EDGE SYMPHONY: A HARNESS-AWARE BENCHMARK FOR HYBRID MOBILE GUI AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Mobile GUI agents that run entirely in the cloud raise privacy concerns and incur substantial API cost. Hybrid agents instead distribute inference between on-device and remote models through an agent harness, which decides where computation runs, what information is exchanged, and when control transfers between models. This raises a central question: what capability does a harness realize in hybrid mobile GUI execution, and at what deployment cost? We introduce EDGE SYMPHONY, a harness-aware benchmark that evaluates hybrid mobile GUI agents on physical smartphones with native on-device inference. EDGE SYMPHONY comprises 258 reproducible tasks across 23 applications, organized into six capability categories and extensible through an automated task curation workflow, and jointly measures Smartness and Cost. We also provide two reference harnesses, POCKETACT (fully local) and RELAYACT (hybrid). Experiments with EDGE SYMPHONY reveal three findings on hybrid agents invisible to model-centric evaluation: efficiency in converting model calls into actions bounds realized capability; harness design relocates cost rather than eliminating it; and model gains does not guarantee a better agent, which is also determined by the harness. These findings provide valuable insights into hybrid mobile GUI agents design.

1 INTRODUCTION

Mobile GUI agents are transitioning from research prototypes toward commercial agentic systems on smartphones, such as Doubao Phone Assistant and AutoGLM (ByteDance, 2025; Liu et al., 2024a). Motivated by concerns about UI privacy and the need for strong remote reasoning, recent work explores **hybrid mobile GUI agents** that distribute computation across smartphones and cloud services (Fan et al., 2025; Yi et al., 2026; Jiang & Huang, 2026). In these systems, limited usable context and reasoning capacity constrain the local models' performance on complex GUI tasks (Liu et al., 2024b; Huang et al., 2026a; Yang et al., 2025). These constraints make the **agent harness** critical for coordinating workload allocation, information exchange, model invocation, and control flow across local and remote models (Fan et al., 2025; Yi et al., 2026). Different harness designs thus yield different task performance and incur different deployment costs, even with the same underlying models. For example, frequent remote verification increases reliability but introduces additional communication latency and API cost (Guo et al., 2026; Yi et al., 2026). Therefore, benchmarking hybrid mobile GUI agents requires joint measurement of agent capability and deployment cost.

Existing mobile GUI agent benchmarks have progressively broadened both capability and cost evaluation. On the capability side, evaluation has expanded beyond end-to-end task success to interaction efficiency, difficulty-aware task organization, cross-app execution, and capability-level diagnosis (Rawles et al., 2025; Chen et al., 2025; Kong et al., 2026; Gong et al., 2026). On the cost side, recent benchmarks have also begun reporting execution-related indicators such as latency, token usage, and API expenditure (Chen et al., 2025; Li et al., 2026b). However, these measurements do not capture the device-side energy and computation incurred by local inference, nor attribute deployment cost to different local–remote execution paths. Separately, PhoneHarness makes the execution harness explicit in phone-use evaluation by coordinating GUI, CLI, and structured-tool actions (Li et al., 2026a), but its orchestration concerns action interfaces rather than the coordination of inference across device and cloud. Because existing benchmarks characterize realized capability, harness effects, and mobile inference cost in isolation, a central question remains unanswered: *what capability does a harness realize in hybrid mobile GUI execution, and at what deployment cost?*

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

Table 1: Comparison of representative mobile GUI agent benchmarks. **Platform** denotes evaluation on an emulator (**E**), real devices (**R**), or both. **On-dev. Inference** denotes model inference executed on the real smartphone instead of only GUI interactions, and **Cap.** denotes a capability-oriented task suite. '–' indicates that the number is not explicitly reported. Task success rate is common to all benchmarks and omitted.

Benchmark	# Apps	# Tasks	Platform	On-dev. Inference	Hybrid Execution	Harness Oriented	Cap. Task	Evaluation & Cost		
								Time	Energy	API Cost
AndroidWorld (Rawles et al., 2025)	20	116	E	✗	✗	✗	✗	✗	✗	✗
SPA-Bench (Chen et al., 2025)	68	340	E+R	✗	✗	✗	✗	✓	✗	✓
GUI-CEval (Li et al., 2026b)	201	4,028‡	R	✗	✗	✗	✓	✓	✗	✗
MobileBench-OL (Wu et al., 2026)	80	1,080§	R	✗	✗	✗	✓	✗	✗	✗
MobileWorld (Kong et al., 2026)	20	201	E	✗	✗	✗	✗	✗	✗	✗
VenusBench-Mobile (Gong et al., 2026)	27	149	E	✗	✗	✗	✓	✗	✗	✗
PhoneHarness (Li et al., 2026a)	-	124	E+R	✗	✗	✓	✗	✓	✗	✗
EDGE SYMPHONY (Ours)	**23**	**258**	**E+R**	✓	✓	✓	✓	✓	✓	✓

‡GUI-CEval counts a huge portion of trajectories-based offline agent evaluation or grounding questions.
§MobileBench-OL includes a Noise-Robust subset that injects environmental noise into the Base task set.

To answer this question, we introduce EDGE SYMPHONY, a harness-aware benchmark for hybrid mobile GUI agents. Specifically, EDGE SYMPHONY consists of three core components:

Harness-aware hybrid execution environment. EDGE SYMPHONY provides a hybrid execution environment that takes the harness as the object of evaluation on physical smartphones with native on-device inference. Users can plug in their own harness with local and remote models of their choice. All harnesses are executed and measured under the same protocol.

Capability-oriented diagnosis. EDGE SYMPHONY provides a capability-oriented task suite that complements aggregate success rates with per-capability diagnosis Ma et al. (2024). It organizes 258 reproducible tasks across 23 applications into six main capability categories, each of which can be evaluated independently to identify specific capability gaps and guide harness development. Most tasks are produced by the built-in automated curation workflow, which allows developers to extend the task suite toward new applications or specific tasks for their own demands.

Joint Smartness–Cost evaluation. EDGE SYMPHONY collects step-level telemetry from the same executions to jointly measure Smartness, i.e., how well tasks are completed across capability categories, together with Cost, i.e., the end-to-end latency, device energy consumption, and remote API expenditure. This connects the capabilities realized by each harness with the actual deployment costs incurred by its execution. We also provide two reference harnesses, POCKETACT for fully local execution and RELAYACT for hybrid execution, as baselines for controlled comparison and custom harness evaluation for developers.

We evaluate a range of fully local, hybrid, and fully remote GUI agent configurations with EDGE SYMPHONY and reveal three key findings about hybrid agent harness hidden from model-centric benchmarks: task success is limited not only by model capability but also by how efficiently a harness converts model calls into GUI actions; running more work on the phone does not remove deployment cost but trades API cost for device energy and latency; and a stronger remote model can even yield a weaker hybrid agent when the harness hands more execution back to the weaker on-device model. *We will release the entire benchmark code on GitHub after the review process.*

Contributions. Our contributions are summarized as follows:

- **Harness-centric real-device benchmark.** We introduce EDGE SYMPHONY, a real-device benchmark that provides a unified environment with native on-device inference support to evaluate hybrid harnesses by jointly measuring their Smartness and Cost.
- **A capability-oriented, extensible task suite.** We construct a task suite with 258 reproducible tasks spanning six capability categories and provide an automated curation workflow for expanding application and task coverage as needed.
- **Two reference harness baselines.** We provide POCKETACT (fully local) and RELAYACT (hybrid) as complementary baselines for controlled comparisons of custom harness designs.
- **Constructive findings of hybrid agent harness.** Our experiments reveal three harness-level effects on capability, cost, and model–harness interaction that model-centric evaluation cannot expose, offering rich insights into the design of hybrid mobile GUI agents.

2 Related Work

2.1 On-Device Mobile GUI Evaluation

Executable mobile GUI agent benchmarks primarily evaluate remotely hosted agents interacting with Android emulators, including Mobile-Bench, AndroidWorld, and MobileWorld (Deng et al., 2024; Rawles et al., 2025; Kong et al., 2026). More recent benchmarks such as GUI-CEval and MobileBench-OL evluate agents using physical smartphones (Li et al., 2026b; Wu et al., 2026), but model inference remains off-device. As a result, these benchmarks do not characterize the device-side execution cost of running the agent model itself, such as on-device inference latency, energy consumption, and local-model usage. In parallel, mobile inference benchmarks measure latency, memory, power, energy, and thermal behavior of models running on phones (Laskaridis et al., 2024; Li et al., 2025b; Xiao et al., 2024), but evaluate isolated inference workloads rather than long-horizon, closed-loop GUI agent tasks as defined in(Xu et al., 2026b). Edge Symphony bridges these settings by supporting on-device model inference and telemetry during reproducible GUI task execution on physical smartphones.

2.2 Hybrid Execution and Harness Evaluation

Recent mobile GUI agents increasingly coordinate on-device and remote models. CORE, EcoAgent, and OpenPhone explore different forms of device–cloud collaboration, while AutoDroid-V2 demonstrates fully on-device GUI automation (Fan et al., 2025; Yi et al., 2026; Jiang & Huang, 2026; Wen et al., 2025). These systems establish concrete local and hybrid execution designs, but evaluate particular agent architectures rather than providing a benchmark for comparing alternative orchestrations. ACE-BENCH systematically compares edge-only, cloud-only, and edge–cloud collaboration strategies over multi-step workspace agents (Anonymous, 2026). However, it does not cover mobile GUI agent tasks or measure realized energy and end-to-end latency on physical smartphones. Similarly, PawBench and Harness-Bench targets on the harnessed agentic workflow but are also not designed for benchmarking mobile GUI agent (The OpenJudge Team, 2026; Yao et al., 2026). Although PhoneHarness brings harness-aware evaluation to mobile agents through mixed GUI, CLI, and structured-tool actions (Li et al., 2026a), its harness primarily orchestrates among action interfaces, rather than coordinating on-device and remote model inference. Edge Symphony instead evaluates harness variation under controlled hybrid mobile GUI execution, where local–remote model orchestration and the resulting capability are measured.

3 The Edge Symphony Benchmark

3.1 Benchmark at a Glance

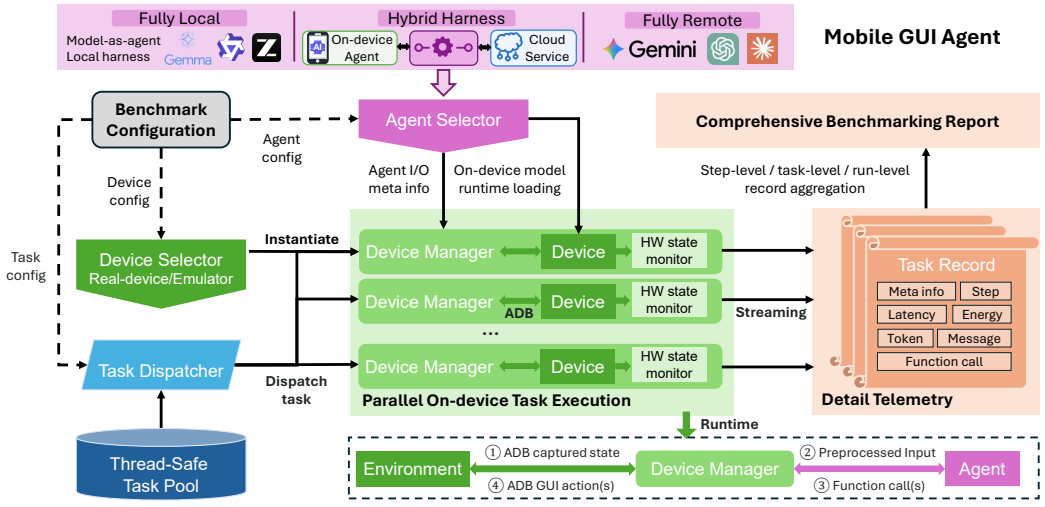


Figure 1: Overview of the Edge Symphony.

Benchmark Infrastructure. Building on the interactive environment architecture of AndroidWorld (Rawles et al., 2025), EDGE SYMPHONY extends it to support parallel execution on physical smartphones and native on-device model inference, served in our current implementation through LiteRT-LM (Google AI Edge, 2026). As illustrated in Figure 1, a host-side orchestrator manages a pool of executable tasks and available devices. A task dispatcher assigns tasks to device managers, each bound to one physical phone or emulator. The device manager bridges the device and agent with unified interface on each side. It transfers captured screen observations (screenshots and UI hierarchies) from devices to the agent as input, and translates agent-issued GUI function calls into ADB commands for GUI execution on devices (details in Appendix A.3). The unified interface also allows custom model replacement without modifying harness logic (Appendix A.1).

Agent Execution. During each task, the agent harness determines how the available models are used to interact with the device. A harness either run entirely on the device, for example by organizing an on-device model into separate roles for planning, execution, and verification, or selectively invoke a remote model for high-level planning and error recovery (Section 4). For the same task, different harnesses can follow different GUI trajectories due to differences in their control logic. All harnesses share the same base set of GUI actions and may introduce additional non-GUI actions as needed (Appendix A.3), so that differences in task outcome and deployment cost can be attributed to the harness rather than the GUI action interface

Evaluation Protocol. Following AndroidWorld (Rawles et al., 2025), each task defines a reproducible initial state, a natural-language goal, a step budget, and success-checking logic that inspects the resulting phone and database states rather than the action sequence. This allows different harnesses to reach the same valid outcome through different trajectories (Rawles et al., 2025; Chen et al., 2025). An episode ends when the agent calls completeness of goal or the step budget is exhausted. A task counts as successful only when completeness is reported together with success-checking logic holds. Reaching the goal state without declaring completion, or declaring completion without reaching it, counts as failure. Alongside the success result, each episode logs multi-dimensional metrics for evaluating Smartness and Cost as discussed in Section 3.3).

3.2 HARNESS-ORIENTED TASK SUITE

A large application set does not necessarily imply broad capability coverage: a suite can span many apps but still consist mostly of short, direct interactions. Real-world user requests instead often require cross-app information transfer, long-horizon planning, conditional execution, disambiguation, and recovery from unsuccessful attempts (Kong et al., 2026; Gong et al., 2026). Meeting these demands falls to the harness rather than to any single model call (Lee et al., 2024; Li et al., 2025a; Wanyan et al., 2025), and in a hybrid harness each demand also raises the question of which model handles it and how information is exchanged. EDGE SYMPHONY therefore evaluates harnesses against these demands through 258 tasks across 23 applications, organized into six capability categories that each stress a different harness function, with most tasks produced by our automated curation workflow in Section 3.2.2. Tasks that pass automated validation are manually executed end-to-end to confirm that they can be completed from the specified initial state., and are further reviewed for goal clarity, realism, and capability assignment. The construction details are provided in Appendix B. Each category can be run independently, allowing developers to diagnose specific capability gaps and iterate on targeted harness improvements without full suite execution.

3.2.1 CAPABILITY TAXONOMY

EDGE SYMPHONY defines six capability categories (Table 2). Basic Knowledge (BK) covers the single-app operations on which all other categories build. The tasks in BK form the basic suite (Appendix B.2); the remaining five categories each correspond to one of the demands introduced above and together form the advanced suite (Appendix B.3). While a task can draw on multiple capabilities, it is categorized by the capability it is primarily designed to evaluate.

Basic Knowledge (BK). This category tests basic usage of each app. Note that BK accounts near half of the task suite as each App exposes multiple basic functions and interaction patterns.

Semantic Migration (SM). Users may need information from one app transformed into an action or record in another, such as making a reservation following the request from a message Liu et al. (2026). SM examines semantic mapping and memory across apps.

Category	Capability Focus
Basic Knowledge (BK)	App-function knowledge and GUI grounding
Semantic Migration (SM)	Cross-app memory and semantic transformation
Long-horizon Planning (LH)	Goal decomposition and progress tracking
Constraint-aware Execution (CA)	Conditional reasoning and state protection
Ambiguous Content (AM)	Contextual disambiguation and intent inference
Interruption and Recovery (IR)	Disruption detection and progress recovery

Table 2: Capability taxonomy in EDGE SYMPHONY.

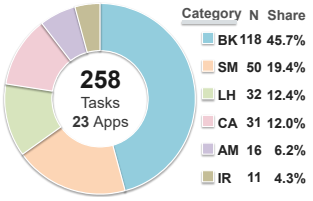


Figure 2: Category distribution.

Long-horizon Planning (LH). Complicated request may delegate workflows involving multiple dependent stages. LH tests goal decomposition, progress tracking, and consistency across subgoals.

Constraint-aware Execution (CA). Users may specify conditions on when to act and what should be preserved. CA tests state inspection, conditional reasoning, and constraint preserving execution.

Ambiguous Content (AM). Users may refer to an target from an incomplete or ambiguous reference, such as people, files, or actions without unique identifiers. AM tests contextual disambiguation and evidence-grounded intent inference.

Interruption and Recovery (IR). Apps may unexpectedly terminate or enter an incorrect state during task execution, interrupting the expected execution path. IR tests disruption detection, progress reconstruction, and recovery without duplicating completed work.

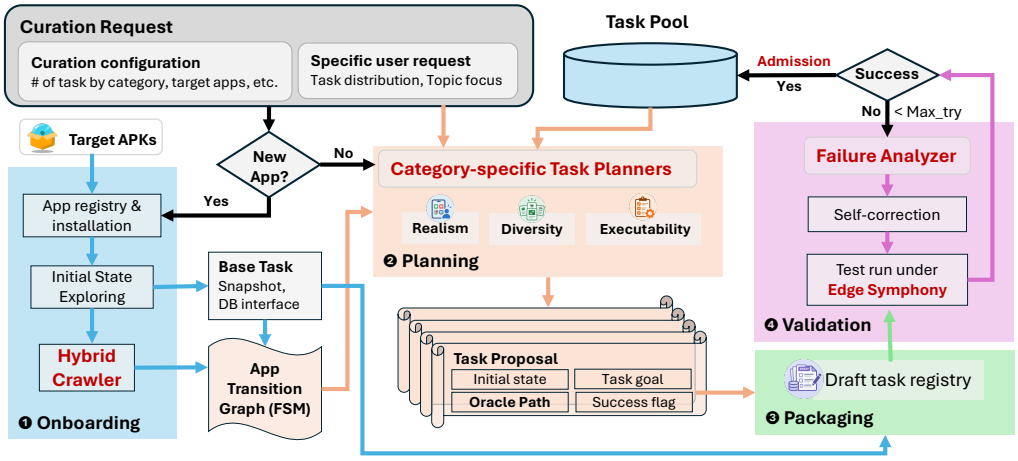


Figure 3: Overview of the Automated Task Curation Workflow.

3.2.2 AUTOMATED TASK CURATION

EDGE SYMPHONY provides an automated task curation pipeline (Figure 3) that lets developers extend the task suite toward their own needs, such as new applications or specific task requirements. It converts a brief, high-level curation request into benchmark-compatible task instances without manual intervention, through four stages: application onboarding, capability-conditioned generation, packaging, and validation and admission. Details and examples are provided in Appendix C.

(1) Application onboarding. Unregistered apps in the curation request are first onboarded as reusable benchmark components. EDGE SYMPHONY installs the app, grants all runtime permissions, and dismisses first-run dialogs until it reaches a stable home screen. It then creates a *base task* that provides a clean-state snapshot for reset and a database interface for seeding initial states and checking success. From this clean state, a script–agent *hybrid crawler* handles open-ended interactions such as form filling and builds an *app transition graph*, whose nodes are UI states and whose edges are executable actions (Appendix C.2).

(2) Capability-conditioned generation. From the app transition graph and the curation request, a category-specific planner generates *task proposals* with a dedicated `skill` per capability category, following three criteria: *realism* (plausible, goal-directed user requests), *diversity* (no overlap with the task pool or other candidates), and *executability* (a reachable task path). Each proposal specifies

an initial state, a task goal, a success check, and an *oracle path*, i.e., a subgraph of the transition graph leading to the goal (Appendix C.3).

(3) Executable packaging. A coder agent compiles each proposal into a draft task that inherits the app-specific base task, so it can be registered through the same interface as existing tasks. The oracle path is encoded into the task goal as execution hints for validation. Additional state construction, data hooks, or app-specific support are resolved in the next stage.

(4) Validation and admission. Each draft is test-run under EDGE SYMPHONY by a strong remote agent guided by the oracle path, and passes only if it can be reproducibly initialized, completed, and verified by its success check. After each failed run, a failure analyzer identifies the root cause and revises the draft. Passed drafts are admitted to the task pool with the oracle path removed, while those still failing after several attempts are discarded (Appendix C.5).

3.3 JOINT SMARTNESS–COST EVALUATION

EDGE SYMPHONY measures Smartness and Cost from the same executions through step-level telemetry. **Step** is a single local or remote model call and serves as the smallest unit of measurement: each step record captures the smartness and cost metrics, local or remote agent I/O messages, issued actions, and runtime anomalies. Each device manager streams its step record to the host once the step completes, so the host is able to track all parallel executions in real time.

Smartness. Smartness characterizes the capability a mobile GUI agent realizes in the task execution. An agentic phone is expected to complete user requests reliably, across diverse kinds of requests, and without unnecessary model calls. We therefore measure:

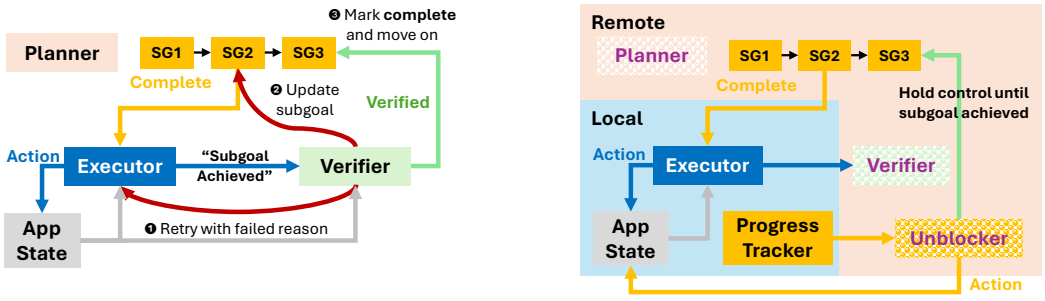
- **Task success:** the fraction of tasks completed under the evaluation protocol in Section 3.1, the standard measure of GUI agent performance (Rawles et al., 2025; Chen et al., 2025).
- **Capability performance:** success rates reported separately for the six capability categories (Table 2). As Basic Knowledge makes up half of the suite, category-level results expose weaknesses in advanced capabilities that aggregate success potentially conceals.
- **Interaction efficiency:** the average number of steps per task (Xu et al., 2026a; Wang et al., 2026). A capable model is able to issue multiple valid function calls within one step and avoids redundant calls. This is essential for evaluating harness efficiency, as calls that produce no GUI action still consume significant inference resources.

Cost. Cost characterizes the deployment burden paid for that capability, which in hybrid execution falls on the user's time, the phone's battery, and the remote-inference bill. All cost metrics are averaged over all executions, including failed and timed-out tasks:

- **End-to-end latency:** the wall-clock time from the first step to episode termination, covering inference, network, harness logic, and GUI execution. Task initialization and watchdog recovery (Appendix A.1.2) are excluded.
- **Device energy:** whole-device energy measured through the On-Device Power Rails Monitor over a window synchronized with that of latency (Appendix A.2.1), which also captures the energy the phone spends communicating with and waiting for remote models.
- **API cost:** the monetary cost of remote model calls, computed from providers' list prices as in Appendix A.4). Unlike benchmarks that use token counts as a cost proxy (Gong et al., 2026), we log tokens for analysis only, since identical token usage can lead to very different API costs (e.g., $10.00 for Claude Fable 5.1 vs. $1.5 for Gemini 3.5 Flash per million input tokens (Anthropic, 2026; Google, 2026)).

4 REFERENCE HARNESSES

EDGE SYMPHONY provides two strong reference harnesses, POCKETACT for fully local execution and RELAYACT for hybrid execution, as anchors for controlled comparison. Comparing models under a fixed reference harness isolates model capability, whereas comparing a custom harness against the references under fixed models isolates orchestration effects.



(a) PocketAct: fully local harness. (b) RelayAct: hybrid harness.

Figure 4: Illustration of two reference harnesses provided by EDGE SYMPHONY.

Shared skeleton. Inspired by existing agent harness designs (Wang et al., 2024; Yao et al., 2023; Shinn et al., 2023), both reference harnesses organize execution around three roles: a planner, an executor, and a verifier (Figure 4). This design reflects the limited context and reasoning capacity of on-device models, spreading the reasoning burden across roles rather than a single call. The planner decomposes the user goal and the initial observation into draft subgoals. The executor receives a compact input and focuses on grounding: it translates the current subgoal into GUI actions, reports its completion with `subgoal_achieved` (Appendix A.3). The verifier reasons over the longer execution history without handling the GUI actions (Dai et al., 2026; Lee et al., 2025), confirming whether a reported subgoal is achieved or diagnosing a false report to executor.

POCKETACT: local planning and recovery. POCKETACT runs all three roles on a single on-device model, so no model input leaves the phone. When the executor stalls or falls into a loop, the verifier replans the stalled subgoal or returns corrective feedback (Appendix D.2).

RELAYACT: remote unblocker. RELAYACT keeps the executor and verifier on the phone but assigns initial planning and recovery to a remote model. **A central challenge in hybrid execution is deciding when control should move between local and remote models.** The local executor may keep acting confidently without making progress, while a remote model, once invoked, needs a clear point at which to return control. RELAYACT uses the subgoal as the unit of both progress measurement and control transfer. A scripted progress tracker escalates when the current subgoal stalls. The remote unblocker then takes over GUI interaction, diagnoses the blocker from the current screen and the tracker's loop/stall diagnosis, and completes the stalled subgoal. It returns control by emitting an updated plan of the remaining subgoals, from which the local executor resumes (Appendix D.3). This concentrates remote assistance on difficult problems and keeps routine execution local.

5 EVALUATION

5.1 EXPERIMENTAL SETUP

Models and environment. Evaluation are conducted on physical Pixel 9 Pro smartphones. To compare harness designs without attributing differences to method-specific pretrained grounding or execution models, all configurations share the same models: local inference uses Gemma4-E4B-LiteRT-LM on the phone's GPU with the common runtime settings in Appendix A.1, and remote inference uses Gemini-3.5-Flash unless otherwise noted. For reproducibility, all local and remote model use greedy decoding (Variance analysis in Appendix E.1. They also share the same base GUI action space, retaining harness-specific non-GUI function calls where applicable (Appendix A.3). Local models take screenshots supplemented with pruned XML as input, whereas remote models receive screenshots only to reduce API cost (Input ablation in Appendix E.2). Every configuration follows the same task settings with 40-step budget (Sensitivity analysis in Appendix E.3).

Baselines. Local-E2E and Remote-E2E prompt a single model to generate GUI actions without a harness. **M3A** (Rawles et al., 2025) runs ReAct-style action selection and post-action reflection on the shared local model, feeding summarized outcomes back into subsequent decisions. **SeeAct-V** (Gou et al., 2025) separates planning from grounding: the remote model describes the next action and its target, which the local model grounds to screen coordinates. **EcoAgent** (Yi et al., 2026) uses a remote Planning Agent to generates action plans with expected outcomes. and a observer checks outcomes from local executor and reports failures back for remote reflection and replanning.

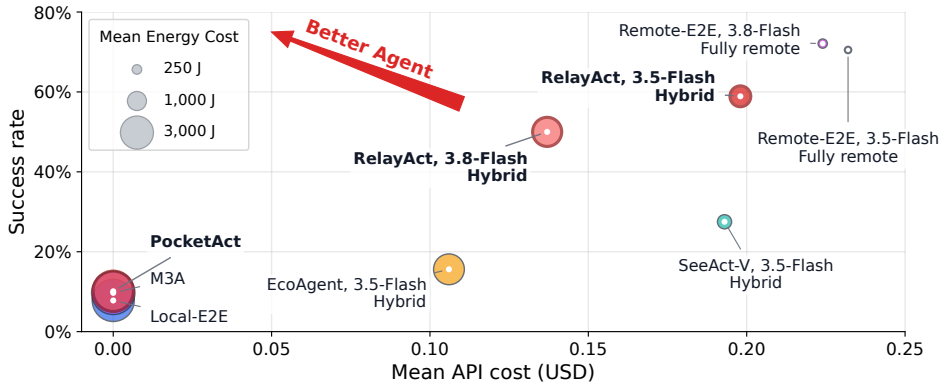


Figure 5: **Visualized Smartness vs. Cost.** Configurations closer to the upper left with smaller bubbles realize more capability at a lower deployment cost. Bold labels denote our reference harnesses.

Table 3: **Major results on EDGE SYMPHONY.**

Configuration	Remote model	Smartness					Cost		
		Overall SR (%)	BK AM	SM IR	LH CA	Steps	Latency (s)	Energy (J)	API Call ($)
Fully local									
Local-E2E	–	7.8	15.3 6.2	0.0 0.0	0.0 3.2	19.2†	1326.7	4856.5	0.0
M3A (Rawles et al., 2025)	–	9.7	10.3 18.8	4.0 18.2	3.1 16.1	34.1	1385.6	5168.5	0.0
PocketAct (Ours)	–	10.1	14.4 6.2	4.0 0.0	0.0 19.4	33.6	1274.6	4481.6	0.0
Hybrid									
SeeAct-V (Gou et al., 2025)	3.5-Flash	27.5	36.4 25.0	20.0 36.4	12.5 19.4	32.6	239.8	553.2	0.193
EcoAgent (Yi et al., 2026)	3.5-Flash	15.6	23.7 18.8	8.3 0.0	3.1 12.9	28.3	760.5	2604.9	0.106
RelayAct (Ours)	3.5-Flash	58.9	62.7 56.2	68.0 63.6	21.9 67.7	25.6	435.2	1195.1	0.198
RelayAct (Ours)	3.8-Flash	50.0	59.3 50.0	50.0 27.3	15.6 58.1	27.1	1586.8	2325.0	0.137
Fully remote									
Remote-E2E	3.5-Flash	70.5	81.4 56.2	76.0 81.8	31.2 64.5	19.0	145.8	142.9	0.232
Remote-E2E	3.8-Flash	72.1	84.7 68.8	74.0 81.8	28.1 64.5	19.6	233.3	236.1	0.224

†Local-E2E often terminates early as it emits malformed outputs due to heavy reasoning load.

5.2 BENCHMARKING AGENT HARNESSES

Table 3 and Figure 5 summarize the Smartness and Cost of all configurations. Task success spreads widely across inference placements. Fully local agents remain below 11%, with POCKETACT achieving the highest success among them at the lowest latency and device energy. Under the same local–remote model pair, hybrid agents range from 15.6% to 58.9%, and Remote-E2E reaches 70.5%. SeeAct-V and RELAYACT incur nearly identical API cost ($0.193 vs. $0.198) yet differ by 2.1× in task success. Across capability categories, Long-horizon Planning is the weakest for every configuration (at most 28.6%), whereas Interruption and Recovery varies most across hybrid agents (0.0% to 63.6%). API cost and device-side cost move in opposite directions: from Remote-E2E to hybrid and fully local agents, API cost falls from $0.232 to $0.106–$0.198 and zero, while device energy rises from 142.9 J to 553–2,605 J and 4,482–5,169. We additionally evaluate RELAYACT and Remote-E2E with Gemini-3.8-Flash (3.8-Flash), as discussed in Section 5.3.

5.3 KEY INSIGHTS

Harness efficiency bounds realized capability. Step-level traces show that hybrid failures mostly come from exhausting the step budget rather than from weaker reasoning or false completion. Because EDGE SYMPHONY counts every model call as a step, harnesses that spend about three calls per GUI action (e.g., planning, grounding, and summarization in SeeAct-V) can take only about 11 actions within the 40-step budget, whereas the tasks they fail typically require about 20 actions under Remote-E2E. The same pressure underlies most hybrid failures on Long-horizon Planning: these tasks nearly exhaust the budget even for Remote-E2E, and in RELAYACT the local executor often consumes 40–60% of the budget before the remote unblocker takes over. Realized capability is therefore bounded not only by model reasoning but also by how efficiently the harness converts model calls into effective GUI actions, a capability that evaluating the model alone cannot reveal.

Harness placement relocates cost rather than eliminating it. As the opposite trends of API and device-side cost show, offloading transfers the deployment burden from the remote bill to the user's time and battery rather than eliminating it. Device energy tracks how much work runs on the phone rather than how efficient the harness is. SeeAct-V illustrates this: it runs only visual grounding locally, emitting a coordinate pair per call, while planning and summarization, about 70% of its model calls, run remotely as the device idles. Each of its on-device calls thus costs about 48 J, versus 110–145 J for agents that generate full actions or verifications locally, so its low total energy reflects less local work rather than higher efficiency. Reporting only API cost or token usage would characterize a configuration by one side of this trade-off; measuring both from the same executions lets EDGE SYMPHONY expose where each configuration's cost is actually paid.

Model gains hinge on model–harness interaction Table 3 reveals a reversal between the two remote models: as standalone agents, 3.8-Flash slightly outperforms 3.5-Flash (72.1% vs. 70.5%), yet under RELAYACT it lowers success from 58.9% to 50.0%. The reversal reflect how each model follows the harness's control-transfer protocol: 3.8-Flash more faithfully returns control after resolving the stalled subgoal, whereas 3.5-Flash often continues executing subsequent subgoals remotely, so under 3.8-Flash more execution shifts back to the weaker on-device model. Step-level traces support this explanation (Table 4): with 3.8-Flash, unblocker sessions are shorter (6.1 vs. 10.6 steps) and more often end with an updated plan (64.3% vs. 42.3%), the unblocker's step share falls from 60.5% to 46.0%, and the share of successful tasks completed by the unblocker drops from 83.6% to 66.7%.

Table 4: **Trace comparison of RELAYACT with 3.5-Flash and 3.8-Flash.** Parentheses are percentages. Δ: 3.8-Flash minus 3.5-Flash.

Metric	3.5-Flash	3.8-Flash	Δ
Successful tasks	152 (58.9)	129 (50.0)	−23
Step allocation			
Unblocker	3,794 (60.5)	3,039 (46.0)	−755
Executor	2,484 (39.5)	3,565 (54.0)	+1,081
Unblocker sessions			
Steps / session	10.6	6.1	−4.5
handbacks	152 (42.3)	320 (64.3)	+168
Successful tasks completed by			
Unblocker	127 (83.6)	86 (66.7)	−41
Executor	25 (16.4)	43 (33.3)	+18

This reversal indicates that upgrading the model thus does not guarantee a better hybrid agent, since realized capability is determined jointly by the models and the harness that coordinates them, an effect invisible to model-centric evaluation. By recording step-level telemetry for every call, EDGE SYMPHONY reveals where control resides and which model completes the task. This allows developers to attribute a performance change to the harness rather than the model and target optimizations on the control handover and agent robustness.

6 CONCLUSION

We introduced EDGE SYMPHONY, a harness-aware benchmark that jointly evaluates the Smartness and Cost of hybrid mobile GUI agents with native on-device inference on physical smartphones. Its capability-oriented tasks, automated curation workflow, and reference harnesses support reproducible and extensible evaluation, and its joint measurement reveals how the capability realized by each harness relates to its actual deployment cost. We believe EDGE SYMPHONY can serve as a shared foundation for developing capable and cost-efficient hybrid mobile GUI agents.

AI USE STATEMENT

Automated task curation. A portion of the task suite described in Section 3.2 was generated by Gemini-3.1pro/3.8-flash with our proposed automated task curation pipeline (Section 3.2.2). These tasks underwent execution-based validation and manual review before release. The proportion of AI-generated tasks and representative examples are reported in Appendix B, with pipeline details provided in Appendix C. We will release this workflow after the review process for developers to support customized EDGE SYMPHONY-compatible task curation and benchmark extension.

Development and manuscript assistance. We used Geimini-3.1pro/3.5Flash to assist with initial code drafting, identifying and completing missing implementation details, and debugging. Before release, all code underwent extensive manual review and readability-focused refactoring by the authors. We used GPT-5.6 Sol/6 Astra to assist with manuscript revision, including identifying potential weaknesses and omissions in the writing and drafting writing improvements for clarity and readability. In summary, we used generative AI tools for implementing methods, and supporting qualitative and thematic data analysis; we have not used them for developing theoretical models or conceptual frameworks, proposing or refining hypotheses, designing or providing feedback on research methodology or experiments, assisting with translation, cleaning and reformatting datasets, or interpreting results. Formulating mathematical claims, providing critical ingredients for proving mathematical claims, and assisting in the writing of proofs are not applicable to this work. The authors take full responsibility for the code, benchmark content, and final manuscript.

REPRODUCIBILITY STATEMENT

Benchmark code, task suite, and reference harnesses will be released after the review process. and all results are reproducible. Fixed application versions, reproducible task initialization, and programmatic outcome verification allow different harnesses to be evaluated under consistent conditions. The supporting implementation details are documented in Section 5.1 and Appendices.

REFERENCES

Android Open Source Project. Power stats hal, 2026. URL https://source.android.com/docs/core/power/power-stats-hal.

Anonymous. Empowering edge agents: A systematic analysis of edge–cloud collaboration in LLM agent execution. Anonymous ACL submission; citation metadata to be updated after public release, 2026.

Anthropic. Claude API pricing, 2026. URL https://platform.claude.com/docs/en/about-claude/pricing.

ByteDance. Doubao Phone Assistant Privacy and Security White Paper, 2025. URL https://lf3-o-website-sdk.doubaocdn.com/obj/o-website/public/assets/pdf/doubao_white_paper_v1.pdf.

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. SPA-Bench: A comprehensive benchmark for smartphone agent evaluation. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/9a75f4dd9679aa4ff80a4e6f0a1dc700-Abstract-Conference.html.

Gaole Dai, Shiqi Jiang, Ting Cao, Yuanchun Li, Yuqing Yang, Rui Tan, Mo Li, and Lili Qiu. V-Droid: Advancing mobile GUI agent through generative verifiers. In *Proceedings of the 32nd Annual International Conference on Mobile Computing and Networking*. ACM, 2026. doi: 10.1145/3795866.3796681. URL https://doi.org/10.1145/3795866.3796681.

Shihan Deng, Weikai Xu, Hongda Sun, Wei Liu, Tao Tan, Jianfeng Liu, Ang Li, Jian Luan, Bin Wang, Rui Yan, and Shuo Shang. Mobile-Bench: An evaluation benchmark for LLM-based mobile agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8813–8831. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.478. URL https://aclanthology.org/2024.acl-long.478/.

Gucongcong Fan, Chaoyue Niu, Chengfei Lyu, Fan Wu, and Guihai Chen. CORE: Reducing UI exposure in mobile agents via collaboration between cloud and local LLMs. In *Advances in Neural Information Processing Systems*, volume 38, pp. 76608–76644, 2025. doi: 10.52202/085713-2311. URL https://openreview.net/forum?id=klOr9y9nMU.

Georgi Gerganov and ggml-org contributors. llama.cpp: LLM inference in C/C++, 2023. URL https://github.com/ggml-org/llama.cpp.

Yichen Gong, Zhuohan Cai, Sunhao Dai, Yuqi Zhou, Zhangxuan Gu, Changhua Meng, and Shuheng Shen. VenusBench-Mobile: A challenging and user-centric benchmark for mobile GUI agents with capability diagnostics. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2604.06182.

Google. Snippet UiAutomator, 2023. URL https://github.com/google/snippet-uiautomator.

Google. Gemini developer API pricing, 2026. URL https://ai.google.dev/gemini-api/docs/pricing.

Google AI Edge. LiteRT-LM: On-device inference framework for large language models, 2026. URL https://github.com/google-ai-edge/LiteRT-LM.

Boyu Gou, Ruohan Wang, Boyuan Zheng, Yanan Xie, Cheng Chang, Yiheng Shu, Huan Sun, and Yu Su. Navigating the digital world as humans do: Universal visual grounding for GUI agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=kxnoqaisCT.

Linqiang Guo, Wei Liu, Li Gu, Yang Wang, and Tse-Hsun (Peter) Chen. StepReflect: Structured UI transition reflection for mobile GUI agents. *arXiv preprint arXiv:2608.05587*, 2026. URL https://arxiv.org/abs/2608.05587.

Hanxian Huang, Igor Fedorov, Andrey Gromov, Bernard Beckerman, Naveen Suda, David Eriksson, Maximilian Balandat, Rylan Conway, Patrick Huber, Chinnadhurai Sankar, Ayushi Dalmia, Zechun Liu, Lemeng Wu, Tarek Elgamal, Adithya Sagar, Vikas Chandra, and Raghuraman Krishnamoorthi. MobileLLM-flash: Latency-guided on-device LLM design for industry scale deployment. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)*, pp. 749–760. Association for Computational Linguistics, 2026a. doi: 10.18653/v1/2026.acl-industry.51. URL https://aclanthology.org/2026.acl-industry.51/.

Runxi Huang, Liyu Zhang, Shengzhong Liu, and Xiaomin Ouyang. MobileExplorer: Accelerating on-device inference for mobile GUI agents via online exploration. *arXiv preprint arXiv:2605.26546*, 2026b. URL https://arxiv.org/abs/2605.26546.

Yangqin Jiang and Chao Huang. OpenPhone: Mobile agentic foundation models. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 30362–30380. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.1518. URL https://aclanthology.org/2026.findings-acl.1518/.

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. 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)*, pp. 6142–6167. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-long.278. URL https://aclanthology.org/2026.acl-long.278/.

Stefanos Laskaridis, Kleomenis Katevas, Lorenzo Minto, and Hamed Haddadi. MELTing Point: Mobile evaluation of language transformers. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, pp. 890–907. ACM, 2024. doi: 10.1145/3636534.3690668. URL https://doi.org/10.1145/3636534.3690668.

Jungjae Lee, Dongjae Lee, Chihun Choi, Youngmin Im, Jaeyoung Wi, Kihong Heo, Sangeun Oh, Sunjae Lee, and Insik Shin. VeriSafe Agent: Safeguarding mobile GUI agent via logic-based action verification. In *Proceedings of the 31st Annual International Conference on Mobile Computing and Networking*. ACM, 2025. doi: 10.1145/3680207.3765248. URL https://doi.org/10.1145/3680207.3765248.

Sunjae Lee, Junyoung Choi, Jungjae Lee, Munim Hasan Wasi, Hojun Choi, Steven Y. Ko, Sangeun Oh, and Insik Shin. MobileGPT: Augmenting LLM with human-like app memory for mobile task automation. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, pp. 1119–1133. ACM, 2024. doi: 10.1145/3636534.3690682. URL https://doi.org/10.1145/3636534.3690682.

Chenxin Li, Zhengyao Fang, Zhengyang Tang, Pengyuan Lyu, Xingran Zhou, Xin Lai, Fei Tang, Liang Wu, Yiduo Guo, Weinong Wang, Junyi Li, Yi Zhang, Yang Ding, Huawen Shen, Sunqi Fan, Shangpin Peng, Zheng Ruan, Anran Zhang, Benyou Wang, Chengquan Zhang, and Han Hu. PhoneHarness: Harnessing phone-use agents through mixed GUI, CLI, and tool actions. *arXiv preprint arXiv:2606.14832*, 2026a. URL https://arxiv.org/abs/2606.14832.

Ning Li, Xiangmou Qu, Jiamu Zhou, Muning Wen, Kounianhua Du, Xingyu Lou, Qiuying Peng, Jun Wang, and Weinan Zhang. MobileUse: A hierarchical reflection-driven GUI agent for autonomous mobile operation. In *Advances in Neural Information Processing Systems*, volume 38, pp. 40361–40388, 2025a. doi: 10.52202/085713-1347. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/3994410d63ec68ce9a66011a34c9a2c4-Abstract-Conference.html.

Yang Li, Yuchen Liu, Haoyu Lu, Zhiqiang Xia, Hongzhen Wang, Kaiyang Han, Changpeng Yang, Jinyang Wu, Jiaming Xu, Runyu Shi, and Ying Huang. GUI-CEval: A hierarchical and comprehensive chinese benchmark for mobile GUI agents. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 20303–20312, 2026b. URL https://openaccess.thecvf.com/content/CVPR2026/html/Li_GUI-CEval_A_Hierarchical_and_Comprehensive_Chinese_Benchmark_for_Mobile_GUI_CVPR_2026_paper.html.

Yilong Li, Jingyu Liu, Hao Zhang, Badri Narayanan Murali Krishnan, Utkarsh Sharma, Shuai Zhang, Yijing Zeng, Jayaram Raghuram, and Suman Banerjee. PalmBench: A comprehensive benchmark of compressed large language models on mobile platforms. In *International Conference on Learning Representations*, 2025b. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/a647405740b28a61311ac9cff28772e5-Abstract-Conference.html.

Guangyi Liu, Pengxiang Zhao, Yaozhen Liang, Qinyi Luo, Shunye Tang, Yuxiang Chai, Weifeng Lin, Han Xiao, WenHao Wang, Siheng Chen, Zhengxi Lu, Gao Wu, Hao Wang, Liang Liu, and Yong Liu. MemGUI-Bench: Benchmarking memory of mobile GUI agents in dynamic environments. *arXiv preprint arXiv:2602.06075*, 2026. URL https://arxiv.org/abs/2602.06075.

Xiao Liu, Bo Qin, Dongzhu Liang, Guang Dong, Hanyu Lai, Hanchen Zhang, Hanlin Zhao, Iat Long Iong, Jiadai Sun, Jiaqi Wang, et al. AutoGLM: Autonomous foundation agents for GUIs. *arXiv preprint arXiv:2411.00820*, 2024a. URL https://arxiv.org/abs/2411.00820.

Zechun Liu, Changsheng Zhao, Forrest Iandola, Chen Lai, Yuandong Tian, Igor Fedorov, Yunyang Xiong, Ernie Chang, Yangyang Shi, Raghuraman Krishnamoorthi, Liangzhen Lai, and Vikas Chandra. MobileLLM: Optimizing sub-billion parameter language models for on-device use cases. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32431–32454. PMLR, 2024b. URL https://proceedings.mlr.press/v235/liu24ce.html.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. AgentBoard: An analytical evaluation board of multi-turn LLM agents. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-2365. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/877b40688e330a0e2a3fc24084208dfa-Abstract-Datasets_and_Benchmarks_Track.html.

OpenAI. OpenAI API pricing, 2026. URL https://developers.openai.com/api/docs/pricing.

Chris 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. AndroidWorld: A dynamic benchmarking environment for autonomous agents. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/01a83bc2f2732a58e6aa731e659e7101-Abstract-Conference.html.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652, 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

The OpenJudge Team. PawBench: A benchmark for evaluating LLM $\times$ harness performance, June 2026. URL https://github.com/agentscope-ai/PawBench.

Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. Mobile-Agent-v2: Mobile device operation assistant with effective navigation via multi-agent collaboration. In *Advances in Neural Information Processing Systems*, volume 37, pp. 2686–2710, 2024. doi: 10.52202/079017-0088. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/0520537ba799d375b8ff5523295c337a-Abstract-Conference.html.

Xuehui Wang, Zhenyu Wu, JingJing Xie, Zichen Ding, Bowen Yang, Zehao Li, Zhaoyang Liu, Qingyun Li, Xuan Dong, Zhe Chen, Weiyun Wang, Xiangyu Zhao, Jixuan Chen, Haodong Duan, Tianbao Xie, Chenyu Yang, Shiqian Su, Yue Yu, Yanting Zhang, Xiangyu Yue, Weijie Su, Xizhou Zhu, Wei Shen, Jifeng Dai, and Wenhai Wang. MMBench-GUI: A unified hierarchical evaluation framework for multi-platform GUI agents. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 6239–6248, 2026. URL https://openaccess.thecvf.com/content/CVPR2026/html/Wang_MMBench-GUI_A_Unified_Hierarchical_Evaluation_Framework_for_Multi-Platform_GUI_Agents_CVPR_2026_paper.html.

Yuyang Wanyan, Xi Zhang, Haiyang Xu, Haowei Liu, Junyang Wang, Jiabo Ye, Yutong Kou, Ming Yan, Fei Huang, Xiaoshan Yang, Weiming Dong, and Changsheng Xu. Look before you leap: A GUI-Critic-R1 model for pre-operative error diagnosis in GUI automation. In *Advances in Neural Information Processing Systems*, volume 38, pp. 4405–4427, 2025. doi: 10.52202/085713-0137. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/05f7fb7bc9a3cc4608f1c6f2cdc79eae-Abstract-Conference.html.

Hao Wen, Shizuo Tian, Borislav Pavlov, Wenjie Du, Yixuan Li, Ge Chang, Shanhui Zhao, Jiacheng Liu, Yunxin Liu, Ya-Qin Zhang, and Yuanchun Li. AutoDroid-V2: Boosting SLM-based GUI agents via code generation. In *Proceedings of the 23rd Annual International Conference on Mobile Systems, Applications and Services*, pp. 223–235. ACM, 2025. doi: 10.1145/3711875.3729134. URL https://doi.org/10.1145/3711875.3729134.

Qinzhuo Wu, Zhizhuo Yang, Hanhao Li, Pengzhi Gao, Wei Liu, and Jian Luan. MobileBench-OL: A comprehensive chinese benchmark for evaluating mobile GUI agents in real-world environment. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 13659–13693. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.668. URL https://aclanthology.org/2026.findings-acl.668/.

Jie Xiao, Qianyi Huang, Xu Chen, and Chen Tian. Large language model performance benchmarking on mobile platforms: A thorough evaluation. *arXiv preprint arXiv:2410.03613*, 2024. URL https://arxiv.org/abs/2410.03613.

Weikai Xu, Zhizheng Jiang, Yuxuan Liu, Pengzhi Gao, Wei Liu, Jian Luan, Yunxin Liu, Yuanchun Li, Bin Wang, and Bo An. SMAN-Bench: A cross-system benchmark for mobile agents under single- and multi-path, ambiguous, and noisy tasks. In *International Conference on Learning Representations*, 2026a. URL https://openreview.net/forum?id=IWDpCaSF9Q.

Yifan Xu, Xiao Liu, Xinghan Liu, Jiaqi Fu, Jiayu Huang, Hanchen Zhang, Bohao Jing, Shudan Zhang, Yuting Wang, Zhao wenyi, and Yuxiao Dong. MobileRL: Online agentic reinforcement learning for mobile GUI agents. In *The Fourteenth International Conference on Learning Representations*, 2026b. URL https://openreview.net/forum?id=C3F0G9nXhl.

Zhen Yang, Zi-Yi Dou, Di Feng, Forrest Huang, Anh Nguyen, Keen You, Omar Attia, Yuhao Yang, Michael Feng, Haotian Zhang, Ram Ramrakhya, Chao Jia, Jeffrey Nichols, Alexander Toshev, Yinfei Yang, and Zhe Gan. Ferret-UI Lite: Lessons from building small on-device GUI agents. *arXiv preprint arXiv:2509.26539*, 2025. URL https://arxiv.org/abs/2509.26539.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Yilun Yao, Xinyu Tan, Chao-Hsuan Liu, Yaoming Li, Zhengyang Wang, Wenhan Yu, Zhewen Tan, Yuxuan Tian, Guangxiang Zhao, Lin Sun, Xiangzheng Zhang, and Tong Yang. HarnessBench: Measuring harness effects across models in realistic agent workflows. *arXiv preprint arXiv:2605.27922*, 2026. URL https://arxiv.org/abs/2605.27922.

Biao Yi, Xueyu Hu, Yurun Chen, Shengyu Zhang, Hongxia Yang, and Fan Wu. EcoAgent: An efficient device-cloud collaborative multi-agent framework for mobile automation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 29838–29846, 2026. doi: 10.1609/aaai.v40i35.40230. URL https://ojs.aaai.org/index.php/AAAI/article/view/40230.

A INFRASTRUCTURE IMPLEMENTATION

This appendix details the evaluation infrastructure summarized in Section 3.1 and Figure 1. Its implementation is guided by three requirements of harness-aware benchmarking on physical smartphones. First, local inference must execute natively on the device, so that measured latency and energy reflect genuine on-device cost (§A.1). Second, telemetry must be collected without perturbing the execution it measures and must scale to parallel devices, since measurement overhead would otherwise be attributed to the agent (§A.2). Third, harnesses must interact with the environment through a common action interface, so that differences in outcomes can be attributed to the harness as expected rather than to asymmetric action definitions; EDGE SYMPHONY therefore shares the same base action space with AndroidWorld (Rawles et al., 2025) (§A.3).

A.1 ON-DEVICE LLM RUNTIME

Evaluating on-device and hybrid mobile agents under realistic physical constraints requires executing multimodal model inference natively on the smartphones. Among prior mobile GUI agent systems that run their models on smartphones, llama.cpp with GGUF-quantized weights is the prevailing runtime as applied in (Wen et al., 2025; Fan et al., 2025; Huang et al., 2026b). The choice of runtime is not neutral: CORE reports roughly $5\times$ higher on-device latency under llama.cpp than under MNN for the same model on the same phone (Fan et al., 2025). Runtime effects can therefore confound the measured cost of an agent unless the runtime is fixed and reported.

EDGE SYMPHONY adopts **LiteRT-LM** (Google AI Edge, 2026) as its device-side runtime. Compared with the llama.cpp-based deployments above, LiteRT-LM offers three properties that match the requirements of closed-loop mobile GUI agents. First, it provides first-party Android support through native C++ and Kotlin APIs. Second, its hardware acceleration is not tied to a single GPU vendor: its GPU backend runs on the Tensor G4 GPU of the Pixel 9 Pro used in our experiments, and it additionally supports NPU acceleration on Qualcomm and MediaTek chipsets, whereas llama.cpp's OpenCL backend for mobile GPUs is designed primarily for Qualcomm Adreno GPUs (Gerganov & ggml-org contributors, 2023). Third, it treats multimodal (vision and audio) input and function calling as built-in runtime capabilities, which closed-loop agents heavily rely on to consume screenshots and emit structured actions. LiteRT-LM also serves on-device generative features in production products such as Chrome and Pixel Watch, and official `.litertlm` builds are distributed for the Gemma models used in our experiments. It packages the converted LiteRT model components, tokenizer files, external weights, and model metadata into a single deployable container. Hardware acceleration is provided by the underlying LiteRT runtime, whose optimized CPU and GPU backends are selected at execution time, and LiteRT-LM further supplies GenAI-specific runtime features such as KV-cache management and prompt templating.

AS the runtime sits under the model-serving interface as described in §A.1.1, additional on-device backends can be registered without modifying any harness. Supporting llama.cpp in a future release would extend EDGE SYMPHONY to the GGUF-distributed models adopted in prior on-device agents, and would make the inference runtime itself a controlled dimension of the deployment configuration.

A.1.1 ON-DEVICE MODEL SERVING VIA THE ADB ENGINE

To bridge LiteRT-LM with the host-side orchestrator, we implement an **ADB Engine** that launches an inference daemon on each phone and connects it to the host-side model client. This design realizes the decoupling described in Section 3.1: local inference runs natively on the phone, so measured costs reflect genuine on-device execution, while model placement and device hardware can be varied without modifying the harness.

Device-side daemon. During device worker provisioning, the ADB Engine verifies on-device storage and pushes the compiled C++ inference daemon (`lit`) together with the `.litertlm` model artifact to an isolated staging directory. It then launches the daemon via `lit serve`, which runs natively on the phone and binds an HTTP server to an isolated on-device port, exposing a REST endpoint compatible with OpenAI and Google GenAI API conventions. At launch, the ADB Engine sets these three main runtime options:

- **Language model backend** (`backend=cpu/gpu`): selects the accelerator for prompt prefill and token generation; the GPU backend runs through OpenCL (e.g., Qualcomm Adreno on Snapdragon or ARM Mali/Immortalis on Tensor/MediaTek SoCs).
- **Vision encoder backend** (`vision_backend=cpu/gpu`): selects the accelerator for image patch embedding and multimodal projection.
- **Memory budget** (`max_num_tokens`): sets the static KV-cache allocation to prevent out-of-memory (OOM) faults under mobile RAM constraints.

Because these options are fixed at daemon launch rather than inside the harness, the same harness can be evaluated across heterogeneous mobile hardware and accelerator settings. In all experiments, we use GPU execution for both backends with `max_num_tokens=2048` unless otherwise specified.

Host-side bridging. The ADB Engine dynamically allocates an isolated host port for each device and establishes TCP port forwarding over ADB, so the host-side model client reaches the on-device endpoint as an ordinary local HTTP service. Since this endpoint follows the same API conventions as remote model services, EDGE SYMPHONY wraps forwarded on-device daemons and remote APIs behind a unified inference interface. Every harness role (e.g., planner, verifier, or executor) thus issues the same requests whether its model runs on the phone's GPU or in the cloud, so model placement can be varied without changing harness logic.

A.1.2 FAULT-TOLERANT WATCHDOG AND AUTOMATED RECOVERY

Sustained, hardware-accelerated multimodal inference on mobile SoCs can occasionally stall the on-device runtime, for example through GPU driver timeouts or engine crashes. To keep such transient failures from aborting large-scale evaluations, the host-side model client runs a watchdog that monitors every on-device request. Upon an HTTP timeout, transport disconnect, or internal engine error, the watchdog terminates the daemon, re-initializes the on-device runtime, re-establishes port forwarding, and warms up the model before retrying the failed request. Because the failure occurs before any action is returned to the harness, the retry does not duplicate GUI actions and remains transparent to the harness. Each recovery is nonetheless recorded as a runtime anomaly in the telemetry (Section 3.3), and the cost for recovery are excluded from the reported per-task cost.

A.2 EFFICIENT ENERGY AND UI STATE CAPTURE ON SMARTPHONE

Cost measurement on physical smartphones must satisfy two constraints: it must not perturb the execution being measured, and it must scale to parallel device fleets. Two common practices violate these constraints. External power analyzers require dedicated wiring for each phone, which obstacles the testbed scalability, while command-line accessibility dumps introduces seconds of stable UI capture for each step, which would inflate the measured end-to-end latency of the agent. EDGE SYMPHONY instead relies on two mechanisms built on Android's inherit system interfaces: the On-Device Power Rails Monitor (ODPM) for step-level energy measurement (§A.2.1), and Snippet UiAutomator (Google, 2023) for low-latency UI hierarchy capture (§A.2.2).

A.2.1 ENERGY MEASUREMENT VIA ODPM

Existing mobile GUI agent benchmarks report task success together with step counts, execution time, or API cost, but none measures the energy consumed on the device. As for the energy measurement for general on-device LLMs they either connects each phone to a Monsoon high-voltage power monitor through a battery bypass that requires disassembling the device (Laskaridis et al., 2024), or measures power with an external USB power meter (Li et al., 2025b). Such instrumentation must be attached to every device and serializes measurement, which is impractical when hundreds of tasks are evaluated across many configurations and repeated runs. The second approach reads the per-application battery usage reported by the operating system (Xiao et al., 2024). It requires no additional hardware but is not suitable to GUI agents, as Android's per-application accounting has traditionally been estimated from predefined power constants rather than measured (Android Open Source Project, 2026). EDGE SYMPHONY instead measures device-level energy with the On-Device Power Rails Monitor (ODPM), which reads the power rails of the phone itself and requires no external instrumentation.

Interface. Modern Android devices expose on-device power rail monitoring through the standard `IPowerStats` hardware abstraction layer (HAL) introduced in Android 10 (Android Open Source Project, 2026). On devices equipped with power-monitoring hardware, such as Google Pixel devices from Pixel 6 onward, ODPM measures current through on-board sense resistors and power management ICs (PMICs) and reports the energy accumulated on each rail since boot, in microwatt-seconds ($1\,\mu\mathrm{W}\cdot\mathrm{s}$). Readings are queried by the host over ADB from the rooted smartphones, so no measurement code runs inside the agent or the applications under test.

Step-level measurement. The host reads the cumulative rail counters at the start and end of each agent step by executing the following adb command:

```
adb shell dumpsys android.hardware.power.stats.IPowerStats/default
                                delta
```

and takes the difference between the two readings. Let $E_r(t)$ denote the cumulative energy of rail r at time t, converted to joules. The energy of a step spanning $[t_0, t_1]$ is

$$E_{\text{step}} = \sum_{r\in\mathcal{R}} \big(E_r(t_1) - E_r(t_0)\big), \tag{1}$$

where $\mathcal{R}$ is the set of monitored rails, which cover the major power consumers of the device, including the CPU clusters, GPU, TPU, DDR memory, display, and cellular and Wi-Fi modems. The consumption on these rails reflects the real overall cost of a GUI agent during each task execution.

A.2.2 UI State Capture via Snippet UiAutomator

A critical latency bottleneck in existing mobile agent benchmarks is the extraction of UI accessibility hierarchies. AndroidWorld obtains it through the accessibility forwarding app of AndroidEnv, which streams the tree to the host over the Android emulator's gRPC interface, and waits for UI elements to stabilize before returning an observation (Rawles et al., 2025). This channel is tied to the emulator and is unavailable on physical smartphones, where the hierarchy is instead retrieved over ADB with the command-line utility

```
adb exec-out uiautomator dump /dev/tty
```

Each invocation spawns a new process, initializes a fresh accessibility connection, serializes the active window hierarchy into XML, and streams it back over ADB, which takes 1.5 to 5 seconds per capture on our devices. This agent-irrelative cost will be paid multiple times and accumulates to several minutes in each task execution, dramatically lowering the benchmarking efficiency.

Edge Symphony replaces it with Snippet UiAutomator (Google, 2023), a persistent on-device instrumentation service built on the Mobly snippet library. Its design consists of two components:

- **Persistent instrumentation service**: A lightweight Android test package is launched once during worker setup as an instrumentation runner. It maintains a long-lived `UiAutomation` connection to Android's accessibility subsystem and listens on an on-device TCP port that is forwarded to the host.
- **In-memory JSON-RPC hierarchy extraction**: When the environment captures an observation or checks for quiescence, the host sends a JSON-RPC request over the forwarded socket. The running service serializes the current hierarchy from memory and returns it, without starting a new process or accessibility connection.

By eliminating per-capture process creation and accessibility initialization, Snippet UiAutomator reduces the average capture latency from about 1850 ms to about 180 ms on our devices, an overhead reduction of over 90%. Since quiescence checks poll several consecutive captures per step, this saving compounds within every step, shortening episodes across parallel device fleets and ensuring that recorded latency reflects model inference and action execution rather than measurement overhead.

A.3 Function Call & Action Space Specification

Edge Symphony defines a flexible, two-tiered action space:

1. **Base Action Space**: Standard GUI operations are unified with **AndroidWorld** to ensure benchmark parity and portability across various agent as provided in Table 5. All coordinate arguments are integers in a normalized $[0, 1000]$ space, where $(0, 0)$ is the top-left and $(1000, 1000)$ the bottom-right corner of the display; the environment rescales them to device pixels before execution, making the interface independent of screen resolution.
2. **Harness-Required Action Space**: Non-GUI signals used by multi-role harnesses for coordination between roles, as given in Table 6. EDGE SYMPHONY provides a dynamic action registration mechanism that allows any agent harness to register custom action declarations and execution handlers to extend the action space as needed.

Table 5: **Base action space.** Coordinates are normalized to $[0, 1000]$ and rescaled to device pixels before execution.

Function Name	Parameters	Description
`tap`	`x`, `y`: int	Taps at (x, y).
`double_tap`	`x`, `y`: int	Double-taps at (x, y).
`long_press`	`x`, `y`: int	Long-presses at (x, y).
`swipe`	`x1`, `y1`, `x2`, `y2`: int; `duration`: int (opt, ms)	Swipes from (x_1, y_1) to (x_2, y_2), taking the $duration$ of milliseconds.
`type_text`	`text`: str; `clear_text`: bool (opt); `x`, `y`: int (opt)	Types text into the focused input field, optionally clearing it first or focusing the field at (x, y).
`launch_app`	`app_name`: str	Opens the specified application.
`navigate_home`	None	Returns to the home screen.
`navigate_back`	None	Presses the system back button.
`wait`	`duration`: float (opt, s)	Pauses for a specified duration to allow asynchronous UI transitions or network requests.
`ask_user`	None	Requests clarification from the user. During evaluation, a simulated user answers by revealing the task's disambiguated goal on the next step.
`goal_achieved`	None	Declares the task complete and ends the episode.

Table 6: **Harness-required action space.** Registered by PocketAct and RelayAct

Action Name	Harness Name	Parameters	Description
`subgoal_achieved`	Hierarchical & Hybrid Harness	None	Marks the active subgoal of the current plan as completed.

A.4 API PRICING REFERENCE

Table 7 lists the API prices of representative models from OpenAI, Anthropic, and Google, collected from the official pricing pages on September 24, 2026 (OpenAI, 2026; Anthropic, 2026; Google, 2026). All prices are standard-tier rates without batch, priority, or caching discounts, and use short-context rates where a provider prices by context length. Each model is listed once at its default reasoning effort to keep the list compact; the per-token prices are the same at other effort levels, which affect cost only through the number of reasoning tokens billed as output. Our experiments use only Gemini models as remote models, and the remaining entries serve as a reference for estimating API cost with other providers. Prices may change after the collection date.

Table 7: API prices of representative models (USD per million tokens).

Model	Input	Output
GPT-6 Astra	10.00	50.00
GPT-6 Sol	2.00	10.00
GPT-6 Luna	0.10	0.50
Claude Fable 5.1	10.00	50.00
Claude Opus 5.5	4.00	20.00
Claude Sonnet 5	2.00	10.00
Claude Haiku 4.5	1.00	5.00
Gemini 3.1 Pro (Preview)	2.00	12.00
Gemini 3.5 Flash	1.50	9.00
Gemini 3.8 Flash[†]	0.75	3.75
Gemini 3.5 Flash-Lite	0.30	2.50

[†]Promotional rate through December 31, 2026; $1.50 / $7.50 from January 1, 2027.

A.5 Inference and Execution Settings

Decoding. Every harness role, on-device or remote, issues its request with temperature 0 (greedy decoding). No other sampling parameters (top-p, top-k, random seed, or thinking/reasoning budget) are set, so each backend uses its provider defaults. On-device requests are served by the LiteRT-LM daemon (§A); remote requests are routed through LiteLLM with the same message and tool schema.

Step semantics and budget. One agent step is one invocation of a single harness role, which corresponds to exactly one model call, followed by execution of the UI actions that call returns (possibly none, e.g., for a Planner or Verifier turn). Each task is allotted a budget of 40 steps; an episode that exhausts the budget without the success check passing counts as a failure. This budget balances benchmark time against evaluation resolution; Appendix E.3 reports success rates under budgets from 10 to 40 steps and validates this choice.

Transient failures. Remote API errors (e.g., rate limits or timeouts) are retried up to six times with exponential backoff and random jitter, and on-device failures are handled by the watchdog (§A.1.2). Retries occur below the harness and do not consume steps. A cooldown of 3 s separates consecutive tasks.

A.6 Observation Space and Shared Tool Interface

To make comparisons across paradigms controlled, every agent in Edge Symphony (Pure Cloud, Pure Local, PocketAct, and RelayAct) receives observations from the same environment interface and acts through the same tool schema. Architectures differ only in *which role* consumes which part of the observation and *which model* (cloud or on-device) runs that role.

A.6.1 Observation Capture

At every step, the environment first waits for the UI to settle, then captures two synchronized views of the screen:

- **Screenshot**: A full-resolution RGB frame, losslessly encoded as PNG and attached to the model input as an image. The screenshot is the primary observation for every role.
- **Accessibility XML**: The raw UI view hierarchy of the active window. The raw XML is never passed to the model verbatim. It is first pruned into a compact list of interactive elements (below).

A.6.2 XML Pruning into Interactive Elements

Raw Android view hierarchies routinely contain hundreds of layout-only nodes. That overflows the limited context budget of $\sim$4B on-device models and dilutes grounding signals. Edge Symphony therefore prunes the hierarchy in four steps:

1. **Interactivity filter**: A node is kept only if it is clickable, focusable, scrollable, or a text input field. Every other node (layout containers, decorative views, static text) is discarded.
2. **Label resolution**: Each kept node gets a human-readable label, taken in priority order from its visible text, its accessibility description, its short resource identifier, and finally its widget type.
3. **Coordinate normalization**: The node's pixel bounding box is reduced to its center point and rescaled into the same $[0, 1000]^2$ normalized space used by all coordinate-bearing tools. This lets the model copy coordinates straight into a tap action.
4. **Affordance annotation and truncation**: Interaction affordances (clickable, scrollable, focusable, long-clickable) are appended to each entry, and the list is capped at the first 50 elements.

The result is injected as a text block alongside the screenshot:

Example Pruned XML Observation

```
<INTERACTIVE_ELEMENTS>
1. 'Search hotels' at (x: 502, y: 118), clickable: true, focusable: true
2. 'Check-in date' at (x: 261, y: 274), clickable: true, focusable: true
3. 'Check-out date' at (x: 739, y: 274), clickable: true, focusable: true
4. 'hotel_list' at (x: 500, y: 640), scrollable: true, focusable: true
5. 'Navigate up' at (x: 56, y: 62), clickable: true
</INTERACTIVE_ELEMENTS>
```

The Executor prompts rely on this block explicitly, telling the model to tap only elements that appear in the list. The loop detector reuses the same hierarchy to tell the model whether a repeatedly tapped coordinate falls on an interactive or a non-interactive element.

A.6.3 Observation Routing per Role

Table 8 summarizes which observation channels and tools each role receives.

History compression. Multi-turn roles (the single-role agents and the Unblocker) keep a running conversation history. Outside the most recent observation turns, every image is replaced with a short text placeholder. The textual trace of prior reasoning and actions is kept, while token cost stays bounded over long horizons. Single-turn roles (Executor and Verifier) instead rebuild their prompt from the shared agent state at every invocation: the goal, the current plan, the recent action history, and any pending feedback.

Table 8: **Observation Channels and Tool Access per Agent Role**

Role	Agent(s)	Screenshot	Pruned XML	Tools
Single-role (cloud)	Pure Cloud	Current frame + sliding window (3 turns)	–	Shared action set in Table 5
Single-role (on-device)	Pure Local	Current frame + sliding window (3 turns)	✓	Shared action set in Table 5
Planner	PocketAct, RelayAct	Initial frame	–	None (text plan output)
Executor	PocketAct, RelayAct	Current frame	✓	Shared action set in Table 5 + subgoal completion signal
Verifier	PocketAct, RelayAct	Before/after frames of recent actions	–	None (text plan / feedback output)
Unblocker (cloud)	RelayAct	Current frame + sliding window (3 turns)	–	Shared action set in Table 5

B CAPABILITY-ORIENTED TASK SUITE

This appendix details the benchmark's application coverage (§B.1), basic task foundation (§B.2), advanced capability task suite (§B.3), and task distribution across applications (§B.4).

B.1 APPLICATION COVERAGE & OPERATING ENVIRONMENT

To provide a rigorous, hermetic, and fully reproducible benchmark across heterogeneous physical devices, EDGE SYMPHONY evaluates agents on **23 Android applications**, consisting of **open-source software (OSS) applications** and **custom benchmark applications** (Table 9). This choice avoids the experimental fragility introduced by commercial mobile benchmarks that rely on closed-source third-party applications (e.g., proprietary shopping, ride-hailing, or social media apps):

1. **Dynamic UI Churn & Server-Side Updates**: Upstream production apps frequently update layouts, run A/B UI experiments, or deprecate API endpoints, rendering recorded trajectories obsolete and breaking evaluation reproducibility.
2. **Authentication & Financial Paywalls**: Live apps require persistent user accounts, phone verification (SMS OTPs), captchas, and active credit cards, creating security risks and preventing automated large-scale fleet evaluation.
3. **Vendor & Regional Fragmentation**: Proprietary apps often depend on Google Play Services or are geo-restricted, preventing execution on open Android Open Source Project (AOSP) builds or diverse international hardware.

In contrast, all application binaries in EDGE SYMPHONY are frozen at fixed release tags, execute entirely on-device without external cloud service dependencies, and feature programmatic state initialization and ground-truth SQLite/ContentProvider inspection.

Scope of the frozen environment. Hermeticity comes at a cost in realism. Because the applications run without backends, the suite does not exercise authentication and account flows, server-driven or personalized content, network latency and failures of application backends, push notifications, or layout drift across application updates. The Interruption and Recovery category (§B.3) emulates part of this variability through injected pop-ups, delayed renders, and state regressions, but it does not replace live services. Success rates on EDGE SYMPHONY should therefore be read as an upper bound on performance for these aspects of production applications.

B.2 BASIC TASK SUITE

B.2.1 TASK DEFINITION & CONSTRUCTION

The basic task suite comprises the 118 tasks of the Basic Knowledge (BK) category. These tasks test app-function grounding, control selection, and action verification, without requiring long-horizon planning or complex reasoning chains. We construct the task suite in following three stages:

- **Custom application tasks**: Table 9 includes seven custom applications developed for EDGE SYMPHONY, covering booking workflows, puzzle and strategy games, spatial navigation, and fine-grained grounding. We manually created 38 basic tasks for these applications and added them to the benchmark.
- **Tasks seeded from AndroidWorld**: To expand the task pool for OSS applications, we randomly sampled 36 tasks from AndroidWorld (Rawles et al., 2025) across Joplin, Simple Calendar Pro, Tasks, Simple SMS Messenger, Contacts, DeskClock, Broccoli, OsmAnd, and Pro Expense. We adapted their implementation, task goals, and success verification to the EDGE SYMPHONY environment. Together with the custom application tasks, these seeded tasks form the starting point of the basic task suite.
- **Automated expansion**: Building on the tasks above, we collected the APKs of the OSS applications and applied the automated task curation workflow (Figure 3, Appendix C) to generate the remaining 44 basic tasks, each admitted only after passing validation.

Table 10 lists a representative sample of tasks from the basic task suite.

Table 9: **Applications in EDGE SYMPHONY. Min-SDK Version: 21**

App Name	Version	Domain	Function
Simple SMS Messenger	v5.18.0	Communication	SMS messaging.
Pro Expense	v1.0.0-beta06	Finance	Expense tracking.
MedTimer	v1.8.5	Health	Medication reminders and intake logging.
Broccoli	v1.4.1	Lifestyle	Recipe management.
ShoppingList	v1.1	Lifestyle	Shopping list management.
Simple Gallery Pro	v6.28.0	Media	Photo and video gallery.
Joplin	v3.5.8	Productivity	Markdown note-taking and notebook management.
Simple Calendar Pro	v6.23.0	Productivity	Calendar and event scheduling.
Tasks	v13.6.5	Productivity	To-do and task management.
Fossify File Manager	v1.3.0	System Utility	Local file management.
OsmAnd	v4.6.13	Travel	Offline maps and navigation.
Breezy Weather	v6.2.1	Utility	Weather forecasts.
DeskClock	v2.31	Utility	Alarms, timers, and stopwatch.
System	AOSP Core	System Utility	Package management.
Settings	AOSP Core	Configuration	Device settings.
Contacts	AOSP Core	Utility	Contact management.
Tic Tac Toe	v1.0, Custom	Simple Game	Tic-tac-toe game.
Grid Touch	v1.0, Custom	Grounding	Grid of fine-grained touch targets.
Sudoku	v1.0, Custom	Complex Logic	Sudoku puzzle.
Buggy Booking	v1.0.0, Custom	Robustness	Booking with injected interruptions and UI faults.
Game 2048	v1.0, Custom	Spatial Logic	2048 sliding-tile puzzle.
3D Maze	v1.0, Custom	Spatial Logic	First-person maze navigation.
Booking Benchmark	v1.0.0, Custom	Travel	Hotel and flight booking.

Table 10: **Representative Basic Tasks Grouped by Target Application.** Goals are reproduced from each task's `goal()`; $\langle\cdot\rangle$ denotes a parameter instantiated at task initialization.

Task Name	Target Apps	Goal Description
`create_joplin_note`	Joplin	Create a new note in Joplin with title ⟨title⟩ and content ⟨content⟩.
`fix_joplin_markdown`	Joplin	Identify and fix the Markdown syntax errors in the headers, lists, and links within the note titled ⟨title⟩.
`normalize_joplin_titles`	Joplin	In the Joplin app, convert the titles to Title Case.
`add_one_event`	Simple Calendar Pro	In Simple Calendar, create a calendar event on ⟨date⟩ at ⟨hour⟩h with the title ⟨title⟩ and the description ⟨description⟩. The event should last for ⟨duration⟩ mins.
`calendar_notifications`	Simple Calendar Pro	In the Simple Calendar app, create an event titled ⟨title⟩ for ⟨time⟩ (device local timezone). Add the following alerts/reminders: 2 hours before, 30 minutes before, and 15 minutes before.
`delete_events`	Simple Calendar Pro	In Simple Calendar Pro, delete all the calendar events on ⟨date⟩.
`tasks_due_on_date`	Tasks	What tasks do I have due ⟨date⟩ in Tasks app? Answer with the titles only. If there are multiple titles, format your answer in a comma separated list.

Table 10: **Representative Basic Tasks Grouped by Target Application (Continued)**

Task Name	Target Apps	Goal Description
simple_sms_reply_most_recent	Simple SMS Messenger	Reply to the most recent text message using Simple SMS Messenger with message: ⟨message⟩
simple_sms_send_clipboard_content	Simple SMS Messenger	Send a message to ⟨number⟩ with the clipboard content (which is ⟨content⟩) in Simple SMS Messenger.
set_mwf_alarm	DeskClock	Create an alarm for 8:00 AM every Monday, Wednesday, and Friday, label it "Morning", and leave it enabled.
edit_alarm_time	DeskClock	Change the existing "Workday" alarm from 6:45 AM to 7:15 AM and leave it disabled. Preserve its repeat days and vibration setting, and preserve the "Restday" alarm.
create_run_timer	DeskClock	Create a five-minute timer named "Tea" and leave it counting down.
create_contacts	Contacts	Add a new contact in the Contacts app with name ⟨name⟩, phone number ⟨phone⟩ and email ⟨email⟩.
add_single_recipe	Broccoli	In Broccoli, add a new recipe with the following details: Title, Description, Servings, Preparation Time, Ingredients, and Directions (⟨values⟩).
delete_duplicate_recipes	Broccoli	In Broccoli, delete all but one of any recipes that are exact duplicates, ensuring at least one instance of each unique recipe remains.
add_map_favorite	OsmAnd	In OsmAnd, find the ⟨location⟩ and set it as a favorite place named ⟨name⟩.
navigate_to2	OsmAnd	In OsmAnd, start navigation from ⟨start⟩ to ⟨destination⟩.
expense_delete_duplicates	Pro Expense	Delete all but one of any expenses in pro expense that are exact duplicates.
expense_delete_multiple	Pro Expense	Delete the following expenses from pro expense: ⟨expense 1⟩, ⟨expense 2⟩, ⟨expense 3⟩.
add_city	Breezy Weather	Add ⟨city⟩ (⟨country⟩) as a new location in the Breeze Weather app.
switch_temp_unit	Breezy Weather	In the Breeze Weather app settings, change the temperature unit to ⟨unit⟩.
create_multiple_shoppinglist	ShoppingList	Create two shopping lists: "Home Supplies" containing 2 Soap, and "Office Supplies" containing 3 Pens.
delete_shopping_item	ShoppingList	Delete Milk from the existing "Weekend" list. Preserve Bread with a quantity of 2 and a unit price of 2.00.
create_daily_reminder	MedTimer	Add a medicine named "Morning Sample" and give it a daily reminder at 8:00 AM with a dosage of 1.
log_extra_dose	MedTimer	Add a medicine named "Dose Sample" with an initial stock of 20 and the unit "tablet". Record an additional dose of 1 tablet taken today at 9:00 AM, leaving a stock of 19.

Table 10: **Representative Basic Tasks Grouped by Target Application (Continued)**

Task Name	Target Apps	Goal Description
`dnd_complex_rules`	Settings	Turn on Do Not Disturb. Set up a Do Not Disturb schedule from ⟨start⟩ to ⟨end⟩ on ⟨days⟩. Please only set up this one schedule.
`delete_app`	System	Delete the ⟨app⟩ app from the device.
`book_hotel_nonref`	Booking Benchmark	In the booking app, search hotels in ⟨city⟩ (check-in ⟨date⟩, check-out ⟨date⟩), book ⟨hotel⟩, select ⟨room type⟩ room, choose the Non-Refundable Rate (15% off) at checkout, with guest name, email, and phone ⟨guest info⟩. Confirm the booking.
`filter_restaurant_japanese`	Booking Benchmark	In the booking app, search restaurants in San Francisco on 2024-04-07, with 2 people, then filter by Japanese cuisine with rating ≥ 4.5.
`search_flight_app_test`	Buggy Booking	In the Buggy Booking app, search a one-way SFO→JFK flight on 2024-04-01. Expected: the search runs and the results screen lists flights from SFO to JFK on that date. File a report on the Test Report screen to report if App works correctly or if any bug is found.
`play_sudoku`	Sudoku	Solve the puzzle. Fill all empty cells with digits 1 through 4 such that each number appears exactly once in every row, column, and 2x2 region.
`play_game2048`	Game 2048	Play the 2048 game and create a tile with the number ⟨target⟩ by swiping Up, Down, Left, or Right to merge equal tiles without filling up the board.
`play_maze3d`	3D Maze	Reach the exit (green wall) of the maze.
`play_tic_tac_toe`	Tic Tac Toe	Play a game of Tic Tac Toe against the AI. Your goal is to not lose. A win or draw is acceptable.

B.3 Advanced Task Suite

The advanced task suite comprises the 140 tasks of the five remaining capability categories defined in Table 2: Semantic Migration (50), Long-horizon Planning (32), Constraint-aware Execution (31), Ambiguous Content (16), and Interruption and Recovery (11). These tasks are generated with the automated task curation workflow (Appendix C) via the capability-specific planner, coder and validator agents, and are manually reviewed before release. Beyond the category definitions, each category imposes specific requirements on task construction and verification:

- **semantic migration** : The source information is seeded into one application during initialization, and success is verified from the resulting records in the target application.
- **Long-horizon Planning**: Each task chains multiple dependent stages and requires more than 15 atomic UI actions; success requires the final state of every stage to be correct.
- **Constraint-aware Execution**: Success requires both reaching the goal state and preserving the state protected by the constraints; any constraint violation counts as failure.
- **Ambiguous Content**: The instruction deliberately omits or conflates a required parameter. The agent is expected to invoke `ask_user` (Table 6), to which the evaluator replies with a programmatic ground-truth clarification; tasks completed on an unconfirmed assumption count as failure.

- **Interruption and Recovery**: Disruptions such as modal pop-ups, delayed renders, state regressions, or system dialogs are injected during execution; success requires completing the task without duplicating already completed work.

Table 11 lists representative tasks from each category.

Table 11: **Representative Advanced Tasks Sampled Across Capability Categories**

Category–Task Name	Target Apps	Goal Description
semantic migration -sms_calendar_event_structured_extraction	Simple SMS, Contacts, Simple Calendar	My landlord Mr. Brown texted me about an inspection. Create a calendar event titled 'Building inspection' at the date and time given in his message.
semantic migration -meeting_note_to_calendar	Joplin, Simple Calendar	Create the corresponding calendar event from the Joplin note "Design Review Schedule."
semantic migration -sms_recipe_joplin_note_verbatim_copy	Simple SMS, Joplin	Emma texted me her grandma's cookie recipe. Save it as a new Joplin note titled 'Grandma's Cookies' containing the full recipe from her message.
semantic migration -broccoli_ingredients_joplin_shopping_list_list_extraction	Broccoli, Joplin	Create a new note in Joplin titled 'Shopping List' inside the notebook 'Personal' that lists every ingredient of the Broccoli recipe 'Pasta Carbonara', one ingredient per line.
semantic migration -expense_category_totals_joplin_summary	Pro Expense, Joplin	Check my expenses in Pro Expense. Create a Joplin note titled 'Expense Summary' with one line per category showing the category name and its total amount.
semantic migration -agenda_to_daily_note	Simple Calendar, Joplin	Summarize the two meetings on October 23 in a Joplin note titled "October 23 Agenda."
long horizon -resolve_expense_review	Joplin, Pro Expense, Tasks	Use the 'Expense Review' note in Joplin to correct the Museum Parking entry in Pro Expense, then update and complete its matching review task in Tasks.
long horizon -double_a_recipe_computation_2_app_back_and_forth	Broccoli, Joplin	I need the Broccoli 'Pancakes' recipe for 4 people instead of 2. Create a Joplin note titled 'Pancakes for 4' listing every ingredient with its quantity doubled, one per line.
long horizon -start_baking_plan	Joplin, Broccoli, DeskClock	Carry out Saturday Baking Plan: favorite its selected Broccoli recipe and start the matching bake timer.
long horizon -expense_dedup_report	Pro Expense, Joplin	In Pro Expense, the $4.50 'Coffee' entry was logged twice — delete one of the duplicates. Then create a Joplin note titled 'July Expense Report' listing each category with its corrected July total and a final line with the grand total.
long horizon -register_allergy_plan	Joplin, MedTimer	Register the medicine, stock, and daily dose schedule from Allergy Plan in MedTimer.
constraint aware -enable_or_create_weekend_alarm	Joplin, DeskClock	The Joplin note "Weekend Plan" specifies a Saturday 07:00 run. If Clock already has an alarm with the same label and time, enable it and turn on vibration; otherwise, create that alarm.
constraint aware -convert_dated_message	Simple SMS, Simple Calendar, Tasks	Read Maya's latest text. If it contains a complete date and time, create the corresponding calendar event; otherwise create "Confirm Meeting Time with Maya."

Table 11: **Representative Advanced Tasks Sampled Across Capability Categories (Continued)**

Category–Task Name	Target Apps	Goal Description
constraint aware -escalate_large_expense	Pro Expense, Simple SMS, Joplin	Check the latest Dining expense this week. If it is at least USD 100, send the amount to "Budget Contact"; otherwise create a note titled "Weekly Dining Check" with the merchant and amount.
constraint aware -schedule_around_conflict	Simple Calendar Pro	Check whether October 20 at 15:30 conflicts with an existing event. If it does, schedule "Travel Preparation Call" at 16:30; otherwise schedule it at 15:30.
ambiguity -delay_evening_medicine	Joplin, MedTimer	Check Joplin for the current Travel Medication Plan and move the evening medication reminder in MedTimer a little later according to it. (Invoking ask_user resolves the target: move the Vitamin D reminder to 21:00.)
ambiguity -disable_doctor_selected_lisinopril	Simple SMS, MedTimer	Check Simple SMS Messenger for Dr. Rivera's text message and disable the Lisinopril reminder in MedTimer that he meant. (Invoking ask_user resolves the target: the Lisinopril 10 mg evening reminder.)
ambiguity -move_jordan_mentioned_meeting	Simple SMS, Simple Calendar	Check Simple SMS Messenger for Jordan's text and move the meeting he mentioned to 4:00 PM in Simple Calendar. (Invoking ask_user resolves the target meeting: Design Critique.)
ambiguity -which_meeting_note	Joplin	Delete my meeting note in Joplin. (Invoking ask_user disambiguates among multiple meeting notes.)
ambiguity -which_alex	Contacts, Simple SMS	Send 'Happy birthday!' to Alex. (Invoking ask_user disambiguates the recipient when multiple contacts share the given name.)
robustness -resume_interrupted_alarm	DeskClock, Settings	Set an alarm for 07:30 this Saturday, enable vibration, and label it "Morning Run." (Once the 07:30 alarm is saved, the Settings app is brought to the foreground; the agent must return to Clock and finish the remaining settings.)
robustness -flight_round_trip_buggy_app_test	Buggy Booking	In the Buggy Booking app, switch the flight search to Round Trip, then search SFO→JFK on 2024-04-01 with a return date of 2024-04-08. Expected: tapping Round Trip activates round-trip mode, a Return Date field appears, and the search is recorded as a round-trip. File a report on the Test Report screen to report if App works correctly or if any bug is found.
robustness -search_hotel_buggy_app_test	Buggy Booking	In the Buggy Booking app, on the hotel search screen, set any check-in / check-out dates, set the guest count to 2 via the guest count stepper button, then search the results. Expected: the Search button runs the search and the results screen lists matching hotels with no error. File a report on the Test Report screen to report if App works correctly or if any bug is found.

B.4 Task Distribution Across Applications

Figure 6 shows the distribution of **application participations**, rather than unique task counts, across the 23 benchmark applications, segmented into **Base Tasks** and **Advanced Tasks**. For visualization,

the five games listed in Table7̃ (Tic Tac Toe, Grid Touch, Sudoku, Game 2048, and 3D Maze) are combined into a single *Games* bar, yielding 19 bars for the 23 applications. Each task contributes one participation to every distinct application it involves, regardless of how many times that application is visited during execution. The 140 advanced tasks involve an average of 2.07 distinct applications. This broad application coverage provide a diverse testbed for evaluating various harness capabilities.

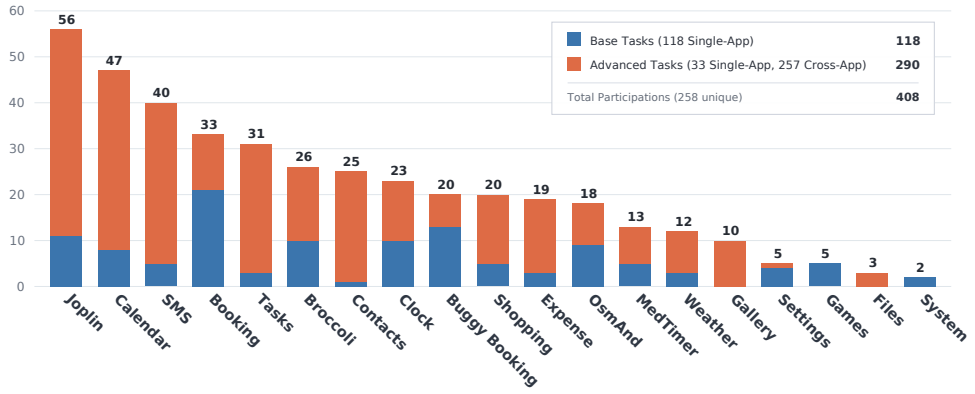


Figure 6: **EDGE SYMPHONY Task Participation Counts Across 23 Applications.**.

C AUTOMATED TASK CURATION FRAMEWORK

Philosophy. Task curation is analogous to preparing an examination: the orchestrator for task curation is the teacher, the application is the subject, and its transition graph and database schema form the textbook. Capability categories specify question types; tasks supply questions; oracle paths and state-based checks supply answer keys and grading rubrics. A separate validator acts as a teaching assistant, testing each question before the evaluated agent attempts it. This motivates the four stages in Section 3.2.2: establish application knowledge, propose capability-conditioned tasks, package executable drafts, and validate them before admission. The Contacts examples below follow this progression from a clean environment to creating and blocking a contact.

C.1 USAGE & PATTERNS

The entry point, `es-automated-task-curation`, converts a natural-language request into `curation_config.yaml` and dispatches the specialized agents. Inputs are target APKs or package names, live devices accessible through ADB, a validator API key, and requested task batches. Supported patterns include local-database OSS apps and system apps, single-app operations, and cross-app tasks with additional semantic or length constraints. Outputs are app registrations, reusable base tasks, app transition graphs, executable task files, and JSON specifications recording validation history.

For example, the supplied request asks for ten Pro Expense tasks, including five hard tasks focused on receipt scanning and text extraction, and five Joplin–OsmAnd tasks exceeding twenty steps. Example C.1 gives the complete configuration, retaining the app list, batch specifications, and original comments; only the local APK path is anonymized. These are requested capabilities, not a claim that the APK exposes them; their realizability must be established downstream.

Example C.1 Request configuration: `curation_config.yaml`

```yaml
# Edge Symphony Task Curation Configuration Template

# Required: API key for Gemini agent validation
api_key: "AIzaSy..."

# Required: ADB connection portal (device serial or host:port)
adb_portal: "localhost:17241"

# List of target apps that will be onboarded and mapped
apps:
  - name: "PRO_EXPENSE"   # Must match AppName enum
    # Required for new apps: Path to the APK
    apk_path: "/path/to/pro-expense.apk"

  - name: "OSMAND"

  - name: "JOPLIN"

# Batches of tasks to generate. The orchestrator routes these to specialized
  planners based on 'category'.
task_batches:
  - category: "Simple"
    target_apps: ["PRO_EXPENSE"]
    count: 10
    extra_note: "Include 5 hard tasks. Focus on receipt photo scanning and text
  extraction."

  - category: "cross2app"
    target_apps: ["JOPLIN", "OSMAND"]
    count: 5
    extra_note: "Make sure the total steps are more than 20."
```

Note. `AIzaSy...` is the placeholder in the supplied example. `Simple` and `cross2app` are the request-level category labels used by the curation workflow; they correspond to Basic Knowledge and cross-application Semantic Migration in the six capability categories defined in the main text.

Note. `AIzaSy...` is the placeholder in the supplied example. `Base Knowledge`, and `Semantic Migration` are retained categories supported by EDGE SYMPHONY; the manuscript uses the six capability categories defined in the main text.

C.2 STAGE 1. APPLICATION ONBOARDING AND HYBRID CRAWLING

Register and establish the clean state. For a new application, `es-app-onboarding` checks the APK, registers the application enum and package mapping, and creates its task directory and build files. It installs the app, extracts and grants its required runtime permissions, and uses a vision agent to bypass first-run screens. The verified bypass sequence and database schema are recorded for reuse. In the supplied Pro Expense example, the sequence is `NEXT` on language selection, then `CONTINUE` on currency selection, reaching the clean home state in Figure 7. Existing onboarded applications reuse their prepared artifacts.

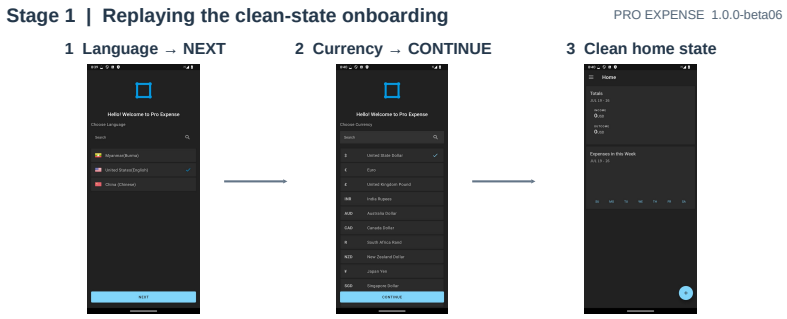


Figure 7: **A verified onboarding sequence.** Original Pro Expense screenshots: language selection, currency selection, and the clean home state. The two actions are recorded for replay.

Persistent onboarding record. Example C.2 records Pro Expense onboarding in `onboarding_status.md`. The metadata identifies the package, version, permissions, replay actions, screenshots, discovered database, and build status; the extracted schemas describe the `expense` and `backup` tables in `accounting.db`, together with shared preferences indicating that the first-run wizard has been completed. The same file is subsequently enriched with previous validations, so its final section records cent-valued amounts, ADB bytes/string handling, the pinned emulator date, database ownership during initialization, cross-app hint injection, and horizontal category scrolling. These later lessons are persistent inputs to downstream agents rather than outputs of the initial onboarding run alone. In addition, with the more curation requested for this app and validation using this app, the persistent record will be extended and thus, making the future curation more efficient with broader and deeper knowledge on this app.

Example C.2 Onboarding record: `onboarding_status.md`

```
# Onboarding Status

- **app_name**: pro_expense
- **app_path**: /path/to/pro_expense_1.0.0-beta06.apk
- **package_name**: com.arduia.expense
- **app_version**: 1.0.0-beta06
- **required_permission**: []
- **click_behavior**:
  - `step1`: Clicked "NEXT" (`com.arduia.expense:id/btn_continue`) to dismiss the
  language selection screen.
  - `step2`: Clicked "CONTINUE" (`com.arduia.expense:id/btn_continue`) to dismiss
  the currency selection screen.
- **screenshot_paths**:
  - click_1.png: `/path/to/edge_symphony/tasks/pro_expense/start_state_screenshot/
  click_1.png`
  - click_2.png: `/path/to/edge_symphony/tasks/pro_expense/start_state_screenshot/
  click_2.png`
```

Example C.2 (continued)

```
  - start_state.png: `/path/to/edge_symphony/tasks/pro_expense/
  start_state_screenshot/start_state.png`
- **discovered_databases**: `accounting.db`
- **build_status**: PASSED

## Extracted DB Schemas

### Database: accounting.db
**Table: expense**
- expense_id (INTEGER)
- name (TEXT)
- amount (INTEGER)
- category (INTEGER)
- note (TEXT)
- created_date (INTEGER)
- modified_date (INTEGER)

**Table: backup**
- backup_id (INTEGER)
- name (TEXT)
- path (TEXT)
- created_date (INTEGER)
- item_total (INTEGER)
- worker_id (TEXT)
- is_completed (INTEGER)

### Shared Preferences: com.arduia.expense_preferences.xml
- update_status: -1
- isFirstTime: false

## Background Knowledge

- **Assert Tolerance**: Database saves money amounts as integers in cents (e.g. `
  $15` is stored as `1500`). Validation functions should check for exact cent
  conversion rather than dollar floats.

### Python 3 Bytes/String TypeErrors in ADB Commands
- **Problem**: When querying the device via `execute_adb_command` (e.g. `sqlite3`
  queries or `pm` tool), the returned output is in `bytes`. Checking `if "string"
  in result:` throws a `TypeError: a bytes-like object is required, not "str"`,
  crashing the oracle script.
- **Solution**: Always use byte literals (`b"string"`) when searching for expected
   substrings in the ADB command output, or decode the output first. Example: `if
  b"exists" in db_status:`.

### Device Date Pinning Constraint
- **Problem**: Android World pins the emulator's date to October 15, 2023. If mock
   database rows are injected with current host time (e.g. 2026), they will be
  hidden from the app UI due to default "Current Month" filters, causing VLM
  failures.
- **Solution**: Always use dates corresponding to October 15, 2023 (e.g.
  `1697384040000` millisecond epoch) for injected rows.

### DB Initialization Permissions
- **Problem**: Performing SQLite inserts via root-executed adb before the app
  starts can create database files owned by root, crashing the app on
  initialization.
- **Solution**: Always let the base class `init_apps` launch the app first to
  establish correct SQLite permissions before executing SQL inserts.

### Cross-App Hint Design (Joplin + Pro Expense)
- **Problem**: Automatic FSM hint injection (`inject_hints.py`) only parses one
  app's FSM (`meta_structure.yaml`). A cross-app task spanning Joplin and Pro
  Expense cannot have automatic hints for both.
- **Solution**: Manually write step-by-step cross-app instructions inside the
  python class's `goal()` method using the `"Hint sequence:"` header. The `
  inject_hints.py` script will detect the prefix and automatically skip it,
  preventing double-hint conflicts while still guiding the agent.
```

Note. Local APK and screenshot paths are anonymized for the review copy.

Reusable base task. The Contacts base-task file appears in Example C.3. Its `init_apps()` first invokes the shared initialization, clears the contacts and blocked-number providers, resolves the installed Contacts package, clears its data, grants contact-related permissions, and returns to the home screen. Subsequent Contacts tasks inherit this clean initialization rather than duplicating it. The empty `goal()` and false `is_success()` make the base class an environment foundation, not an independently evaluable benchmark question.

Example C.3 Base task: `_base_contacts_task.py`

```python
"""Base class for Contacts tasks."""

from edge_symphony.apps import app_packages
from edge_symphony.env import device_manager
from edge_symphony.tasks import task


class BaseContactsTask(task.Task):
  """Base class for Contacts tasks."""

  def __init__(self):
    super().__init__()
    self._app_name = app_packages.AppName.CONTACTS
    self._package_name = app_packages.APP_TO_PACKAGE_NAME[self._app_name]

  def goal(self) -> str:
    return ""

  def is_success(self, device: device_manager.DeviceManager) -> bool:
    return False

  def init_apps(self, device: device_manager.DeviceManager) -> None:
    """Clear contacts app data and navigate to home screen."""
    super().init_apps(device)
    device.controller().execute_adb_command(
        "shell pm clear com.android.providers.contacts"
    )
    device.controller().execute_adb_command(
        "shell pm clear com.android.providers.blockednumber"
    )
    package_name = device.controller().get_installed_package_name(self._app_name)
    if not package_name:
      raise ValueError(f"App {self._app_name} is not installed.")
    device.controller().execute_adb_command(
        f"shell pm clear {package_name}"
    )
    for permission in [
        "android.permission.READ_CONTACTS",
        "android.permission.WRITE_CONTACTS",
        "android.permission.GET_ACCOUNTS",
    ]:
      device.controller().execute_adb_command(
          f"shell pm grant {package_name} {permission}"
      )
    device.controller().navigate_home()
```

Explore with a script–agent loop. Starting from the base-task reset, `es-build-meta-state` uses an agent-directed depth-first search (DFS). The script manages the pending stack, device backtracking, and XML extraction. At a new UI state, it pauses with `pending_elements.json`; a steering agent reads the screenshot and XML, selects meaningful actions, supplies form or search inputs, and prunes dead ends such as third-party sign-in. Its `decision.json` resumes scripted exploration. Traversal stops at the configured depth or transition limit, and a finalization agent assigns semantic screen names (Figure 8).

App transition graph. Example C.4 shows and example of app transition graph, including both contact-editing transitions and the later settings and scrolling transitions. Each edge

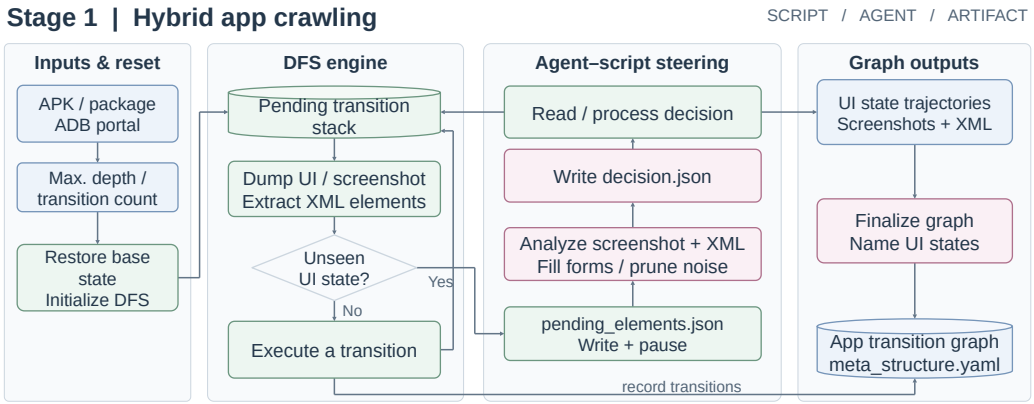


Figure 8: **Hybrid crawling within Stage 1.** Scripts perform traversal and replay; agents steer new states and name the final graph. Screenshots and XML traces remain linked to the transition records.

records a source screen, action, target element, resulting screen, and transition label, written as $\langle s_{\text{parent}}, a, o, s_{\text{target}}, \tau \rangle$. Here τ is `ui_changed`, `db_content`, or `db_modified`, not a full database diff. The transition IDs support downstream path references, while semantic state names identify the screens connected by each action. The bounded graph records discovered interactions rather than establishing exhaustive app coverage.

Example C.4 App transition graph: `meta_structure.yaml`

```
# Constructed meta-state graph for com.google.android.contacts.
# Generated via autonomous explore_app.py DFS script using DeviceManager.

transitions:
  7: ["people_contacts", "click", "Create contact", "editor", "ui_changed"]
  73: ["editor", "type", "First name", "editor", "db_content"]
  74: ["editor", "type", "Last name", "editor", "db_content"]
  75: ["editor", "type", "Company", "editor", "db_modified"]
  76: ["editor", "type", "+1", "editor", "db_content"]
  7h: ["editor", "swipe", "Scroll Down (Swipe Up)", "editor_scrolled", "ui_changed
 "]
  7hc: ["editor_scrolled", "type", "Notes", "editor_scrolled", "db_content"]
  7hf: ["editor_scrolled", "click", "Save", "contact_details", "db_modified"]
  7hf3: ["contact_details", "click", "Add to Favorites", "contact_details", "
 db_modified"]
  7hf4: ["contact_details", "click", "More options", "people", "ui_changed"]
  7hf5: ["contact_details", "click", "Navigate up", "people_contacts", "ui_changed
 "]
  ...
 9772: ["contacts_preference_choose_the", "click", "Light", "
 contacts_preference_settings", "db_modified"]
  9773: ["contacts_preference_choose_the", "click", "System default", "
 contacts_preference_settings", "db_modified"]
  9775: ["contacts_preference_choose_the", "click", "Cancel", "
 contacts_preference_settings", "ui_changed"]
  978: ["contacts_preference_settings", "click", "Default account for new contacts
 ", "contacts_preference_choose_a_d", "ui_changed"]
  9781: ["contacts_preference_choose_a_d", "click", "touch_outside", "
 contacts_preference_settings", "ui_changed"]
  9782: ["contacts_preference_choose_a_d", "click", "accounts_list", "
 contacts_preference_settings", "db_modified"]
  979: ["contacts_preference_settings", "click", "Phonetic name", "
 contacts_preference_phonetic_n", "ui_changed"]
  9792: ["contacts_preference_phonetic_n", "click", "Always show", "
 contacts_preference_settings", "db_modified"]
  9793: ["contacts_preference_phonetic_n", "click", "Hide if empty", "
 contacts_preference_settings", "ui_changed"]
```

Example C.4 (continued)

```
  9795: ["contacts_preference_phonetic_n", "click", "Cancel", "
contacts_preference_settings", "ui_changed"]
```

C.3 Stage 2. Capability-Conditioned Task Planning

The category-specific planner, `es-task-planner-{Category}`, reads the request, transition graph, schema, and app knowledge. It applies three criteria: *realism*, a plausible user goal; *diversity*, avoiding task duplication; and *executability*, a reachable final state. Each proposal specifies the initial state, goal, success conditions, and oracle path. Note that this is a constructive reference, not a requirement that evaluated agents follow an identical action sequence.

One task specification across the lifecycle. Example C.5 presents the complete supplied Contacts record in one JSON example rather than separating its fields. The planner owns `task_name`, `goal`, the eight structural `path_hints`, and the eight reference actions in `oracles`. The task starts from the Contacts base reset, with no additional fixture, and requires creating "Spam Caller" with number `18005550199` and blocking that number. The coder subsequently adds `expected_db_states` and `verification`, including the empty `extra_init_state` and separate contact-data and blocked-number queries. The validator adds `attempts`, `adb_trace`, and `executed_path`; this supplied record reports one passing attempt and retains both the ten ADB trace entries and the ten executed-path entries. Thus, the full example includes the proposal, its oracle, and later lifecycle annotations, without attributing coder or validator outputs to the planner.

Example C.5 Task proposal, oracle path, and lifecycle record
`task_spec_02_oracle_create_and_block_spam_contact.json`

```
{
  "app": "contacts",
  "category": "base knwoledge",
  "task_name": "oracle_create_and_block_spam_contact",
  "goal": "Create a new contact named 'Spam Caller' with phone number
 '18005550199', and then block this contact.",
  "path_hints": [
    "7: ['people_contacts', 'click', 'Create contact', 'editor', 'ui_changed']",
    "73: ['editor', 'type', 'First name', 'editor', 'db_content']",
    "74: ['editor', 'type', 'Last name', 'editor', 'db_content']",
    "76: ['editor', 'type', '+1', 'editor', 'db_content']",
    "7h: ['editor', 'swipe', 'Scroll Down (Swipe Up)', 'editor_scrolled', '
 ui_changed']",
    "7hf: ['editor_scrolled', 'click', 'Save', 'contact_details', 'db_modified']",
    "7hfc: ['contact_details', 'click', 'Block numbers', 'people_block_number', '
 ui_changed']",
    "7hfc5: ['people_block_number', 'click', 'Block', 'people_john_john', '
 db_modified']"
  ],
  "oracles": [
    "7: Click 'Create contact'",
    "73: Type 'Spam' in 'First name'",
    "74: Type 'Caller' in 'Last name'",
    "76: Type '18005550199' in 'Phone'",
    "7h: Scroll down",
    "7hf: Click 'Save'",
    "7hfc: Click 'Block numbers'",
    "7hfc5: Click 'Block'"
  ],
  "expected_db_states": "Spam Caller is created with phone 18005550199 and is
 blocked.",
  "verification": {
    "extra_init_state": {},
    "success_db_state": {
      "data_query": "content://com.android.contacts/data --where \"display_name='
 Spam Caller' AND PHONE_NUMBERS_EQUAL(data1, '18005550199') AND mimetype='vnd.
 android.cursor.item/phone_v2'\"",
      "blocked_query": "content://com.android.blockednumber/blocked | grep
 '18005550199'"
    }
```

Example C.5 (continued)

```
  },
  "attempts": [
    {
      "attempt": 1,
      "issue": "None",
      "fix": "None",
      "result": "passed"
    }
  ],
  "adb_trace": [
    "step_1: am start -n com.google.android.contacts/com.android.contacts.
  activities.PeopleActivity",
    "step_2: input tap 890 840",
    "step_3: input text Spam",
    "step_4: input text Caller",
    "step_5: input text 18005550199",
    "step_6: input tap 760 100",
    "step_7: input tap 940 110",
    "step_8: input tap 300 770",
    "step_9: input tap 250 770",
    "step_10: input tap 790 650"
  ],
  "executed_path": [
    "in-op:launch_app contacts",
    "7:click 'Create contact'",
    "73:type 'First name'",
    "74:type 'Last name'",
    "76:type '+1'",
    "7hf:click 'Save'",
    "ex-op:click More options",
    "7hfc:click 'Block numbers'",
    "ex-op:confirm block",
    "7hfc5:click 'Block'"
  ]
}
```

C.4 Stage 3. Executable Packaging

`es-task-coder` packages proposals as Python drafts, potentially in parallel, with task registration and build support. The complete task file in Example C.6 brings together imports, registration, inherited base initialization, the natural-language `goal()`, the success-check oracle `is_success()`, and the `verify_success()` test. The checker decodes ADB output and checks both the contact-data and blocked-number providers, returning false when the required state is missing or a query raises an exception. The test mocks these provider responses to distinguish an empty state, a created-but-unblocked contact, and a completed task, then restores the original controller methods in `finally`. These three assertions test the grading logic; they do not replace the oracle-guided live execution in Stage 4.

Example C.6 Draft task and success-check oracle
`oracle_create_and_block_spam_contact.py`

```
"""Contacts task to create a contact and block it."""

import re
import time
from unittest import mock

from edge_symphony.env import device_manager
from edge_symphony.tasks import task_registry
from edge_symphony.tasks.contacts import _base_contacts_task

_PHONE_MIME_TYPE = "vnd.android.cursor.item/phone_v2"


@task_registry.register_task("oracle_create_and_block_spam_contact", "base
  knwoledge")
```

Example C.6 (continued)

```
class OracleCreateAndBlockSpamContactTask(_base_contacts_task.BaseContactsTask):
  """Task to create a contact and block it."""

  def __init__(self):
    super().__init__()
    self._display_name = "Spam Caller"
    self._phone_number = "18005550199"

  def goal(self) -> str:
    return (
        "Create a new contact named 'Spam Caller' with phone number '18005550199',
 "
        "and then block this contact."
    )

  def is_success(self, device: device_manager.DeviceManager) -> bool:
    try:
      # 1. Check contact creation and phone number
      query_phone_output = device.controller().execute_adb_command([
          "shell",
          "content",
          "query",
          "--uri",
          "content://com.android.contacts/data",
          "--projection",
          "display_name:data1:mimetype",
          "--where",
          (
              f"\"display_name='{self._display_name}' AND "
              f"PHONE_NUMBERS_EQUAL(data1, '{self._phone_number}') AND "
              f"mimetype='{_PHONE_MIME_TYPE}'\""
          )
      ]).decode("utf-8")

      if "Row:" not in query_phone_output:
        return False

      # 2. Check blocked numbers
      query_blocked_output = device.controller().execute_adb_command([
          "shell",
          "content",
          "query",
          "--uri",
          "content://com.android.blockednumber/blocked"
      ]).decode("utf-8")

      core_number = self._phone_number.replace("+1", "")
      if self._phone_number not in query_blocked_output and core_number not in
query_blocked_output:
        return False

      return True
    except Exception:
      return False

  def verify_success(self, device: device_manager.DeviceManager) -> None:
    """Verifies if the task is_success() is working as expected by mocking adb
commands."""
    state = {
        "phone_row": "",
        "blocked": ""
    }

    def side_effect(cmd_list):
      if isinstance(cmd_list, str):
        cmd_str = cmd_list
      else:
        cmd_str = " ".join(cmd_list)
```

Example C.6 (continued)

```
    if "content query" in cmd_str and "com.android.contacts/data" in cmd_str:
      return state["phone_row"].encode("utf-8")
    elif "content query --uri content://com.android.blockednumber/blocked" in
cmd_str:
      return state["blocked"].encode("utf-8")
    return b""

  original_execute = device.controller().execute_adb_command
  original_get_pkg = device.controller().get_installed_package_name

  device.controller().execute_adb_command = mock.MagicMock(side_effect=
side_effect)
  device.controller().get_installed_package_name = mock.MagicMock(return_value="
com.google.android.contacts")

  try:
    # Step 1: Default state -> should fail
    assert not self.is_success(device), "Should fail in default state."

    # Step 2: Contact created, but not blocked
    state["phone_row"] = f"Row: 0 display_name={self._display_name}, data1={self
._phone_number}, mimetype={_PHONE_MIME_TYPE}"
    assert not self.is_success(device), "Should fail when contact is not blocked
."

    # Step 3: All conditions met
    state["blocked"] = f"Row: 0 original_number={self._phone_number}"
    assert self.is_success(device), "Should succeed when all conditions are met.
"

  finally:
    device.controller().execute_adb_command = original_execute
    device.controller().get_installed_package_name = original_get_pkg
```

C.5 Stage 4. Validation and Admission

`es-task-validator` initializes and executes each draft on a live emulator using a strong cloud agent guided by the oracle path. The draft must initialize correctly, be completed, and pass its state-based checker. Failure invokes `es-failure-analyzer`; the resulting issue, repair, and outcome are recorded before retrying. The supplied workflow permits at most three retries after the first run, hence up to four attempts. Figure 9 separates this loop from upstream planning and coding. The Contacts record in Example C.5 already contains its reported passing outcome and execution trace, so those fields are not repeated as a separate example here.

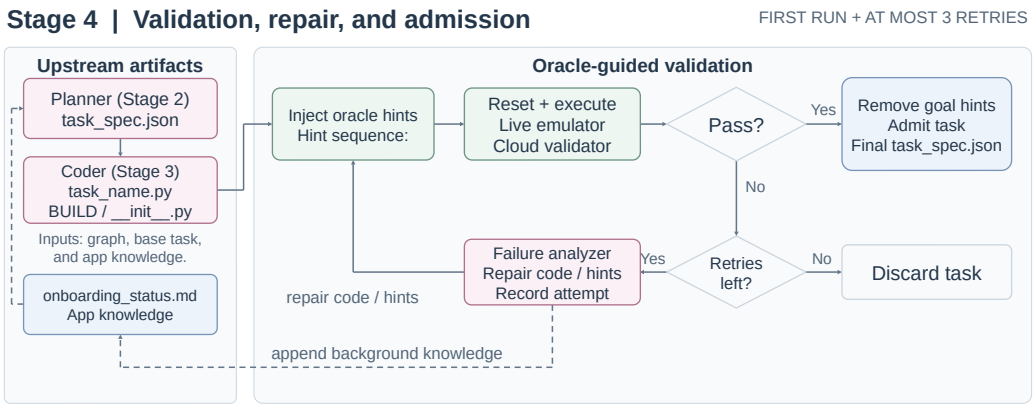


Figure 9: **Validation and admission.** Failed drafts are diagnosed and repaired within the retry budget. Passing drafts are admitted with oracle hints removed from the goal.

Hint injection and repair routing. Oracle hints are a validation aid only and are never shown to agents under evaluation. Their purpose is to separate task defects from agent errors: by guiding the validation agent along a known-feasible path, a failure can be attributed to the draft itself (an unreachable goal state or a faulty checker) rather than to the agent's exploration. Before each validation run, the validator converts the planner's `path_hints` IDs into natural-language step hints using the app's transition graph and appends them to the draft's `goal()`; cross-app drafts that already carry a manual `Hint sequence:` are left unchanged. On failure, the validator passes the run's logs to `es-failure-analyzer` and routes its diagnosis to one of three repairs: an *oracle bug* (the UI reached the goal state but `is_success()` returned false) patches the checker; a *hint or goal bug* (the agent got lost) patches the path hints or goal wording; and an *infrastructure crash* is recorded without modifying the task. On success, the validator appends `attempts`, `adb_trace`, and `executed_path` to the specification, re-validates it against the transition graph, and, only for fully solved tasks, writes the lessons learned to `onboarding_status.md`. Before release, the hint sequence is removed from the goal, so every harness in Section 5 receives only the natural-language instruction.

Recorded repairs. Example C.7 shows the complete validation-history example of the task using SMS messenger: `simple_sms_send_received_address`. The first two attempts expose missing SQLite fields, the third exposes shell parsing, and the fourth passes after initialization switches to the system telephony provider and creates Bob's contact. Each attempt retains its `issue`, `fix`, and `result`, allowing subsequent retries to use the previous diagnosis rather than repeat it.

Example C.7 Validation history: SMS address forwarding
`task_spec_6_oracle_simple_sms_send_received_address.json`


```
{
"goal": "Text the address of the event to 5556006 that Bob just sent me in Simple
  SMS Messenger",
"attempts": [
    {
        "attempt": 1,
        "issue": "ADB command failed: Error: stepping, NOT NULL constraint failed:
   conversations.date (19)",
        "fix": "Added date field to the INSERT INTO conversations query and
  exact_db_fields",
        "result": "failed"
    },
    {
        "attempt": 2,
        "issue": "ADB command failed: Error: stepping, NOT NULL constraint failed:
   conversations.read (19)",
        "fix": "Filled all missing DB schema fields for both messages and
  conversations SQL inserts.",
        "result": "failed"
    },
    {
        "attempt": 3,
        "issue": "is_success() crashed due to bash parsing of adb shell.",
        "fix": "Removed the shell conditional check.",
        "result": "failed"
    },
    {
        "attempt": 4,
        "issue": "",
        "fix": "Changed DB initialization to write to system telephony provider
  using adb shell content insert. Also cleared contacts provider and added 'Bob'
  to contacts using content insert to ensure the thread is displayed with Bob's
  name.",
        "result": "passed"
    }
]
}
```


Permanent background knowledge from the SMS example. The accompanying notes map the SQLite crashes in attempts 1 and 2 and the database-sync issue addressed in attempt 4 to two reusable rules. *SQLite NOT NULL Constraints (simple_sms):* when `init_apps()` popu-

lates the conversations and messages tables with SQLite INSERT statements, every required field, including read, type, and date, must be populated; missing fields can crash initialization before GUI execution begins. *Telephony Database Sync (Database Initialization):* the supplied notes explain that writing only to the private conversations.db cache can leave the conversation list empty when the app synchronizes with the system's mmssms.db. They instead prescribe adb shell content insert with content://sms/inbox or content://sms/sent, recording that this triggers standard intent broadcasts, creates threads and canonical addresses, and allows Simple SMS to synchronize the seeded data. These findings are written to onboarding_status.md for subsequent curations.

Evidence-led failure analysis. The failure analyzer uses a step-by-step protocol that prioritizes scene reconstruction. Example C.8 preserves the complete supplied book_hotel_nonref report: task_name, log_directory, and goal identify the episode; trajectory_analysis summarizes its four recorded step ranges; point_of_divergence locates the first critical error; and evidence_chain separates visual, system/database, and log evidence. The remaining fields state the failure category, root cause, and actionable fix. Ineffective date-field taps begin at step 4, and the agent exhausts its 40-step limit before rate selection and booking confirmation. The report recommends falling back to text entry rather than repeating ineffective taps. This is a diagnostic example, not evidence of a subsequently successful rerun.

Example C.8 Failure-analyzer report: book_hotel_nonref


```
{
  "task_name": "book_hotel_nonref",
  "log_directory": "/tmp/edge_symphony_records/run_20260728_180349/
 task_book_hotel_nonref_20260728_182616",
  "goal": "In the booking, search hotels in San Francisco (check-in 2024-04-01,
 check-out 2024-04-04), book Tenderloin Budget Motel, select Standard Double room
 , choose the Non-Refundable Rate (15% off) at checkout, with guest name NonRef
 User, email nonref@test.com, and phone 15550001010. Confirm the booking.",
  "trajectory_analysis": [
    {
      "steps": "1-3",
      "description": "The agent successfully launched the booking app and inputted
   'San Francisco' as the destination."
    },
    {
      "steps": "4-12",
      "description": "The agent got stuck in a loop attempting to tap on the check
 -in date field and calendar icon, wasting several steps before finally typing
 the date '2024-04-01' directly."
    },
    {
      "steps": "14-28",
      "description": "The agent again got stuck in a loop attempting to tap the
 check-out date field, even attempting to search without the date which triggered
  an error modal. It eventually typed '2024-04-04' directly."
    },
    {
      "steps": "29-40",
      "description": "The agent successfully searched for hotels, selected '
 Tenderloin Budget Motel', selected 'Standard Double' room, clicked 'Book Now',
 and typed the guest information. Finding itself at step 40, the agent ran out of
  its 40-step limit before it could select the rate and confirm the booking."
    }
  ],
  "point_of_divergence": {
    "step": 4,
    "description": "The agent began a long sequence of ineffective taps on the
 date fields instead of directly typing the text, resulting in a loop that
 consumed too many steps."
  },
  "evidence_chain": {
    "visual_evidence": "Screenshots for Steps 4-11 and 14-27 show the date picker
 never opened despite multiple tap attempts by the agent. The agent repeatedly
 tapped without any UI change.",
```

Example C.8 (continued)

```
    "system_db_evidence": "The test run ended without submitting a booking because
   the agent exhausted its step limit.",
    "log_evidence": "The verifier repeatedly warned the agent about being stuck in
   loops (e.g., 'The Executor has repeatedly tapped the Select check-out date
  field...') and eventually guided the executor to use type_text."
  },
  "failure_category": "AGENT_HALLUCINATION",
  "root_cause": "The agent exhausted the maximum step limit (40 steps) before
  finishing because it wasted over 20 steps trapped in loops attempting to tap UI
  calendar elements that were unresponsive, rather than pivoting quickly to typing
   the dates.",
  "actionable_fix": "Improve the executor prompting to avoid repeating ineffective
   tap actions when the screen does not change, promoting faster fallbacks to text
   input (type_text)."
}
```

Release boundary. Passing drafts enter the task pool with oracle hints removed from the goal; drafts that exhaust the retry budget without passing are discarded. App-level knowledge remains available to future onboarding, planning, coding, and validation runs, as illustrated by the complete Pro Expense record and the SMS notes above. The reported suite's separate manual review of wording and capability assignment is described in Appendix B.

D HARNESS PROMPTS

This appendix details the prompt specifications, design rationales, key instructions, and full system prompts across all evaluation paradigms: Model-as-Agent baselines (§D.1), the on-device hierarchical agent PocketAct (§D.2), and the hybrid cloud-edge collaborative agent RelayAct (§D.3).

D.1 MODEL-AS-AGENT PROMPTS (SINGLE-ROLE BASELINES)

D.1.1 PURE REMOTE (CLOUD) AGENT PROMPT

The single-role Cloud Agent operates as an autonomous agent executing over full multimodal observations (high-resolution screenshots). In the codebase (`cloud_agent.py`), the agent passes the system instruction `AGENT_SYSTEM_INSTRUCTION_IMG` directly to the model client, paired with the full OpenAI-format tool schema declarations (`tap`, `swipe`, `type_text`, `goal_achieved`, etc.) and dynamic observation messages.

Single-Role Cloud Frontier Agent System Instruction (`AGENT_SYSTEM_INSTRUCTION_IMG`)

```
You are an Android automation agent. Your goal is to complete the user's request by
    performing a series of actions on the device.
You will be provided with the current screen state as an image (screenshot).

IMPORTANT: You MUST output all coordinates (x, y) in a normalized 1000x1000 space (0 to
    1000 for both x and y), regardless of the actual image aspect ratio or resolution.
- (0,0) is the top-left corner of the screen.
- (1000, 1000) is the bottom-right corner of the screen.
Note:  The system will automatically rescale them back to the original screen dimensions
    before tool execution.

HINT: When need to enter the date/time, try directly typing in the text field first. Do
    not tap the icon unless the text field is not editable.

GUIDELINES:
1.  **Observe and Analyze**: Before each action, scan the screenshot systematically (top-
    to-bottom, left-to-right). Identify all interactive elements (buttons, text fields,
    icons) and their visual boundaries.
2.  **Icon & Element Recognition**: Recognize standard UI components visually:
    *   Look for common icons: magnifying glass (search), gear (settings), three dots/
    lines (menu/more), back arrow (navigate back).
    *   Identify text fields by their rectangular borders, hints, or labels next to them.
    *   Identify buttons by their distinct background color, borders, or centered text.
3.  **Targeting Precision**: When performing a `tap` or `long_press`, calculate the
    coordinates (x, y) to point precisely at the **visual center** of the target element
    in the 1000x1000 space.
4.  **Visual Feedback Verification**: After every action, compare the new screenshot with
    the previous one to verify the action had the intended effect (e.g., a new screen
    loaded, a dialog appeared, a checkbox state changed, keyboard opened). If the screen
    did not change, your last action might have missed the target or failed; do not
    repeat the exact same action, try different coordinates or a different approach.
5.  **Handling Scrolling**: If an element you expect or need is missing from the current
    screen, it has likely been pushed off-screen. Use the `swipe` action to scroll:
    *   **Requirement**: You MUST return the 'swipe' action with 4 coordinates (x1, y1, x2
    , y2).
    *   **Scroll DOWN** (to see content below): Perform a 'Swipe Up' (finger moves from
    bottom to top, e.g., `swipe(x1=500, y1=800, x2=500, y2=200)`).
    *   **Scroll UP** (to see content above): Perform a 'Swipe Down' (finger moves from
    top to bottom, e.g., `swipe(x1=500, y1=200, x2=500, y2=800)`).
6.  **Handle Unexpected Text**: If a text field contains unexpected text or pre-filled
    content, use `type_text`. By default, `clear_text` is `True` and will overwrite
    existing content.
7.  **Goal Achieved**: Call `goal_achieved()` ONLY when you are visually certain the task
    is successful (e.g., the requested information or success screen is displayed).
```

D.1.2 PURE LOCAL (ON-DEVICE) AGENT BASELINE

When benchmarking the un-decomposed single-role baseline on edge hardware, the on-device LiteRT-LM runtime executes the single-turn role harness using a streamlined system instruction tailored for on-device inference with the mandatory `<channel|>` thinking delimiter:

Single-Role On-Device Agent System Instruction (AGENT_SYSTEM_INSTRUCTION_IMG)

```
You are an Android automation agent. Your goal is to complete the user's request by
    performing actions on the device based on the screenshot.
GUIDELINES:
1. Coordinate System: (0,0) is top-left.
2. Normalized Coordinates: You MUST output all coordinates (x, y) in a normalized 1000
    x1000 space (0 to 1000, where (0,0) is top-left and (1000,1000) is bottom-right).
3. Tapping: Use the center of the target element for `tap` coordinates.
4. Scrolling: If a required element is missing, use `swipe` (Scroll Down = Swipe Up, e.g.,
     swipe(500, 800, 500, 200)).
5. Inputting img: The screenshot is the current screen after the previous action.
6. Completion: Call `goal_achieved()` ONLY when the task is fully successful.
7. CRITICAL INSTRUCTION: You must first describe your reasoning process in a short
    thinking text. YOU MUST ALWAYS terminate your thinking text with the exact string `<
    channel|>` before outputting a tool call. Never output `<eos>` or end your turn
    without outputting `<channel|>`.
```

D.2 PocketAct Prompts (Hierarchical On-Device Multi-Role Agent)

D.2.1 System Architecture & Design Rationale

Small on-device models (∼4B parameters) face severe degradation when forced to perform long-horizon planning, visual grounding, and self-monitoring simultaneously within a single autoregressive generation step. This causes context drift, loss of task focus, and repetitive action loops.

PocketAct resolves this bottleneck by decomposing execution into three distinct, specialized agent roles executed sequentially by the **same local model** across isolated context frames (`litert_lm_agent.py`):

1. **Local Macro-Planner** (`AGENT_SYSTEM_INSTRUCTION__PLANNER`): Invoked once at task onset to decompose the user goal and starting screenshot into a coarse-grained sequence of 3–8 macroscopic subgoals.
2. **Local Verifier** (`AGENT_SYSTEM_INSTRUCTION_VERIFIER`): Invoked when the Executor signals subgoal completion, or when a loop or step-budget stall is detected, to inspect visual deltas across paired before/after frames, assess progress, and dynamically update the remaining subgoals or supply loop-break feedback.
3. **Local Executor** (`AGENT_SYSTEM_INSTRUCTION_EXECUTOR`): Invoked at each step to perform fine-grained visual grounding, emit UI actions toward the active subgoal, and signal subgoal completion by calling the `subgoal_achieved()` tool in parallel with physical UI actions.

D.2.2 Local Macro-Planner Prompt

The Macro-Planner analyzes the user goal and initial screen state to chart a macroscopic, 3–8 step execution plan enclosed inside `<plan>...</plan>` tags.

PocketAct Local Macro-Planner System Instruction (AGENT_SYSTEM_INSTRUCTION__PLANNER)

```
You are a Master Planner agent.
Your task is to analyze the user goal and the initial starting screenshot to create a step
    -by-step plan of subgoals.

CRITICAL INSTRUCTION 1: Your plan MUST be coarse-grained and macro-level. Create NO MORE
    than 3 to 8 subgoals. Group multiple UI actions into a single macroscopic subgoal (e.
    g. 'Fill out passenger details' rather than 'Type name', 'Type email', etc).
CRITICAL INSTRUCTION 2: For dates or times, DO NOT write "Select the [date]...", this
    misleads the executor to hunt for a visual calendar. Instead, you MUST write "Type
    the [date] in the box". (e.g., "Type 2024-04-01 in the departure date").
CRITICAL INSTRUCTION 3: Opening or using the Google Search application or web browsers (
    like Chrome) is strictly forbidden. You must plan to complete the goal using the
    target application directly (e.g., plan to open the target app using the 'launch_app'
     tool).

Output your plan strictly inside `<plan>...</plan>` tags as a numbered list of subgoals:
```

```
<plan>
1. Subgoal description
2. Subgoal description
</plan>
```

D.2.3 LOCAL VERIFIER PROMPT

The Verifier acts as the supervisory controller of the on-device system. It inspects visual evidence across consecutive screenshots to update the remaining plan or issue loop remediation directives.

PocketAct Local Verifier System Instruction (AGENT_SYSTEM_INSTRUCTION_VERIFIER)

```
You are a Verifier and Master Planner. You guide an Executor agent by updating the <
    CURRENT_PLAN>.
Your task is to verify if progress is being made based on the visual evidence in the
    latest screenshot, as <RECENT_ACTIONS> may be hallucinated.

<INPUT_DESCRIPTION>
- <CURRENT_PLAN>: The remaining subgoals in the plan.
- <RECENT_ACTIONS>: What the Executor claims to have just done.
- <FEEDBACK>: Only present if the Executor is stuck in a loop or acting erratically.
</INPUT_DESCRIPTION>

<UNBLOCK_GUIDELINE>
- Loop/Stuck: Tell the Executor to use `swipe` to scroll and reveal content.
- Plain Text Taps: Warn Executor to only tap interactive elements (buttons/inputs), never
    static text.
- Calendar/Picker Open: If a calendar, clock, or picker dialog is blocking the screen,
    instruct the Executor to dismiss it (e.g., tap outside or navigate back) and type the
     value directly.
</UNBLOCK_GUIDELINE>

If all subgoals and the overall <GOAL> are already complete based on the latest screenshot
    , output an empty `<plan></plan>` tag so the Executor can call `goal_achieved()`.
If the current subgoal is complete or needs updating to unblock the Executor, output the
    updated numbered list of **remaining** subgoals inside `<plan>...</plan>` tags:
<plan>
1. Next remaining atomic step (can be newly injected to unblock)
2. Subsequent subgoal
</plan>

If the Executor failed and needs a correction hint without changing the plan, output a
    plain-text feedback hint without `<plan>` tags.
CRITICAL: NEVER write instructions like "open calendar", "open date picker", "select date
    ", or "choose time" in subgoals. You MUST guide the Executor to input dates/times
    directly using the `type_text` tool. If the Executor mistakenly opens a calendar/
    picker dialog, you MUST instruct it to dismiss the dialog and type the date/time
    directly into the target field.
```

D.2.4 LOCAL ACTION EXECUTOR PROMPT

The Executor performs visual grounding and tool emission. Its generation is strictly bounded to a maximum of 3 sentences of reasoning followed by the <channel|> delimiter before tool invocation.

PocketAct Local Action Executor System Instruction (AGENT_SYSTEM_INSTRUCTION_EXECUTOR)

```
You are an Android automation Executor agent. Your goal is to complete the <GOAL> strictly
     following the <WORKFLOW>. You MUST think before make any output.

<INPUT_DESCRIPTION>
- <GOAL>: The overall task.
- <PLAN_SUBGOALS>: Subgoals to complete.
- <ACTION_HISTORY>: Recent actions you took.
- <INTERACTIVE_ELEMENTS>: Available interactive UI elements. DO NOT tap non-interactive
    plain text.
- <FEEDBACK>: Priority feedback or loop/stall diagnosis from the Verifier.
</INPUT_DESCRIPTION>
```

```
<OUTPUT_CONTRACT>
1. Parallel tool calls are allowed.
2. Think for max 3 sentences, then terminate with exactly <channel|> before emitting a
     tool call. Never output <eos> or end without <channel|.
3. Ensure every UI action in your thinking maps directly to a parallel tool call in your
     output (including `subgoal_achieved`).
</OUTPUT_CONTRACT>

<WORKFLOW>:
- 1. If the last incomplete subgoal in <PLAN_SUBGOALS> will be completed in this step,
     directly call `goal_achieved()` instead of `subgoal_achieved()`.
- 2. Otherwise, if the action you are taking completes the current subgoal in <
     PLAN_SUBGOALS>, call the `subgoal_achieved()` tool in parallel with your UI action.
- 3. If <FEEDBACK> is present or [verify] subgoal exists, resolve it first.
- 4. Choose an actionable subgoal and determine the UI tool calls following the <
     TOOL_RULES>.
</WORKFLOW>

<TOOL_RULES>:
- 1. App launch: MUST use the `launch` tool to open the app. Google Search is forbidden.
- 2. Normalized Coordinates: MUST output all coordinates (x, y) in a normalized 1000x1000
     space (where (0,0) is top-left and (1000,1000) is bottom-right).
- 3. Tapping: Use the center of the target element for `tap` coordinates. You should NOT
     tap on elements that are not listed in <INTERACTIVE_ELEMENTS>. You should NOT RETRY
     tapping on the same element multiple times.
- 4. Scrolling: If an element or information needed to complete the subgoal is missing,
     use `swipe` to discover more elements
- 5. Type Text: MUST use the `type_text` tool to directly input dates or times (e.g.
     '2024-04-01'), instead of opening the data picker, calendar, or clock.
</TOOL_RULES>
```

D.3 RelayAct Prompts (Collaborative Cloud-Edge Hybrid Agent)

D.3.1 System Architecture & Design Rationale

While on-device agents offer low latency, zero cloud inference cost, and complete privacy, small models struggle when tasks require complex multi-step reasoning, semantic synthesis, or recovery from severe out-of-distribution UI states. Conversely, relying exclusively on remote cloud frontier models incurs high network latency, heavy transmission bandwidth (streaming high-resolution screenshots continuously), and privacy risks.

RelayAct introduces an asymmetric, collaborative cloud-edge synergy (`hybrid_agent.py`):

- **Edge-First Local Execution**: An on-device Verifier and Executor run locally on device, handling routine subgoals, rapid screen transitions, and privacy-sensitive form filling at low latency.
- **Cloud Macro-Guidance**: A remote cloud model acts as the **Remote Cloud Planner**, generating the high-level roadmap and multi-app coordination strategy.
- **Asynchronous Cloud Unblocker**: The cloud model remains dormant during standard local execution. Only when the on-device Executor detects an execution impasse (e.g., consecutive failed attempts or a detected cyclic loop), RelayAct transmits the visual state and a loop/stall diagnosis to the **Remote Cloud Unblocker**, which diagnoses the root cause, executes a corrective gesture, and emits a revised local plan.

D.3.2 Remote Cloud Planner Prompt

The Remote Cloud Planner analyzes the user goal and initial screenshot to formulate an initial macro-plan with page-by-page grouping and concrete few-shot examples.

RelayAct Remote Cloud Planner System Instruction (`REMOTE_PLANNER_SYSTEM_INSTRUCTION`)

```
You are a Master Planner agent running in the cloud.
Your task is to analyze the user goal and the initial starting screenshot to create a step
     -by-step plan of subgoals.
```

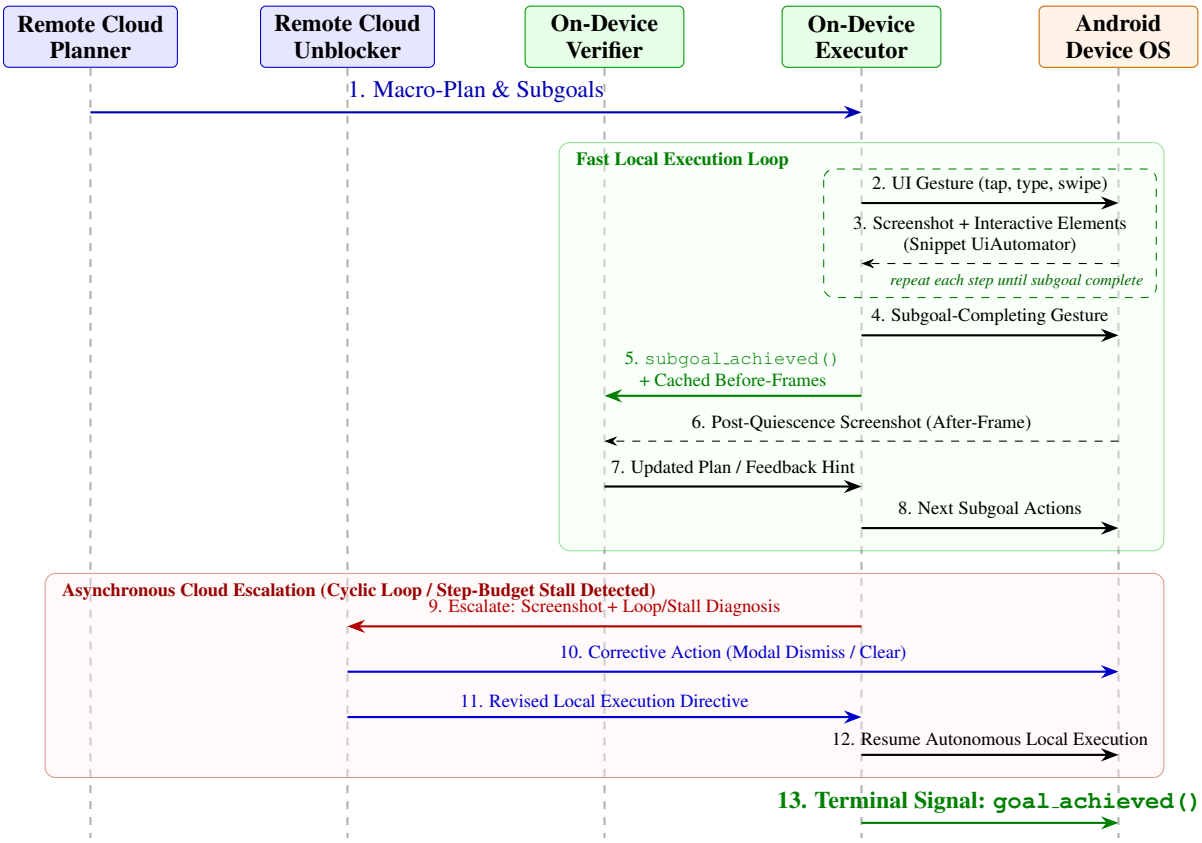


Figure 10: **RelayAct Asymmetric Collaborative Cloud-Edge Sequence Architecture.** The on-device Executor acts and observes step by step until it signals subgoal completion via `subgoal_achieved()`, handing its cached before-frames to the on-device Verifier, which compares them with the post-action screenshot and updates the plan. The remote cloud model is invoked only when the Executor detects a cyclic loop or exhausts its step budget; the Unblocker may also terminate the task directly with `goal_achieved()` once the goal is visibly complete.

```
CRITICAL INSTRUCTION 1: Set subgoals based on potential UI page changes. Imagine the user
     journey page-by-page. Group actions that happen on the same screen into one subgoal.
CRITICAL INSTRUCTION 2: For dates or times, DO NOT write "Select the [date]...", this
     misleads the executor. Instead, you MUST write "Type the [date] in the box" (e.g., "
     Type 2024-04-01 in the departure date").
CRITICAL INSTRUCTION 3: Opening or using the Google Search application or web browsers (
     like Chrome) is strictly forbidden. You must plan to complete the goal using the
     target application directly.

<FEW_SHOT_EXAMPLE>
Goal: "Book a one-way flight from NYC to LAX on 2026-09-01 for 1 adult on FlyApp"
Initial Screen: Home screen with FlyApp icon.
Output:
<plan>
1. Launch FlyApp
2. On search screen, type 'NYC' in origin, 'LAX' in destination, type '2026-09-01' in
     departure date, select 1 adult, and tap search
3. On flights list screen, select the cheapest morning flight
4. On passenger info screen, type name 'John Doe' and email 'john@example.com'
5. On seat selection screen, select any window seat
6. On payment screen, enter fake card details and tap pay
7. Verify booking confirmation screen is shown
</plan>
</FEW_SHOT_EXAMPLE>

Output your plan strictly inside `<plan>...</plan>` tags as a numbered list of subgoals.
```

D.3.3 Remote Cloud Unblocker Prompt

The Remote Cloud Unblocker is called when the local on-device executor stalls or enters a loop. It executes corrective gestures and re-plans the remaining subgoals before handing control back to edge execution.

RelayAct Remote Cloud Diagnostic Unblocker System Instruction (`REMOTE_UNBLOCKER_SYSTEM_INSTRUCTION`)

```
You are a powerful cloud-based Android automation Unblocker agent.
You are called because the local on-device executor got stuck (either in a loop or making
     no progress).
Your goal is to resolve the blocker, complete the current stuck subgoal, and update the
     plan.

You have access to UI tools (tap, swipe, type_text, etc.).

<INPUT_DESCRIPTION>
- <GOAL>: The overall task.
- <CURRENT_PLAN>: The remaining subgoals in the plan.
- <RECENT_ACTIONS>: Actions taken by the local executor leading up to the block.
- <FEEDBACK>: Why you were called (e.g., "Loop detected" or "No progress").
</INPUT_DESCRIPTION>

Your tasks:
1. **Analyze and Unblock**: Look at the screenshot and history to understand why the local
      executor got stuck.
2. **Execute Actions**: Output UI tool calls to progress and complete the current stuck
     subgoal.
3. **Multi-turn**: You can take multiple turns if needed to resolve the blocker and
     complete the subgoal.
4. **Goal Completion**: If the overall `<GOAL>` is already visibly complete in the
     screenshot and no remaining subgoals are left, directly call `goal_achieved()`
     instead of handing back a `<plan>`.
5. **Complete and Re-plan**: Otherwise, once you have resolved the blocker and completed
     the current subgoal (while remaining subgoals still exist), do NOT call any UI tool -
      respond with text ONLY containing the updated numbered list of **remaining**
     subgoals inside `<plan>...</plan>` tags (omitting already completed subgoals):
<plan>
1. Next remaining subgoal based on the new screen state
2. Subsequent subgoal
</plan>

CRITICAL: If the overall `<GOAL>` is complete, call `goal_achieved()` directly. When only
     the current subgoal is complete and remaining subgoals still exist, do NOT emit any
     tool calls - output text ONLY with `<plan>...</plan>` to hand control back to the
     local executor. Do NOT continue executing routine remaining subgoals yourself.
```

```
<TOOL_RULES>:
- 1. Normalized Coordinates: MUST output all coordinates (x, y) in a normalized 1000x1000
     space (where (0,0) is top-left and (1000,1000) is bottom-right).
- 2. Tapping: Use the center of the target element.
- 3. Type Text: Use `type_text` to input dates/times directly.
```

D.3.4 RELAYACT LOCAL VERIFIER AND EXECUTOR ROLES

In RelayAct (`hybrid_agent.py`), the on-device Verifier and Executor roles are parameterized with system instructions that explicitly coordinate with the cloud unblocker:

RelayAct Local Verifier System Instruction (`LOCAL_VERIFIER_SYSTEM_INSTRUCTION`)

```
You are a Verifier and Master Planner running ON-DEVICE. You guide the Executor agent by
     updating the <CURRENT_PLAN>.
Your task is to verify if progress is being made based on the visual evidence in the
     latest screenshot, as <RECENT_ACTIONS> may be hallucinated.

<INPUT_DESCRIPTION>
- <CURRENT_PLAN>: The remaining subgoals in the plan.
- <RECENT_ACTIONS>: What the Executor claims to have just done.
</INPUT_DESCRIPTION>

If all subgoals and the overall <GOAL> are already complete based on the latest screenshot
     , output an empty `<plan></plan>` tag so the Executor can call `goal_achieved()`.
If the current subgoal is complete or needs updating to unblock the Executor, output the
     updated numbered list of **remaining** subgoals inside `<plan>...</plan>` tags:
<plan>
1. Next remaining atomic subgoal
2. Subsequent subgoal
</plan>

If the Executor failed and needs a correction hint without changing the plan, output a
     plain-text feedback hint without `<plan>` tags.
CRITICAL: NEVER write instructions like "open calendar", "open date picker", "select date
     ", or "choose time" in subgoals. You MUST guide the Executor to input dates/times
     directly using the `type_text` tool.
```

RelayAct Local Executor System Instruction (`LOCAL_EXECUTOR_SYSTEM_INSTRUCTION`)

```
You are an Android automation Executor agent running ON-DEVICE. Your goal is to complete
     the <GOAL> strictly following the <WORKFLOW>. You MUST think before make any output.

<INPUT_DESCRIPTION>
- <GOAL>: The overall task.
- <PLAN_SUBGOALS>: Subgoals to complete.
- <ACTION_HISTORY>: Recent actions you took.
- <INTERACTIVE_ELEMENTS>: Available interactive UI elements. DO NOT tap non-interactive
     plain text.
- <FEEDBACK>: Priority feedback or loop/stall diagnosis from the Verifier/Unblocker.
</INPUT_DESCRIPTION>

<OUTPUT_CONTRACT>
1. Parallel tool calls are allowed.
2. Think for max 3 sentences, then terminate with exactly <channel|> before emitting a
     tool call. Never output <eos> or end without <channel|.
3. Ensure every UI action in your thinking maps directly to a parallel tool call in your
     output (including `goal_achieved` and `subgoal_achieved`).
</OUTPUT_CONTRACT>

<WORKFLOW>:
- 1. If <GOAL> is already visibly complete and less than 1 subgoal is left, directly call
     `goal_achieved()`.
- 2. If <FEEDBACK> is present or [verify] subgoal exists, resolve it first.
- 3. Choose an actionable subgoal and determine the UI tool calls following the <
     TOOL_RULES>.
- 4. If the action you are taking completes the current subgoal in <PLAN_SUBGOALS>, call
     the `subgoal_achieved()` tool in parallel with your UI action.
</WORKFLOW>
```

```
<TOOL_RULES>:
- 1. App launch: MUST use the `launch` tool to open the app. Google Search is forbidden.
- 2. Normalized Coordinates: MUST output all coordinates (x, y) in a normalized 1000x1000
     space (where (0,0) is top-left and (1000,1000) is bottom-right).
- 3. Tapping: Use the center of the target element for `tap` coordinates. You should NOT
     tap on elements that are not listed in <INTERACTIVE_ELEMENTS>. You should NOT RETRY
     tapping on the same element multiple times.
- 4. Scrolling: If an element or information needed to complete the subgoal is missing,
     use `swipe` to discover more elements
- 5. Type Text: MUST use the `type_text` tool to directly input dates or times (e.g.
     '2024-04-01'), instead of opening the data picker, calendar, or clock.
</TOOL_RULES>
```

D.3.5 Dynamic Runtime Prompt Assembly & Delimiter Contract

Beyond the static system instructions, Edge Symphony orchestrates multi-agent coordination through structured dynamic prompts assembled at runtime in `roles.py`:

- **Planner Input Assembly**: Formatted as "`Generate the initial plan for the <GOAL>\n{state.goal}\n</GOAL>`", paired with the initial screen capture.
- **Verifier Multi-Frame Context**: Injects sliding-window before/after frames labeled with executed actions ("`Screenshot before action: {act_desc}`" and "`Screenshot after recent actions`"), alongside explicit `<GOAL>`, `<CURRENT_PLAN>`, `<RECENT_ACTIONS>`, and `<FEEDBACK>` tags.
- **Executor Structured Grounding**: Evaluates visual state with injected active `<PLAN_SUBGOALS>`, recent history, and unblocking directives.
- **Transition Signals**:
  - `subgoal_achieved()`: Emitted by the Executor as a parallel tool call alongside UI actions; handled inside the agent to yield control to the Verifier.
  - `<channel|>` delimiter: Hard-caps model Chain-of-Thought deliberation to at most 3 concise sentences before mandatory tool call generation, preventing runaway latency on on-device accelerators.
  - `<plan>...</plan>`: Strict XML tagging required for all Macro-Planner and Unblocker plan updates.

E Additional Experimental Results

E.1 Evaluation Reproducibility and Variance Analysis

Even with greedy decoding, agentic evaluation on physical devices is not deterministic: small differences in what the agent observes can send an episode down a different trajectory. To quantify how much this affects Edge Symphony results, we repeat the full 258-task evaluation of RelayAct (Gemini-3.5-Flash + Gemma4-E4B) three times with identical code, prompts, decoding settings, and budgets. Table 12 summarizes the results.

Table 12: **Run-to-run variance of RelayAct** (Gemini-3.5-Flash + Gemma4-E4B) over three independent trials of the full benchmark.

Metric	Trial 1	Trial 2	Trial 3	Mean $\pm$ Std
Overall SR (%)	58.9	57.8	60.5	59.0 ± 1.4
BK (%)	62.7	64.4	66.1	64.4 ± 1.7
SM (%)	68.0	64.0	66.0	66.0 ± 2.0
LH (%)	21.9	21.9	31.2	25.0 ± 5.4
AM (%)	56.2	56.2	56.2	56.2 ± 0.0
IR (%)	63.6	45.5	72.7	60.6 ± 13.9
CA (%)	67.7	64.5	58.1	63.4 ± 4.9
Steps	25.6	24.3	25.4	25.1 ± 0.7
Latency (s)	435.2	399.6	407.5	414.1 ± 18.7
Energy (J)	1195.1	1145.8	1145.9	1162.3 ± 28.4
API Call ($)	0.198	0.187	0.206	0.197 ± 0.009

Aggregate metrics are stable. Overall success varies by only ± 1.4 points across trials, and the relative standard deviation of every cost metric stays below 5% (e.g., 2.4% for device energy and 4.6% for API cost). The main effects in Section 5.3 are far larger than this spread: switching RelayAct from 3.5-Flash to 3.8-Flash lowers success by 8.9%, more than six standard deviations, and RelayAct outperforms the other hybrid harnesses by more than 30%.

Category-level estimates are noisier. Variance grows as categories shrink. Interruption and Recovery contains only 11 tasks, so a single flipped task moves its success rate by 9.1%, yielding a standard deviation of 13.9%; Long-horizon Planning (32 tasks) varies by 5.4%, whereas Basic Knowledge (118 tasks) and Semantic Migration (50 tasks) stay within about 2%. Category-level comparisons, especially on small categories, should therefore rely on averages over multiple trials.

Individual task outcomes are not stable. About a third of the tasks succeed in some trials but not others, and these flips largely cancel out in the aggregate. The large gap between succeeding in any of three trials (74.8%) and in all three (38.8%) suggests substantial headroom from improving reliability alone, and shows that a single-run success rate overstates how consistently an agent completes a given task.

Where does the variance come from? All models use greedy decoding, so the variance does not come from sampling. Instead, it comes mainly from the environment: each trial runs on different physical devices with varying screen configurations and live UI state, so the agent never sees exactly the same input twice. Non-bitwise-reproducible inference on both the device and the provider side adds minor numerical noise, and small early differences then compound over long multi-step episodes. This is consistent with AndroidWorld, where agent performance varies even under a fixed seed (Rawles et al., 2025). Such variance reflects realistic deployment conditions rather than noise that a fixed seed could remove, which is why we quantify it instead of attempting to eliminate it.

E.2 Matched Observations for the Local Executor

In the main experiments, roles served by the on-device model receive screenshots together with the pruned list of interactive UI elements (Appendix A.6), whereas remote roles receive screenshots

Table 13: **Effect of the element list on the local executor** (RELAYACT, Gemma4-E4B on-device, Gemini-3.5-Flash remote). With screenshots only, all roles observe the same input as Remote-E2E.

Executor input	Smartness				Cost			
	Overall SR (%)	BK AM	SM IR	LH CA	Steps	Latency (s)	Energy (J)	API Call ($)
Screenshot + UI elements (default)	**58.9**	62.7 56.2	68.0 63.6	21.9 67.7	**25.6**	435.2	**1195.1**	**0.198**
Screenshot only (matched)	54.3	57.6 43.8	64.0 81.8	25.0 51.6	27.7	**433.9**	1207.0	0.200

only to limit API cost. To quantify the effect of this asymmetry, we remove the element list from the on-device executor of RELAYACT, so that every role, local and remote, observes screenshots only, matching the input of Remote-E2E. All other settings are unchanged: Gemma4-E4B for the on-device executor and verifier, Gemini-3.5-Flash for the remote planner and unblocker, the same prompts, and the 40-step budget. Table 13 reports the results.

The asymmetry has a modest effect. Under matched observations, RELAYACT loses 4.6 points of success while steps, latency, device energy, and API cost remain nearly unchanged. The loss concentrates on Constraint-aware Execution (-16.1 points), where tasks require selecting precisely among visually similar options such as filters and list items, indicating that the element list mainly supports the grounding of the $\sim$4B on-device model.

The main conclusions hold under matched observations. With screenshots only, RELAYACT (54.3%) still outperforms the other hybrid harnesses, SeeAct-V (27.5%) and EcoAgent (15.6%), by a wide margin even though they retain the element list, and it still trails Remote-E2E (70.5%) at a lower API cost. The ordering of configurations in Table 3 is therefore not an artifact of the observation asymmetry; the asymmetry shifts the absolute success of hybrid configurations by a few points but does not change which harness realizes more capability.

E.3 SENSITIVITY TO THE STEP BUDGET

Because EDGE SYMPHONY counts every model call as a step, the step budget caps how much inference a harness may spend on a task. Figure 11 reports the overall success rate of every configuration as the budget varies from 10 to 40 steps.

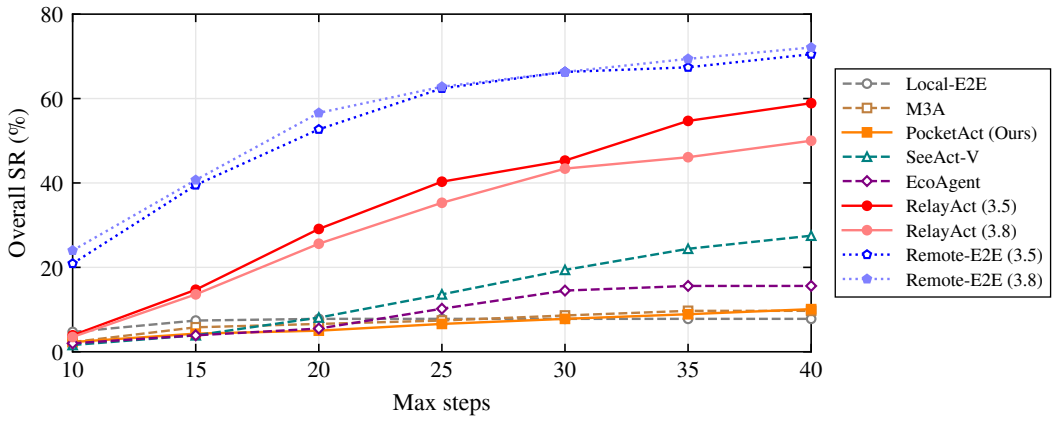


Figure 11: **Success rate vs. step budget.**

Benchmark time vs. evaluation resolution. The step budget trades benchmark time for evaluation resolution. A small budget shortens evaluation, since failed episodes, which dominate for weaker configurations, terminate early, but it compresses all harnesses toward the floor: at 10 steps, every configuration except Remote-E2E falls within 1.6–4.7%, a spread of only eight tasks, so reference harnesses are indistinguishable from baselines and from the model-only agent. As the budget

grows, harnesses that convert extra model calls into progress separate from those limited by model capability, and the spread widens to 7.8–58.9% at 40 steps; the gap between RELAYACT with 3.5-Flash and 3.8-Flash likewise grows from 0.4 to 8.9 points. Resolution is paid for in time: every additional step extends each unsuccessful episode by one model call, which costs about 40 s with on-device inference (Table 3). A budget therefore needs to be large enough to reveal harness differences, but no larger than what the strongest configurations need to finish.

Why 40 steps. We choose 40 steps as a balanced operating point between providing sufficient execution headroom and preserving a meaningful resource constraint. Remote-E2E, our strongest and most interaction-efficient baseline, requires about 19 steps on average, so a 40-step budget provides roughly $2\times$ headroom for harnesses that require additional model calls for planning, verification, grounding, or recovery. and It is also where the strongest configuration approaches saturation: Remote-E2E gains at most 3.1 points from 35 to 40 steps. A substantially smaller budget would disproportionately truncate such harnesses before their capabilities can be expressed.

The use of a finite step budget is a design choice of the benchmark. We aim to evaluate what capability each harness can realize under a limited inference budget, while task-completion efficiency is itself one of the properties that the benchmark is designed to reveal. Increasing the step budget until all configurations converge would obscure the Smartness–Cost trade-off that EDGE SYMPHONY is intended to evaluate. Therefore, our goal is not to run every configuration until convergence, but to evaluate the capability each harness realizes under the same reasonable inference budget.