MobileGym: A Verifiable and Highly Parallel Simulation Platform for Mobile GUI Agent Research

Dingbang Wu[1,2,*] **Rui Hao**[3,1,*] **Haiyang Wang**[4] **Shuzhe Wu** **Han Xiao**[5]
Zhenghong Li[1] **Bojiang Zhou**[1] **Zheng Ju**[1] **Zichen Liu**[1]
Lue Fan[1,5,6,8,†,‡] **Zhaoxiang Zhang**[1,7,8,9,†]
[1]Institute of Automation, Chinese Academy of Sciences (CASIA)
[2]School of Advanced Interdisciplinary Sciences, University of Chinese Academy of Sciences
[3]School of Artificial Intelligence, University of Chinese Academy of Sciences
[4]Peking University [5]The Chinese University of Hong Kong [6]CPII
[7]University of Chinese Academy of Sciences (UCAS)
[8]New Laboratory of Pattern Recognition (NLPR)
[9]State Key Laboratory of Multimodal Artificial Intelligence Systems (MAIS)
lue.fan@ia.ac.cn, zhaoxiang.zhang@ia.ac.cn
[*]Equal contribution. [†]Corresponding authors. [‡]Project lead.
Project page: https://mobilegym.github.io

Abstract

We present MobileGym, a **browser-hosted, lightweight, fully controllable simulation platform for everyday mobile use**, targeting *interaction fidelity* without replicating proprietary backends. It enables two capabilities previously out of reach for everyday apps: **verifiable outcome signals** through deterministic state-based judging over structured JSON state, and **scalable online RL** through low-cost parallel rollouts. The full environment state is captured, configured, forked, and compared as structured JSON, and a single server can host hundreds of parallel instances (∼400 MB each, ∼3 s cold start). A layered state model and a declarative task-definition framework keep state programmability and task creation practical at scale, and a single programmatic judging mechanism delivers both deterministic evaluation verdicts and dense RL rewards. The accompanying MobileGym-Bench provides 416 parameterized task templates (256 test + 160 train) over 28 apps, with deterministic judges and a structured AnswerSheet protocol that avoids free-text matching failures. In a Sim-to-Real case study, GRPO on Qwen3-VL-4B-Instruct gains +12.8 pt on the 256-task test set, and on a 59-task real-device signal subset, real-device execution retains 95.1% of the simulation-side training gain.

1 Introduction

Mobile GUI agents have advanced rapidly in operating smartphones from screenshots and natural-language instructions (Qin et al., 2025; Liu et al., 2024; Venus-Team et al., 2026; Xu et al., 2026; Xiao et al., 2025), yet current evaluation and training environments remain divided by a basic trade-off. Emulator-based environments such as AndroidWorld and AndroidLab (Rawles et al., 2025; Xu et al., 2025) offer repeatable evaluation but mainly cover system utilities and simple open-source apps, and scaling to online training requires many heavyweight emulator instances. Real-device benchmarks such as MobileBench-OL (Wu et al., 2026a) reach everyday apps, but live accounts, backend state, app-version drift, real-world consequences, and the cost of maintaining many devices and accounts make episodes difficult to control, reproduce, and parallelize. Neither route provides the combination needed for progress. First, environments need **verifiable outcome signals**, so benchmark verdicts and RL rewards are deterministic and grounded in actual task state rather than unreliable VLM judgments. Sec-

Figure 1: **Example screens from MobileGym.** Annotated launcher and messaging screens showing MobileGym's configurable and sandboxed capabilities.

Proceedings of the 2026 Conference on Empirical Methods in Natural Language Processing, pages 23223–23247
October 24-29, 2026 ©2026 Association for Computational Linguistics

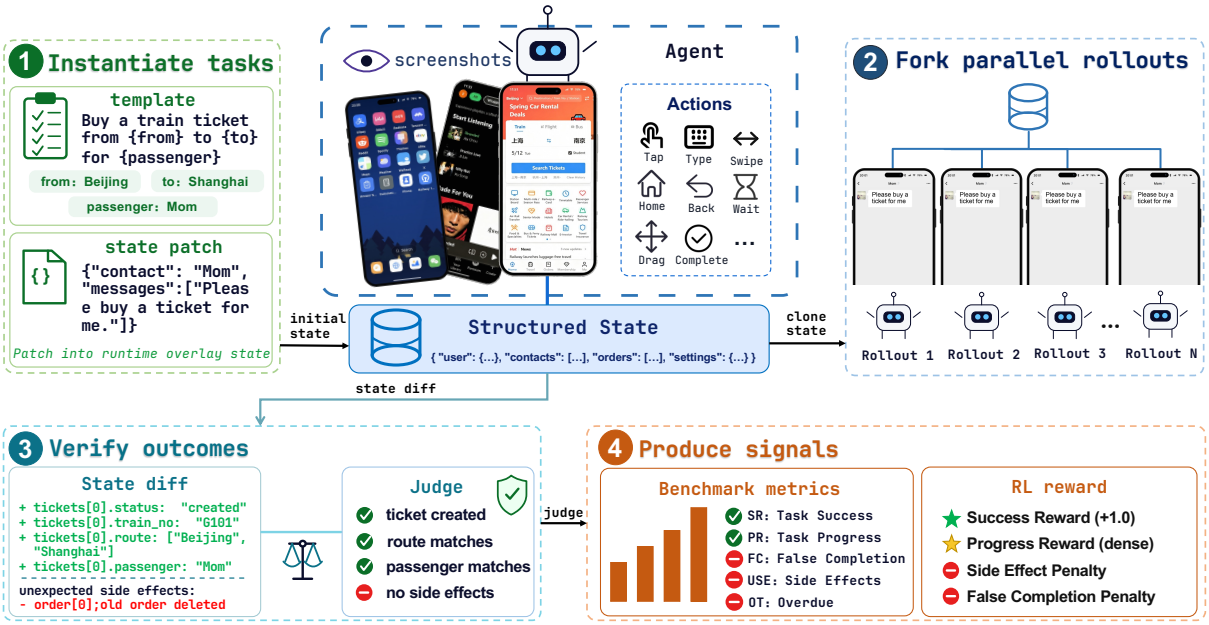


Figure 2: **End-to-end workflow of MobileGym.** A structured state supports task instantiation, parallel rollout forking, and state-diff verification. The resulting judgments are then converted into benchmark metrics and RL rewards.

ond, they need **scalable online training**: online RL has become an important capability driver for GUI agents (Venus-Team et al., 2026; Wang et al., 2025; Zhang et al., 2025), while offline trajectories struggle to cover dynamic GUI variations (Zhou et al., 2025).

The barriers are inherent to how everyday apps work. Everyday-app state is **unreadable**: internal state such as balances and orders is difficult to inspect through adb and accessibility trees, while VLM judges are intrinsically unreliable and further constrained by discrete screenshots that provide only partial evidence. It is **unwritable**: reproducible evaluation and online RL require resetting to known initial conditions, yet task-relevant state is split across proprietary storage, caches, and remote services, making desired states difficult to configure or restore. It is **unforkable**: large-scale online training benefits from parallel rollouts, and group-based methods such as GRPO further require multiple rollouts from identical initial states, yet live apps provide neither cheap replication nor state forking. Finally, many actions are **irreversible**, risking real messages, real transfers, or permanent account changes. These constraints make everyday apps structurally resistant to reproducible experimentation, even though they are natural targets for mobile-agent research. Scalability poses a further challenge: even for the apps emulators do support, each instance requires gigabytes of RAM, making large-scale parallel rollouts impractical on commodity hardware — let alone for everyday apps that are resource-intensive or restrict emulator execution.

Yet GUI agents observe only screenshots and act through discrete actions, so a lightweight simulator with fully programmable state can be sufficient — it only needs *interaction fidelity*, producing realistic screens in response to agent actions. We introduce MobileGym, a browser-hosted Android-like simulation environment built on this principle. App data, OS state, and device context are represented as structured JSON, and the same mechanism makes state readable for deterministic outcome checking, writable for configuration and reset, forkable for parallel rollouts, and fully sandboxed for high-consequence actions. Agents observe only screenshots, while researchers retain full programmatic control. Each browser instance uses roughly 400 MB of RAM and cold-starts in about 3 s, enabling hundreds of parallel instances on a single server. For query tasks, a structured AnswerSheet protocol replaces brittle free-text matching with typed, GUI-submitted fields. Figure 1 shows example simulated screens, and Figure 2 shows the end-to-end pipeline.

Our main contributions are:

- **The MobileGym platform** (§3): a

lightweight, browser-hosted Android-like simulation environment, including 12 everyday apps covering the major categories of daily mobile use and 16 system apps. Its modular app architecture and declarative task framework support easy extension, and a single machine can host hundreds of parallel instances.

- **Programmable state and verification mechanisms** (§3.2, §4.2): full-environment state represented as structured JSON that supports deterministic judging, snapshot-based rollout forking, side-effect detection, and a typed AnswerSheet protocol that avoids free-text matching failures.

- **MobileGym-Bench** (§4): 416 parameterized task templates (256 test + 160 train) covering major categories of everyday mobile use, with deterministic judges, empirically calibrated difficulty strata, and diagnostic metrics.

- **Empirical validation** (§5): benchmark results across 9 agents (9.4%–58.8% SR), a GRPO training study gaining +12.8 pt on the 256-task test set, a real-device study retaining 95.1% of the simulated gain on a real-device signal subset, and a VLM-judge audit showing 10.2% misjudgment.

2 Related Work

Real-device and emulator route. Existing mobile GUI agent environments run tasks on a heavyweight Android emulator or physical device and judge them externally, either through programmatic queries to interfaces such as `adb`, accessibility trees, UI-tree, or XPath rules, or through VLM-based screenshot judges. On system utilities and open-source apps, deterministic verification is feasible: AndroidWorld (Rawles et al., 2025) judges 116 emulator tasks through `adb`, AndroidLab (Xu et al., 2025) adds UI-tree matching with an LLM verifier for query-answer subtasks, and MobileWorld (Kong et al., 2026) queries backend databases directly. A3 (Chai et al., 2026) targets 20 mainstream Google Play apps via Appium and adopts MLLM-as-judge to handle their dynamic content, trading determinism for coverage. MobileBench-OL (Wu et al., 2026a) runs 1080 tasks across 80 Chinese-language everyday apps on physical phones, the closest prior attempt at real everyday-app evaluation. Its XPath rules are brittle to unexpected popups and to minor app or backend updates, and the physical-device setup does not support parallel rollouts. All inherit the constraints discussed in §1. Table 1 compares representative environments.

Other mobile GUI benchmarks. SPA-Bench (Chen et al., 2025), Mobile-Bench (Deng et al., 2024), ProBench (Yang et al., 2026), MVISU-Bench (Huang et al., 2025), UI-NEXUS (Guo et al., 2025), and ColorBench (Song et al., 2025) contribute task suites along axes orthogonal to environment infrastructure, and inform MobileGym-Bench's taxonomy design (§4.1).

Synthesis and trajectory-replay environments. GUI-Genesis (Cao et al., 2026) reconstructs real apps as lightweight web environments from interaction trajectories with code-native rewards, but each environment covers only a single task trajectory. UISim (Xiang et al., 2025) and ViMo (Luo et al., 2026) adopt image-generation approaches. However, visual prediction errors can accumulate over long horizons, making these environments less suitable for RL with deterministic state transitions. OpenApps (Ullrich et al., 2026) focuses on reliability measurement with 6 FastHTML applications and shares the lightweight design philosophy of MobileGym, while pursuing a different goal.

Verifiable environments in other domains. Beyond mobile, verifiable interactive environments have been built in the web domain (WebShop (Yao et al., 2022), WebArena (Zhou et al., 2024), VisualWebArena (Koh et al., 2024), WebGym (Bai et al., 2026), AutoWebWorld (Wu et al., 2026b), InfiniteWeb (Zhang et al., 2026)), the desktop OS domain (OSWorld (Xie et al., 2024), macOSWorld (Yang et al., 2025)), and over simulated Python APIs (AppWorld (Trivedi et al., 2024)).

RL-based GUI agent training. DigiRL (Bai et al., 2024) demonstrates a substantial advantage of online RL over SFT for device control. UI-TARS-2 (Wang et al., 2025) deploys thousands of VMs to enable large-scale RL rollouts. UI-Venus-1.5 (Venus-Team et al., 2026) introduces full-trajectory online RL with model fusion and achieves 77.6% SOTA on AndroidWorld. GUI-Owl-1.5 (Xu et al., 2026) proposes the MRPO algorithm to address conflicts in multi-platform RL training. MobileGUI-RL (Shi et al.,

	AndroidWorld	AndroidLab	MobileWorld	MobileBench-OL	MobileGym
Runtime	Emulator	Emulator	Emulator	Real device	Browser
App types	System + open-source	System + open-source	System + surrogates	Real everyday apps	Simulated everyday + system
Everyday app coverage	✗	✗	Surrogates	✓ (real)	✓ (simulated)
Apps / task units	20 / 116 templates	9 / 138 instances	20 / 201 tasks	80 / 1080 instances	28 / 416 templates
Verification	adb programmatic	UI-tree + LLM	SQL + hooks	XPath rules	State-based programmatic
Full environment state comparison	✗	✗	✗	✗	✓
Snapshot & restore	App-data snapshot	AVD snapshot	AVD snapshot	✗	JSON (ms-level)
State customization	Limited	Limited	Partial (SQL)	✗	Full (JSON)
Online RL-ready	Limited	Limited	Limited	✗	✓
Memory per instance	$\sim$4.5 GB^{M}	$\sim$6 GB^{R}	$\geq$4.5 GB^{L}	N/A	$\sim$400 MB^{M}
Disk footprint	$\sim$20 GB^{M}	$\sim$9 GB^{R}	$\geq$20 GB^{L}	N/A	$\sim$50 MB^{M}
Cold start	$\sim$78 s^{M}	—	—	N/A	$\sim$3 s^{M}
Parameterized tasks	✓	✗	✗	✗	✓
Sim-to-Real validation	N/A	N/A	N/A	N/A	Validated

Table 1: Comparison of mobile GUI agent benchmarks and infrastructures. Task-unit labels follow each benchmark's native counting unit. AndroidLab additionally releases 10.5k offline SFT trajectories, not counted here. *Validated* denotes the real-device transfer study in §5.2, where 95.1% of the simulation-side training gain on the 59-task signal subset is retained. Resource figures are ^{M}measured by us, ^{R}repository-reported configurations, or ^{L}lower bounds inherited from a shared infrastructure baseline; measurement details are in Appendix M.

2025), Mobile-R1 (Gu et al., 2026), UI-R1 (Lu et al., 2026), GUI-R1 (Luo et al., 2025) explore curriculum-style and R1-style training.

3 The MobileGym Platform

MobileGym is a browser-hosted Android-like simulation environment. Its app data, OS settings, and device properties are represented as explicit structured state, which the benchmark layer can configure, reset, snapshot, fork, and compare (Figure 3).

3.1 System Design

Interaction fidelity target. MobileGym does not aim to reproduce real everyday app backends or pixel-level Android internals. Its target is the interaction surface available to GUI agents: visual screens, touch and typing responses, navigation, cross-app handoffs, and task-relevant state transitions. As summarized in Figure 3, this requires Android-like runtime mechanisms such as task stacks, keyboard, notification, and permission flows, shared resources, intent routing, content sharing, and back-key dispatch. These mechanisms are implemented in the browser over structured local state, making the same interaction semantics readable, writable, and forkable for evaluation and RL. Implementation details are in Appendix A.

Layered state model. The environment separates large, mostly read-only *world data*, compact per-environment *runtime state*, and OS runtime state. World data contains public entities such as posts and products, while runtime state contains data that can be changed by the agent, such as the current user's profile or app settings. Agent operations write only to runtime state, and views are produced by overlaying this layer on the read-only world data. Only runtime state is exposed for configuration, reset, judging, and comparison, keeping snapshots small and stable while preserving all agent-induced changes for full-environment state comparison.

Declarative navigation specification. The UI navigation of every app is modeled as a declarative finite-state machine, built at development time into a per-app specification file. The same file drives runtime navigation and static analysis, including task-trajectory enumeration, and auto-generation of new tasks. The formal definition and guard syntax are provided in Appendix B.

Interface and extensibility. The Benchmark layer maps agent outputs to a unified 17-action abstraction (Appendix C), executes actions through Playwright with coordinates normalized to $[0, 1000]$, and returns only screenshots. On the app side, MobileGym provides a repeatable module architecture that separates UI pages, app-local runtime state, declarative navigation, replaceable default data, and world data, allowing new apps and features to reuse the shared OS lifecycle, reset, snapshot, rollout, and judging interfaces (Appendix A).

3.2 State Programmability

Verifiable outcome signals. Task success is judged by **programmatic state verification**: each task has a deterministic judge that inspects environment state. This provides deterministic, fine-grained outcome signals without unreliable VLM

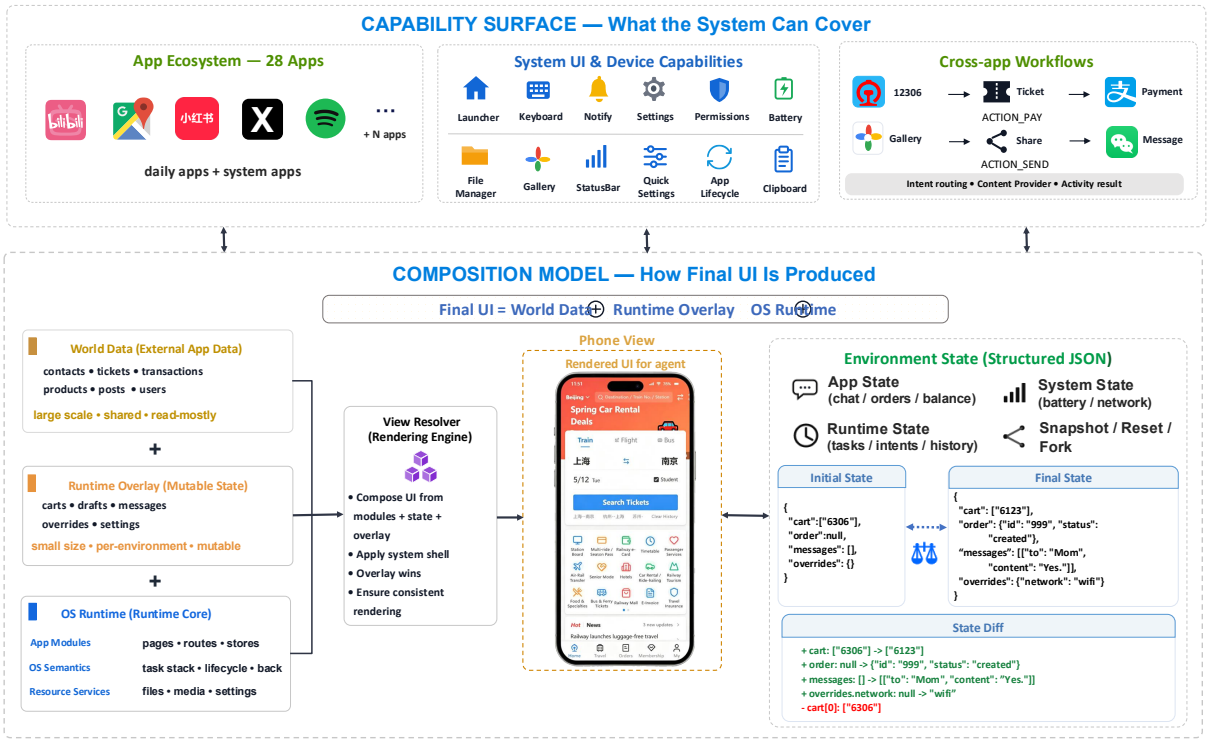


Figure 3: **System capabilities and state model of MobileGym.** App views are produced by composing read-mostly *World Data*, a per-environment *Runtime Overlay*, and the *OS Runtime*. The resulting structured environment state supports snapshot/reset/fork and deterministic state-diff judging.

judgments.

State serialization and multi-instance replication. The full environment state can be serialized as structured JSON and restored on demand, enabling exact reset and forking from any snapshot, supporting RL methods such as GRPO. For irreversible operations (transfers, deactivation, deletions, etc.), the consequence-free simulator allows full restoration after each trajectory.

Full environment state comparison. The fully structured state enables full-environment state comparison between an episode's initial and terminal states, reporting any mutation outside the task's expected outcome as an *unexpected side effect*. For personal mobile agents, this distinction is critical: an agent may complete the requested goal while, for example, sending an unintended message. This mechanism defines the **Unexpected Side Effects** metric (§4.3). Existing programmatic mobile benchmarks do not provide this environment-wide signal, and VLM judges can only approximate it from screenshots without deterministic guarantees.

4 The MobileGym-Bench

MobileGym-Bench is a suite of 416 parameterized task templates (256 test + 160 train, strictly disjoint) built on top of the MobileGym platform. It covers major categories of everyday mobile use. Detailed information about the 28 apps and representative task examples is listed in Appendix D.

4.1 Task Taxonomy

Prior task taxonomies often couple unrelated dimensions, such as mixing app count with subtask count (Deng et al., 2024). We factor the task space along four orthogonal axes:

- **Scope** — how many apps a task involves. S1: single-app, S2: two-app, S3: three or more.
- **Objective** — what the task asks for. Operate: state-changing actions, query: information retrieval, hybrid: both.
- **Composition** — how subtasks are structured. Atomic: a single action, sequential: an ordered chain, transfer: cross-app handoff, deep-dive: multi-step drill-down.
- **Difficulty** — how hard the task is for current models. L1: easy, L2: moderate, L3:

hard, L4: very hard. Calibrated post-hoc using eight reference models, details in §4.4.

Each task is additionally annotated with 1–4 capability tags from a 13-tag vocabulary. The full taxonomy and tag definitions are provided in Appendix E.

4.2 Task Design

Two design choices shape the task suite: parameterized instantiation for diversity, and AnswerSheet fields for query-task judging reliability.

Parameterized task instantiation. The 416 entries in MobileGym-Bench are **templates**, not fixed instances. Each template is instantiated at runtime through three sources of variation: **(i) instruction variation**, where semantically equivalent goal phrasings are sampled; **(ii) parameter sampling**, where slot values are drawn from curated sets, numeric ranges, or the current environment state; and **(iii) environment configuration**, where app state such as contacts or order history is set through shared base data or per-task injections before rollout. Together, these variations reduce memorization of fixed instances and expand task diversity without requiring each instance to be authored separately. Across finite parameter ranges, they yield over 27,000 distinct task instances, not counting templates with continuous ranges that contribute unbounded additional instances.

The AnswerSheet protocol. Existing mobile benchmarks often judge free-text query answers with string-similarity or substring heuristics (Rawles et al., 2025; Kong et al., 2026), which can reject equivalent phrasings or falsely accept answers that leak reasoning text containing the gold answer. MobileGym instead moves answer submission into the GUI: query tasks end with the agent filling an **AnswerSheet** form whose fields declare types and show format hints (Figure 4). This preserves a natural form-filling interaction for GUI-specialized agents, while the submitted typed state is checked by type-specific matchers such as exact text, numeric tolerance, format, or choice checks. Details are in Appendix F.

4.3 Evaluation Protocol

We report success, progress, termination, and side-effect diagnostics under fixed step budgets.

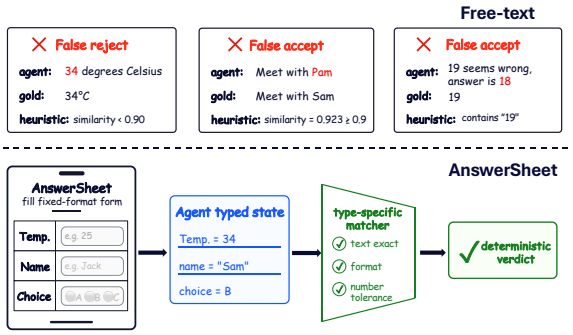


Figure 4: **AnswerSheet protocol.** Free-text heuristics can reject equivalent answers or accept leaked reasoning that contains the gold answer. AnswerSheet instead uses GUI form filling and type-specific checks over typed fields.

Metrics. **Success Rate (SR)**, the fraction of tasks judged successful, is the primary metric. Diagnostics include **Progress Rate (PR)**, the fraction of subtasks passed; **False Complete (FC)**, episodes where the agent declares completion without success; **Unexpected Side Effects (USE)**, episodes with unexpected state changes; and **Overdue Termination (OT)**, episodes where the agent reaches the goal but continues until truncation.

Execution setup. The simulator is reset before each task, and agents observe only screenshots. Each task is assigned one of four step budgets (15, 30, 45, or 60), manually verified to comfortably exceed its optimal completion length. Tasks with AnswerSheet receive an additional 15-step budget.

4.4 Model-Calibrated Difficulty Strata

Motivated by benchmark-curation precedents such as BBH, which identifies hard tasks using prior model and human-rater performance (Suzgun et al., 2023), four difficulty levels are assigned by **post-hoc empirical calibration**. We evaluate eight reference models[1] on the test set and stratify tasks by mean SR and PR: L1 (SR$\geq$75%, PR$\geq$75%, n=20), L2 (remaining tasks with SR$\geq$25%, PR$\geq$50%, n=73), L3 (remaining tasks with SR$>$0, PR$\geq$25%, n=83), and L4 (otherwise, n=80). These are diagnostic strata rather than intrinsic labels, and the calibration excludes Qwen3-VL-4B-Instruct and its fine-tuned variants used in §5.2. A reference-model robustness check is reported in Appendix I.

[1]Gemini 3.1 Pro, Doubao-Seed-2.0-Pro, Qwen3.6-Plus, AutoGLM-Phone-9B, UI-TARS-1.5-7B, UI-Venus-1.5-8B, GUI-Owl-1.5-8B-Think, Step-GUI-4B.

5 Experiments

We evaluate 9 agents on MobileGym-Bench (Table 2). Open-source models use 4 trials with resampled parameters; proprietary models use one due to API cost, with one additional run for Gemini 3.1 Pro, the strongest model, to estimate variation.

5.1 Benchmark Results

Two observations stand out from Table 2. Additional experimental results are in Appendix H.

Difficulty stratification. SR decreases monotonically from L1 to L4 for all 9 models, while overall SR spans 9.4%–58.8%, giving a 6× performance range without top saturation or bottom floor effects. L1 already separates proprietary and open-source agents, and L4 acts as the frontier discriminator: only Gemini 3.1 Pro retains meaningful performance at 21.9%, while all other proprietary models reach at most 6.2% and all open-source GUI specialists at most 1.9%.

Unexpected side effects. USE captures unintended agent operations that modify state unrelated to the task. It does not simply decrease with model capability: across the 9 models it ranges from 4.7% to 14.5%, and even open-source GUI specialists with similar SRs (12.9–15.4%) differ nearly 2× in USE (7.6–14.1%). This diagnostic is enabled by MobileGym's full-environment state comparison. Screenshot or UI-tree judges cannot reliably expose off-target changes hidden in app-internal or backend state.

5.2 Sim-to-Real Transfer

We view this real-device experiment as an existence proof that training in MobileGym can produce behavior that survives real-device execution, not as a comprehensive sim-to-real study.

We fine-tune Qwen3-VL-4B-Instruct with GRPO (Shao et al., 2024) on MobileGym's 160-task train set for 10 steps, using a single node with 3 RTX Pro 6000s and 96 parallel environment instances. Key hyperparameters are lr $= 10^{-6}$, group size k=8, batch size bs=12, KL 0.01, DAPO (Yu et al., 2025)-style asymmetric clip-higher (0.2/0.28). The reward is a PR-shaped dense signal, with multiplicative penalties for AnswerSheet error, side effects, false completion, and overdue/post-success abort. Details are provided in Appendix G.

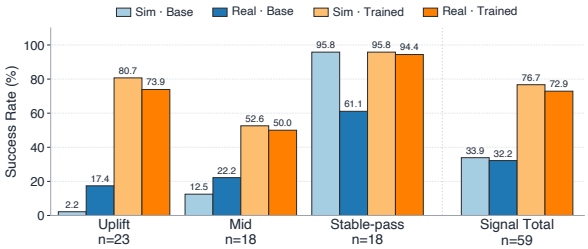


Figure 5: **Sim-to-Real transfer of GRPO training gains.** Per-bucket Success Rate on the 59-task signal-bucket subset and the overall Signal Total. In the legend, Sim/Real denotes the evaluation environment and Base/Trained denotes before/after GRPO. Sim columns are 4-seed averages, Real columns are pass@1 and all manually audited (Appendix J).

Training gains on the simulation side. Training raises overall SR from 9.4% to 22.2% (+**12.8 pt**) on the 256-task MobileGym-Bench test set. Broken down by difficulty, SR changes from 71.2% to 92.5% on L1, 12.3% to 37.7% on L2, 0.6% to 11.7% on L3, and 0.3% to 1.2% on L4. The lift is largest on L2 and nearly flat on L4, suggesting that training is most effective where the base model already exhibits moderate capability, while the hardest tasks remain capacity-limited. The trained 4B model surpasses the 9B AutoGLM-Phone-9B on L1–L3, while both remain near zero on L4.

Real-device evaluation design. We evaluate on a Redmi Note 12 Turbo (1080 × 2400). We stratify the 256-task test set by the base/trained models' pass counts over four simulator rollouts: Uplift (base $\leq$1, trained $\geq$3; 26 tasks), Stable-pass (both $\geq$3; 21 tasks), Mid (all remaining cases; 20 tasks), Regression (base $\geq$3, trained $\leq$1; 0 tasks), and Stable-fail (both $\leq$1; 189 tasks). The three signal buckets (Uplift, Mid, Stable-pass) contain 67 tasks, of which 59 can be safely and equivalently run on the real device after excluding 8 tasks involving unreproducible account state or irreversible operations. Running all 189 Stable-fail tasks on a single real device serially would be costly and manual state restoration, while these tasks, by definition, exhibit no simulation-side training gain to transfer. We therefore randomly sample 15 as a negative-control check. The real-device setting differs from simulation in UI details, app data, real-app variability, and task entities such as contacts or POIs. Details in Appendix H.5.

Results. On the 59-task signal-bucket subset, training raises the real-device pass rate from

Model	Overall (%) SR	PR	Difficulty SR (%) L1 (20)	L2 (73)	L3 (83)	L4 (80)	Diagnostics (%) FC	OT	USE
Proprietary models									
Gemini 3.1 Pro	58.8 ±1.4	72.1	97.5	83.6	63.3	21.9	34.0	0.2	5.5
Doubao-Seed-2.0-Pro	52.0†	63.6	100.0	93.2	48.2	6.2	33.6	0.4	4.7
Qwen3.6-Plus	45.7†	59.2	100.0	78.1	44.6	3.8	34.0	0.4	14.5
Open-source GUI-specialized models									
AutoGLM-Phone-9B	20.0 ±1.3	35.3	86.2	33.6	9.6	1.9	39.6	0.6	12.6
UI-TARS-1.5-7B	13.8 ±1.7	26.3	77.5	21.9	3.0	1.6	38.6	0.2	11.0
UI-Venus-1.5-8B	15.4 ±2.4	28.3	85.0	21.9	6.0	1.9	22.9	0.5	7.7
GUI-Owl-1.5-8B-Think	15.1 ±0.9	28.8	76.2	26.0	4.2	1.2	30.4	0.9	14.1
Step-GUI-4B	12.9 ±1.1	25.7	83.8	17.8	2.4	1.6	37.0	0.8	7.6
Open-source generalist models									
Qwen3-VL-4B-Instruct	9.4 ±0.6	20.1	71.2	12.3	0.6	0.3	15.9	0.4	10.0

Table 2: Main results on the MobileGym-Bench test set (256 tasks). Overall reports Success Rate (SR) and Progress Rate (PR); Difficulty SR reports SR within calibrated difficulty strata L1–L4, with task counts in parentheses; Diagnostics report False Complete (FC), Overdue Termination (OT), and Unexpected Side Effects (USE). ± denotes standard deviation across trials; † marks single-run results. See §4.3 for metrics and Appendix H for details.

32.2% to 72.9% (+**40.7 pt**), closely matching the simulation-side increase from 33.9% to 76.7% (+**42.8 pt**). This corresponds to **95.1% retained gain**. The absolute sim–real gaps are small for both the base model (1.7 pt) and the trained model (3.8 pt) (Figure 5). The 15 randomly sampled Stable-fail tasks yield 0/15 success for both models on the real device, consistent with simulation. Because the real-device environment and task entities differ from the simulator, the lift more plausibly reflects transferable policies than memorization.

Trajectory-length check. Successful trajectories on `operate` tasks have similar lengths in simulation and on device: 5.00 vs. 6.03 steps for the base model and 10.08 vs. 12.20 for the trained model. We exclude `query/hybrid` tasks because AnswerSheet adds simulator-only steps. Details in Appendix H.6.

Failure-recovery example. In a real-device Reddit post-creation task, the selected community required a flair before the Post button could be enabled. The base model looped on the disabled button until the step budget was exhausted, while the trained model identified the missing flair requirement and succeeded in 22 steps (Appendix K).

VLM judge error analysis. Manual review of all 118 signal-bucket real-device trajectories identifies 12 errors for Qwen3.6-Plus (5/59 base, 7/59 trained; 10.2% overall). Re-judging the same trajectories with GPT-5.4 (OpenAI, 2026) also yields 12/118 errors, although on a partially different subset. These results suggest that VLM-judge errors are not specific to a single judge model, while programmatic state verification avoids this failure mode. Detailed counts are in Appendix J.

5.3 Efficiency Analysis

MobileGym runs in the browser with roughly 1/10 the memory and under 1/100 the disk footprint of emulator-based setups (Table 1). In our measurements, 256 parallel instances on one server used <10% CPU and ∼100 GB RAM, completing a full 256-task benchmark evaluation in about 6 minutes. By contrast, MAI-UI (Zhou et al., 2025) reports requiring 10 bare-metal cloud servers (960 vCPUs, 3,840 GB RAM total) to reach 512 parallel Android-emulator instances for online RL. This single-node parallel capacity makes concurrent online RL (§5.2) feasible without dedicated cluster infrastructure. Appendix N quantifies the API cost of using VLM judges instead of code-level judging.

6 Conclusion

MobileGym turns everyday mobile use into a fully controllable simulation platform for GUI agent research. By targeting interaction fidelity rather than replicating proprietary backends, MobileGym makes everyday-app state readable for deterministic verification, writable for reset and configuration, forkable for parallel online RL, and consequence-free for high-risk operations. The MobileGym-Bench suite operationalizes this environment with 416 parameterized task templates, calibrated difficulty strata,

structured AnswerSheet-based evaluation, and diagnostic metrics including unexpected side effects. Experiments across 9 agents show substantial headroom on everyday mobile tasks, and the Sim-to-Real study shows that most of the simulation-side training gain transfers to real-device execution. The same controllable infrastructure could also be used for safety-alignment research, robustness testing, and training-data generation (Appendix L). More broadly, MobileGym shows that interaction-fidelity simulation can make everyday mobile tasks available for reproducible research and scalable training, without relying on real accounts, device farms, or proprietary backends.

Limitations

Visual appearance modeling. Observed visual differences between MobileGym and the corresponding real apps mainly include subtle layout details, animations, and certain app-specific icons. The Sim-to-Real experiment (§5.2) provides one quantitative datapoint that this level of visual similarity can support behavioral policy transfer. Tasks that depend heavily on recognizing exact app-specific icons may still be affected during transfer.

Backend and dynamic-content modeling. MobileGym models agent-facing interaction semantics rather than real service backends. Server-driven content such as ads, pop-ups, recommendation feeds, and real-time messages is represented as controllable JSON state, which favors deterministic reset, reproducible evaluation, and stable RL reward signals. This design does not capture backend-only or stochastic phenomena such as live recommendation dynamics, fraud checks, latency spikes, or server-side policy changes unless they are explicitly modeled as controllable state. Controllable dynamic-content injection is architecturally supported and left for future study.

Functional coverage of simulated apps. Each simulated app implements the main everyday-use scenarios of its real counterpart rather than the full feature surface. Less common features remain out of scope. Expanding within-app coverage is future work.

Ethical Considerations

MobileGym is a **fully sandboxed** research infrastructure. All simulation of commercial apps is disconnected from any real service, real account, real funds, or personal data.

Legality of commercial-app simulation. The commercial apps reproduced in MobileGym are used only for academic research and model evaluation. Their trademarks, brand names, and visual elements remain the property of their respective owners. MobileGym does not reuse or distribute any official code or client components. The simulator UI is independently implemented with LLM-assisted programming and, due to the limits of model-based reproduction, differs from the real apps in pixel-level visual details. The environment runs in the browser, is offline, and never touches real accounts or funds. We do not claim any commercial or derivative use.

Double-edged nature of evaluating high-risk operation capabilities. The high-risk subset (Appendix Table 10) consists of 14 tasks: 7 standalone payment operations (Payment) and 7 high-consequence tasks drawn from Test256 (Test256-Risk, including account deactivation, large transfers, bulk deletions, etc.). Gemini 3.1 Pro reaches 64.3% on Payment and 71.4% on Test256-Risk, while smaller open-source GUI specialists remain at $\leq$10.7% on Payment. Trajectory inspection finds no evidence of explicit refusal in either tier: frontier models attempt the operation and largely succeed, whereas open-source models attempt but fail. **We explicitly state that this is a report of execution capability, not an endorsement of such autonomous operations.** Under the current training paradigm, "execution capability" and "operational caution" are not decoupled. Frontier models, when instructed, execute irreversible operations with high success and no intrinsic caution gating. We argue that capability evaluation must be paired with safety alignment. The ability of MobileGym to simulate irreversible operations provides a no-real-risk testing infrastructure for follow-up safety-alignment research, which is an important part of its value.

Misuse risk and mitigation. Any GUI-agent training infrastructure could potentially be used to automate malicious behavior. MobileGym is, by design, a research tool for capability evaluation and safety research, not a production deployment. We encourage using MobileGym for defensive research as well—safety alignment, prompt-injection robustness, and refusal training.

Societal impact. The safe-simulation properties of MobileGym—zero-consequence operations, one-click reset, built-in difficulty levels—naturally make it suitable for **digital-literacy education**. Learners can repeatedly practice tasks such as contact lookup, mobile payment, and ticket booking in a fully simulated environment without any real consequences. We encourage the community to explore positive social applications of MobileGym in digital inclusion, customer-service training, and AI-safety education.

Acknowledgments

This work was supported by the Jiangsu Science and Technology Major Project (BG2025035) and in part by the National Natural Science Foundation of China (Nos. 62320106010, 62506356, and 62536010). This work was also supported by the Beijing Natural Science Foundation (Nos. L257004 and L257015).

We thank the reviewers and area chairs for their constructive feedback. Cursor, Claude Code, and Codex supported code implementation.

References

Hao Bai, Alexey Taymanov, Tong Zhang, Aviral Kumar, and Spencer Whitehead. 2026. Webgym: Scaling training environments for visual web agents with realistic tasks. *arXiv preprint arXiv:2601.02439*.

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. In *Advances in Neural Information Processing Systems 37 (NeurIPS)*.

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, Wenbin Ge, Zhifang Guo, Qidong Huang, Jie Huang, Fei Huang, Binyuan Hui, Shutong Jiang, Zhaohai Li, Mingsheng Li, and 45 others. 2025. Qwen3-vl technical report. *arXiv preprint arXiv:2511.21631*.

ByteDance Seed Team. 2026. Seed2.0 model card. `https://seed.bytedance.com/en/seed2`.

Yuan Cao, Dezhi Ran, Mengzhou Wu, Yuzhe Guo, Xin Chen, Ang Li, Gang Cao, Zhi Gong, Hao Yu, Linyi Li, Wei Yang, and Tao Xie. 2026. Gui-genesis: Automated synthesis of efficient environments with verifiable rewards for gui agent post-training. *arXiv preprint arXiv:2602.14093*.

Yuxiang Chai, Shunye Tang, Han Xiao, Weifeng Lin, Hanhao Li, Jiayu Zhang, Liang Liu, Pengxiang Zhao, Guangyi Liu, Guozhi Wang, Shuai Ren, Rongduo Han, Haining Zhang, Siyuan Huang, and Hongsheng Li. 2026. A3: Android agent arena for mobile GUI agents with essential-state procedural evaluation. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 3774–3789, San Diego, California, United States. Association for Computational Linguistics.

Jingxuan Chen, Derek Yuen, Bin Xie, Yuhao Yang, Gongwei Chen, Zhihao Wu, Yixing Li, Xurui Zhou, Weiwen Liu, Shuai Wang, Kaiwen Zhou, Rui Shao, Liqiang Nie, Yasheng Wang, Jianye Hao, Jun Wang, and Kun Shao. 2025. Spa-bench: A comprehensive benchmark for smartphone agent evaluation. In *The Thirteenth International Conference on Learning Representations (ICLR)*.

Shihan Deng, Weikai Xu, Hongda Sun, Wei Liu, Tao Tan, Jianfeng Liu, Ang Li, Jian Luan, Bin Wang, Rui Yan, and Shuo Shang. 2024. Mobile-bench: An evaluation benchmark for llm-based mobile agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 8813–8831.

Google DeepMind. 2026. Gemini 3.1 pro model card. `https://deepmind.google/models/model-cards/gemini-3-1-pro/`.

Jihao Gu, Qihang Ai, Yingyao Wang, Pi Bu, Jingxuan Xing, Yue Cao, Zekun Zhu, Wei Jiang, Ziming Wang, Yingxiu Zhao, Ming-Liang Zhang, Jun Song, Yuning Jiang, and Bo Zheng. 2026. Mobile-r1: Towards interactive capability for VLM-based mobile agent via systematic training. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 30802–30820, San Diego, California, United States. Association for Computational Linguistics.

Yuan Guo, Tingjia Miao, Zheng Wu, Pengzhou Cheng, Ming Zhou, and Zhuosheng Zhang. 2025. Atomic-to-compositional generalization for mobile agents with a new benchmark and scheduling system. *arXiv preprint arXiv:2506.08972*.

Zeyu Huang, Juyuan Wang, Longfeng Chen, Boyi Xiao, Leng Cai, Yawen Zeng, and Jin Xu. 2025. Mvisu-bench: Benchmarking mobile agents for real-world tasks by multi-app, vague, interactive, single-app and unethical instructions. *arXiv preprint arXiv:2508.09057*.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Chong Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan Salakhutdinov, and Daniel Fried. 2024. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 881–905.

Quyu Kong, Xu Zhang, Zhenyu Yang, Nolan Gao, Chen Liu, Panrong Tong, Chenglin Cai, Hanzhang Zhou,

Jianan Zhang, Liangyu Chen, Zhidan Liu, Steven Hoi, and Yue Wang. 2026. MobileWorld: Benchmarking autonomous mobile agents in agent-user interactive and MCP-augmented environments. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6142–6167, San Diego, California, United States. Association for Computational Linguistics.

Xiao Liu, Bo Qin, Dongzhu Liang, Guang Dong, Hanyu Lai, Hanchen Zhang, Hanlin Zhao, Iat Long Iong, Jiadai Sun, Jiaqi Wang, Junjie Gao, Junjun Shan, Kangning Liu, Shudan Zhang, Shuntian Yao, Siyi Cheng, Wentao Yao, Wenyi Zhao, Xinghan Liu, and 11 others. 2024. Autoglm: Autonomous foundation agents for guis. *arXiv preprint arXiv:2411.00820*.

Zhengxi Lu, Yuxiang Chai, Yaxuan Guo, Xi Yin, Liang Liu, Hao Wang, Han Xiao, Shuai Ren, Pengxiang Zhao, Guangyi Liu, Guanjing Xiong, and Hongsheng Li. 2026. Ui-r1: Enhancing efficient action prediction of gui agents by reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 17608–17616.

Dezhao Luo, Bohan Tang, Kang Li, Georgios Papoudakis, Jifei Song, Shaogang Gong, Jianye Hao, Jun Wang, and Kun Shao. 2026. ViMo: A generative visual gui world model for app agents. In *The Fourteenth International Conference on Learning Representations*.

Run Luo, Lu Wang, Wanwei He, Longze Chen, Jiaming Li, Min Yang, and Xiaobo Xia. 2025. Gui-r1: A generalist r1-style vision-language action model for gui agents. *arXiv preprint arXiv:2504.10458*.

OpenAI. 2026. Gpt-5.4 thinking system card. `https://openai.com/index/gpt-5-4-thinking-system-card/`.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, Wanjun Zhong, Kuanye Li, Jiale Yang, Yu Miao, Woyu Lin, Longxiang Liu, Xu Jiang, Qianli Ma, Jingyu Li, and 16 others. 2025. Ui-tars: Pioneering automated gui interaction with native agents. *arXiv preprint arXiv:2501.12326*.

Qwen Team. 2026. Qwen3.6-Plus: Towards real world agents. `https://qwen.ai/blog?id=qwen3.6`.

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

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

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*.

Yuanyi Song, Heyuan Huang, Qiqiang Lin, Yin Zhao, Xiangmou Qu, Jun Wang, Xingyu Lou, Weiwen Liu, Zhuosheng Zhang, Jun Wang, Yong Yu, Weinan Zhang, and Zhaoxiang Wang. 2025. Colorbench: Benchmarking mobile agents with graph-structured framework for complex long-horizon tasks. *arXiv preprint arXiv:2510.14621*.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V. Le, Ed H. Chi, Denny Zhou, and Jason Wei. 2023. Challenging big-bench tasks and whether chain-of-thought can solve them. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 13003–13051.

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 (ACL)*, pages 16022–16076.

Karen Ullrich, Jingtong Su, Claudia Shi, Arjun Subramonian, Amir Bar, Ivan Evtimov, Nikolaos Tsilivis, Randall Balestriero, Julia Kempe, and Mark Ibrahim. 2026. Openapps: Simulating environment variations to measure ui agent reliability. In *The Fourteenth International Conference on Learning Representations*.

Venus-Team, Changlong Gao, Zhangxuan Gu, Yulin Liu, Xinyu Qiu, Shuheng Shen, Yue Wen, Tianyu Xia, Zhenyu Xu, Zhengwen Zeng, Beitong Zhou, Xingran Zhou, Weizhi Chen, Sunhao Dai, Jingya Dou, Yichen Gong, Yuan Guo, Zhenlin Guo, Feng Li, and 8 others. 2026. Ui-venus-1.5 technical report. *arXiv preprint arXiv:2602.09082*.

Haoming Wang, Haoyang Zou, Huatong Song, Jiazhan Feng, Junjie Fang, Junting Lu, Longxiang Liu, Qinyu Luo, Shihao Liang, Shijue Huang, Wanjun Zhong, Yining Ye, Yujia Qin, Yuwen Xiong, Yuxin Song, Zhiyong Wu, and 1 others. 2025. Ui-tars-2 technical report: Advancing gui agent with multi-turn reinforcement learning. *arXiv preprint arXiv:2509.02544*.

Qinzhuo Wu, Zhizhuo Yang, Hanhao Li, Pengzhi Gao, Wei Liu, and Jian Luan. 2026a. Mobilebench-ol: A comprehensive chinese benchmark for evaluating mobile gui agents in real-world environment. *arXiv preprint arXiv:2601.20335*.

Yifan Wu, Yiran Peng, Yiyu Chen, Jianhao Ruan, Zijie Zhuang, Cheng Yang, Jiayi Zhang, Man Chen, Yenchi Tseng, Zhaoyang Yu, Liang Chen, Yuyao Zhai, Bang Liu, Chenglin Wu, and Yuyu Luo. 2026b. Autowebworld: Synthesizing infinite verifiable web environments via finite state machines. *arXiv preprint arXiv:2602.14296*.

Jiannan Xiang, Yun Zhu, Lei Shu, Maria Wang, Lijun Yu, Gabriel Barcik, James Lyon, Srinivas Sunkara, and Jindong Chen. 2025. Uisim: An interactive image-based ui simulator for dynamic mobile environments. *arXiv preprint arXiv:2509.21733*.

Han Xiao, Guozhi Wang, Yuxiang Chai, Zimu Lu, Weifeng Lin, Hao He, Lue Fan, Liuyang Bian, Rui Hu, Liang Liu, Shuai Ren, Yafei Wen, Xiaoxin Chen, Aojun Zhou, and Hongsheng Li. 2025. UI-Genie: A self-improving approach for iteratively boosting MLLM-based mobile GUI agents. In *Advances in Neural Information Processing Systems*, volume 38.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. 2024. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems*, volume 37, pages 52040–52094.

Haiyang Xu, Xi Zhang, Haowei Liu, Junyang Wang, Zhaozai Zhu, Shengjie Zhou, Xuhao Hu, Feiyu Gao, Junjie Cao, Zihua Wang, Zhiyuan Chen, Jitong Liao, Qi Zheng, Jiahui Zeng, Ze Xu, Shuai Bai, Junyang Lin, Jingren Zhou, and Ming Yan. 2026. Mobile-agent-v3.5: Multi-platform fundamental gui agents. *arXiv preprint arXiv:2602.16855*.

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

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*.

Leyang Yang, Ziwei Wang, Xiaoxuan Tang, Sheng Zhou, Dajun Chen, Wei Jiang, and Yong Li. 2026. Probench: Benchmarking gui agents with accurate process information. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 27547–27555.

Pei Yang, Hai Ci, and Mike Zheng Shou. 2025. macosworld: A multilingual interactive benchmark for gui agents. In *Advances in Neural Information Processing Systems*, volume 38, pages 148529–148571.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2022. Webshop: Towards scalable real-world web interaction with grounded language agents. In *Advances in Neural Information Processing Systems 35 (NeurIPS)*.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, and 17 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. In *Advances in Neural Information Processing Systems*, volume 38.

Zhong Zhang, Yaxi Lu, Yikun Fu, Yupeng Huo, Shenzhi Yang, Yesai Wu, Han Si, Xin Cong, Haotian Chen, Yankai Lin, Jie Xie, Wei Zhou, Wang Xu, Yuanheng Zhang, Zhou Su, Zhongwu Zhai, Xiaoming Liu, Yudong Mei, Jianming Xu, and 6 others. 2025. AgentCPM-GUI: Building mobile-use agents with reinforcement fine-tuning. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 155–180, Suzhou, China. Association for Computational Linguistics.

Ziyun Zhang, Zezhou Wang, Xiaoyi Zhang, Zongyu Guo, Jiahao Li, Bin Li, and Yan Lu. 2026. Infiniteweb: Scalable web environment synthesis for gui agent training. *arXiv preprint arXiv:2601.04126*.

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

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. Webarena: A realistic web environment for building autonomous agents. In *The Twelfth International Conference on Learning Representations (ICLR)*.

A System Implementation Details

This appendix provides implementation details that are omitted from §3 for space.

A.1 TaskManager and the Activity stack

The application life-cycle management of MobileGym mirrors the `ActivityTaskManager` of Android. Each app runs in its own **Task**, and each Task maintains its own **Activity stack**. The TaskManager handles requests in a Reducer pattern: `LAUNCH_APP` (start a new Task or reuse an existing one), `GO_HOME` (return to desktop), `SHOW_RECENTS` (open the recent-tasks

list), CLOSE_TASK (close and destroy React components), PUSH_ACTIVITY/POP_ACTIVITY (Activity push / pop).

App keep-alive is implemented by setting backgrounded Activity containers to display:none rather than unmounting the React component. The React state tree is therefore preserved, so when the user switches back, the interface can be restored without rebuilding the component tree. For example, when a user types half a draft in WeChat, switches to Alipay to complete a transfer, and then switches back to WeChat, the draft remains available. This behavior is important for the interaction fidelity of cross-app tasks. The TaskManager uses a non-persistent store, so a browser refresh is equivalent to a device reboot that returns to the desktop.

A.2 State layer aligned with the data model of Android

Android tier	MobileGym implementation	Persisted
Build Props	OsStateStore.build	✓
Settings	OsStateStore.settings	✓
Hardware/Sensor	+ Managers	✓
App Data	createAppStore (per-app)	✓
Runtime	Volatile stores	×

Table 3: Android data model → MobileGym state-layer mapping

The persistence policy follows the core semantics "browser refresh = device reboot": user data is preserved across refreshes, while runtime state is reset on refresh. All stores are created through a unified factory and automatically registered in a global registry, so the entire environment state can be snapshotted or reset with a single call. This infrastructure supports bit-level consistent reset and programmatic state verification.

Hardware-state constraint logic is encapsulated in a Manager layer: ConnectivityManager (handles airplane-mode → cascading WiFi/Bluetooth/cellular shutdown), BatteryManager (battery and charging), AudioManager (volume and DND), and DisplayManager (brightness and zoom). A Manager acts as a write-side facade for the OsStateStore: when the Benchmark layer injects airplane_mode:true through setState(), the Manager automatically cascades the dependent state.

A.3 Cross-app communication

Intent system. Each app declares the Intent types it can handle in its manifest. When the IntentResolver receives an Intent, it scans every manifest for matches. With a unique match, it transitions directly; with multiple matches, it displays a Chooser. Cross-app calls with callbacks (startActivityForResult) are also supported.

ContentProvider. Shared data is accessed via the content:// protocol; we currently implement Contacts, Sms, and Media providers, supporting CRUD operations and change notifications.

BroadcastBus. System-level events are dispatched through a broadcast mechanism that is semantically aligned with sendBroadcast / registerReceiver in Android.

A.4 Back-key dispatch

The BackDispatcher implements priority-chain dispatch: permission dialog (1000) > system shade (800) > keyboard (700) > app page (100) > return-to-desktop (0). Back-key events are propagated from the highest priority downwards, and the first handler that returns true consumes the event. A frame-level deduplication mechanism (back lock) prevents the edge gesture and a backdrop click from double-triggering within the same frame.

A.5 Standardized App-layer architecture

Every app follows the same directory structure: manifest.ts (declares the app identity), *App.tsx (entry component, using MemoryRouter), state.ts (Zustand-based store), navigation.declaration.ts (declarative navigation specification), and data/defaults.json (replaceable initial data). At compile time, the OS uses Vite import.meta.glob to scan apps/*/manifest.ts and system/*/manifest.ts. This design supports **zero-registration auto-discovery**: once an app module provides a manifest, the OS automatically places it on the desktop and matches its Intents, removing OS-side registration work; implementing the app itself still requires page components, state stores, navigation declarations, and data.

A.6 Input injection and coordinate transformation

The screenshot pixel coordinates observed by the agent go through the following transfor-

mation chain: (1) screenshot pixel → CSS viewport coordinates, scaled by the ratio between screenshot resolution and viewport size; (2) the target DOM element is located through `document.elementFromPoint`; and (3) standard PointerEvent / TouchEvent sequences are generated and dispatched into the React event system. For apps that declare a `designViewportWidth` (e.g. the 412 px design width of WeChat), an additional inverse transform is needed when CSS zoom is in effect.

A.7 End-to-end App, Task, and Judge development workflow

App development and validation. Development begins with human exploration of the corresponding real App. Developers examine its functional structure, state-dependent entry points, action branches, navigation and back behavior, and user-observable outcomes, and organize these observations into implementation requirements. Coding Agents then implement pages, structured states, business logic, synthetic data, resources, and a declarative navigation specification under MobileGym's shared architecture and platform contracts. The standardized module structure, automatic discovery, reusable system services, and Vite hot-module replacement (HMR, where code edits become effective in <1 second) reduce repeated infrastructure work.

Automated checks verify consistency between navigation declarations and their implementations and generate navigation graphs and reachable paths. Human developers subsequently operate the simulated App and compare its major implemented functionality against observations from the real App, including functional logic, state changes, navigation, back behavior, and cross-App outcomes. Discrepancies are reported to Coding Agents for iterative correction. Together, the automated checks and human comparative testing verify the behavioral agreement of the implemented, user-observable functionality.

Task and Judge construction. Task construction has three stages. First, human developers derive objectives from realistic mobile scenarios and specify the natural-language instruction, parameters, initial state, and expected outcome. Second, Coding Agents inspect the relevant structured-state interfaces and translate the task intent into instruction templates, task configurations, success and progress conditions, and permitted state changes. Simple state-changing tasks reuse shared base classes and declare a small number of target states; more complex tasks compose multiple conditions or introduce custom Judge logic. Automated tests cover successful execution, failure, partial completion, and unintended side effects.

Third, after the automated tests pass, human developers execute every benchmark task end to end. They verify that the instruction reflects a realistic and unambiguous user need, that reasonable completion paths and valid alternative outcomes are covered, and that the interaction outcomes, terminal state, and programmatic Judge decision are consistent. Any ambiguity, boundary-condition error, missing valid outcome, or semantic mismatch triggers another Code-Agent revision followed by repeated automated testing and manual validation. Validated Apps and tasks then directly reuse the platform interfaces for state injection, reset, snapshotting, cloning, parallel rollouts, evaluation, and programmatic RL rewards.

Development effort. According to our development records, implementing a typical everyday App and its major functionality required approximately **3–4 person-days**, while a focused system App generally required less than one person-day. Implementing all 28 simulated Apps required approximately 60 person-days in total. These figures include functional exploration, implementation, business logic, synthetic data, debugging, and manual validation. Under the mature task framework, one developer assisted by Coding Agents recently added **30 tasks in approximately 8 person-hours**. The measurement covers task design, task and Judge implementation, automated testing, human review, revision, and manual end-to-end validation; approximately 80% of the tasks were directly usable after human review, while 20% required additional Code-Agent iteration. These internal engineering estimates are derived from project development records. The recent 30-task measurement characterizes the current marginal-extension cost under the mature shared task abstractions, Judge components, and testing tools.

The separation of App content and logic allows benchmark data to be replaced or varied independently of App code.

B EFSM Formalization and Declaration Syntax

In MobileGym, the UI navigation of every app is formalized as an extended-finite-state-machine tuple:

$$\mathcal{M} = (S,\ \Sigma,\ \Delta,\ s_0,\ D,\ G,\ U)$$

where S is the set of UI states (each state corresponds to a unique combination of route path + query parameters); Σ is the input alphabet (user actions); $\Delta : S \times \Sigma \times G \to S \times U$ is the transition function with guards and update operations; s_0 is the initial state; D is the set of application state variables; G is the set of guards; and U is the set of update operations on D. Compared with a classical FSM, the EFSM extension has three roles: (1) guards allow the same input to trigger different transitions under different data states; (2) data-driven expansion allows the state space to grow according to configuration data; and (3) compound UI states model different visual presentations (pop-ups, drawers, tabs) that share the same route path as distinct state nodes.

Guard examples.

```
// URL-based "from" constraint
from: { path: '/book/:id',
        search: { modal: null } }    //
            modal must be absent
from: { path: '/book/:id',
        search: { tab: 'comment' } } //
            tab must equal 'comment'

// AppState-based conditional branch
cases: [
  { to: '/user/:mid',
    search: { panel: 'recommend' },
    when: { op: 'eq',
            left: { ref: 'appState',
                    key: 'isFollowing'
                        },
            right: false } },
  { to: '/user/:mid',
    search: { menu: 'unfollow' },
    when: { op: 'always' } }, //
        fallback
]

// Data-driven entry-visibility
    condition
ui: { condition: {
  op: 'memberOf',
  ref: 'initialShelf',
  param: 'bookId' } }
```

Runtime interface and DOM tagging.

```
// Hook
const { go, back } = useAppNavigate();
go('book.modal.open', { bookId: '60' });
back();

// DOM tagging (the agent does not use
    it directly)
<button data-trigger="book.modal.open"
        data-trigger-params='{"bookId
            ":"60"}'>
  Open Book
</button>
```

The declarative navigation specification is both **executable and analyzable**: at runtime it drives UI navigation (a transition is fired through `go(transitionId, params)`); statically it supports consistency checking, BFS path enumeration (for candidate trajectory generation and shortest-path enumeration), and navigation-graph construction.

C Full Action Space

Category	Action	Parameters & description
Physical touch	`CLICK`	`point=[x,y]`, single tap
	`DOUBLE_TAP`	`point=[x,y]`, double tap
	`LONG_PRESS`	long press to trigger context menu
	`TYPE`	`value=str` (optional point/clear), supports Chinese pinyin IME
	`SWIPE`	`point1, point2`, with inertia
	`DRAG`	`point1, point2`, no inertia
System keys	`BACK`	invoke the BackDispatcher priority chain
	`HOME`	return to desktop
	`RECENT`	open recent-tasks list
	`ENTER`	fire the Enter key
Control	`WAIT`	`value=seconds`
	`AWAKE`	`value=app_id`, launch an app
Termination/answer	`ANSWER`	`value=str`, submit an answer (does not terminate)
	`COMPLETE`	terminate the episode, declaring success
	`ABORT`	terminate the episode, declaring inability to complete
Other	`INFO`	`value=str`, ask the user a clarifying question (does not terminate)
	`NOOP`	no-op (used internally by the agent)

Table 4: MobileGym action space (the Benchmark-layer `Action` abstraction, 17 in total)

General-purpose models under the `generic_v2` agent template use the full action space above (excluding `NOOP` and `INFO`, which are reserved for agent-internal use); GUI-specialized models (AutoGLM-Phone-9B, UI-TARS-1.5-7B, UI-Venus-1.5-8B, GUI-Owl-1.5-8B-Think, Step-GUI-4B) use their own native action spaces, and the agent adapter maps them to the MobileGym environment-layer `Action` abstraction. Coordinates are uniformly normalized to $[0, 1000] \times$

$[0, 1000]$.

D App Coverage and Representative Tasks

Type / Category	Apps	#Routes
Everyday apps (12)		
Social/Comm.	WeChat	100
	RedNote	41
Finance	Alipay	48
Video/Ent.	Bilibili	52
Travel	Maps, 12306	17, 48
Reading/Music	WeChat Reading, Spotify	30, 35
Social media	Reddit, X (Twitter)	17, 33
Business/Prod.	Tencent Meeting, eBay	18, 9
System apps (16)		
Launcher	Launcher	$-^*$
Core utilities	Settings, Contacts, SMS	$138^\dagger$, $39^\dagger$, 8
Productivity	Calendar, Notes, Calculator, Calculator (AOSP), AnswerSheet	10, 7, 1, 1, $-^*$
System apps	Browser, File Manager, Clock, Theme Store	2, 7, 5, 2
Info/Nav	Weather, Compass, Gallery	9, 4, 5

Table 5: Simulated app coverage (12 everyday + 16 system = 28 apps). **#Routes** counts route objects declared in each app's `navigation.declaration.ts`; compound `uiStates` (popups, drawers, tabs sharing a route path) and runtime modals are *not* counted as separate routes, so the figure underestimates the number of distinguishable UI states. *Launcher and AnswerSheet are single-screen apps without a navigation declaration. †Settings and Contacts use the Android AOSP data-driven preference pattern: a single `/page/:pageId` route mounts content from a page registry, so the React-level route count alone severely under-represents user-visible screens. Their figures add the reachable preference pages: Settings = 3 routes + 135 reachable pages out of 623 defined (89 with interactive controls; 46 are read-only placeholder screens); Contacts = 10 routes + 29 phone-preference pages (28 of them interactive).

Every app is populated with public or synthetic data that mimics the structure and content style of real platforms, loaded from a configurable `defaults.json`. Across the 12 everyday apps, the released world data contains over 190K entities (over 350K structured records including auxiliary indices and relations), making feeds, search results, comments, products, maps, and travel pages information-dense enough to support parameterized search, query, and deep-dive tasks.

E Detailed Task Taxonomy

Test and Train are strictly disjoint. The Train set is mostly composed of single-app tasks that cover the core skills of the 12 everyday apps; 36% of the Test set consists of cross-app tasks, which extends beyond the training distribution and supports OOD-generalization diagnostics for cross-app performance. All tasks support parameter sampling; L3/L4 tasks additionally provide 2–3 instruction variants, which combine orthogonally with parameters to further increase diversity.

Test256 distribution by dimension. Difficulty: L1=20, L2=73, L3=83, L4=80. Objective: operate=170, query=48, hybrid=38. Composition: atomic=22, sequential=110, transfer=56, deep_dive=68.

Capability tags. Each task carries 1–4 tags from the following 13-tag vocabulary:

- `nav` — navigate to a specific page or screen.
- `settings` — modify app or system settings.
- `search` — locate content through explicit search, filtering, or sorting mechanisms.
- `create` — create new content (post, message, order, etc.).
- `edit` — modify existing content or records.
- `delete` — remove content, records, or accounts.
- `social` — interact with other users (follow, like, comment, etc.).
- `extract` — extract and report specific information from the UI.
- `handoff` — transfer context or data across apps.
- `finance` — perform financial operations (payment, transfer, etc.).
- `reasoning` — require multi-step inference or comparison.
- `explore` — locate content by navigating feeds, comment threads, long lists, or unknown page hierarchies without an explicit search entry point.
- `image` — require understanding image content to complete the task.

Objective	Composition	Diff.	App	Instruction example
operate	atomic	L1	Clock	Turn on the 7:30 alarm for me
query	atomic	L1	Weather	Tell me what the temperature is in Beijing right now
operate	sequential	L2	Spotify	Add "Shape of You" to my Liked Songs in Spotify
query	deep_dive	L3	Alipay	In the Alipay bill, accumulate the amounts and tell me which counterparty has the largest total
hybrid	transfer	L3	RedNote → WeChat	Search "camping" in RedNote and send the title of the first note to Xiaohong via WeChat
query	sequential	L4	eBay	I want to buy a brand-new Sony Bluetooth headset shipped from Japan; which one is the cheapest, and how much is it including shipping?
hybrid	deep_dive	L4	eBay+Alipay → Notes	Find the cheapest brand-new AirPods on eBay, see how much my Alipay balance would have left after buying it, and write the product name, price, and remaining balance to Notes

Table 6: Representative task examples

Scope	Test (256)	Train (160)
S1 (Single-app)	163 (64%)	141 (88.1%)
S2 (Cross-app, 2 apps)	65 (25%)	17 (10.6%)
S3 (Multi-app, 3+ apps)	28 (11%)	2 (1.3%)
Total	**256**	**160**

Table 7: Composition of Test256 / Train160 (by Scope)

F AnswerSheet Protocol Design Details

F.1 Field types and matchers

The AnswerSheet, as a system app, provides an answer form. Each field declares a type and matcher:

- `choice` field—choose from enumerated options, paired with the `exact` matcher.
- `number` field—numeric input, paired with the `number` matcher (with floating-point tolerance).
- `text` field—text input, paired with `exact` / `date` / `time` / `duration` matchers.
- `repeatable` multi-value list—supports scenarios that require listing multiple answers.

F.2 Design motivation

Eliminating natural-language false negatives. "34°C" / "34 degrees" / "about 34 Celsius" all map to the same numeric value `34` under a `number` field. The judge can therefore perform a floating-point comparison with tolerance, without relying on preset string-normalization rules.

Eliminating false positives from enumeration / mixed-in thinking. A `number` field can hold only one numeric value, and a `choice` field can hold only one enumerated value. Typing "34" and typing "33 or 34" produce **physically different states**. The agent must navigate to the app, locate the field, and enter the value; each step is observable at the state layer.

Compatibility with small models. Some small GUI agents (e.g. AutoGLM-Phone-9B) often place the entire think trace in the `<answer>` field and may not stably emit purely structured answer text. Requiring them to "fill the answer" through GUI operations better matches their interaction-oriented training distribution.

F.3 Format expectations and hints

Each field carries a `hint` string, which is shown to the agent in the UI as a placeholder and explicitly indicates the expected input form, e.g. "Temperature (Celsius, integer)", "Date (YYYY-MM-DD)", "Amount (CNY, two decimal places)". The task author is responsible for providing an unambiguous format constraint in the hint. If the model still writes "34°C" into a `number` field, or writes the date as "tomorrow" instead of a concrete date, the judge marks it wrong. This failure reflects insufficient ability to follow the output format rather than ambiguity in the evaluation design.

F.4 Compensation for execution cost

The AnswerSheet introduces an additional "switch app + fill form" GUI workflow for query tasks. To compensate for this execution cost, tasks with `answer_fields` are **given an additional 15-step budget** (i.e. L1: $15+15=30$, L4: $60+15=75$), corresponding to a reasonable number of actions for "switching to AnswerSheet + filling fields + submitting".

G Detailed Experimental Configuration

Inference configuration. General-purpose models (`generic_v2`) uniformly use the following settings: decoding `temperature`=0.1, `top_p`=0.95, `frequency_penalty`=0, `max_tokens`=4096; single-step LLM call timeout 300 s; a 0.8 s wait after each action to allow the UI render to stabilize; and `loop_detect`=10 (early termination when the same action is repeated $\geq$10 times in a row). The screenshot is provided at full resolution (1080×2400 physical $\rightarrow$ 0–1000 normalized coordinates), and the dialogue history is managed in a "current step carries the screenshot + earlier steps keep only the LLM text response" format to avoid context growth over long episodes. The prompt templates and decoding configurations of GUI-specialized models follow their original papers / official implementations, and the execution layer shares the environment constraints above.

GRPO training configuration. The Sim-to-Real training run uses Qwen3-VL-4B-Instruct as the initial policy, 3 GPUs on a single node, 96 parallel environment instances. Rollouts use `vLLM` asynchronous mode with group size $k = 8$, train batch size 12, PPO mini-batch size 12, and per-GPU micro-batch size 2. The optimizer learning rate is 10^{-6}, gradient clipping is 1.0, $\gamma = 1.0$, $\lambda = 1.0$, KL coefficient is 0.01, and the asymmetric clipping range is 0.2/0.28. We use maximum prompt length 32768, maximum response length 1024, rollout maximum model length 40960, and training-time agent decoding temperature 0.7 (validation temperature 0.1). The environment pool uses page-level isolation, a 0.8 s delay after each action.

Reward function. The training reward is computed from structured rollout artifacts. Let $p \in [0, 1]$ denote task progress, i.e., the fraction of goal checks passed. The base reward is $r = p$. For AnswerSheet tasks, if the agent submits the sheet but any answer field is wrong, the reward is recomputed after removing the bookkeeping check `answer_sheet.submitted`, so submitting an incorrect sheet does not provide extra progress credit. Multiplicative discounts are then applied:

$$
\begin{aligned}
r \leftarrow p' &\cdot 0.8^{\mathbb{I}[\text{goal success} \wedge \neg \text{clean}]} \\
&\cdot 0.8^{\mathbb{I}[\text{false complete} \wedge p' > 0]} \\
&\cdot 0.5^{\mathbb{I}[\text{post-success abort}]} \cdot 0.5^{\mathbb{I}[\text{overdue}]},
\end{aligned}
$$

where p' is either the original progress p or the AnswerSheet-adjusted progress described above. *Goal success* means the task goal state is reached, *clean* means no unexpected state changes are detected, and *false complete* means the agent terminates with `COMPLETE` but the episode is not a full success. The final two terms penalize cases where the goal state is reached but the agent does not correctly declare completion: *post-success abort* means it terminates with `ABORT`, while *overdue* means it keeps acting until truncation. Binary task correctness for reporting is still the simulator's final success signal, not the shaped reward.

Notes on the evaluated models.

- Gemini 3.1 Pro (Google DeepMind, 2026): a reasoning model from Google DeepMind.
- Doubao-Seed-2.0-Pro (ByteDance Seed Team, 2026): a multimodal model from ByteDance Seed.
- Qwen3.6-Plus (Qwen Team, 2026): a multimodal model from Alibaba Tongyi Qianwen.
- AutoGLM-Phone-9B (Liu et al., 2024): a mobile-oriented GUI agent from Zhipu AI.
- UI-TARS-1.5-7B (Qin et al., 2025): a GUI agent from ByteDance Seed.
- UI-Venus-1.5-8B (Venus-Team et al., 2026): full-trajectory online RL training; Android-World SOTA.
- GUI-Owl-1.5-8B-Think (Xu et al., 2026): introduced in Mobile-Agent-v3.5; multi-platform RL with MRPO.
- Step-GUI-4B (Yan et al., 2025): Calibrated Step Reward System.
- Qwen3-VL-4B-Instruct (Bai et al., 2025): a vision-language model from Qwen, used as the base model for our Sim-to-Real training experiments.

H Full Result Decomposition

H.1 SR by taxonomy dimension

Table 8 decomposes the test-set Success Rate along all three taxonomy axes (Difficulty, Objective, Composition; §4.1), extending the difficulty-only breakdown in Table 2.

Model	Difficulty				Objective			Composition			
	L1 (20)	L2 (73)	L3 (83)	L4 (80)	oper. (170)	query (48)	hybrid (38)	atom. (22)	seq. (110)	trans. (56)	deep (68)
Proprietary models											
Gemini 3.1 Pro	97.5	83.6	63.3	21.9	56.8	64.6	60.5	81.8	76.8	33.0	43.4
Doubao-Seed-2.0-Pro	100.0	93.2	48.2	6.2	52.4	52.1	50.0	77.3	70.0	25.0	36.8
Qwen3.6-Plus	100.0	78.1	44.6	3.8	41.2	50.0	60.5	81.8	61.8	16.1	32.4
Open-source GUI-specialized models											
AutoGLM-Phone-9B	86.2	33.6	9.6	1.9	20.7	23.4	12.5	56.8	25.7	7.1	9.6
UI-TARS-1.5-7B	77.5	21.9	3.0	1.6	14.0	20.3	4.6	37.5	20.2	2.7	4.8
UI-Venus-1.5-8B	85.0	21.9	6.0	1.9	16.9	15.6	8.6	37.5	20.7	6.2	7.4
GUI-Owl-1.5-8B-Think	76.2	26.0	4.2	1.2	16.9	14.6	7.9	44.3	20.9	2.7	6.6
Step-GUI-4B	83.8	17.8	2.4	1.6	16.3	7.8	3.9	30.7	21.4	0.4	3.7
Open-source generalist models											
Qwen3-VL-4B-Instruct	71.2	12.3	0.6	0.3	12.9	3.6	0.7	25.0	15.5	0.9	1.5

Table 8: Success Rate (%) decomposed by Difficulty / Objective / Composition

H.2 Trajectory-length diagnostics

Table 9 reports the mean episode length (**Steps**) and the mean length of successful trajectories (**Steps✓**) on the MobileGym-Bench test set. Successful trajectories cluster around 8–14 steps across models, while overall episode length varies substantially, reflecting how often weaker models exhaust the step budget without succeeding.

Model	Steps	Steps✓
Proprietary models		
Gemini 3.1 Pro	16.4	13.4
Doubao-Seed-2.0-Pro	17.1	12.8
Qwen3.6-Plus	20.6	14.0
Open-source GUI-specialized models		
AutoGLM-Phone-9B	26.8	13.1
UI-TARS-1.5-7B	27.6	13.0
UI-Venus-1.5-8B	21.6	11.0
GUI-Owl-1.5-8B-Think	24.9	11.7
Step-GUI-4B	31.7	11.8
Open-source generalist models		
Qwen3-VL-4B-Instruct	17.6	7.9

Table 9: Trajectory-length diagnostics on the MobileGym-Bench test set. **Steps**=mean episode length; **Steps✓**=mean length of successful trajectories.

H.3 High-Risk subset

The High-Risk subset comprises 7 standalone `Payment` tasks (money transfer, card binding, subscription renewal, etc.) plus 7 high-risk tasks within test256 (e.g., account registration, account deactivation, large transfers, and message/data deletion), 14 in total. This subset characterizes execution capability in **irreversible / high-consequence** scenarios; it differs from a safety evaluation that tests refusal of harmful or inappropriate instructions. The numbers in this table are completion success rates and do not measure whether the model should refuse the operation.

Model	Payment (7)	T256-Risk (7)	All (14)
Proprietary models			
Gemini 3.1 Pro	64.3	71.4	67.9
Doubao-Seed-2.0-Pro	0.0	71.4	35.7
Qwen3.6-Plus	28.6	42.9	35.7
Open-source GUI-specialized models			
AutoGLM-Phone-9B	3.6	28.6	16.1
UI-TARS-1.5-7B	0.0	25.0	12.5
UI-Venus-1.5-8B	10.7	14.3	12.5
GUI-Owl-1.5-8B-Think	3.6	25.0	14.3
Step-GUI-4B	0.0	14.3	7.1
Open-source generalist models			
Qwen3-VL-4B-Instruct	0.0	28.6	14.3

Table 10: High-Risk subset SR (%): financial operations / account-credential modifications / irreversible deletions

See the Ethical Considerations section.

H.4 Sim-to-Real simulation-side breakdown by difficulty

Model	L1 (20)	L2 (73)	L3 (83)	L4 (80)	All
Qwen3-VL-4B-Instruct (base)	71.2	12.3	0.6	0.3	9.4 $\pm$0.6
Qwen3-VL-4B-10s (trained)	92.5	37.7	11.7	1.2	22.2 $\pm$1.2
Δ (training gain, pt)	**+21.3**	**+25.4**	**+11.1**	**+0.9**	**+12.8**

Table 11: Sim-to-Real training gain on the simulation side, broken down by difficulty (256-task test set)

H.5 Sim-to-Real outcome-stratified task sampling

Tasks are bucketed by the number of passes out of 4 base / 4 trained rollouts: **Uplift** (base $\leq$1/4, trained $\geq$3/4, all 26 selected); **Stable-pass** (both $\geq$3/4, all 21 selected); **Mid** (partial uplift, all 20 selected); **Regression** (base $\geq$3/4, trained $\leq$1/4, 0 instances, suggesting that no severe regression is observed under this sampling protocol); and **Stable-fail** (both $\leq$1/4, 189 tasks). From the three signal buckets we select all 67 tasks (uplift 26 + stable-pass 21 + mid 20); from Stable-fail we additionally sample

15 tasks at random as a sanity check, re-sampling any task that cannot be equivalently reproduced on the real device.

Of the 67 signal-bucket tasks, 8 are dropped because they cannot be equivalently reproduced on the real device: 3 irreversible account-level modifications, 1 non-reversible consumption-style operation, and 4 tasks that require preset states the real device cannot equivalently reproduce (synthetic meeting histories, preset message sessions, etc.). The final **59 signal-bucket tasks** are the headline subset; combined with the 15 stable-fail sanity-check tasks, 74 tasks are run on the real device in total. These 8 unrun tasks illustrate the complementary value of the simulator: within MobileGym, they can be configured to arbitrary initial states and rolled back without real-world consequences, while on a real device such configurations are either not equivalently reproducible or require prohibitive cost.

H.6 Same-outcome Trajectory-Length Breakdown

Table 12 compares trajectory length only on same-outcome pairs: real-device successes are paired with simulator successful rollouts for the same task, and real-device failures are paired with simulator failed rollouts for the same task.

Model	Slice	Pairs	Sim	Real	Δ	MAE
Trained	**operate-succ.**	**91**	**10.08**	**12.20**	**+2.12**	**5.40**
	operate-fail	12	23.75	17.42	-6.33	8.50
	query-succ.	28	13.71	5.54	$-8.18^{\star}$	8.18
	hybrid-succ.	17	18.06	15.06	$-3.00^{\star}$	3.94
Base	**operate-succ.**	**38**	**5.00**	**6.03**	**+1.03**	**1.34**
	operate-fail	57	16.74	38.32	$+21.58^{\dagger}$	21.75
	query-succ.	6	15.50	6.00	$-9.50^{\star}$	9.50
	query-fail	28	16.21	30.57	$+14.36^{\dagger}$	27.79
	hybrid-fail	39	17.64	34.23	$+16.59^{\dagger}$	19.97

* Query/hybrid success only; sim includes the AnswerSheet submission stage absent on real. † Sim-side loop-detect early stop ($\geq$10 identical actions) vs. real-side run-to-budget on base-model flailing.

Table 12: Same-outcome paired trajectory length on sim vs. real, broken down by Objective and outcome. $\Delta = \text{Real} - \text{Sim}$.

Three observations.

(i) Operate-success length remains comparable. On paired tasks that succeed in both environments, real-device operate trajectories are only modestly longer than simulator trajectories (trained +2.12 steps; base +1.03 steps), consistent with similar successful-operation path lengths across environments.

(ii) Query/hybrid rows reflect protocol asymmetry. Real-device query successes are shorter because real-device query tasks do not include the simulator-only AnswerSheet submission workflow.

(iii) Failure rows are termination diagnostics, not trajectory-length evidence. Simulator runs use loop-based early stopping, while the real-device runs in this study do not; consequently, many base failures run until the task step budget on the real device.

I Reference-Model Sensitivity of the L1–L4 Stratification

The primary L1–L4 strata in §4 are calibrated by the joint SR+PR criterion over eight reference models. The criterion is applied sequentially: L1 requires mean SR $\geq$ 75% and mean PR $\geq$ 75%; L2 contains remaining tasks with mean SR $\geq$ 25% and mean PR $\geq$ 50%; L3 contains remaining tasks with mean SR $>$ 0 and mean PR $\geq$ 25%; L4 contains the rest. To test sensitivity to the choice and number of reference models, we re-run the same calibration using four reference models: {Gemini 3.1 Pro, Doubao-Seed-2.0-Pro, UI-Venus-1.5-8B, Step-GUI-4B}. Qwen3-VL-4B-Instruct and its trained variant remain held out in both calibrations, avoiding calibration bias toward the Sim-to-Real lift analysis.

The bucket counts shift from $20/73/83/80$ under the 8-model calibration to $25/99/58/74$ under the 4-model calibration. The corresponding mean SR/PR values remain well separated: 8-model L1–L4 means are $(88.3, 90.7)$, $(47.0, 64.0)$, $(22.7, 38.3)$, $(5.0, 15.0)$, while 4-model means are $(89.0, 91.2)$, $(50.4, 63.3)$, $(24.6, 38.5)$, $(3.5, 18.4)$. Table 13 reports the per-model SR breakdown under both calibrations.

Two qualitative observations from §5.1 and §5.2 are robust:

1. **Sim-to-Real lift concentrates on L1–L2 and diminishes sharply at L3–L4 under both calibrations.** The primary 8-model calibration yields $+21.3/+25.4/+11.1/+0.9$ pt, while the 4-model calibration yields $+23.0/+22.5/+7.3/+0.7$ pt. In both cases, most of the training lift lies in L1–L2 and nearly vanishes on L4.

2. **L4 isolates the frontier under both calibrations.** Under the 8-model calibration, only

Gemini 3.1 Pro stays meaningfully above the floor on L4 (21.9%), while every other model is $\leq$ 6.2%. Under the 4-model calibration, Gemini remains the only model above 10% on L4 (12.2%), while all other models are $\leq$ 8.1%. The trained 4B model also exceeds AutoGLM-Phone-9B on L1–L2 under both calibrations.

The full overall SR (9.4% $\rightarrow$ 22.2%, +12.8 pt for trained vs. base) is invariant by construction because the test set is fixed at 256 tasks.

J Detailed VLM-Judge Misjudgment Audit

We use Qwen3.6-Plus as the VLM judge for real-device evaluation because it is among the closed-source models with strong multimodal reasoning capability and relatively low API cost. The real-device pass/fail labels used to compute Figure 5 are corrected according to this manual audit. The 12 misjudgment instances cover 9 unique tasks; some tasks are misjudged for both the base and trained models. The 30 stable-fail trajectories ($n = 15$ tasks $\times$ 2 models) incur 0 misjudgments because both models fail in obvious patterns that the VLM judge reads correctly. We therefore report 10.2% on the 118-trajectory signal-bucket subset, where misjudgments by construction occur on non-trivial trajectories, as the headline rate, with the 8.1% (12/148) on the broader pool noted for completeness. The misjudge rate of the trained model (11.9%) is slightly higher than that of the base model (8.5%) because the trained model produces more complex trajectories, which give the VLM more "declarative-statement" surface that can lead to errors. This phenomenon warrants further study.

Judge-model robustness check. To test whether the observed errors are specific to Qwen3.6-Plus, we re-judge the same saved real-device trajectories with GPT-5.4 (OpenAI, 2026) without re-running the agents. We use the same manually audited labels as ground truth and exclude protocol-level manual exceptions caused by real-app anomalies from the judge-error count. GPT-5.4 yields the same aggregate error rate, but with a different distribution across base and trained trajectories (Table 15).

K Case Study

Sim-to-Real OOD generalization: a case study. The real-device community used in `Reddit_CreatePostToCommunity` requires a flair tag before a post can be submitted. In a single real-device trial, the base model repeats clicks on the grayed-out "Post" for the full 60-step budget until the trajectory is truncated, without recognizing the disabled state. The trained model, after two failed "Post" clicks, attends to the required-field asterisk on the `Add tags & flair` entry, opens the flair selector, picks a flair (`Tech`), applies it, and submits in 22 steps (Figure 6). At step 15, the verbatim think trace of the trained model—translated from Chinese—explicitly verbalizes the bridging inference ("the button is still clickable but its color is gray, which may indicate the system has not detected all required fields… the 'Add tags & flair' entry has an asterisk indicating it is a required field"; full trace in Listing 1).

The recovery occurs on a real-device gate condition not encountered in the training rollouts, illustrating Sim-to-Real OOD generalization at the level of individual interaction primitives. The structural barriers discussed in §1 make real-device online RL difficult to run at scale; simulator-based online RL on MobileGym offers a practical alternative. This case provides one instance in which the learned behavior transfers to a community-specific posting constraint.

Listing 1: Verbatim think trace on `Reddit_CreatePostToCommunity`, translated from Chinese; ellipses indicate omitted boilerplate. The reasoning of the trained model at step 15 attends to the visual cue (grayed button, asterisked label) and re-plans; the base model emits byte-for-byte identical reasoning at every step from 10 onward.

```
=== TRAINED MODEL (single real-device trial; 22
     steps total) ===

[Step 13]  (first click on the grayed-out "Post"
     button)
  "Title and body are correctly filled. The 'Post'
       button in the
   top-right is in clickable state. Next: click '
        Post' to publish."
  -> CLICK [880, 78]

[Step 14]  (second click on the grayed button;
     reasoning essentially
            identical to step 13)
  -> CLICK [880, 78]

[Step 15]  *** insight ***
  "Title and body are filled, all required
       information is ready.
   The 'Post' button is still clickable, BUT ITS
        COLOR IS GRAY,
   which may indicate the system has not detected
```

Model	8 reference models (primary)				4 reference models (sensitivity check)			
	L1 (20)	L2 (73)	L3 (83)	L4 (80)	L1 (25)	L2 (99)	L3 (58)	L4 (74)
Proprietary models								
Gemini 3.1 Pro	97.5	83.6	63.3	21.9	98.0	85.4	56.0	12.2
Doubao-Seed-2.0-Pro	100.0	93.2	48.2	6.2	100.0	88.9	34.5	0.0
Qwen3.6-Plus	100.0	78.1	44.6	3.8	100.0	66.7	34.5	8.1
Open-source GUI-specialized models								
AutoGLM-Phone-9B	86.2	33.6	9.6	1.9	78.0	23.5	8.2	5.1
UI-TARS-1.5-7B	77.5	21.9	3.0	1.6	67.0	15.9	0.4	3.4
UI-Venus-1.5-8B	85.0	21.9	6.0	1.9	79.0	15.9	5.2	1.4
GUI-Owl-1.5-8B-Think	76.2	26.0	4.2	1.2	64.0	19.2	5.6	0.7
Step-GUI-4B	83.8	17.8	2.4	1.6	79.0	11.6	2.6	0.3
Open-source generalist models (Sim-to-Real subject)								
Qwen3-VL-4B-Instruct (base)	71.2	12.3	0.6	0.3	63.0	7.6	0.9	0.3
Qwen3-VL-4B-10s (trained)	92.5	37.7	11.7	1.2	86.0	30.1	8.2	1.0
Trained − base lift (pt)	**+21.3**	**+25.4**	**+11.1**	**+0.9**	**+23.0**	**+22.5**	**+7.3**	**+0.7**

Table 13: Reference-model sensitivity of the L1–L4 stratification under the joint SR+PR criterion. Qwen3-VL-4B-Instruct and Qwen3-VL-4B-10s are excluded from both calibrations. All SR numbers are mean across the same trial counts as Table 2.

Model	FP	FN	Total	Misjudge
Base (Qwen3-VL-4B-Instruct)	4	1	5/59	8.5%
Trained (+Sim RL, `train10s`)	4	3	7/59	11.9%
Total	8	4	12/118	**10.2%**

Table 14: Manual review of real-device VLM-judge misjudgments on the 59-task signal-bucket subset (118 trajectories = 59 tasks $\times$ 2 models). The 30 stable-fail trajectories (15 tasks $\times$ 2 models) incur 0 misjudgments and are not included in the rate; including them would dilute the rate to 12/148 = 8.1%.

Judge	Base	Trained	Total	Error rate
Qwen3.6-Plus	5/59	7/59	12/118	10.2%
GPT-5.4	3/59	9/59	12/118	10.2%

Table 15: Robustness check: VLM-judge error rate when the same saved real-device trajectories are re-judged with GPT-5.4.

```
        all required
   fields (such as tags or labels) or other
        preconditions.
   ... The 'Add tags & flair' button on the current
        screen has an
   ASTERISK indicating it is a REQUIRED FIELD.
   Therefore, next: click 'Add tags & flair' to fill
         in the
   missing field, ensuring the post meets the
        publication
   requirements."
  -> CLICK [202, 255]      (the flair pill)

[Step 16]  (entered the tag selector)
  "Currently in the 'Add Tag' interface. 'None' is
       selected and the
   system indicates 'required'. ... 'Tech' is closer
         to the 'review'
   context, so select 'Tech'."
  -> CLICK [78, 668]       ('Tech' option)

[Step 17]  (after picking 'Tech')
  "'Tech' tag successfully selected, satisfying the
       required-field
   condition. The 'Apply' button is clickable. Click
         'Apply'."
  -> CLICK [909, 128]

[Step 18]  (back at post page; 'Post' button now
     blue)
  "Title, body, and tag are all correctly filled.
       The 'Post' button
   is now blue, indicating it can be clicked. Click
         'Post' to
   finally publish."
  -> CLICK [880, 78]       (succeeds)


=== BASE MODEL (single real-device trial; 60 steps =
      max budget) ===

[Step 10]
  "Title and body are filled, community is selected.
        Next: click
   'Post' to publish. The button is in the top-right
         ."
  -> CLICK [880, 78]

[Steps 11..60]  (50 further responses)
  "Title and body are filled, community is selected.
        Next: click
   'Post' to publish. The button is in the top-right
         ."
```

```
-> CLICK [880, 78]
(byte-for-byte identical reasoning across all 51
     click steps; no
 recognition of the disabled state or the required
      -field asterisk)
```

The AnswerSheet protocol as a conservative gate. Query/hybrid tasks in MobileGym end with the AnswerSheet protocol. The AnswerSheet **neither leaks the answer nor reduces the task to multiple choice**; it only adds an additional barrier of format and submission. On the test set, we observe a consistent pattern: on tasks containing an AnswerSheet, the simulation and real-device performance of the trained model are closely aligned (over all 19 AnswerSheet tasks: sim 71.1% vs real 73.7%; in the uplift subset of 12: sim 79.2% vs real 83.3%). For the base model, however, real-device free-text evaluation scores higher than simulation-side AnswerSheet evaluation (uplift subset: sim 2.1% vs real 25.0%). This pattern suggests that AnswerSheet imposes an additional submission-format barrier on weak models, and that **the AnswerSheet pass rate is a conservative lower bound of the free-text success rate**, not an inflated version of it. A strict ablation (toggling the AnswerSheet on the same rollout) is left to future work.

L Broader Uses of MobileGym

Beyond benchmarking, MobileGym enables several research directions that require controllable mobile environments.

Custom mobile environments and benchmarks. Because apps, world data, task templates, and judges are modular, MobileGym can be extended beyond MobileGym-Bench to build targeted mobile benchmarks. Researchers can instantiate domain-specific environments, such as mobile finance, travel planning, social-media safety, or digital-literacy training, while retaining the same reset, snapshot, and state-based judging interface.

Controlled robustness and safety evaluation. Programmable state and event injection make it possible to evaluate agents under systematic variations rather than incidental real-device conditions. The same task can be run under different balances, permissions, network states, incoming messages, popups, or phishing-like content. This supports controlled studies of robustness, prompt-injection susceptibility, caution gating, side effects, and recovery behavior.

Low-cost online RL research for GUI agents. Because MobileGym can fork identical initial states into many lightweight browser instances, it provides a practical testbed for studying online RL in GUI environments without large emulator clusters or real-device farms. Researchers can compare reward designs, state-diff penalties, rollout grouping strategies, and Sim-to-Real behavior under reproducible initial states and deterministic outcome signals.

Controlled training data synthesis. Each interaction step yields a five-tuple $(s_t^{\text{vis}}, s_t^{\text{json}}, a_t, s_{t+1}^{\text{vis}}, s_{t+1}^{\text{json}})$ of paired visual and structured state transitions. Because the environment is fully controllable, this data can be generated with intentional state coverage rather than incidental device logs, supporting training of mobile UI world models, state predictors, reward models, or trajectory verifiers.

M Detailed Footnotes for the Resource-Efficiency Comparison

Detailed footnotes complementing the efficiency rows in Table 1:

- The ∼50 MB core disk footprint of MobileGym contains the framework code and the code of the 28 apps (JavaScript bundle, component code, state/navigation definitions, CSS, IME dictionary, and icon fonts). App content data (images, virtual-filesystem presets, in-app corpora, etc.) scales linearly with the number of apps and content richness as an optional layer; the current full deployment is around 1.5 GB and can be slimmed or replaced as needed. Android-emulator environments are dominated by the Android system image, which is largely independent of benchmark task count or content data.
- AndroidWorld README states that the emulator guest memory is 2 GB; in our Docker measurements, host occupation steadily sits at ∼4.5 GB (containing emulator + FastAPI server + Android 13 system image, without `/dev/kvm`).
- The Docker image of AndroidWorld totals 20.2 GB, of which the Android 13 system image accounts for 9.5 GB. Multiple emulator instances can share the same image, but each instance still has a ∼1 GB `userdata.img`.

- MobileWorld is built on top of the AndroidWorld emulator stack; we therefore report its memory and disk figures as lower bounds ($\geq$4.5 GB and $\geq$20 GB) inherited from the AndroidWorld baseline rather than direct measurements. The actual figures cannot fall below this baseline but were not separately benchmarked. AndroidLab's numbers ($\sim$6 GB, $\sim$9 GB) are taken from its repository's stated emulator configuration; we did not independently measure them.

- The emulator boot of AndroidWorld has a measured median of 78 s without `/dev/kvm`; with KVM enabled, it is usually faster. After boot, the FastAPI server still has to perform automatic app setup (Chrome, Contacts, etc.), and the time-to-fully-ready can reach the minute level in our test environment.

- The `/reset` endpoint of AndroidWorld does not reboot the emulator or wipe app data; it only performs `press_home` + clear interaction_cache. Task-level state restoration is achieved via `app_snapshot` (file-level copy of `/data/data/<package>`), which is constrained by what the OS surfaces at the file-system layer. App-internal in-memory state and account/backend state are not captured. MobileGym restores state via direct JSON setState injection into the same in-memory stores that the apps read from, so the restoration scope matches the state the agent can affect.

- The AndroidWorld repository does not provide an in-process multi-session runner; running multiple sessions concurrently requires launching an independent Docker container per session (with its own emulator, app data, snapshot, and host port), so resource overhead scales linearly. MobileGym can directly manage multiple browser contexts / pages within a single process via `EnvPool`, with no duplicated OS or system-image overhead.

N Cost Table if Switching to a VLM Judge

To anchor the abstract cost argument to concrete scenarios, we use **one full evaluation run on the 256-task MobileGym test set** as the unit and compute the VLM-judge API cost in two typical scenarios. The estimate is based on a sampled VLM audit of 546 screenshots, where each trajectory consumes on average $\sim$29.8K input tokens including screenshots and $\sim$924 output tokens.

Scenario	**Qwen3.6-Plus**[†]	**GPT-5.4**[‡]
One eval over 1$\times$256 tasks	$\sim$\$2.6 (¥18)	$\sim$\$23 (¥158)
GRPO 1 step (96 traj)	$\sim$\$1 (¥7)	$\sim$\$8.5 (¥59)
Code-level judging (this paper): \$0 (any scale)		

Table 16: Per-run cost comparison if a VLM judge were used

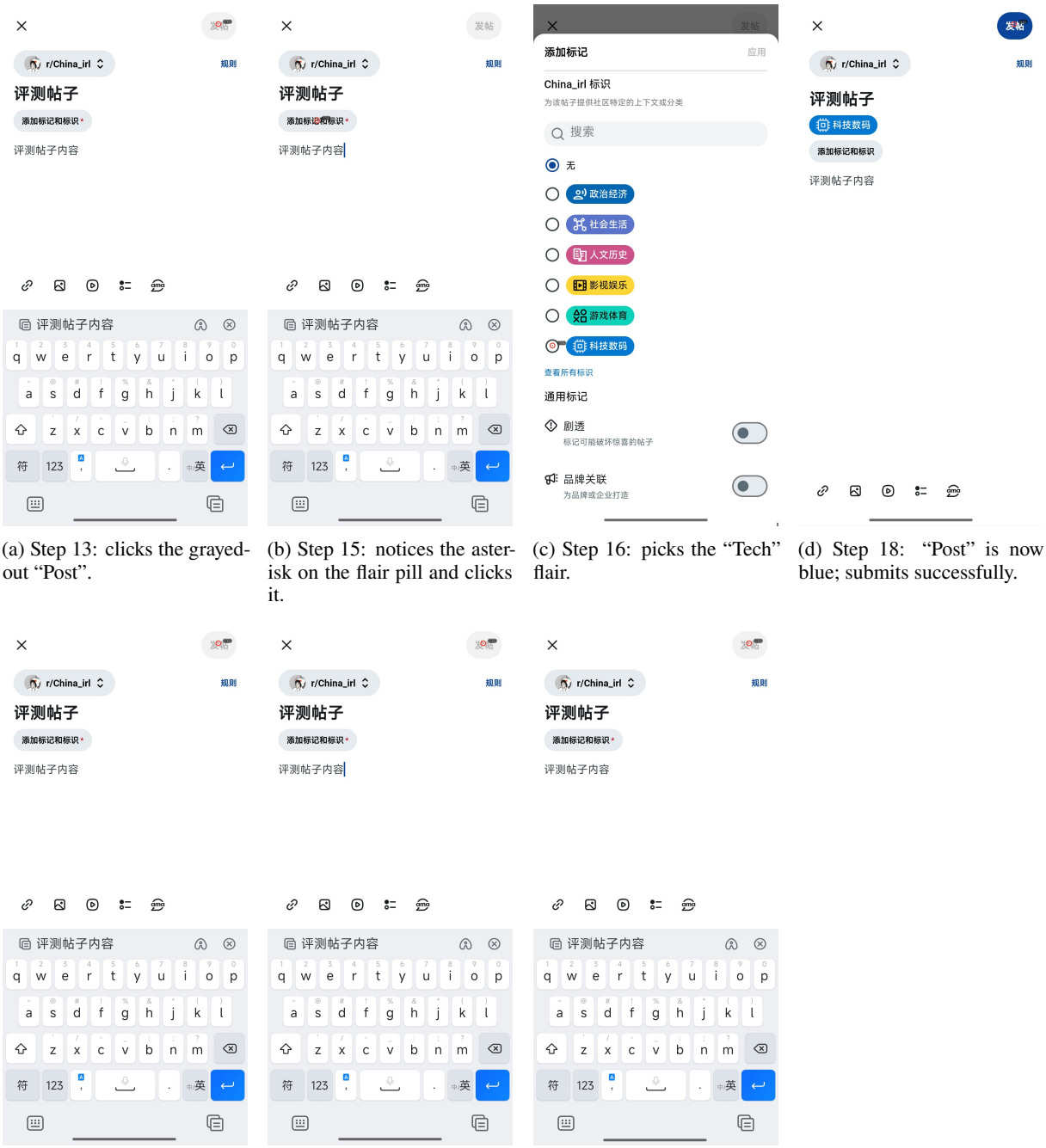
[†]Aliyun Bailian pricing ¥2/M input + ¥12/M output (within 256K). [‡]OpenAI API pricing for GPT-5.4: \$2.50/M input + \$15/M output, converted at a 7$\times$ exchange rate. The GPT-5.4 path is roughly 8.75$\times$ as expensive as the Qwen path.

Scale	**Qwen3.6-Plus**	**GPT-5.4**	**Code-level**
100 step (9.6K traj)	$\sim$\$100	$\sim$\$850	\$0
1K step (96K traj)	$\sim$\$1,000	$\sim$\$8.5K	\$0
10K step (960K traj)*	$\sim$\$10K	$\sim$\$85K	\$0

Table 17: Cumulative cost at large-scale RL training if a VLM judge were used

*Comparable to the "millions of interactive rollouts" training scale publicly reported by UI-TARS-2 (Wang et al., 2025).

We emphasize that the above is only the **VLM-judge API cost**: a complete real-environment RL training run also incurs the cost of cloud devices / emulator rentals. GUI-Genesis (Cao et al., 2026) reports that, in their WeChat mini-program experiments, the real-environment + VLM-reward configuration costs as much as \$240 per step (\$0.17/min cloud-device rental + \$0.005/trajectory VLM verification), and a single epoch (1K env $\times$ 12 rollout = 12K trajectories) costs approximately **\$28,000**; infrastructure costs clearly dominate in that setting. Because MobileGym runs as a browser environment on local machines, it avoids this category of infrastructure cost in our setup.



(a) Step 13: clicks the grayed-out "Post".

(b) Step 15: notices the asterisk on the flair pill and clicks it.

(c) Step 16: picks the "Tech" flair.

(d) Step 18: "Post" is now blue; submits successfully.

(e) Step 10: first click on the grayed-out "Post".

(f) Step 30: still clicking; identical reasoning.

(g) Step 60: 60-step budget exhausted.

From step 10 onward the base model produces byte-for-byte identical reasoning at every step (see Listing 1, bottom block), repeatedly clicking the disabled "Post" until the budget expires.

Figure 6: **Sim-to-Real OOD generalization on `Reddit_CreatePostToCommunity`.** The real-device r/China_irl community requires a flair tag before submission. **Top row**—trained model recovery (4 keyframes from a 22-step trajectory): step 13 clicks the grayed "Post"; step 15 attends to the asterisk on the `Add tags & flair` pill and infers that flair is a required field; step 16 picks the "Tech" flair; step 18 the "Post" button has turned blue and the model submits successfully. **Bottom row**—base model loop (3 keyframes spanning a 60-step trajectory): the screen and reasoning remain unchanged from step 10 onward, and the model clicks the disabled "Post" until the step budget is exhausted. Red `CLICK` badges mark the tap target of the model on each frame; device status bars have been cropped.