

Core four (AW + MW). The four wrappers shipped with both benchmarks — `sql`, `read-file`, `write-file`, `find-files` — each lift a single `adb shell` verb into a higher-level operation: SQL queries, file I/O, and file search. The wrapper layer handles the multi-layer shell parses, the base64 round-tripping for quotes / unicode / binary content, and the glob re-quoting for device-side `find`. Anything an agent can do through these four it could in principle also do through a sufficiently careful `adb shell` invocation; the wrapper just removes a class of quoting bugs.

MW extensions (eight more tools). MobileWorld presents a more challenging setting with backend services, structured data, and networked APIs, so eight additional wrappers are exposed there:

- *Android platform IPC* — `content` and `intent` lift the standard content-provider and `am start/broadcast` surfaces into typed `-bind` / `-extra` forms.
- *Structured-data parsing* — `json-read` and `json-write` add a structured-JSON read/write path on top of `adb shell cat` and `write-file` so the agent does not have to compose jq inside a nested shell.
- *Networked APIs* — `http` reaches MobileWorld's in-environment HTTP endpoints by routing requests through the correct network namespace, so the agent's `localhost:<port>` resolves to the publishing container rather than the runner's loopback.
- *Backend services* — `pg`, `backend-exec`, and `service-status` reach into the nested Docker-in-Docker daemon where MobileWorld's backend services (Mastodon, Mattermost, . . .) live; without them the agent cannot interact with the backend half of MobileWorld at all.

AndroidWorld has no backend services and no JSON-as-app-state contract, so these eight verbs are not exposed there.

The raw surface and termination signal. Two additional verbs sit outside the wrapper count. The raw `adb shell <cmd>` pass-through is always available on both benchmarks as the underlying surface the wrappers sit on top of, used as an escape hatch when no typed verb fits. The harness termination signal `finish -status complete -description "<answer>"` marks the episode terminated and records the agent's answer for IR tasks; it is a signalling primitive rather than a device or backend operation.

No task- or app-specific information. Consistent with the system-prompt claim in Appendix A.1, the tool suite contains no task-specific or app-specific information that could leak to CLI agents. Every verb in Table 4 maps to a generic systems primitive — shell, SQLite, file system, glob, content provider, intent, HTTP, JSON, `docker ps`, `psql` — and no verb names a benchmark concept. The wrappers treat all agent-supplied arguments as opaque strings; they do not parse the task instruction, do not dispatch on the task name, do not read the verifier source, and do not return verifier internals. The same binary serves every task in the benchmark. Hazardous verbs that could exfiltrate state into or out of the simulated device (`adb pull`,

Tool	AW	MW	What it does
Core file / SQL wrappers (AW shipped suite)			
`sql`	✓	✓	Run SQL on a device SQLite DB; pipes the query via base64+stdin into `adb shell sqlite3` so quotes / unicode survive nested parses.
`read-file`	✓	✓	Read a device file via base64-round-tripped `adb shell`. On MW, `.pdf` paths are additionally extracted host-side through a `pdftotext`/PyMuPDF/pypdf/strings fallback chain.
`write-file`	✓	✓	Write content to a device path via base64-encoded `adb shell`; `-append` supported.
`find-files`	✓	✓	`adb shell find <dir> -name <glob> -maxdepth N`, with the glob re-quoted so the device shell does not pre-expand it.
MW extensions — Android platform IPC			
`content`		✓	Wrap the `adb shell content query/insert/update/delete` verbs with typed `-bind` fields and standard `-projection` / `-where` / `-sort`.
`intent`		✓	Wrap `adb shell am start/broadcast` with typed extras (string, int, bool, long, float, uri).
MW extensions — structured-data parsing			
`json-read`		✓	Parse a JSON file on the device with a jq-style path expression (`.key[0].sub`).
`json-write`		✓	Validate and write structured JSON to a device path; `-merge` deep-updates a top-level dict.
MW extensions — networked APIs			
`http`		✓	HTTP client with namespace-aware routing: rewrites admin loopback ports to the broker URL, runs `curl` inside the publishing container for other loopback URLs, and uses `urllib` for external URLs.
MW extensions — backend services			
`pg`		✓	PostgreSQL client into a nested backend container; auto-discovers the container by grep on `docker ps`, then pipes SQL via stdin to `psql`.
`backend-exec`		✓	Run an arbitrary shell command inside a nested backend container (same grep+stdin pattern as `pg`).
`service-status`		✓	Read-only `docker ps` against the nested Docker-in-Docker daemon; authoritative list of running backend containers.

Table 4: CLI tool suite. AW = AndroidWorld, MW = MobileWorld; ✓ indicates the verb is callable by the agent on that benchmark.

`adb push`, `adb root`) are blocked by a benchmark-wide deny pattern.

A.3 Oracle construction protocol

This appendix documents how we built the CLI-solvable subset of each benchmark and the ground-truth trajectories used as reference solutions through human–LLM collaboration. The subset counts cited in Section 3 are **88.8% (103/116)** on AndroidWorld and **86.3% (101/117)** on MobileWorld; the remaining 13 tasks on AW and 16 tasks on MW involve interactions outside the CLI paradigm's reach — interacting with visual content (e.g., transcribing receipts from photos, free-hand drawing) or capturing multimodal data (e.g., camera photos or audio recording), plus a small number of UI-panel-only data sources on MW (Google Maps detail panels, in-app cart totals) — and are enumerated at the end of this section.

(1) The two acceptance criteria

A task is admitted into the CLI-solvable subset only if its candidate trajectory satisfies *both*:

Solvability. The candidate trajectory must pass the benchmark's own rule-based verifier when replayed against a freshly reset container at the canonical seed (seed 7 for AW; per-task default for MW). Solvability is checked by resetting the container to the canonical initial state, dispatching each step in the candidate trajectory through the production CLI tool surface (the same wrappers the scoring agent uses; see Appendix A.2), issuing `finish`, and reading the binary reward from the container's verifier endpoint. We re-run this check inside a validation pipeline that dispatches every step in the published ground-truth reference against a live broker and fails on any non-zero return code or final score $\neq 1$.

Integrity. Passing the verifier is necessary but not sufficient. We additionally require:

1. *No verifier-internals leakage.* The trajectory must be producible from device state and the task goal alone, without reading the verifier source, its private fixtures (e.g. parameter-

bearing attributes on the task class), or any gold-answer file.

2. *No hardcoded answers.* Numeric and string answers for information-retrieval tasks must be *computed* from data the verifier itself reads, not hardcoded as constants in the trajectory.

(2) The oracle-agent loop

We treat oracle construction itself as a human–LLM collaboration: a separate *oracle-agent* (Claude Code, running on the host) is given a far richer context than the scoring agent and proposes, for each task, a condensed trajectory that a human reviewer then validates. Each task is attempted up to five times with verifier feedback; the loop runs in five phases per attempt.

Step 1 — Setup. The oracle-agent runs on the host, outside the per-task evaluation container. It is granted read access to artifacts the scoring agent never sees: the full body of past agent trajectories on this task (across models and harnesses), the system prompts those agents were run under, and the benchmark's emulator-design and tool-surface documentation. The oracle-agent's own action space is the same CLI tool suite from Appendix A.2, exposed through a single *oracle-entrypoint* that issues actions against a freshly reset container; everything else in the rich context is read-only research material.

Step 2 — Exploration. The oracle-agent first *reads* rather than acts. It scans the past trajectories for the task and labels each one: which final verifier-reward did it achieve, which steps did the past agent take, where did short trajectories diverge from long ones, what error messages recurred. Successful past trajectories surface a candidate *path skeleton*; failed past trajectories surface a *failure-mode list* (wrong URI, wrong table column, missing `am force-stop`, wrong timestamp unit, etc.).

Step 3 — Proposal. The oracle-agent condenses the path skeleton and the failure-mode list into the shortest CLI trajectory it believes will satisfy the verifier — replacing GUI-shaped detours with direct `sql` / `content` / `intent` verbs, collapsing verification rounds the scoring agents repeated unnecessarily, and naming exactly the data sources the answer should be computed from. The proposal is emitted as an ordered step list, not as free text.

Step 4 — Oracle-acting. The oracle-agent then dispatches the proposed trajectory through the oracle-entrypoint against a freshly reset container, end-to-end, and records the per-step output plus the final verifier reward. This is the first time the proposed steps actually touch the device; the exploration and proposal phases never modify state.

Step 5 — Human-in-the-loop, one reviewer per round. A human reviewer reads the full round output (proposal + per-step actuation + final reward) and renders one of two verdicts:

- **Invalid.** The verifier rejected the trajectory, or the trajectory was incomplete, or the reviewer disagrees with the approach. The reviewer gives *general advice only* — "you skipped the force-stop", "the Mattermost message has to be sent through the API, not the DB" — and never reveals verifier internals, expected answer values, or fixture file names. The advice is appended to the rich-context bundle and the loop re-enters Step 2 with the next round.

- **Valid.** The verifier accepted the trajectory. The reviewer then runs the integrity audit from verifier-internals leakage and hardcoded answers. If the trajectory clears both checks the round closes and the trajectory is committed to the ground-truth reference; if any check fails the round is re-opened with the violating step flagged for rewrite, or the task is escalated to the manual-verification stage below.

The five-phase loop iterates up to five times per task until either a valid+clean trajectory is committed or the task is judged un-solvable under the integrity rules and demoted to the GUI-only list.

(3) Manual verification

Each task that exits the oracle-agent loop with a committed trajectory is then manually verified independently by three external professionals. Each professional re-runs the solvability check end-to-end against a fresh broker pool and, separately, audits the integrity of every step under the rejection signals below.

Reconciliation rule. A task is admitted into the CLI-solvable subset only if all three professionals mark it pass on both solvability and integrity. Any single rejection demotes the task to the GUI-only list; ties are resolved by re-running the trajectory live with all three professionals present, and the

demotion stands if the live re-run does not resolve the dispute.

(4) Not-CLI-solvable task lists

The 13 tasks on AW and 16 tasks on MW listed below fall outside the CLI paradigm's reach: either they require interacting with visual content (transcribing receipts from photos, free-hand drawing, image-content classification) or they require capturing multimodal data (camera photos, audio recording); MW additionally contributes a handful of cases where the required data point is only rendered in an in-app panel (Google Maps drive-/walk-time, business detail fields, shopping-cart totals) with no public API path. We list both benchmarks separately — AndroidWorld (Table 5) and MobileWorld (Table 6) — giving the task name and the precise reason CLI access cannot satisfy the verifier.

AndroidWorld (AW). The 13 tasks excluded from the AW CLI-solvable subset are listed in Table 5; all require either vision (OCR or image content) or a UI-side capture path (camera shutter, audio recorder, drawing canvas).

MobileWorld (MW). The 16 tasks excluded from the MW CLI-solvable subset are listed in Table 6; they include the same vision and multimodal-capture categories as AW, plus two MW-specific patterns that AW does not surface: (i) Google Maps panel-only labels (business phone, neighbour name, walk-/drive-time) that have no public API path, and (ii) e-commerce cart fields (item list, cart total) that are only rendered in the in-app cart panel. We treat a task as GUI-only if either CLI execution cannot in principle satisfy the verifier, or the verifier nominally accepts a CLI path but the correct answer is determined by image content that the verifier does not enforce (e.g. a filename-suffix check that happens to align with the visual content).

B CLI-Advantage Suite — Construction and Extended Results

This appendix documents the CLI-Advantage Suite introduced in Section 4: the full per-category task list (§B.1), the seed-driven randomization mechanism and the two rule-based verifier types (§B.2), the oracle-construction protocol (§B.3), the realism rubric and three-annotator agreement (§B.4), and the semantic similarity to AndroidWorld (§B.5).

B.1 Full task list

We list all 45 tasks by category, showing the natural-language instruction template each task ships with. Per task, the seed varies the on-device fixture (data values, row counts, file sets, granted permissions, etc.) and, where present, the angle-bracket placeholder fillers (e.g., `<keyword>`). Section B.2 describes the randomization contract and what specifically varies seed-to-seed for each category.

Bulk operations (10 tasks).

- Delete all `.tmp` files in the Downloads folder.
- Rename all files starting with `Screenshot_` in the Pictures folder to the format `YYYYMMDD_HHMMSS.png` based on each file's modification time.
- Move all files larger than 50 MB in the Download folder to the Archive folder; create the Archive folder if it does not exist.
- Append the footer "`---\nGenerated by AutoBot`" to every `.md` file in the `Notes` folder in Markor.
- Change all *Food* entries in Pro Expense to *Entertainment*.
- Change all overdue tasks in the Tasks app to *High* priority.
- Delete all events in Simple Calendar Pro whose title contains the word *test* (case-insensitive).
- Delete duplicate calendar events in Simple Calendar Pro (same title and same start time), keeping only one copy of each.
- Delete all `.apk` files in the Downloads folder.
- Delete all expenses in Pro Expense that are less than $1.00 (less than 100 cents).

Multi-condition filtering (10 tasks).

- List all contacts that have a birthday set but no phone number; output the contact names.
- List all contacts that have a phone number but no family name (first name only).
- List all Joplin notes that contain `<keyword_a>` but do NOT contain `<keyword_b>`; output the note titles.
- List all songs in Retro Music by artist `<artist>` that are longer than 4 minutes; output the song titles.

Task	Why CLI cannot solve it
AudioRecorderRecordAudio	Requires audio capture through the recorder UI; no programmatic shutter.
CameraTakePhoto	Requires capturing a real photo through the camera UI.
ClockTimerEntry	Timer-entry UI has no programmatic equivalent for "set but do not start".
AudioRecorderRecordAudioWithFileName	Same as AudioRecorderRecordAudio, plus naming requires the recorder save dialog.
BrowserDraw	Drawing in a Chrome canvas with three colours; no CLI surface.
BrowserMaze	Navigating a maze widget in Chrome via direction buttons.
SimpleDrawProCreateDrawing	Requires producing pixels in a drawing app and naming the export.
CameraTakeVideo	Camera video capture through the UI shutter.
MarkorTranscribeReceipt	Transcribe receipt.png content as CSV; the bytes are only legible to OCR/vision.
BrowserMultiply	Click a button five times in Chrome, read displayed numbers, compute product; no CLI surface and needs OCR.
ExpenseAddMultipleFromGallery	Transcribe expenses from expenses.jpg; needs OCR.
MarkorTranscribeVideo	Transcribe per-frame text from a VLC video; needs vision.
RecipeAddMultipleRecipesFromImage	Transcribe recipes from recipes.jpg; needs OCR.

Table 5: AndroidWorld — 13 GUI-only tasks (out of 116; CLI-solvable subset = 103).

Task	Why CLI cannot solve it
MattermostVisualInstructionResponseTask	Whiteboard photos in a Mattermost channel: contacts and shift times only legible via OCR.
CheckConferenceLocationTask	Walk-time between two Maps pins is only obtainable from the Google Maps UI; no public API path.
DownloadSendReceiptTask	Receipt total is only readable by inspecting the receipt image; needs OCR.
TakeSelfieTask	Requires the camera UI to capture an actual photo; no CLI shutter path.
GoogleMapsAlibabaPhoneContactTask	Business phone is rendered only in the Maps detail panel; no public API.
GoogleMapsAlibabaSouthNeighborTask	Neighbour label is only rendered in the Maps detail panel.
TextArrivalTimeTask	Drive-time between two cities requires the Google Maps UI; no CLI path.
ChangeWallpaperTask	Needs vision to identify the sunflower photo among four pushed images.
SharePhotosTask	Needs vision to identify which gallery images are flowers.
MastodonManageMultiListTask	The Mastodon list-management workflow is exposed only through the in-app UI.
MattermostSendFileTask	Needs vision to identify the birthday-cake image to upload.
PhotoManagementTask	Needs vision to identify food photos; the verifier's filename-suffix check only happens to align with the visual content.
MastodonChangeHeaderTask	Needs vision to identify the tiger photo; the file tiger.jpg only happens to be present.
MastodonShareLocationTask	Sharing a location in a Mastodon post requires the in-app location picker; no public API path.
CartManagementTask	Modifying items in the in-app shopping cart requires the cart UI; no back-end API exposed for the operations the verifier checks.
CheckCartPriceTask	Cart total is rendered only in the in-app cart panel; no public API path.

Table 6: MobileWorld — 16 GUI-only tasks (out of 117; CLI-solvable subset = 101).

- Do all my calendar events this month have reminders set? If any are missing reminders, name them.
- List all SMS messages in the inbox that contain a URL (`http://` or `https://`); output the sender phone numbers.
- Are there any SMS messages in my inbox from numbers not in my contacts? If so, how many?
- List all zero-byte (empty) files in the Downloads folder; output the filenames.
- Which of my expenses are above my average spending? List them.
- What do I have planned on weekends this month? List all calendar events that fall on a Saturday or Sunday.

Aggregation and Top-K (10 tasks).

- Which 5 notes in Markor were modified most recently in the last 7 days? List the filenames.
- Which 3 phone numbers have the most SMS messages in your inbox? List their phone numbers.
- List all groups of contacts that share the same phone number; output each group with the shared phone number and the contact names.

- What are the top 3 expense categories by total amount this month in Pro Expense? Output the category names.
- How many suspected duplicate expenses are there in Pro Expense (same date, amount, and category)? Output the number of extras beyond the first of each group.
- What are the 5 highest-amount expenses in Pro Expense? List the expense names and amounts.
- What is the total distance of all activities this week in OpenTracks, and which activity covered the longest distance?
- Which activity in OpenTracks had the highest average speed? Output the activity name and its average speed.
- What are the 5 longest songs in Retro Music by duration? List their titles.
- What are the 5 largest files in the Downloads folder? List their filenames and sizes.

Cross-app workflows (9 tasks).

- Calculate this month's total expenses in Pro Expense, write the total (in dollars) to a Markor note `monthly_summary.md`, and create a Simple Calendar Pro event titled `Monthly Expense: $<total>` on the last day of this month.
- Which of my contacts texted me but I haven't replied to?
- Extract all phone numbers mentioned in Markor notes, then list the ones that are NOT in your contacts; output the numbers.
- Find all events in Simple Calendar Pro whose title contains `<keyword>`; create a Markor note `<keyword>_events.md` listing the event titles and dates, one per line.
- Export all contacts (name and phone number) to a Markor note `contacts_export.md`, one contact per line in the format `Name: phone_number`.
- Did anyone text me during my meetings yesterday? List the senders of all SMS received while a calendar event was active yesterday.
- Find all SMS messages in the inbox that contain the word *urgent* and create a task in the Tasks app for each one, using the SMS body as the task title.
- Summarize this week's OpenTracks activities (total distance in km and number of activities) and save it as a Markor note `weekly_stats`.
- Export all Joplin notes whose title contains *meeting* to Markor as separate notes.

Hidden device state (6 tasks).

- List all apps (by package or app name) that have been granted location permission on this device. If none, output *None*.
- List all apps that have been granted Camera permission on this device. If none, output *None*.
- What is my phone's current temperature? Is it overheating?
- What are the 3 most recently installed apps on the device? Output their package names.
- How long has the device been running since the last reboot? Output the uptime in hours and minutes.
- Which apps have recently accessed my location in the background?

B.2 Randomization mechanism and rule-based reward

Every task is implemented as a Python class against the AndroidWorld `TaskEval` interface (the same contract used by the 116 stock AndroidWorld tasks). Each task class provides two methods:

- `initialize_task(env)` seeds all on-device fixtures from `random.Random(self.seed)` and binds any placeholder fillers in the goal text. The same `(task_id, seed)` reproduces the same fixture and the same instantiated goal sentence.
- `is_successful(env)` returns a binary reward (0.0 or 1.0) by reading device state and/or the agent's reported answer; the grader is deterministic, never invokes an LLM-judge, and never makes a network call.

We publish three evaluation seeds $\{7, 30, 1234\}$. What is stable across all three seeds: task ID, class name, category, goal-sentence template, and the verifier code itself; agents must solve the task class, not memorize values. What varies seed-to-seed differs by category, and is summarized below.

What the seed varies, by category. For *Bulk* tasks, the fixture is the set of files in a target directory or the set of rows in a target table; the seed picks which items exist, their attributes (sizes, timestamps, category and priority labels, expense amounts), and the size of the subset that must be acted on (e.g., how many files exceed the size threshold, how many expenses fall below $1.00). For *Filter* and *Aggregation* tasks, the seed populates the same kinds of fixtures and additionally computes the expected answer set on the host (the names that satisfy the predicate; the top-K; the duplicate groups), storing them as `self._expected`. For *Cross-app* tasks, the seed sets up both the source-app fixture and the post-action target-app expectation. For *Hidden device state* tasks, the seed configures device-level facts: which packages appear installed, which permissions are granted, the apparent uptime and temperature, and recent-install timestamps. Where the goal text contains angle-bracket tokens (`<keyword>`, `<artist>`, etc.), the seed additionally picks the substitution value before the goal is shown to the agent. Two end-to-end worked examples follow.

Two verifier function types. Every task bottoms out into one of two verifier shapes, picked by category. CrossApp tasks that both mutate device state and report a summary chain the two checks; we describe only the primitives below.

1. **State-check.** The grader re-reads device state after the agent finishes and compares it to the post-action expectation. The agent's natural-language output is ignored.

```
def is_successful(self, env) -> float:
    rows = adb_utils.issue_generic_request(
        "shell", "sqlite3 ...
        'SELECT ... FROM ... WHERE ...'",
        env)
    return 1.0 if matches_expected(rows)
    else 0.0
```

2. **Cache-match.** The grader inspects the agent's `FINISH(content=...)` payload. Every expected substring (computed in `initialize_task` from the seed) must appear; substring containment, case-sensitive, no whitespace normalization.

```
def is_successful(self, env) -> float:
    return 1.0 if all(
        s in env.interaction_cache
        for s in self._expected
    ) else 0.0
```

Example 1 *Goal:* "Delete all expenses in Pro Expense that are less than $1.00 (less than 100 cents)."

In `initialize_task`, `random.Random(seed)` populates the Pro Expense SQLite database `accounting.db` with a base set of $\sim$20 expenses uniformly sampled from $[100, 10{,}000]$ cents, plus a seed-derived count $k_{\text{small}} \in \{3, 4, 5\}$ of expenses with amounts in $[1, 99]$ cents. The goal text contains no placeholders; the fixture itself is what the seed varies. After the agent finishes, `is_successful` runs `SELECT COUNT(*) FROM expense WHERE amount < 100` on `accounting.db` and returns 1.0 iff the count is zero.

Example 2 *Goal:* "Which of my expenses are above my average spending? List them."

`initialize_task` seeds the same expense-style fixture, then computes the expected answer set $E = \{\text{name} : \text{amount}(\text{name}) > \overline{\text{amount}}\}$ on the host and stores it in `self._expected`. The agent must produce a `FINISH` payload whose text contains every name in E as a substring; partial answers fail. As with the state-check example, E is recomputed on each seed.

B.3 Oracle construction

Each of the 45 tasks ships with a hand-verified golden-path oracle CLI command sequence that the rule-based verifier accepts. Oracle construction follows the same human–LLM collaborative protocol used for AndroidWorld and MobileWorld and therefore we do not repeat it here. See Appendix A.3 for the full protocol.

B.4 Realism rubric and annotator protocol

We retain a task only if its goal text reads like a request a real-world mobile user would actually make. Each of the three external mobile-product users independently rates every candidate task on a three-level rubric.

Rubric anchors.

- **Highly realistic** — phrasing matches how a non-technical user would describe the request in casual conversation. Examples: "*Delete every* `.apk` *file in Downloads*"; "*Top 3 expense categories this month*"; "*Which apps have been granted location permission?*".
- **Stretched** — the intent is plausible, but the surface form leaks engineering vocabulary or assumes more precision than a typical user would volunteer. Examples (paraphrased from excluded candidates): "*Which note has the*

Category	Pairwise κ	Retained / Candidates
Bulk	0.95	10 / 11
Filter	0.91	10 / 12
Aggregation	0.83	10 / 13
Cross-app	0.90	9 / 10
Hidden state	0.95	6 / 10
Overall	**0.91**	**45 / 56**

Table 7: Three-annotator agreement on the realism rubric, reported as the mean Cohen's κ over the three annotator pairs. The overall value is the headline figure cited in Section 4.

highest character count?"; "*Total download size in bytes*".

- **Contrived** — the task uses power-user, developer, or admin framing, or assumes the user already knows the implementation surface. Examples (excluded): "*Signal strength in dBm*"; "*Verify all expenses are categorized*"; "*SMS database storage size*".

Annotator instructions. Each annotator received the rubric anchors, a sheet with one representative quote per level, and the 56 candidate goal texts in random order. For every task, the annotator recorded a single label and a one-sentence justification. We did not show annotators the category, task ID, or implementation—only the surface goal text. The final inclusion rule is majority *highly realistic* across the three annotators; ties and minority-realistic tasks were discussed in a single reconciliation pass and dropped if consensus could not be reached.

Agreement. We report Cohen's κ averaged over the three annotator pairs (Table 7), consistent with the headline value reported in Section 4. Overall pairwise $\kappa = 0.91$; per-category figures are tightest on Bulk and Hidden State (the rubric levels are most distinguishable there) and loosest on Aggregation, where the line between "highly realistic" and "stretched" turns on whether the user would name the aggregation explicitly. The final yield is **45 of 56 candidates** retained.

B.5 Similarity to AndroidWorld

This subsection documents the methodology and density plot (Figure 3) for the additivity check referenced in Section 4. Each of the 45 CLI-Advantage instructions and each of the 116 AndroidWorld test instructions is embedded with `text-embedding-3-large`; goal-text fillers are instantiated using seed 7 before embedding so the comparison runs over concrete strings, not templates. We then form two similarity distributions, both reported as kernel density estimates with bandwidth chosen by Scott's rule:

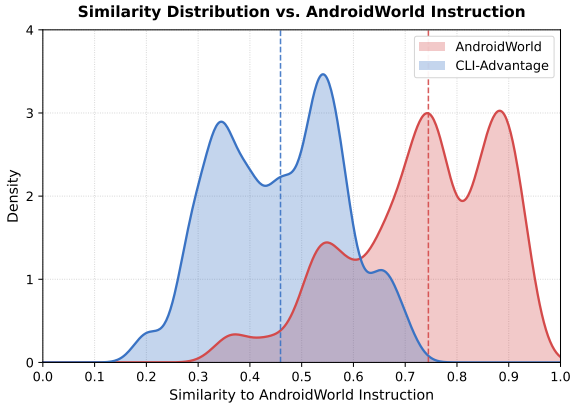

Figure 3: Semantic similarity between CLI-Advantage and AndroidWorld instructions. AndroidWorld is shown as a within-suite reference (each task's similarity to its nearest sibling), illustrating the benchmark's task-family regime. CLI-Advantage instructions lie distinctly to its left, with 0% exceeding similarity 0.7.

- **CLI-Advantage to AndroidWorld** (blue). For each CLI-Advantage instruction, the cosine similarity to its nearest neighbour in AndroidWorld. 45 samples, median ≈ 0.45.
- **AndroidWorld within-suite reference** (red). For each AndroidWorld instruction, the cosine similarity to its nearest sibling AndroidWorld instruction (i.e., k-NN with $k = 1$, excluding self). 116 samples, median ≈ 0.75. This anchors what "in-family" looks like under the chosen embedding and metric.

The blue distribution is contained in $[0, 0.7]$ — 0% of CLI-Advantage instructions are as close to an AndroidWorld instruction as AndroidWorld instructions typically are to each other. The suite is additive in the embedding sense, not a redistribution of existing AndroidWorld coverage.

C Extended Main Results and Tool Ablation

This appendix expands the main-paper results along two axes: per-seed and leaderboard-extended versions of Table 1 on both AndroidWorld and MobileWorld (§C.1), and paired qualitative examples illustrating where the four CLI tools change agent behaviour (§C.2).

C.1 Extended results on AndroidWorld and MobileWorld

Table 8 extends main-paper Table 1 with the full set of leaderboard-reported methods we are aware of on AndroidWorld (AW) and MobileWorld (MW). AW leaderboard numbers are sourced from the public GUI-Owl/MAI-UI benchmark comparison page; MW leaderboard numbers are sourced from the official MobileWorld `leaderboard.json` snapshot at `https://tongyi-mai.github.io/MobileWorld/`, accessed 2026-05-24 (31 entries). Several methods are reported under slightly different display names across the two leaderboards (e.g. `Qwen3-VL-32B-Instruct` on AW vs. `Qwen3-VL-32B` on MW, `MAI-UI-8b` vs. `MAI-UI-8B`); we collapse these to a single row when the upstream identifier and release date match. **SR-CLI** and **Avg Step** columns are only populated for our reproduced GUI baselines and our CLI agents (mean (std) across three seeds, $\{7, 30, 1234\}$); leaderboard-only rows release neither per-task verifier outputs nor step counts, so those cells are marked "–". "–" in the SR columns means the method does not currently appear on that benchmark's leaderboard.

Reproduction sanity. The three reproduced GUI baselines fall within $\sim$1.5 pp of the corresponding leaderboard rows: GUI-Owl-1.5-32B (AW 69.3 reproduced vs. 69.8 leaderboard), MAI-UI-8B (AW 68.1 vs. 70.7), and Qwen3-VL-32B (AW 57.8 vs. 57.3). The MW gaps are similarly small for GUI-Owl-1.5-32B (reproduced 43.2 vs. leaderboard 43.9) and MAI-UI-8B (26.3 vs. 27.5); Qwen3-VL-32B reproduces above its leaderboard MW number (13.3 vs. 11.9), which we attribute to seed variance (4.9 pp std on AW, 0.7 pp std on MW).

Cross-benchmark difficulty drop. Every leaderboard method that reports both AW and MW drops substantially on MW — e.g. MAI-UI-235B-A22B from 76.7 to 39.7, GUI-Owl-1.5-32B-Instruct from 69.8 to 43.9, Qwen3-VL-235B-A22B from 63.7 to 12.8 — consistent with MW's longer-horizon, cross-application, backend-driven task design (Section 3; 27.8 vs. 14.3 average steps; 62.2% vs. 9.5% multi-app). The single highest leaderboard SR on MW (63.2, Seed-2.0-Pro) and the strongest leaderboard end-to-end LLM (Claude-Opus-4.7, 56.4) both exceed our Claude Code + Opus-4.7 CLI configuration (51.9); these leaderboard rows are reported under MW's native screenshot-based protocol and a different harness, so the comparison is across rather than within paradigm.

C.2 Tool ablation examples

We provide two paired Codex trajectories that pin all variables except the toolset (same harness, same model, same seed) and explain the two metric shifts in Table 2. Table 9 (Task 082) is the SR mechanism: `bash-only` loses the trajectory to nested-quote and heredoc errors that prevent it from reading the schema, and ultimately writes `NULL` into fields the verifier expects to be ''; `+tools` sidesteps the same quoting failures with two `sql` calls, observes the existing-row format, and inserts correctly. Table 10 (Task 052) is the cost mechanism: both runs reach reward 1, but `bash-only` burns ten steps recovering from a single apostrophe in the note body, while `+tools` writes the file in one `write-file` call (the wrapper base64-encodes the body internally). Both cases reproduce on a second harness; full traces are in the released artifact.

D Trajectory-Level Error Analysis — Details

This appendix details the full taxonomy (§D.1), the construction pipeline together with its reliability and judge validation (§D.2), and the extended per-agent results behind the trajectory-level error analysis introduced in Section 5.2 (§D.3).

D.1 Full taxonomy

This section defines the shared classification scale used by the trajectory-level error analysis in Section 5.2 across all three comparison axes: (i) GUI vs. CLI paradigms (Figure 2(a)), (ii) `bash-only` vs. `+tools` for GPT-5.3 Codex (Figure 6), and (iii) model APIs and harnesses jointly (Figure 5). How the scale was instantiated on our trajectory pool — the Docent clustering setup, the Opus 4.7 cluster→leaf mapping, and the two-annotator rubric calibration — is documented separately in Appendix D.2.

Two-level paradigm-agnostic taxonomy. The paradigm-agnostic backbone is a two-level adaptation of MAST (Cemri et al., 2026), single-agent subset, following Terminal-Bench's mobile adaptation (Merrill et al., 2026). *Level 1* is three top-level error classes — *Execution*, *Coherence*, *Verification* — that partition the agent's cognitive loop. *Level 2*,

Method	Model API / Notes	AndroidWorld			MobileWorld		
		SR↑	SR-CLI↑	Avg Step↓	SR↑	SR-CLI↑	Avg Step↓
Leaderboard-reported — General-purpose LLMs (no specialized GUI training)							
Seed-1.8	ByteDance, end-to-end	70.7	–	–	52.1	–	–
Seed-2.0-Pro	ByteDance, end-to-end	—	–	–	63.2	–	–
SeedVL-1.5	ByteDance, end-to-end	62.1	–	–	—	–	–
Claude-Sonnet-4.5	Anthropic, end-to-end	56.0	–	–	47.8	–	–
Claude-Opus-4.6	Anthropic, end-to-end	—	–	–	44.5	–	–
Claude-Opus-4.7	Anthropic, end-to-end	—	–	–	56.4	–	–
Gemini-2.5-Pro	Google, end-to-end	69.7	–	–	—	–	–
Gemini-3-Pro	Google, end-to-end	—	–	–	51.3	–	–
Gemini-3.1-Pro-Preview	Google, end-to-end	—	–	–	58.1	–	–
GPT-5.5	OpenAI, end-to-end	—	–	–	62.4	–	–
OpenAI CUA o3	OpenAI, end-to-end	31.3	–	–	—	–	–
Kimi-K2.5	Moonshot, end-to-end	—	–	–	49.6	–	–
Kimi-K2.6	Moonshot, end-to-end	—	–	–	55.6	–	–
Qwen3.5-397B-A17B	Alibaba, end-to-end	—	–	–	42.7	–	–
Qwen3-VL-8B-Instruct	Alibaba, end-to-end	47.6	–	–	9.4	–	–
Qwen3-VL-32B-Instruct	Alibaba, end-to-end	57.3	–	–	11.9	–	–
Qwen3-VL-235B-A22B-Instruct	Alibaba, end-to-end	63.7	–	–	12.8	–	–
Leaderboard-reported — GUI specialists (mobile post-trained)							
UI-Venus-1.5-30B-A3B	Ant Group	77.6	–	–	17.1	–	–
UI-Venus-72B	Ant Group	—	–	–	16.4	–	–
UI-Venus-7B	Ant Group	—	–	–	8.5	–	–
MAI-UI-8B	Alibaba	70.7	–	–	27.5	–	–
MAI-UI-32B	Alibaba	73.3	–	–	36.2	–	–
MAI-UI-235B-A22B	Alibaba	76.7	–	–	39.7	–	–
UI-TARS-72B-DPO	ByteDance	46.6	–	–	—	–	–
UI-TARS-1.5	ByteDance	64.2	–	–	—	–	–
UI-TARS-2	ByteDance	73.3	–	–	—	–	–
Doubao-1.5-UI-TARS	ByteDance	—	–	–	26.3	–	–
Step-GUI-8B	StepFun	67.7	–	–	16.1	–	–
GELab-Zero-4B	StepFun	63.9	–	–	16.1	–	–
GUI-Owl-7B	Alibaba	66.4	–	–	7.7	–	–
GUI-Owl-32B	Alibaba	—	–	–	8.5	–	–
GUI-Owl-1.5-2B-Instruct	Alibaba	67.9	–	–	32.2	–	–
GUI-Owl-1.5-4B-Instruct	Alibaba	69.8	–	–	—	–	–
GUI-Owl-1.5-8B-Instruct	Alibaba	69.0	–	–	38.2	–	–
GUI-Owl-1.5-8B-Thinking	Alibaba	71.6	–	–	—	–	–
GUI-Owl-1.5-32B-Instruct	Alibaba	69.8	–	–	43.9	–	–
GUI-Owl-1.5-32B-Thinking	Alibaba	69.8	–	–	—	–	–
Leaderboard-reported — Agentic frameworks (planner + grounder composition)							
GPT-5 + UI-Ins-7B	OpenAI + Ant Group	—	–	–	54.0	–	–
Gemini-3-Pro + UI-Ins-7B	Google + Ant Group	—	–	–	55.6	–	–
Claude-4.5-Sonnet + UI-Ins-7B	Anthropic + Ant Group	—	–	–	47.8	–	–
Reproduced GUI baselines (our infrastructure)							
GUI-Owl-1.5-32B	–	69.3 (0.5)	71.6 (1.5)	15.5 (2.5)	43.2 (2.6)	38.1 (2.0)	34.7 (0.7)
MAI-UI-8B	–	68.1 (0.6)	69.3 (0.3)	18.4 (1.2)	26.3 (1.3)	26.0 (2.0)	39.7 (0.3)
Qwen3-VL-32B	–	57.8 (4.9)	62.4 (5.6)	17.4 (1.8)	13.3 (0.7)	11.7 (0.3)	25.6 (0.6)
CLI agents (ours)							
Claude Code	Claude Opus 4.7	71.8 (1.8)	80.9 (2.0)	15.2 (0.6)	51.9 (1.0)	60.1 (1.1)	30.2 (0.6)
mini-swe-agent	Claude Sonnet 4.6	70.7 (2.3)	79.6 (2.6)	12.1 (0.3)	43.6 (2.6)	50.5 (3.0)	33.5 (0.4)
Terminus-2	GPT-5.3 Codex	63.2 (0.5)	71.2 (0.6)	7.1 (0.1)	36.2 (1.0)	41.9 (1.1)	24.5 (1.9)
Terminus-2	MiniMax M2.7	55.2 (3.8)	62.1 (4.2)	14.5 (0.5)	16.8 (1.3)	19.5 (1.5))	32.9 (0.8)
Oracle (CLI ceiling)	–	88.8	100.0	3.7	86.3	100.0	5.6

Table 8: **Full leaderboard-extended version of Table 1.** SR is success rate (%); SR-CLI restricts the denominator to the CLI-solvable subset of each benchmark (103/116 on AW; 101/117 on MW; see Appendix A.3); Avg Step is mean agent steps per task. Reproduced and CLI rows report mean (std) across three seeds, $\{7, 30, 1234\}$. Leaderboard rows are taken from the AndroidWorld GUI-Owl/MAI-UI comparison page and the MobileWorld official `leaderboard.json` snapshot (accessed 2026-05-24) and report a single number per benchmark per the conventions of those leaderboards; SR-CLI and Avg Step are unavailable for these rows because per-task and per-step traces are not released. "–" marks unavailable metrics; "—" marks methods absent from a given leaderboard. Methods reported under slightly different display names across the two leaderboards (e.g. `MAI-UI-8b`/`MAI-UI-8B`, `Qwen3-VL-32B-Instruct`/`Qwen3-VL-32B`) are collapsed to a single row.

underneath each class, is a small set of **named failure modes** (the MAST *leaves*); the nine MAST leaves we inherit retain their original semantics but are re-stated in Android-specific terms (Table 11). All nine leaves are defined for both paradigms with identical operational meanings; only their prevalence differs. Using the same Level-1/Level-2 scale for both paradigms is what lets us write a single

bash-only

17 steps + finish, 221,689 input tokens, reward 0

#	Action → Result
1	`pm list packages | grep -Ei 'markor|broccoli'; find ... -iname recipes.txt` → both packages + `/sdcard/Documents/Markor/recipes.txt`
2	`cat recipes.txt; run-as broccoli ...` → recipes read; `run-as` blocked (not debuggable)
3	probe external dir + `cmd package resolve-content-provider` (4 guesses) → ext dir empty; `Unknown command`
4	list ext files + `awk -F'|' 'NR==1 || $4=="4 hrs"'` → empty; 3 four-hr rows found
5	probe 7 candidate `content://com.flauschcode.broccoli*` URIs → 6 `Could not find provider`, 1 `No result found`
6	`find` Broccoli-named files → only `recipes.txt`
7	`logcat -c; force-stop; monkey; logcat -d` → discovers DB at `/data/.../broccoli/databases/broccoli` from `FileUtils: chmod` log
8	`cmd package query-activities -a SEND -t text/plain` → matches `CreateAndEditRecipeActivity`
9	`am start ... SEND text/plain ... -es TEXT 'Title|desc|...'` → "supplied URL is malformed" (SEND handler scrapes URLs)
10	enumerate 7×7 action/MIME combos via nested `"$out"` → `/system/bin/sh: no closing quote`
11	retry intent enumeration with cleaner quoting → only SEND text/plain matches
12	direct `sqlite3 broccoli '.tables'` → tables listed
13	`adb shell "sqlite3 broccoli \".schema recipes; .schema categories...\""` → `Usage: .schema ?--indent? ?LIKE-PATTERN?` (nested-quoted dot-cmd)
14	heredoc `cat <<'SQL' | sqlite3 broccoli` → schema returned; sample queries fail with `no such column: id`
15	heredoc with `json_each(readfile(...))` + nested `substr/instr` parser → `Error: near "(": syntax error`; `json_each` not built
16	`awk -F'|' '... printf "INSERT...VALUES(...NULL...NULL...);" ...' | sqlite3` → `/system/bin/sh: no closing quote`
17	heredoc `.import recipes.txt _import_recipes; INSERT INTO recipes SELECT title, NULL, ..., NULL, ..., 0 FROM _import_recipes WHERE preparationTime='4 hrs'` → 3 rows inserted with `imageName=NULL, source=NULL`
18	`finish "Inserted 3 recipes ..."` → **Reward 0** — `imageName/source` should be `''`, not `NULL`

+tools

11 steps + finish, 92,826 input tokens, reward 1 (−58%)

#	Action → Result
1	`pm list packages | grep -Ei 'markor|broccoli'; find ... -name recipes.txt` → both packages + recipes.txt path
2	`cat recipes.txt; run-as broccoli ...; dumpsys package | grep provider` → recipes read; `run-as` blocked
3	probe ext dir + `find /sdcard *broccoli*` + recipe-file candidates → only `recipes.txt` matches
4	`ls /data/.../broccoli/databases + find ... -name '*.db'` → **`broccoli` DB world-readable**
5	`sql broccoli "SELECT type,name,sql FROM sqlite_master ..."` → full schema in one call; FTS triggers identified
6	`sql broccoli "SELECT recipeId, title, imageName, ..., source, ... FROM recipes LIMIT 10"` → **observes `imageName=''`, `source=''`** in every existing row
7	`sql broccoli "SELECT preparationTime, COUNT(*) FROM recipes GROUP BY preparationTime"` → 0 existing 4-hr rows; safe to insert
8	`python3 --version 2>/dev/null || echo NO_PYTHON` → `NO_PYTHON`
9	`awk -F'|' 'NR==1 || $4=="4 hrs" {print NR"|"$0}' recipes.txt` → exactly 3 source rows (Margherita, Grilled Cheese, Avocado Toast)
10	`sql broccoli "BEGIN; INSERT INTO recipes (..., imageName, ..., source, ...) VALUES ('...', '', ..., '', ...); ... COMMIT;"` → **3 rows inserted with `imageName=''`, `source=''`**
11	`am force-stop com.flauschcode.broccoli` → —
12	`finish "Added the 3 recipes ..."` → **Reward 1**

Table 9: **Case 1 — Task 082 (Broccoli recipes), Codex.** Full `bash-only` (17-step) and `+tools` (11-step) trajectories; same harness, same seed. The SR shift (0 → 1) traces to repeated nested-quote and heredoc errors in steps 10–16 of `bash-only` that prevent it from reading the existing-row format, leading it to insert `NULL` where the verifier expects `''` (step 17). `+tools` reads the schema and existing-row format in two `sql` calls (steps 5–6) and inserts correctly (step 10).

classification statement about any trajectory regardless of action interface.

Paradigm-specific clusters underneath the taxonomy. Below the two-level taxonomy we further annotate each trajectory with a Docent (Meng et al., 2025) cluster that captures the concrete mobile mechanism behind the failure. These clusters are paradigm-specific because the action interfaces

bash-only

17 steps + finish, 445,700 input tokens, reward 1

#	Action → Result
1	`pm list packages | grep -i markor` → `net.gsantner.markor`
2	`ls /sdcard/Documents/; ls /sdcard/Markor/; ls /sdcard/ | head -30` → finds `Markor` under `/sdcard/Documents/`
3	`ls -la /sdcard/Markor/` → no such directory
4	`ls -la /sdcard/Documents/` → confirms `Markor/` subfolder
5	`ls -la /sdcard/Documents/Markor/` → empty
6	`run-as net.gsantner.markor cat shared_prefs/*.xml | grep dir|folder|path` → (no output)
7	`adb shell 'echo "Lunch ... Cafe L'"'"'amour." > /sdcard/.../eHwd_helpful_jacket.md && ls -la ... && cat ...'` → `no closing quote; adb: ls requires an argument; adb: unknown command cat` (&& parsed as separate adb subcommands)
8	`adb shell 'cat > /sdcard/.../eHwd_helpful_jacket.md'` → (no output, stdin not connected)
9	`ls -la && cat` chain → same quoting error
10	`adb shell ls -la /sdcard/Documents/Markor/` → file exists but **0 bytes**
11	`adb shell 'cat > /sdcard/.../...'` → still empty
12	`adb shell ls -la ...` → 0 bytes
13	`adb shell 'echo "Lunch meeting ... Cafe L'"'"'amour." > /sdcard/.../...'` (no && chain) → written
14	`adb shell ls -la ...` → **47 bytes**
15	`adb shell cat ...` → `Lunch meeting with Sarah at 1 PM Cafe L'amour.`
16	`am broadcast MEDIA_SCANNER_SCAN_FILE` → `Broadcast completed: result=0`
17	`am force-stop net.gsantner.markor` → —
18	`finish "Created note ..."` → **Reward 1**

+tools

7 steps + finish, 242,191 input tokens, reward 1 (−46%)

#	Action → Result
1	`pm list packages | grep -i markor` → `net.gsantner.markor`
2	`run-as markor ls shared_prefs/; ls /sdcard/Documents/; ls /sdcard/Markor/; ls /sdcard/` (one batched probe) → `run-as` blocked; `Markor` under `/sdcard/Documents/`
3	`ls -la /sdcard/Documents/Markor/` → empty
4	`find-files /sdcard "*.md"` → (no output — confirms no stray notes)
5	`write-file /sdcard/.../eHwd_helpful_jacket.md` content: `Lunch meeting with Sarah at 1 PM Cafe L'amour.` → written (wrapper base64-encodes body internally; **apostrophe never enters the shell**)
6	`read-file /sdcard/.../eHwd_helpful_jacket.md` → `Lunch meeting ... Cafe L'amour.` (apostrophe preserved)
7	`am broadcast MEDIA_SCANNER_SCAN_FILE; am force-stop net.gsantner.markor` → `Broadcast completed: result=0`
8	`finish "Created ... with content: ..."` → **Reward 1**

Table 10: **Case 2 — Task 052 (Markor note with apostrophe), Codex.** Full `bash-only` (17-step) and `+tools` (7-step) trajectories. Both runs reach reward 1; the cost difference comes from `bash-only` burning steps 7–13 escaping a single apostrophe in the note body (combined with an `&&`-in-`adb-shell` parsing interaction). `+tools`'s `write-file` base64-encodes content internally, so neither the apostrophe nor any other content character is ever exposed to shell parsing.

themselves are paradigm-specific: a CLI *wrong write surface* has no GUI analogue, and a GUI *wrong-row tap* has no CLI analogue. The bottom-up clustering pass surfaced 20 CLI clusters and 18 GUI clusters (Tables 12 and 13); every classified trajectory is assigned to exactly one cluster, and each cluster's primary MAST leaf is fixed by the mapping step in Appendix D.2. Clusters therefore play the role of a paradigm-specific *third* layer beneath the paradigm-agnostic taxonomy: they refine, but do not extend, the MAST leaves.

Top-level classes and leaves. Table 11 lists the three classes, the nine MAST-derived leaves grouped under them, the operational definition of each leaf, and the primary-label prevalence we observe. Five leaves are active in at least one paradigm; four are uniformly zero in our setting (rationale at the end of this section). We retain the four inactive leaves in the published taxonomy so that future longer-horizon or multi-agent extensions can populate them without redefining the tree. The reading of the multi-label Figure 2(a) vs. the primary-label table here is also discussed in Appendix D.2.

Paradigm-specific clusters — CLI. Table 12 enumerates the 20 CLI clusters grouped under

Class	Leaf	Operational definition	CLI %	#CLI	GUI %	#GUI
Execution	**Disobey Specification (DS)**	Action is well-formed at the interface but does not satisfy the verifier contract — wrong surface, wrong value, or wrong artifact for the same goal.	**76.7**	16	**40.4**	11
	Step Repetition (SR)	Same action repeated on the same observation with no progress; loop continues until the step budget exhausts.	0.0	0	**35.9**	4
	Unaware of Termination	Agent does not realize the task is done and keeps issuing actions past the success state.	0.0	0	0.0	0
Coherence	**Reasoning-Action Mismatch (RAM)**	The action does not implement the predicate the reasoning describes; the trajectory commits to a divergent action because it runs cleanly.	**14.0**	2	0.0	0
	Context Loss	Earlier observation or instruction is no longer reflected in the agent's state; the agent acts on stale context.	0.0	0	0.0	0
	Task Derailment	Agent veers to a related but unrequested goal in the middle of the trajectory.	0.0	0	0.0	0
Verification	**Premature Termination (PT)**	Agent calls `finish/submit` without a post-action read of the device state; the closing handshake is decoupled from what actually happened.	**5.7**	1	**23.7**	3
	No or Incorrect Verification	A verification command is issued but its result is logically mis-interpreted (e.g., empty result read as success).	0.0	0	0.0	0
	Weak Verification (WV)	Verification reads the same surface that was just written, while the grader reads a different surface; the agent's confirmation is non-informative for the actual contract.	**3.6**	1	0.0	0

Table 11: **Two-level paradigm-agnostic failure taxonomy: three top-level MAST classes and the nine MAST-derived leaves underneath.** The **CLI %** and **GUI %** columns give the primary-label share of failed trajectories assigned to each leaf (active-leaf shares sum to 100% within each paradigm). The **#CLI** and **#GUI** columns count the paradigm-specific clusters under each leaf (Tables 12 and 13). Bolded leaves are active in at least one paradigm; **DS** is the cross-paradigm dominant mode. Construction details, including how primary-label vs. multi-label shares relate to Figure 2(a), are in Appendix D.2.

each active leaf (Disobey Specification, Reasoning-Action Mismatch, Premature Termination, Weak Verification). The identifiers are the canonical cluster names used throughout our released artefact, including in Figure 6 (Appendix D.3). The third column gives the dominant Android mechanism behind the failure; the **Share** column reports each cluster's share of the classified CLI pool.

Paradigm-specific clusters — GUI. Table 13 enumerates the 18 GUI clusters grouped under each active leaf (Disobey Specification, Step Repetition, Premature Termination); none of the four Coherence or remaining Verification leaves are activated, consistent with the per-leaf zeros in Table 11. The GUI mass is more concentrated than the CLI mass: a single Step-Repetition cluster (`identical_action_loop_until_step_budget_exhausted`) absorbs more than a quarter of all GUI failures, and three clusters above 10% jointly account for over half of the pool.

Departures from MAST. We follow Terminal-Bench in dropping MAST's inter-agent coordination subcategories (*Specification Issues*, *Inter-Agent Misalignment*) because all our agents are single-agent. Within the remaining tree, the four uniformly-zero leaves in Table 11 have a single-agent rationale: *Unaware of Termination* does not fire because both paradigms expose an explicit `finish/submit` action with hard semantics; *Context Loss* and *Task Derailment* do not fire because our tasks are short-horizon ($\leq$ 50 steps) and the goal text is re-displayed every step in all three harnesses; *No or Incorrect Verification* is empirically subsumed by Weak Verification (verifying on the wrong surface) in our setting, because the cases where the verification *logic* is malformed turn out

to coincide with cases where the verification *surface* is also wrong.

D.2 Construction

This appendix documents how the taxonomy in §D.1 was instantiated on our trajectory pool and how its reliability was validated.

Pipeline. The labelling pipeline runs in three stages:

- **Per-trajectory failure summary.** Each rollout is distilled into a structured, rubric-shaped paragraph by Claude Opus 4.7, recording the agent's surface, value, and verification choices.
- **Bottom-up clustering (Docent (Meng et al., 2025)).** Docent agglomeratively clusters the per-trajectory summaries within each paradigm separately, yielding the paradigm-specific clusters listed in Tables 12 and 13.
- **Cluster → leaf mapping.** Claude Opus 4.7 reads each cluster's exemplars and assigns it a single MAST leaf, using a rubric that was *calibrated against two human annotators* on a 20-trajectory calibration subset before being released to the rest of the pool.

Docent setup. We use Docent (Meng et al., 2025) (`v0.4.2`) with its default setup. The CLI and GUI failure pools are clustered separately so that paradigm-specific mechanisms surface as distinct clusters. GUI trajectories are pre-processed by rendering each rollout as an action+observation transcript: action JSONs are kept verbatim and the corresponding accessibility-tree screenshots are replaced by a one-sentence Claude-Opus 4.7 caption of the rendered screen, keeping the input length comparable to a CLI rollout. Docent surfaces 20 CLI clusters and 18 GUI clusters; singletons are discarded and the residual "noise" bucket (cluster -1) accounts for 4.1% of CLI failures and 2.4% of GUI failures, which we exclude from the per-cluster prevalence numbers.

Cluster→leaf prompt (verbatim). The Opus 4.7 judge receives the prompt in Figure 4 for every cluster. `{cluster_id}` is the canonical Docent name, `{exemplars}` is a random sample of eight trajectory summaries drawn from the cluster, and the two rubric files are passed verbatim in the system prompt.

Rubric structure. The rubric (`rubric_v2.md`) is a flat, 9-row file: one row per MAST leaf, each row giving (i) a one-sentence operational definition, (ii) an Android-specific framing in the spirit of Section §D.1, and (iii) 2–3 positive exemplars and 1–2 negative exemplars drawn from the calibration subset (see below). The assignment procedure is the top-down tree walk in the prompt above: a cluster is first placed in one of three top-level classes (Execution/Coherence/Verification), then in the best-fitting leaf within that class. The companion file `rubric_v2_clarifications.md` contains 16 numbered tie-breakers covering ambiguities that surfaced during calibration — e.g. same-surface verification (DS vs. WV), timezone misinterpretation (DS, wrong value), Q&A from the wrong provider (PT vs. WV), GUI long-press idiom (DS, wrong API level), single-slot clipboard during multi-source merge (DS, wrong protocol). Each tie-breaker is a single paragraph stating the pattern, the leaf it maps to, and a worked example, and is referenced by number from the judge prompt.

Inter-annotator agreement. On the 20-trajectory calibration subset, two human annotators (one author and one external researcher with 5+ years of Android-development experience, blinded to the cluster identities) independently assigned each trajectory a top-level class. They agreed on $18/20$ trajectories (raw agreement $P_o = 0.90$); the two disagreements were both at the Coherence/Execution boundary. Cohen's $\kappa = 0.85$ (95% CI $[0.62, 1.00]$, 1,000-sample bootstrap) — the gap from P_o to κ reflects the chance-agreement correction at three roughly balanced classes ($P_e \approx 1/3$).

Judge validation against human labels. On a disjoint, held-out validation set of $M = 60$ trajectories (20 per top-level class, stratified by paradigm and harness), the Opus 4.7 judge agrees with the consensus human top-level-class label on $56/60$ trajectories ($P_o = 0.933$); Cohen's $\kappa = 0.90$ (95% CI $[0.80, 0.98]$, 1,000-sample bootstrap), at or above the inter-human $\kappa = 0.85$ on the calibration subset, so we treat the judge as a substitute for a third human annotator.

Cluster→leaf mapping table (released artefact). The full 38-row mapping — every Docent cluster, its assigned primary leaf, the tie-breaker invoked (if any), the secondary leaf (if any), and a flag for

```
You are mapping a Docent failure-mode cluster onto our two-level mobile failure taxonomy:
3 top-level classes (Execution, Coherence, Verification) and 9 named leaves underneath.
The taxonomy is single-agent MAST adapted to Android per Terminal-Bench.

INPUTS
  cluster_id     : canonical Docent cluster name
  exemplars      : 8 per-trajectory failure summaries from this cluster
  rubric         : rubric_v2.md (one-sentence operational definition of each of the
                   9 leaves; Android-specific framings)
  tie_breakers   : rubric_v2_clarifications.md (16 numbered rules that resolve the
                   most common 2-leaf ambiguities; each rule has a pattern,
                   a "maps to" leaf, and an example)

TASK
  1. Read the 8 exemplars. State the dominant Android failure mechanism in one
     sentence, in your own words (no taxonomy jargon).
  2. Walk the tree top-down. Which top-level class (Execution, Coherence, or
     Verification) best describes the failure? Justify in one sentence.
  3. Within the chosen class, pick the single best-fitting leaf.
  4. If two leaves are equally plausible, consult tie_breakers and cite the rule
     number that decides (e.g. "TB7: timezone misinterpretation -> Disobey
     Specification, wrong output protocol").
  5. Optionally record one secondary leaf, only if at least 3 of the 8 exemplars
     also support it. Never list more than one secondary.

OUTPUT  (strict JSON, no prose outside the object)
  {
    "cluster_id":     "...",
    "mechanism":      "one sentence in plain English",
    "primary_class":  "Execution | Coherence | Verification",
    "primary_leaf":   "...",
    "tie_breaker":    "TB# or null",
    "secondary_leaf": "... or null",
    "rationale":      "2-4 sentences grounded in specific exemplars"
  }
```

Figure 4: **Verbatim Opus 4.7 prompt for the cluster→leaf mapping step (Appendix D.2).** `rubric_v2.md` and `rubric_v2_clarifications.md` are passed in the system prompt. The judge runs in JSON mode with temperature 0, reasoning effort high, and a maximum of 4,096 output tokens.

whether the Opus 4.7 judge agreed with both human annotators — is too large to reproduce inline; it is released as `cluster_to_leaf_mapping.csv` in the analysis bundle. Across all 38 clusters, the judge agreed with the consensus human label on 36 (94.7%).

D.3 Extended results

This subsection covers the two extensions of Section 5.2 that did not fit in the main paper: the joint model × harness breakdown that motivates Axis (iii) of our error analysis (Figure 5), and the sub-cluster decomposition of the headline Axis (ii) result on *Disobey Specification* (Figure 6).

Failure-mode prevalence by model and harness (Figure 5). Figure 5 expands the cross-paradigm view of Figure 2(a) into the full model × harness matrix used by the CLI block of Table 1: rows of the figure are sub-level failure categories from the taxonomy of Appendix D.1, and columns are the eight CLI cells (three harnesses — Claude Code, mini-swe-agent, Terminus-2 — crossed with the four model APIs of Section 5.1, with the Claude Code × {Codex, MiniMax} cells empty because the Claude Code SDK is Claude-only). The takeaway, complementing the SR-level robustness claim of Section 5.1, is that *the failure-mode distribution is shaped primarily by the model API, not the harness*: holding the model fixed, the three harness columns sit within a few percentage points of each other on every active sub-category. Holding the harness fixed and varying the model, by contrast, moves the bars substantially — most visibly, GPT-5.3 Codex carries a much larger *Premature Termination* mass than the Claude models on the same harness, which is consistent with its bash-only SR deficit in Table 2 and with the Axis (ii) mechanism described next. All bars are computed over the failed CLI trajectories on the 103-task CLI-solvable AndroidWorld subset (Appendix A.3), averaged across three seeds.

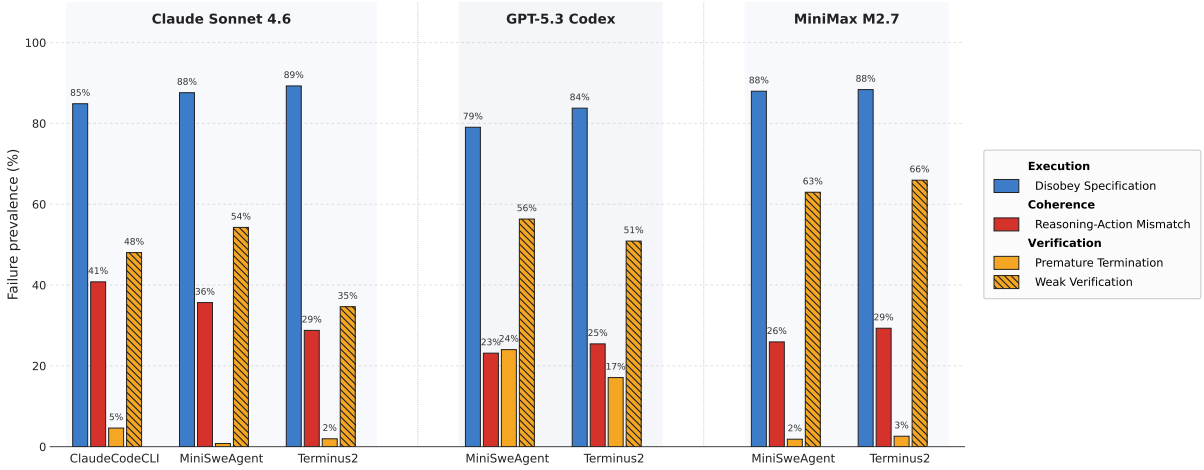

Figure 5: **Failure-mode prevalence by model and harness in the CLI paradigm.** Each column corresponds to one (model × harness) cell on the fair-comparison subset (1,508 failed trajectories across the 103 CLI-solvable AndroidWorld tasks). Bars give the prevalence of each sub-level category. The empty Codex / MiniMax slots under ClaudeCodeCLI reflect that ClaudeCodeCLI is a Claude-only harness.

Disobey Specification sub-category shifts under `+tools` for Codex (Figure 6). Figure 6 drills into the headline Axis (ii) result of Section 5.2 — the four-tool wrapper's effect on GPT-5.3 Codex — at the sub-cluster level of the dominant *Disobey Specification* mode. The figure decomposes which fine-grained sub-clusters drive that net change. Blue (top) bars are the sub-clusters that shrink under the wrapper; orange (bottom) bars are the ones that grow. The blue mass concentrates on the shell-mechanical failures the four tools are designed to neutralize — nested-quote and heredoc errors, apostrophe-in-text escaping, unicode round-trip corruption, `adb shell` double-escaping — consistent with the Codex case studies in Appendix C.2, where `+tools` converts a bash-only failure into a success by absorbing exactly this category of quoting failure into the `sql` / `write-file` wrappers. The orange mass shifts toward higher-level specification mismatches (wrong URI, wrong column, wrong timestamp unit) that are visible only once the agent can execute the right shell at all — in other words, removing the shell-level wall does not eliminate the trajectory's failure modes, it advances them downstream. This is the precise mechanism behind the Axis (ii) summary in Section 5.2 (*Premature Termination* cut by half, *Reasoning-Action Mismatch* and *Weak Verification* both up by $\sim$ 10 pp) and behind the same-direction SR / cost shift in Table 2: the tools convert a class of bash-mechanics failures into a smaller class of downstream reasoning errors, raising SR while leaving the harder semantic failures for the model to solve.

E LLMs Usage

ChatGPT5 is used solely as a general-purpose writing assistant. Specifically, we apply a fixed prompt template — "Polish the writing in a concise and academic way" — to improve grammar, clarity, and style of text written by the authors. The LLM did not contribute to research ideation, methodology design, experimental setup, analysis, or result interpretation.

F Potential Risks and Impact Statement

We flag two privacy risks specific to deploying CLI mobile agents outside the benchmark setting. First, on a real device the CLI surface reads user-private storage directly through `adb shell` and `content` queries, without the permission prompts, SAF intents, or provider-authority checks that mediate ordinary app access; any deployed CLI agent is therefore a privileged on-device process that sees the user's data in clear form by construction. Second, the agents we evaluate are driven by frontier model APIs hosted in the cloud, which means that whatever the on-device CLI reads — including the user-private data above and any task context that quotes it — is sent off-device to the model provider on every step; a careless deployment leaks user information to a third party.

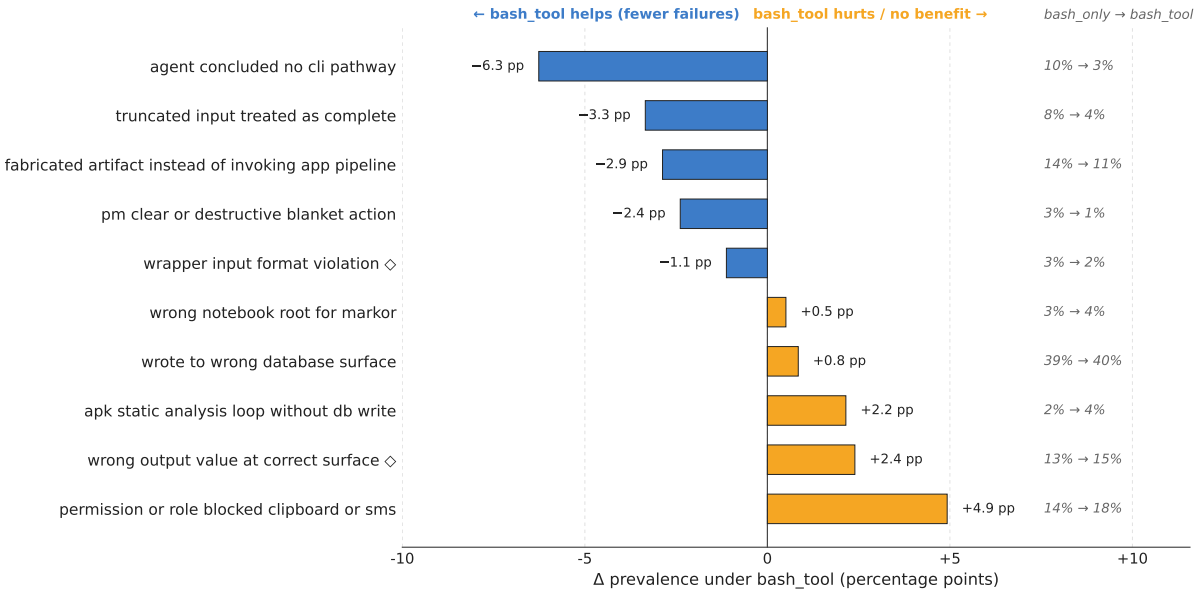

Figure 6: **Disobey Specification sub-cluster shifts under the four-tool wrapper (GPT-5.3 Codex).** Each row is one fine-grained sub-cluster within Disobey Specification; bar length is the change in prevalence from `bash-only` to `+tools` on the fair-comparison subset. Blue bars (top) shrink under the wrapper; orange bars (bottom) grow. The italic column reports the absolute prevalence pair (`bash-only` → `+tools`).

Leaf	Docent cluster (canonical name)	Dominant Android mechanism	Share
DS	wrote_to_wrong_database_surface	Persisted state to a SQLite file or content provider that the target app does not consume (e.g., wrote SMS rows to AOSP mmssms.db when the verifier reads Simple SMS's own conversations.db).	20.7%
	ocr_or_vision_tool_hunt_without_writing_target	Burned the entire step budget probing for OCR / vision tools (tesseract, strings/xxd over image bytes, MediaStore description columns) and never executed the target INSERT or write-file.	8.9%
	task_required_gui_only_no_cli_pathway	Task ground truth is gated by an on-screen tap or rendered DOM state with no shell write surface (Chrome canvas verifiers, Camera shutter, AudioRecorder UI). Agent computed answers offline via awk/bc or fabricated artifacts but the verifier reads the live foreground UI.	8.2%
	fabricated_artifact_instead_of_invoking_app_pipeline	Hand-crafted the artifact (base64 JPEG, screenrecord MP4, GPX with invented coords, raw contacts2.db INSERT) instead of driving the responsible app to produce real provenance bytes/rows.	8.2%
	permission_or_role_blocked_clipboard_or_sms	Looped on shell calls to clipboard / SMS surfaces blocked by Android 13's permission gate, often misreading a SecurityException parcel as "clipboard empty" rather than pivoting to a foreground app or the default-SMS role.	6.3%
	time_or_timezone_misinterpretation	Encoded ambiguous time literals as the wrong period (PM vs. AM, UTC vs. device-local), or used overlap (start<end AND end>start) instead of containment in time-window queries.	4.4%
	recurrence_or_filter_predicate_omission	SQL or content-query predicate dropped a required column the schema exposed: repeat_interval on recurring events, dueDate vs. completed on org.tasks, hideUntil/importance/parent on subtasks.	4.0%
	byte_exact_file_content_mismatch	Markor merge/edit/paste produced almost-right bytes but introduced \n\n separators where a single \n was required, dropped a trailing newline, or substituted file extension; cat-based self-checks could not catch the byte-level disagreement.	3.0%
	apk_static_analysis_loop_without_db_write	Locked into 30–50 turns of strings/unzip -p/xxd/dexdump against base.apk trying to recover an enum-or-ordinal, and never executed the required INSERT.	3.0%
	truncated_input_treated_as_complete	Read a source file or dump where the harness cut output mid-row, then synthesized invented continuations rather than paginating with sed/tail/wc -l or re-issuing with a projection.	2.8%
	wrong_notebook_root_for_markor	Inferred Markor's notebook directory from the presence of seeded .md files instead of reading pref_key__notebook_directory from shared_prefs, writing notes one directory level too deep or too shallow.	2.3%
	screenrecord_or_screencap_substitute_for_camera	Substituted screenrecord MP4 or screencap -p PNG (sometimes renamed .jpg) for the camera shutter; verifier checked MediaStore.owner_package_name and rejected the screen-capture provenance.	2.3%
	pm_clear_or_destructive_blanket_action	Used pm clear, rm -rf of an entire directory, or DELETE FROM sqlite_master where the verifier expected a single-row delete; wiped sibling data that was meant to remain intact.	0.8%
	shell_quoting_or_harness_parse_blocked_action	Command died in the wrapper due to nested-quote / apostrophe issues (L'amour inside an inner-shell INSERT), multi-JSON-per-turn parse errors, or max_tokens truncation; zero commands reached the device.	0.6%
	chrome_firstrun_or_file_scheme_blockade	Burned steps fighting Chrome's FirstRunActivity via shared_prefs edits and am start ChromeTabbedActivity hacks instead of tapping the FRE accept button through Files-app→Chrome.	0.6%
	harness_verb_or_command_prefix_violation	Emitted bare Android verbs (settings, cmd, svc, am) without the adb shell prefix the wrapper requires; the parser rejected every command before it reached the device.	0.4%
RAM	treated_broadcast_or_intent_dispatch_as_state_change	Fired am broadcast/am start with an invented or AOSP-only action string and accepted result=0 or a dumpsys top-activity match as proof of the state change, without inspecting persisted shared_prefs / SQLite / UI.	7.6%
	inverted_or_guessed_enum_mapping	Wrote a guessed integer for an unknown enum (importance where 0 means *High*; repeat_rule weekday bitmask; Pro Expense category; DeskClock timer_state_0) after reasoning admitted the mapping was uncertain.	6.3%
PT	answer_question_through_wrong_provider	For app-specific Q&A (Simple Calendar Pro, OpenTracks, org.tasks), queried the AOSP system provider, accepted No result found. as final, and answered None; never tried the app-private SQLite under /data/data/<pkg>/databases/.	5.7%
WV	self_confirming_select_loop	After any write (sqlite3 INSERT, content insert, settings put), verified success by re-reading the exact same surface with the inverse verb, never round-tripping through the consumer's actual read path (content query, app relaunch, MediaStore re-scan).	3.6%

Table 12: **Paradigm-specific CLI clusters grouped under their assigned MAST leaf.** The **Share** column reports each cluster's primary-label share of the classified CLI pool; within-leaf shares sum to the leaf totals in Table 11, and cross-leaf shares sum to 100%. Cluster→leaf assignments, the rubric and tie-breakers, and the two-annotator agreement statistics are reported in Appendix D.2.

Leaf	Docent cluster (canonical name)	Dominant Android mechanism	Share
DS	skipped_source_image_then_fabricated_destination_entries	Launched the viewer app for a named source media file, switched apps within 1–2 steps with no full-screen render observed, then typed invented destination values (Pro Expense rows, Broccoli recipe lines, Markor transcriptions).	11.8%
	missing_two_step_record_or_count_observation	Collapsed an observe-between-actions pattern (start-record/stop-record; tap-then-count) into back-to-back taps at the same coordinate with no intervening observation, so no artifact or correct aggregate is produced.	4.9%
	stuck_on_search_or_filter_dialog_with_wrong_field_type	Typed human-readable date strings or category phrases into a title-only search field (Tasks, Broccoli, OpenTracks), got empty results, then answered `0`/`""` or looped on the same wrong query.	4.5%
	deletion_or_move_targeted_wrong_row_or_first_match_only	Long-pressed a visually similar but wrong filename (`2023_02_13_shy_king_copy.md` instead of `shy_king_copy.md`) or deleted only the first visible row of a multi-row category.	4.5%
	wrong_menu_path_for_markor_rename	Cycled through Markor's editor overflow menu (*File settings* → *Format*) instead of long-pressing the file-list row, so the filename never changes.	4.1%
	missing_long_press_gesture_for_selection_or_marker	Used `click` or `scroll` where the Android idiom requires `long_press` (Markor delete, OsmAnd marker, VLC multi-select, text-selection handles) and looped on the wrong gesture indefinitely.	2.4%
	single_tap_on_html_canvas_instead_of_swipe_stroke	On the `task.html` drawing subtask in Chrome, issued a single `click` at one canvas coordinate per colour instead of a `swipe`/drag, so no stroke was registered and the canvas was submitted blank.	2.0%
	wrong_clock_face_digit_in_time_picker	On Simple Calendar Pro's Material 24-hour TimePicker, tapped a digit on the wrong ring leaving the default 16:00 unchanged, or set only the end time without touching start.	2.0%
	answer_string_emitted_as_unknown_action_or_omitted_entirely	Had the correct comma-separated titles in reasoning but emitted them under `action_type:"unknown"/"status"/"wait"` instead of `"answer"`; verifier reports a missing answer payload.	1.6%
	single_clipboard_slot_overwritten_during_multi_source_merge	Chained *Copy* across all source files before any *Paste*, so each new Copy overwrote Android's single-slot clipboard and final Pastes delivered only the last file's content.	1.6%
	sent_message_in_wrong_conversation_thread	Read the source SMS in the sender's thread, then composed and sent the reply inside the same thread rather than backing out to start a new conversation with the named recipient.	0.8%
SR	identical_action_loop_until_step_budget_exhausted	Fired the exact same action (click at one coordinate, scroll in one direction, `wait`, `open_app`) for 30–50 consecutive steps with byte-identical reasoning, never branching when the screen failed to advance.	27.3%
	fixed_coordinate_delete_macro_on_reflowing_list	Recorded a 4-tap delete sequence (row long-press → overflow → Delete → Confirm) at hard-coded y-coordinates on Broccoli/Markor and replayed it 5–9 times without re-reading the list as rows shifted upward.	5.3%
	horizontal_chip_row_scroll_failure_in_pro_expense	Issued ≥ 10 consecutive `scroll(left)` (or alternating left/right) actions against Pro Expense's category chip strip, never tapping a visible chip or trying vertical scroll, never reaching SAVE.	2.0%
	calendar_chevron_oscillation_without_view_switch	Alternated forward and backward chevron taps in the Calendar day/month header for 30–44 steps, undoing its own navigation, never switching to week/agenda view or emitting an `answer`.	1.2%
PT	answered_immediately_after_open_app_without_reading_screen	For retrieval tasks, emitted an `answer` action one step after `open_app` with no scroll/click/wait in between, returning a guessed number, a fabricated comma-separated list, or `0`.	11.8%
	declared_complete_before_dialog_confirmed	Filled a dialog (folder name, brightness slider, time picker, delete confirmation) and emitted `status=complete` before tapping the final OK/confirm button or screenshotting the post-tap state.	9.0%
	filled_only_one_form_instance_for_multi_item_task	Task required N items (recipes, transactions, expenses) but the agent saved only one entry and declared complete, or treated a multi-recipe source file as a single copy-paste into one Title field.	2.9%

Table 13: **Paradigm-specific GUI clusters grouped under their assigned MAST leaf.** The **Share** column reports each cluster's primary-label share of the classified GUI pool; within-leaf shares sum to the leaf totals in Table 11, and cross-leaf shares sum to 100%. The GUI mass concentrates in Step Repetition and Premature Termination ($35.9 + 23.7 = 59.6\%$), whereas the CLI mass concentrates in Disobey Specification (76.7%): the same paradigm-agnostic two-level tree separates the two action interfaces cleanly without redefining any leaf. Construction details are in Appendix D.2.