ANDROIDFLUX: EVALUATING AND REWARD MODELING FAILURE RECOVERY CAPABILITIES IN MOBILE-USE MULTIMODAL AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Mobile-use multimodal agents are typically evaluated by task success from a clean initial state, which does not isolate their ability to recover from intermediate errors. This capability also matters when a stronger agent takes over an unfinished task and inherits another agent's mistakes. Hence, We introduce **AndroidFlux**, a failure-recovery benchmark built on AndroidWorld. Recorded trajectories are restored to controlled checkpoints before, at, and after the first recoverable error, allowing different agents to continue from the same device state and history. A complementary reward-modeling benchmark evaluates recovery-action preferences using downstream rollout outcomes. Moreover, we release a reward-modeling training dataset of 1.8k first-error contexts with candidate recovery actions and an offline reinforcement learning (RL) dataset of 8.5k contexts to enhance both error recovery and overall task success. We also introduce **Latent Peer Voting** (LPV): each expert scores actions proposed by its peers using conditional likelihoods, and these scores are normalized within each candidate set and combined with performance-based weights to produce preference supervision. Our evaluations show that recovery performance only partially correlates with clean-start success, and that explicit reasoning does not consistently improve recovery. Also, we show that LPV aggregates the internal action preference representation in expert policies and its collective preference can be distilled into a single reward model.

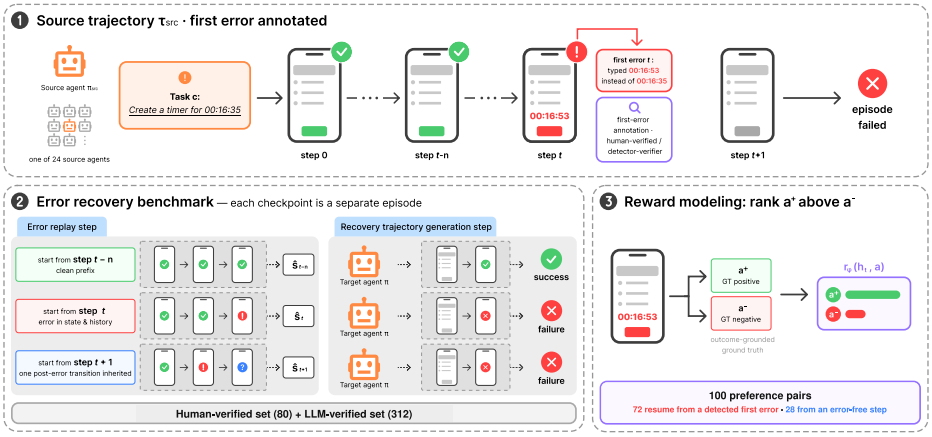


Figure 1: **Overview of AndroidFlux** A source agent commits a first error at step t. The error recovery benchmark replays to a checkpoint around t and resumes a target agent from the restored state. The reward modeling benchmark evaluates whether a reward model can rank recovery actions.

1 INTRODUCTION

Multimodal agents now operate real mobile devices, completing user tasks by reading screens and issuing touch and text actions over long horizons (Rawles et al., 2024; Xu et al., 2024; Kong et al., 2025). Progress on these agents is measured almost entirely by terminal task success from a clean start. That measure is saturating on the strongest models and, more fundamentally, too sparse to describe what happens inside an episode: one binary outcome cannot separate an agent that derails at

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.

			Error recovery							
Benchmark	Platform	Live	Eval.	Error src.	Src. agents	Full trans.	$t{+}1$ rec. action	RM eval	Starts at	Scale
AndroidControl	Mobile	✗	✗	–	–	✗	–	✗	any step	8,444 steps
AndroidWorld	Mobile	✓	✗	–	–	✗	–	✗	step 0	116 tasks
AndroidLab	Mobile	✓	✗	–	–	✗	–	✗	step 0	138 tasks
MobileWorld	Mobile	✓	✗	–	–	✗	–	✗	step 0	201 tasks
OSWorld	Desktop	✓	✗	–	–	✗	–	✗	step 0	369 tasks
GUI-RobustEval	Desktop	✓	✓	Policy	12	✗	Neg. only	✗	$t{+}n^{-}$	1,216 cases
AndroidFlux (ours)	**Mobile**	✓	✓	**Policy**	**21**	✓	**Pos. + Neg.**	✓	$t{-}n, t, t{+}1^{\pm}$	**392 cases, 100 pairs**

Table 1: Comparison with other GUI benchmarks. *Full trans.* marks whether all three phases around the first error are covered; $t{+}1$ *rec. action* gives which recovery actions are supplied at the post-error state; *RM eval* marks whether the benchmark offers action preference evaluation for reward models.

the first unexpected state from one that notices the problem and repairs it. Clean-start rollouts are also a poor instrument for isolating that distinction, since such states arise only incidentally, each costs a full episode to reach, and the episode is scored as a whole rather than at the moment recovery succeeds or fails. The distinction matters most on mobile, where one wrong tap, an incorrect app transition, or a failed text entry leaves the device in a state that appears in no clean demonstration, and interruptions and asynchronous changes make such states routine. Recovery is also a deployment primitive: cost-aware routing (Chen et al., 2023; Ong et al., 2024) and step-level agent cascades (Wei et al., 2026) escalate to a stronger model when a weaker one stalls, handing it a foreign context and a device state it did not produce—yet no mobile-use benchmark measures whether it can recover.

Whether a stronger agent can complete a failed task cannot be inferred from its clean-start success: it must be tested from the intermediate failed state to measure its recovery performance. Thus, We introduce **AndroidFlux** (Figure 1) to make this comparison possible through controlled replay of policy-induced failures in AndroidWorld. We annotate the first error step t and restore checkpoints at $t{-}n$ (correct prefix), t (error committed), and $t{+}1$ (after a positive or negative recovery action). Target agents resume from matched device states with histories re-rendered in their own formats, allowing us to examine how policies handle the same failure and how recovery depends on the preceding actions. Across human-verified and larger LLM-verified sets, recovery performance only partially correlates with clean-start success, showing that standard evaluations provide an incomplete basis for selecting a takeover agent. We find that explicit reasoning also does not consistently improve recovery: general reasoning ability alone does not ensure effective action in a disrupted GUI state. We further investigate whether the action preferences encoded in expert policies can provide supervision for recovery. Our **Latent Peer Voting** (LPV) reward modeling method aggregates experts' conditional likelihoods of peer-proposed actions through within-set normalization and performance-based weighting. Evaluated with our reward modeling benchmark, these collective preferences improve recovery-action ranking and can be partially distilled into a single reward model with our 1.8k reward-modeling dataset. Furthermore, We release an offline RL dataset of 8.5k decision contexts and show that our offline RL dataset improves both recovery and clean-start success, providing the research community with a practical resource for studying and improving failure recovery without expensive online RL training.

2 Related Work

Mobile/GUI benchmarks. GUI-agent benchmarks differ mainly in whether the agent acts in a live environment and what state each episode begins from. AndroidControl supplies static UI-control data along demonstrations, scorable at any step but without consequences (Li et al., 2024). AndroidWorld, AndroidLab, MobileWorld, and OSWorld provide interactive environments with programmatic verifiers, but most episodes start from the home screen and return one terminal outcome, so recovery is not separated from success (Rawles et al., 2024; Xu et al., 2024; Kong et al., 2025; Xie et al., 2024). Concurrent with this work, GUI-RobustEval benchmarks post-error recovery on desktop agents by resuming after a policy-induced error (Bu et al., 2026). AndroidFlux instead targets mobile agents and spans controlled checkpoints before, at, and after the first error, enabling cross-agent takeover from inherited context. It further evaluates outcome-grounded recovery-action preferences, asking not only whether an agent recovers but whether a reward model can identify the action that makes recovery more likely.

Reward modeling. Mobile GUI agents are typically trained against sparse terminal success (Xu et al., 2025), while process reward models supply denser signal by scoring intermediate decisions (Lightman et al., 2023; Choudhury, 2025; Hu et al., 2025; Chae et al., 2025), and conventional reward models are trained with the Bradley–Terry interface (Bradley & Terry, 1952; Stiennon et al., 2020; Ouyang et al., 2022). UI-Genie (Xiao et al., 2025) is the closest mobile use precedent, training action- and task-level rewards from rule verification and synthetic corruption. Its negatives, however, are perturbations of correct behavior rather than actions a policy took at a state it had actually broken, and the reward model is evaluated inside a self-improvement loop rather than on whether its rankings hold at recovery states. Hence, reward model for failure recovery remains underexplored.

3 Problem Formulation

Preliminaries. We model mobile interaction as a finite-horizon POMDP with agent-specific context,

$$\mathcal{M} = (\mathcal{S}, \mathcal{O}, \mathcal{A}, P, \Omega, r, H, \mu_0), \qquad h_t^\pi = \Phi_\pi(c, o_{0:t}, y_{0:t-1}), \tag{1}$$

where s_t is latent device state, o_t the observation, a_t an AndroidWorld action, r the terminal programmatic verifier, c the task, $y_i = (u_i, a_i)$ a response with optional reasoning u_i, and Φ_π the target agent's context formatting. The screenshot does not expose application databases, activity stacks, or filesystem state, making recovery inherently history-dependent. Recorded responses are stored schema-neutrally and re-rendered per policy, so cross-agent comparison does not copy one model's prompt format into another. The detailed action space of a_t is reported in Appendix A.

Replay and resume from the first error. For source trajectory τ^{src} and checkpoint k, replay re-executes the recorded action prefix under the same task seed, falling back to an emulator snapshot where replay is not reproducible; k is thus the last replayed step supplied in context, and the target's first new action occurs at $k+1$. Letting $e_i \in \{0, 1\}$ indicate whether a_i fails to advance the task from s_i, the first error is the earliest such step:

$$\hat{s}_k = \mathrm{Replay}(\tau^{\mathrm{src}}, k), \qquad t = \min\{\, i : e_i = 1 \,\}. \tag{2}$$

Any y_i with $i > t$ is taken at a state already carrying the effect of a_t, so it is an attempt to act from a broken state rather than an independent failure; all such actions are recovery actions, and t is the only error committed under a correct prefix. We consider only recoverable first errors.

Resume runs target policy π from checkpoint k using the restored state $\hat{s}_k$ and the reconstructed context h_k^π, generating a recovery trajectory whose terminal state the verifier scores:

$$\tau^{\mathrm{rec}} = \mathrm{Resume}(\pi, \hat{s}_k, h_k^\pi) = (y_{k+1}, \ldots, y_T), \qquad \mathrm{SR}(\pi, k) = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}\left[r(s_{T_i}^{(i)}) = 1\right]. \tag{3}$$

Trajectory generation thus continues under a new policy from a prefix another agent authored: if $\pi = \pi_{\mathrm{src}}$ this measures self-recovery, otherwise recovery from a foreign source agent. Varying k relative to t separates the cost of a committed error from the cost of inherited foreign history.

Recovery-action reward modeling. The error recovery task resumes from the state at $k = t$ and asks which action should be taken next. With $x = (h_t, o_t)$, a reward model assigns a scalar score $r_\phi(x, a)$ to each candidate action, and preferences follow from score ordering; accuracy is the fraction of outcome-labeled pairs (a^+, a^-) with $r_\phi(x, a^+) > r_\phi(x, a^-)$. Both actions in a pair share the task, state, and history, so accuracy isolates the judgment of the action itself. We train the baseline reward model with a Bradley–Terry (BT) objective over preferred–rejected pairs,

$$\mathcal{L}_{\mathrm{BT}}(\phi) = -\mathbb{E}_{(x,a^+,a^-)}\left[\log \sigma\big(r_\phi(x, a^+) - r_\phi(x, a^-)\big)\right], \tag{4}$$

where σ is the logistic function, so the model learns to score a^+ above a^-.

4 AndroidFlux: Error Recovery & Recovery Action Reward Modeling Benchmark

To measure whether a policy repairs a broken state and whether a reward model can identify the action that repairs it, we introduce **AndroidFlux**: a failure-recovery benchmark and a recovery-action reward modeling benchmark, both built on the AndroidWorld environment and tasks. Table 1 situates them against existing GUI agent benchmarks.

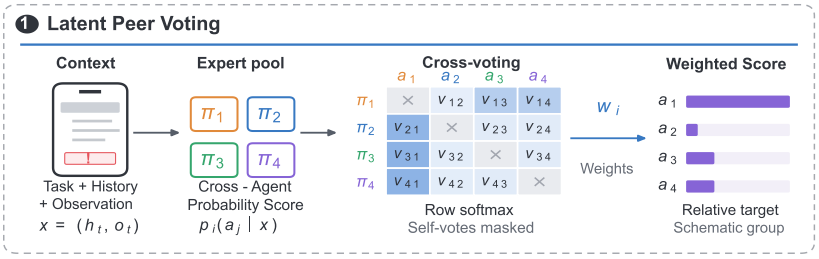


Figure 2: **Latent Peer Voting.** LPV aggregates the preferences of multiple experts. Each expert scores its peers' candidate actions by likelihood, with self-votes masked, and the votes are normalized and combined with performance weights into a single ranking.

4.1 Error Recovery Benchmark

The error recovery benchmark supplies restored resume points drawn from real policy failures. Its human-verified set is fixed at 80 cases from 6 source models, 65 restored by replaying the recorded action prefix and 15 from saved emulator snapshots where exact replay is not reliable. Its LLM-verified set contains 312 replay-based cases over 104 tasks from 21 source models, extending coverage across a far wider pool of failure sources.

4.1.1 Annotation of the First Error

The first error is the initial error in the trajectory a model executes toward its goal, and the only one unaffected by a preceding error: every later error occurs in a state an earlier error already modified, so it cannot be attributed to the model as an independent failure. For the human-verified set, human experts reviewed the trajectory step by step and labeled the earliest error. For the LLM-verified set, a two-stage detector–verifier framework implemented with Gemini Flash 3.0 (Google DeepMind, 2025) proposes candidate error steps and re-checks each against the resulting UI state, taking the earliest surviving step as the first error. Because a false label relocates the state at which every downstream recovery experiment begins, the framework is tuned toward precision at the expense of recall, reaching 90.8% step-level precision on the annotated set and so identifying policy-induced first errors reliably enough to anchor evaluation. Each error is then assigned to one of three categories—Stagnation, Direction, and Execution. Further details are reported in Appendix C.7 and C.9.

4.2 Recovery Action Reward Modeling Benchmark

This benchmark holds the resume state fixed and varies the action taken from it. It contains 100 recovery states over 26 AndroidWorld task types: 72 resume from a detected first error and 28 from an error-free step. We generate source trajectories from ten agents, annotate their first errors with the detector–verifier pipeline above, and resume five recovery models from the restored states to obtain candidate recovery actions. Each candidate is executed, and a panel of twelve agents spanning a wide range of recovery ability rolls out from the resulting state. Preference thus follows from consequence rather than rule or judge: at each state we pair the two candidates whose panel success counts differ most, keeping the pair only when the margin is at least two. The resulting 100 pairs share task, state, screenshots, and history, differing only in the recovery action. Details on the trajectory and recovery actions are in Appendix D.

5 Latent Peer Voting

Terminal outcomes give sparse supervision for intermediate recovery actions, and annotating each action's utility requires judging long-term consequences that the immediate UI transition may not reveal. We therefore introduce **Latent Peer Voting** (LPV), which pools the implicit preferences of mobile-use experts into a single scalar reward model, a form of collective distillation. Trained on different datasets and interaction experience, these experts acquire complementary grounding priors, app coverage, and action preferences that are encoded in their policies but not necessarily expressible as rules or verbal judgments. DPO motivates reading such signals from likelihoods: under KL-regularized reward maximization, a policy's implied reward is a scaled log-likelihood ratio

against a reference policy (Rafailov et al., 2023). LPV uses conditional likelihoods as preference signals, without recovering that reward exactly or asking a judge to reconstruct it in language.

Given a recovery context $x = (h_t, o_t)$, the experts propose candidates $\mathcal{A}(x) = \{a_1, \ldots, a_n\}$, and each voter scores its peers' complete responses, including the end-of-sequence token, while excluding its own, because a policy assigns high likelihood to its own samples regardless of their quality. Raw likelihoods are not comparable across agents, so each voter's scores are normalized over its eligible peer candidates and then aggregated with performance weights. Let $\mathcal{V}$ denote the voters, $s(j)$ the source of a_j, and $\ell_{ij} = \frac{1}{|a_j|} \log p_i(a_j \mid x)$ the response log-likelihood averaged over its $|a_j|$ tokens, so that longer responses are not penalized for their length. With temperature τ (we use $\tau = 1$),

$$v_{ij} = \begin{cases} \dfrac{\exp(\ell_{ij}/\tau)}{\sum_{k:\, s(k)\neq i} \exp(\ell_{ik}/\tau)}, & s(j) \neq i, \\ 0, & s(j) = i, \end{cases} \qquad R_j = \sum_{i \in \mathcal{V}} \frac{w_i}{\sum_{m \in \mathcal{V}} w_m} v_{ij}, \tag{5}$$

where w_i is agent i's performance-based weight, a weighted average of its recovery success rates on the human-verified and LLM-verified sets, so stronger recovery agents weigh more. The rewards R_j order the candidates, and we evaluate this ordering directly on the recovery-action reward modeling benchmark. Because the likelihoods are computed once, candidate groups and weights can be varied without rescoring. LPV rewards target relative ordering, not absolute estimates of success.

LPV can serve directly as the reward in policy optimization, since group-relative methods such as GRPO (Shao et al., 2024) need only the relative order of sampled candidates. Scoring each candidate, however, requires a forward pass from every voter, which makes LPV costly to query at every step of online training. As in RLHF, where human preferences are distilled into a learned reward model rather than queried directly, the rich and sharply discriminative preferences that LPV extracts can instead be distilled into a single scalar reward model. Such a model scores candidates in one forward pass, making LPV-derived supervision practical for efficient online reinforcement learning.

5.1 Dataset: Reward Modeling and Offline RL Training Dataset for Error Recovery

Reward-modeling dataset. The corpus contains 1,803 first-error contexts across 88 task types, each with six recovery candidates (10,818 total). Contexts include the task goal, action history through the first error at t, and up to five recent screenshots from the same trajectory. Gemini 3.8 Flash annotates whether termination is appropriate. Candidates come from Qwen3-VL-2B, Qwen3-VL-8B, GUI-Owl-7B, GUI-Owl-1.5-8B, and MAI-UI-8B. Each proposes an action at $t{+}1$ from the replayed history and recorded screen without emulator execution. Coordinates are converted to screen pixels. The five open-weight generators also act as voters, scoring each peer candidate under the request their own agent sends. Rankings are derived with LPV (Eq. 5), and pairwise preferences take the candidates with the highest and lowest weighted LPV score as the preferred and rejected actions.

Offline RL dataset. The RL dataset contains 8,499 policy-induced contexts built from source trajectories collected before replay filtering, sampled with seed 42 from four slices: $t{-}n$ (2,097), $t{-}1$ and t (2,080 each), and successful contexts with verified environment success (2,242). Each is annotated with whether its goal is already satisfied, making termination the correct next action. A checkpoint is the last action in the history, so $t{-}1$ prompts the erroneous action and t prompts a recovery at $t{+}1$; no history extends past the first error. Construction details and the training mixtures sampled from this corpus are given in Appendix E.1 for details.

6 Main Results

6.1 Evaluation Setup

All experiments run in the containerized AndroidWorld environment released with the benchmark, which we customize for reproducibility and stability. Every evaluation in this paper is split across container shards running on H100, A40, L40s, and RTX 4090 GPUs.

Model	Human-verified										LLM-verified				
	Scratch	$t{-}5_{\text{rep}}$	$t{-}3_{\text{rep}}$	$t{-}1_{\text{rep}}$	t_{rep}	t_{snap}	t_{all}	$t{+}1^{+}$	$t{+}1^{-}$	$t{+}1_{\text{all}}$	Scratch	$t{-}1$	t	$t{+}1^{+}$	$t{+}1^{-}$
Qwen3-VL-2B† (Bai et al., 2025)	16.25	13.85	16.92	23.08	12.31	26.67	15.00	28.85	17.31	23.08	22.12	27.56	24.68	41.83	32.69
Qwen3-VL-4B† (Bai et al., 2025)	26.25	29.23	24.62	27.69	21.54	46.67	26.25	34.62	34.62	34.62	38.46	40.71	32.37	50.96	41.83
GUI-Owl-32B (Xu et al., 2026)	53.75	40.00	43.08	35.38	21.54	46.67	26.25	28.85	34.62	31.73	60.58	41.35	28.21	41.35	42.79
Mobile-Agent-v3-32B	57.50	44.62	49.23	47.69	41.54	46.67	42.50	44.23	48.08	46.15	61.54	51.28	46.15	67.31	55.29
MAI-UI-8B (Zhou et al., 2025)	37.50	40.00	36.92	40.00	30.77	53.33	35.00	38.46	38.46	38.46	46.15	41.35	31.73	50.96	48.56
Gemini Flash 3.0†	40.00	7.69	10.77	29.23	24.62	40.00	27.50	42.31	34.62	38.46	46.15	41.35	39.74	49.52	48.56

Table 2: Human-verified (HV) and LLM-verified (LV) sets on error recovery over the HV set source agents. Scratch runs from step 0 of all 80 HV tasks. HV has 65 replay-based (t_{rep}) and 15 snapshot-based (t_{snap}) cases. At $t{+}1$, $^{+}$ and $^{-}$ take the recovery action from the successful and failed recovery trajectory, and $t{+}1_{\text{all}}$ aggregates both. †Evaluated with the Qwen3-VL AndroidWorld prompt.

6.2 Human-Verified Set and LLM-Verified Set

In Table 2, the human-verified and LLM-verified sets show the same central pattern: performance degrades when agents resume from states containing an annotated first error. Before the error, a correct foreign prefix can still benefit some models: Qwen3-VL-2B and Qwen3-VL-4B outperform scratch from several $t{-}n$ checkpoints, suggesting that inherited progress can place the target in a useful state it may not reliably reach on its own while providing enough context to continue. Together, the $t{-}n$ checkpoints capture how inherited progress affects continuation before the first error, with $t{-}1$ providing the closest comparison to t. The $t{+}1$ conditions show that a single recovery action can substantially alter downstream recoverability, depending on both the injected action and the recovering policy. Refer to Appendix C for details on the HV and LV set.

6.3 Failure Recovery Across Agent Architectures

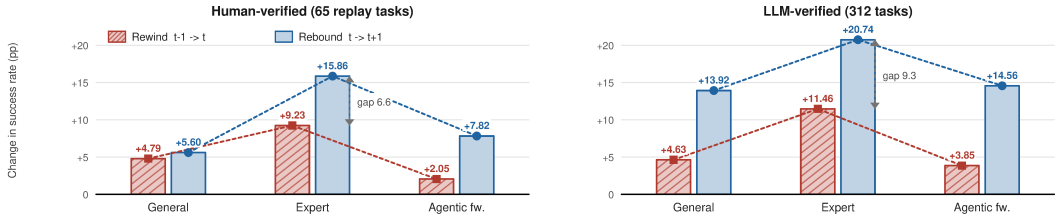


Figure 3: Rewind and rebound by agent class, from the class means of Table 3 (rewind $= t{-}1 - t$, rebound $= t{+}1 - t$). Both moves are largest for GUI experts: they lose the most at the first error and regain the most from a single supplied action.

Table 3 shows a clear progression in recovery capability across agent architectures. Figure 3 shows the same ordering in the moves themselves: both the rewind and the rebound are largest for GUI experts. General-purpose VLMs can recover from inherited failures, but domain-specialized GUI agents are generally more robust around the first-error checkpoint. On the human-verified set, GUI-Owl-1.5-8B achieves 40.0 at t_{all}, compared with 28.8, 30.0, and 27.5 for Qwen3-VL-8B, Qwen3-VL-32B, and Gemini Flash 3.0. The same pattern appears from $t{-}1$ to t: most of models are affected by the committed error, but stronger GUI-specialized policies retain more of their pre-error performance. This suggests that training for interactive GUI control provides robustness to off-distribution states beyond clean-start capability. Agentic frameworks strengthen this effect through inference-time state reassessment and replanning. Mobile-Agent-v3-7B reaches 35.4 at t_{rep}, compared with 24.6 for GUI-Owl-7B, with the gap widening on the LLM-verified set. MobileRun provides the strongest example: using Gemini Flash 3.0 within the framework raises recovery from 27.5 to 67.5 at t_{all} on the human-verified set and from 39.7 to 67.0 at t on the LLM-verified set. It also remains strong after either injected recovery action. These results show that recovery depends not only on the underlying model, but also on inference-time scaling mechanisms for reflection, correction, and replanning.

Table 4 and Figure 4 compare matched instruct and explicit Chain-of-Thought (CoT) reasoning variants within each model family, with Gemini Flash 3.0 providing a same-model thinking on/off control. Explicit CoT does not uniformly improve recovery: Qwen3-VL-32B and Gemini Flash 3.0 perform worse at t with reasoning enabled, whereas GUI-Owl-1.5-32B and GPT-5.6 Terra improves substantially. This suggests that reasoning acquired through general-domain RL does not necessarily transfer to interactive GUI recovery. In contrast, GUI-Owl and GPT-5.6 Terra applies reasoning-

Type	Model	Human-verified								LLM-verified				
		Scratch	$t-1_{\text{rep}}$	t_{rep}	t_{snap}	t_{all}	$t{+}1^{+}$	$t{+}1^{-}$	$t{+}1_{\text{all}}$	Scratch	$t-1$	t	$t{+}1^{+}$	$t{+}1^{-}$
General	Qwen3-VL-8B-Instruct	41.25	33.85	24.62	46.67	28.75	36.54	26.92	31.73	49.04	36.86	29.49	48.56	43.27
	Qwen3-VL-32B-Instruct	36.25	36.92	26.15	46.67	30.00	38.46	30.77	34.62	45.19	42.63	33.97	50.96	48.56
	Qwen3.8-27B	45.00	30.77	27.69	53.33	32.50	40.38	30.77	35.58	50.00	47.12	37.82	57.21	50.00
	Gemma-4-31B-IT (M3A)	37.50	40.00	36.92	66.67	42.50	48.08	36.54	42.31	43.27	51.60	44.55	62.50	61.06
	Gemini Flash 3.0†	40.00	29.23	24.62	40.00	27.50	42.31	34.62	38.46	46.15	41.35	39.74	49.52	48.56
	Gemini Flash 3.8†	51.25	52.31	50.77	66.67	53.75	50.00	53.85	51.92	58.65	58.33	61.54	74.52	69.23
	Gemini 3.1 Pro†	55.00	55.38	47.69	60.00	50.00	63.46	51.92	57.69	61.54	62.18	58.33	78.85	65.87
	GPT-5.6 Luna†	27.50	33.85	27.69	60.00	33.75	28.85	21.15	25.00	33.65	35.26	31.73	41.35	40.38
	GPT-5.6 Terra†	38.75	40.00	43.08	53.33	45.00	46.15	38.46	42.31	53.85	46.15	42.63	60.10	59.62
Expert	MAI-UI-8B	37.50	40.00	30.77	53.33	35.00	38.46	38.46	38.46	46.15	41.35	31.73	50.96	48.56
	GUI-Owl-7B	46.25	41.54	24.62	53.33	30.00	36.54	34.62	35.58	56.73	40.38	27.24	45.19	41.83
	GUI-Owl-1.5-8B (Xu et al., 2026)	56.25	47.69	36.92	53.33	40.00	53.85	46.15	50.00	62.50	49.68	44.23	62.98	58.65
	Gemini Flash 3.5 Computer Use† (Quiros, 2026)	38.75	30.77	30.77	53.33	35.00	61.54	63.46	62.50	40.38	57.69	40.06	74.04	70.19
Agentic framework	Mobile-Agent-v3-7B	58.75	44.62	35.38	53.33	38.75	51.92	40.38	46.15	63.46	49.68	45.19	61.54	56.73
	MobileRun (Li et al., 2026)	78.75	63.08	64.62	80.00	67.50	82.69	75.00	78.85	79.81	69.87	66.99	75.96	82.21
	FinalRun (FinalRun, 2026)	66.25	38.46	40.00	60.00	43.75	42.31	34.62	38.46	68.27	43.27	39.10	59.13	54.33

Table 3: General-purpose VLMs, GUI experts, and agentic frameworks on error recovery. Left block: human-verified set. Right block: LLM-verified set. The columns follow the recovery checkpoints defined in Table 2. All agentic frameworks are driven by Gemini Flash 3.0 and all proprietary models† are evaluated with reasoning enabled and Qwen3-VL AndroidWorld prompt.

Model	Human-verified t			LLM-verified t		
	Inst.	Reas.	Δ	Inst.	Reas.	Δ
Without GUI Domain Specific Reasoning RL						
Qwen3-VL-32B	30.00	17.50	−12.50	33.97	21.15	−12.82
Gemini Flash 3.0	41.25	27.50	−13.75	46.79	39.74	−7.05
Qwen3-VL-8B	28.75	16.25	−12.50	29.49	27.88	−1.60
Gemma-4-31B-IT (M3A)	42.50	41.25	−1.25	44.55	47.44	+2.88
With GUI Domain Specific Reasoning RL						
GUI-Owl-1.5-32B	32.50	37.50	**+5.00**	36.22	44.55	**+8.33**
GPT-5.6 Terra	27.50	45.00	**+17.50**	32.05	42.63	**+10.58**

Table 4: Instruct vs. reasoning modes at the first-error checkpoint t, for models without and with GUI domain specific reasoning RL.

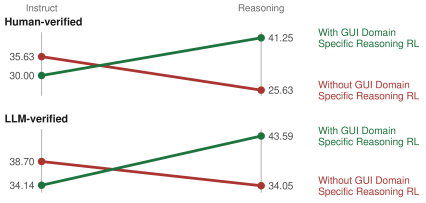


Figure 4: Average of Table 4: models with the domain specific reasoning RL gain, models without lose.

oriented RL in the target GUI domain, and its reasoning variant consistently improves recovery, suggesting that domain-aligned RL enhances error recovery performance.

6.4 Rewinding vs. Injection by Failure Timing

Steps before the error	n	Gain (pp) Rewind	Inject.	Δ
≤ 3	116	+8.76	−1.01	−9.77
4–6	100	+7.67	+7.55	−0.11
≥ 7	96	+3.41	+5.86	+2.44
Latest − earliest				**+12.22**

Table 5: Gain from rewinding ($t-1$ vs. t) and injection ($t{+}1^{+}$ vs. $t{+}1^{-}$) by steps taken before the error.

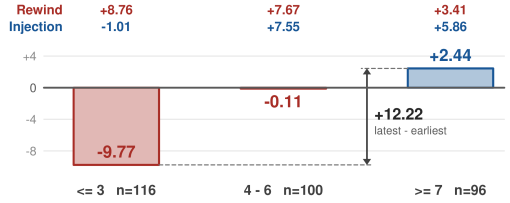


Figure 5: Δ rises monotonically with failure step: rewinding beats injection on the earliest failures, injection wins on the latest.

An agent's mistake can be remedied in two ways: *rewinding* puts it back one step and lets it try again ($t-1$ vs. t), and *injection* hands it an action taken from a successful recovery and lets it proceed ($t{+}1^{+}$ vs. $t{+}1^{-}$). Which one helps depends on how late the failure occurs. Splitting the 312 episodes by how many steps the agent had taken before the error, rewinding is worth +8.76pp on the earliest failures but only +3.41pp on the latest, while injection rises from roughly zero to +5.86pp (Table 5, Figure 5). The contrast between the two grows by 12.22pp, with a 95% confidence interval of $[+4.1, +20.8]$. The interval resamples episodes rather than model–episode pairs: every model attempts every episode, so rows within an episode are correlated (intra-class correlation 0.09–0.11), which more than halves the effective sample. The sign survives this correction and several further checks. It holds for 17 of the 18 models individually (sign test $p = .0001$), for all four difficulty quartiles, for all three trajectory-length terciles, and in both halves of a random episode split ($p = .029$ and $p = .044$). The tercile check matters because longer tasks also have longer prefixes, so the effect is not explained by task length alone. Across these strata the contrast ranges from +3.29pp to +21.03pp, but each stratum is small, so the checks establish a consistent sign rather than significance within each; the contrast is individually significant in the hardest and easiest quartiles and the two shorter terciles, and

			AndroidFlux						AndroidFlux		
Type	Scorer	UI-Genie	Overall	Error	Clean	Type	Scorer	UI-Genie	Overall	Error	Clean
Single agent	MAI-UI-8B	0.736	0.60	0.597	0.607	LPV (ours)	$\alpha = 0$	0.761	0.64	0.639	0.643
	GUI-Owl-1.5-8B	0.744	0.59	0.569	0.643		$\alpha = 0.35$	0.760	**0.65**	**0.653**	0.643
	GUI-Owl-7B	0.705	0.58	0.569	0.607		$\alpha = 0.7$	0.759	0.63	0.625	0.643
	Qwen3-VL-8B	0.747	0.52	0.500	0.571		$\alpha = 1$	0.759	0.62	0.611	0.643
	Qwen3-VL-2B	0.679	0.52	0.514	0.536	UI-Genie RM	BT-128K	0.904	0.49	0.444	0.607
Naive	Clean SR	–	0.49	0.500	0.464		BT-256K	**0.915**	0.52	0.486	0.607
	Error t SR	–	0.50	0.514	0.464		SFT-128K Hard	0.681	0.16	0.097	0.321
	Average	–	0.51	0.528	0.464		SFT-128K Soft	0.878	0.45	0.403	0.571
Oracle	5 agents	0.928	0.82	0.819	0.821	AndroidFlux RM	BT-Naive	0.711	0.48	0.528	0.357
	5 agents + BT-256K	0.989	0.87	0.875	0.857		BT-LPV (Ours)	0.913	0.55	0.528	0.607

Table 6: Pairwise accuracy on UI-Genie (1,000 pairs) and AndroidFlux (100 pairs), split into error-path and clean-path pairs. *Naive* ranks by the generator's success rate, *Single agent* and *LPV* by likelihood, and *Oracle* is correct if any listed scorer is. *UI-Genie RM* is trained on UI-Genie; *AndroidFlux RM* starts from the UI-Genie BT-256K model and is trained on AndroidFlux with Naive or LPV labels. Details are in Appendix D.

keeps the same sign in the longest, though not significantly. Bucket boundaries were fixed in advance rather than searched. When the failure comes late in the trajectory, putting the agent back one step helps less than handing it the next move. Refer to Appendix C.6 for details.

6.5 Recovery-Action Reward Modeling

Table 6 compares training-free scorers with trained reward models. Reward models trained on generic GUI supervision transfer poorly to recovery. BT fits UI-Genie above 0.90 but stays near chance on AndroidFlux, and doubling its data barely helps; SFT falls below chance under both readouts. Naive scoring, which ranks an action by the success rate of its proposing agent, is also at chance. However, the GUI experts' own likelihoods carry much of the missing signal, but it only shows up under recovery evaluation. The experts trail BT-256K by 17–21 points on UI-Genie yet reach 0.58–0.60 on AndroidFlux, while Qwen3-VL-8B shows the reverse, so generic GUI accuracy is a poor guide to recovery preferences. LPV combines the five open-weight generators with voter weights $w_i = \alpha\,\bar{c}_i + (1-\alpha)\,\bar{e}_i$ over normalized clean-start and first-error success rates. Without training, it reaches 0.65 at $\alpha = 0.35$, 13 pairs above BT-256K and five above the best single voter. Most of the gain comes from error-path pairs, whose accuracy peaks at this α, so weighting voters partly by recovery ability pays off. LPV also beats every single voter on UI-Genie, though in-domain BT stays ahead there.

The oracle rows show how much headroom remains. At least one agent ranks 82 of the 100 pairs correctly, and adding BT-256K raises this to 87, since agent likelihoods and the UI-Genie reward model fail on different pairs. The signal needed for most of the benchmark already exists across these models; what is missing is a better way to aggregate it.

We then ask whether this signal can be distilled into a single reward model. Both AndroidFlux RMs are initialized from UI-Genie BT-256K and are trained with LoRA (Hu et al., 2022) for 8 epochs on the AndroidFlux reward-modeling dataset, differing only in their labels. Naive labels always prefer the agent with higher recovery success, whereas LPV uses performance only to weight votes, so the same agent can win in one context and lose in another. Both labelings raise error-path accuracy from 0.486 to 0.528, so training on recovery contexts helps either way. The difference is in what is lost. Naive labels force the model to prefer, for example, GUI-Owl-7B over Qwen3-VL-2B regardless of the action, dropping clean-path accuracy from 0.607 to 0.357 and UI-Genie accuracy from 0.915 to 0.711, which leaves BT-Naive at 0.48, below its initialization. LPV labels leave both intact, and BT-LPV rises to 0.55. With the same initialization, data, and budget, the labels alone decide whether recovery gains cost general preferences. BT-LPV still under-performs training-free LPV, so our distillation captures only part of the signal. This likely reflects our limited setup. As we trained with LoRA under a fixed budget, longer or full-parameter training may narrow the gap.

6.6 Offline Reinforcement Learning

We further examine whether AndroidFlux provides useful training data for policy improvement. To isolate the effect of the training distribution, all experiments use UI Genie RM BT 256K, optimization

Training data	Clean	Error-t	Mean
Baseline (Qwen3-VL 4B)	20.0	21.5	20.8
UI 100%	23.1	16.9	20.0
t 100%	18.5	**30.8**	24.6
UI 50% / t 25% / $t-1$ 25%	**29.2**	21.5	25.4
UI 40% / t 35% / $t-1$ 25%	27.7	26.2	**26.9**

Table 7: **Offline RL Experiments.** Success rate (%) on 65 HV tasks. All RL runs use the same UI Genie RM BT 256K and a total budget of 2,400 training prompts.

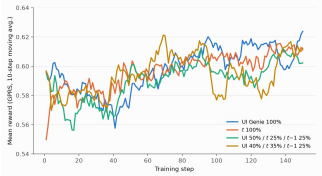


Figure 6: Mean GPRS reward per step. Because GPRS is a within-group win share, its mean can also increase as sampled actions become less diverse.

recipe, and budget of 2,400 training prompts; only the source of the prompts varies. Building on prior work that optimizes VLM and LLM based agents with offline RL (Qian et al., 2025; Chen et al., 2025) and online RL (Feng et al., 2025; Yang et al., 2026), we finetune Qwen3-VL-4B-Instruct with LoRA using offline GRPO with Group Preference Reward Shaping (GPRS) (Zhu et al., 2025), without executing sampled actions in the environment during training. We compare generic UI-Genie contexts with AndroidFlux contexts collected at the first-error checkpoint t, $t-1$, and mixtures of these sources. Further details are provided in Appendix E.2.

Table 7 shows that the source of offline training contexts substantially affects downstream behavior. Training on generic UI-Genie contexts alone provides little overall benefit and reduces recovery from first-error states. In contrast, every run containing AndroidFlux contexts outperforms UI-Genie-only training on mean success. Training exclusively at the first-error checkpoint produces the strongest recovery but degrades clean-start performance, indicating that concentrating only on failure states can over-specialize the policy.

Combining generic and failure-centered contexts largely removes this trade-off. In particular, the UI 40% / t 35% / $t-1$ 25% mixture improves both clean-start execution and error recovery over UI-Genie-only training, while also outperforming the base policy under both evaluation conditions. Moreover, the mixtures improve clean performance despite using fewer generic UI-Genie samples, suggesting that states surrounding first errors provide useful supervision beyond recovery itself. These results indicate that AndroidFlux is not only an evaluation resource: its policy-induced failure contexts can serve as informative offline training data for strengthening GUI agents without additional environment interaction.

Figure 6 further shows that optimization remains stable across all data mixtures. The mean GPRS reward increases similarly across runs, but it should not be interpreted as a direct measure of policy quality because the within-group statistic is affected by duplicate sampled actions. Indeed, the UI-Genie-only run achieves a comparable final GPRS reward despite yielding the weakest downstream performance. The underlying reward-model scores reveal a clearer difference: UI-Genie-only training plateaus and begins to decline during the latter part of optimization, whereas runs containing AndroidFlux contexts continue improving through the final training window. Together with the evaluation results, this suggests that failure-centered AndroidFlux contexts provide a more persistent learning signal than generic UI states under the same optimization budget.

7 Limitations and Conclusion

AndroidFlux covers only AndroidWorld and recoverable first errors that replay or saved snapshots can reconstruct, leaving irreversible failures, and other GUI environments for future work. The LLM-verified set inherits residual noise from the detector–verifier pipeline, and recovery success remains sensitive to environment nondeterminism. LPV also requires token likelihoods, limiting voters to open-weight models, and the reward-modeling benchmark is small enough that differences of a few pairs should be read with care. Despite these limitations, our results show that recovery from a committed error is distinct from clean-start success, and that agentic frameworks and GUI-domain reasoning RL improve it most. Generic GUI reward models rank recovery actions near chance, whereas LPV gives the strongest training-free preferences and, as training labels, adds recovery signal without erasing general preferences. Failure-centered contexts also improve offline RL. These findings establish failure recovery and recovery-aware reward modeling as key targets for robust GUI agents in the wild.

AI USE STATEMENT

We used generative AI tools as part of the benchmark pipeline: Claude and ChatGPT implement codes and writings broaderly for this work. The detector–verifier framework that annotates first errors in the LLM-verified set (Appendix C.9) and generates the schema-neutral history summaries (Appendix C.8), and produces the terminal-state labels for the offline RL contexts. The detector–verifier labels were validated against human annotations (Table 24); and to verify the quality of the published dataset. In addition, we used AI coding assistants to help implement our methods and experimental code, and generative AI tools to help draft and edit parts of the paper. All AI-assisted code was reviewed and tested by the authors, and all AI-assisted text was checked for accuracy by the authors. We did not use generative AI tools to propose or refine hypotheses, design the methodology or experiments, interpret results, or translate the paper. Developing theoretical frameworks, formulating mathematical claims, and writing proofs are not applicable to this work. All of our core ideas are original and we have reviewed all AI-assisted work and take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, et al. Qwen3-VL Technical Report. *arXiv preprint arXiv:2511.21631*, 2025.

Ralph A. Bradley and Milton E. Terry. Rank Analysis of Incomplete Block Designs: I. The Method of Paired Comparisons. *Biometrika*, 39(3/4):324–345, 1952.

Tianpeng Bu, Xin Liu, Qihua Chen, Hao Jiang, Shurui Li, et al. Recovering Policy-Induced Errors: Benchmarking and Trajectory Synthesis for Robust GUI Agents. *arXiv preprint arXiv:2605.29447*, 2026.

Hyungjoo Chae, Seonghwan Kim, Junhee Cho, Seungone Kim, Seungjun Moon, Gyeom Hwangbo, Dongha Lim, Minjin Kim, Yeonjun Hwang, Minju Gwak, Dongwook Choi, Minseok Kang, Gwanhoon Im, ByeongUng Cho, Hyojun Kim, Jun Han, Taeyoon Kwon, Minju Kim, Beong-woo Kwak, Dongjin Kang, and Jinyoung Yeo. Web-Shepherd: Advancing PRMs for reinforcing web agents. In *Advances in Neural Information Processing Systems*, 2025.

Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance. *arXiv preprint arXiv:2305.05176*, 2023.

Ziyuan Chen, Zhenghui Zhao, Zhangye Han, Miancan Liu, Xianhang Ye, Yiqing Li, Hongbo Min, Jinkui Ren, Xiantao Zhang, and Guitao Cao. TGPO: Tree-guided preference optimization for robust web agent reinforcement learning, 2025. URL https://arxiv.org/abs/2509.14172.

Sanjiban Choudhury. Process reward models for LLM agents: Practical framework and directions, 2025.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. Retool: Reinforcement learning for strategic tool use in llms, 2025. URL https://arxiv.org/abs/2504.11536.

FinalRun. FinalRun AndroidWorld Benchmark, 2026. URL https://finalrun.app/benchmark/.

Google DeepMind. Gemini 3 Flash Model Card, December 2025. URL https://deepmind.google/models/model-cards/gemini-3-flash/.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Zhiyuan Hu, Shiyun Xiong, Yifan Zhang, See-Kiong Ng, Anh Tuan Luu, Bo An, Shuicheng Yan, and Bryan Hooi. Guiding VLM agents with process rewards at inference time for GUI navigation, 2025.

Quyu Kong, Xu Zhang, Zhenyu Yang, Nolan Gao, Chen Liu, et al. MobileWorld: Benchmarking Autonomous Mobile Agents in Agent-User Interactive, and MCP-Augmented Environments. *arXiv preprint arXiv:2512.19432*, 2025.

Sijia Li, Xiaoyu Tan, Shahir Ali, Niels Schmidt, Gengchen Ma, and Xihe Qiu. Curiosity Driven Knowledge Retrieval for Mobile Agents. *arXiv preprint arXiv:2601.19306*, 2026.

Wei Li, William Bishop, Alice Li, Christopher Rawles, Folawiyo Campbell-Ajala, Divya Tyamagundlu, and Oriana Riva. On the Effects of Data Scale on UI Control Agents. *arXiv preprint arXiv:2406.03679*, 2024.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, et al. Let's Verify Step by Step. *arXiv preprint arXiv:2305.20050*, 2023.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, et al. RouteLLM: Learning to Route LLMs with Preference Data. *arXiv preprint arXiv:2406.18665*, 2024.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, et al. Training Language Models to Follow Instructions with Human Feedback. *arXiv preprint arXiv:2203.02155*, 2022.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. Toolrl: Reward is all tool learning needs, 2025. URL https://arxiv.org/abs/2504.13958.

Mateo Quiros. Introducing Computer Use in Gemini 3.5 Flash. Google DeepMind, June 2026. URL https://blog.google/innovation-and-ai/models-and-research/gemini-models/introducing-computer-use-gemini-3-5-flash/.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct Preference Optimization: Your Language Model is Secretly a Reward Model. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pp. 53728–53741, 2023.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, et al. AndroidWorld: A Dynamic Benchmarking Environment for Autonomous Agents. *arXiv preprint arXiv:2405.14573*, 2024.

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

Nisan Stiennon, Long Ouyang, Jeff Wu, Daniel M. Ziegler, Ryan Lowe, et al. Learning to Summarize from Human Feedback. *arXiv preprint arXiv:2009.01325*, 2020.

Jinbiao Wei, Kangqi Ni, Yilun Zhao, Guo Gan, and Arman Cohan. Step-level Optimization for Efficient Computer-use Agents. *arXiv preprint arXiv:2604.27151*, 2026.

Han Xiao, Guozhi Wang, Yuxiang Chai, Zimu Lu, Weifeng Lin, et al. UI-Genie: A Self-Improving Approach for Iteratively Boosting MLLM-based Mobile GUI Agents. *arXiv preprint arXiv:2505.21496*, 2025.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, et al. OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments. *arXiv preprint arXiv:2404.07972*, 2024.

Haiyang Xu, Xi Zhang, Haowei Liu, Junyang Wang, Zhaozai Zhu, Shengjie Zhou, et al. Mobile-Agent-v3.5: Multi-platform Fundamental GUI Agents. *arXiv preprint arXiv:2602.16855*, 2026.

Yifan Xu, Xiao Liu, Xueqiao Sun, Siyi Cheng, Hao Yu, et al. AndroidLab: Training and Systematic Benchmarking of Android Autonomous Agents. *arXiv preprint arXiv:2410.24024*, 2024.

Yifan Xu, Xiao Liu, Xinghan Liu, Jiaqi Fu, Hanchen Zhang, et al. MobileRL: Online Agentic Reinforcement Learning for Mobile GUI Agents. *arXiv preprint arXiv:2509.18119*, 2025.

Rui Yang, Qianhui Wu, Yuxi Chen, Hao Bai, Wenlin Yao, Hao Cheng, Baolin Peng, Huan Zhang, Tong Zhang, and Jianfeng Gao. OpenWebRL: Demystifying online multi-turn reinforcement learning for visual web agents, 2026. URL https://arxiv.org/abs/2606.02031.

Hanzhang Zhou, Xu Zhang, Panrong Tong, Jianan Zhang, Liangyu Chen, Quyu Kong, et al. MAI-UI Technical Report: Real-World Centric Foundation GUI Agents. *arXiv preprint arXiv:2512.22047*, 2025.

Huaisheng Zhu, Siyuan Xu, Hangfan Zhang, Teng Xiao, Zhimeng Guo, Shijie Zhou, Shuyue Hu, and Vasant G. Honavar. Reinforcement learning for large language models via group preference reward shaping. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 21387–21400, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.1085. URL https://aclanthology.org/2025.emnlp-main.1085/.

A AndroidWorld & AndroidFlux Action Space

AndroidFlux inherits the action space of AndroidWorld, summarized in Table 8. Every action carries an `action_type` together with typed arguments, and `status` and `answer` terminate the episode. Fixing one grammar is what makes cross-agent replay well defined—candidate actions produced by different agents become comparable strings over a single action language, so a recorded action can be re-executed by the replay operator and a recovery action produced by one model can be scored against one produced by another.

Table 8: The action space of AndroidWorld and AndroidFlux. Actions with † terminate the episode.

Action type	Typed arguments	Effect
`click`	`index` or `(x,y)`	Tap a UI element or screen coordinate
`double_tap`	`index` or `(x,y)`	Double tap
`long_press`	`index` or `(x,y)`	Press and hold
`scroll`	`direction`	Scroll the focused container
`swipe`	`direction`	Swipe across the screen
`input_text`	`text, index`	Enter text into a field
`keyboard_enter`	–	Submit the soft keyboard
`open_app`	`app_name`	Launch an application
`navigate_home`	–	Return to the launcher
`navigate_back`	–	Back navigation
`wait`	–	Yield one step for asynchronous state
`answer`†	`text`	Return a textual answer
`status`†	`goal_status`	Declare the episode complete or infeasible

B AndroidWorld

Table 9: Completed AndroidWorld clean-start measurements (116 tasks).

Model	Type	Reported SR (%)	Emulator Correct	Emulator SR (%)	Docker Correct	Docker SR (%)
Qwen3-VL-2B-Instruct	General	36.4	38/116	32.76	33/116	28.45
Qwen3-VL-4B-Instruct	General	45.3	55/116	47.41	51/116	43.97
Qwen3-VL-8B-Instruct	General	47.6	56/116	48.28	63/116	54.31
Qwen3-VL-30B-A3B-Instruct	General	54.3	56/116	48.28	27/116	23.28
Qwen3-VL-32B-Instruct	General	57.3	N/R	N/R	59/116	50.86
Qwen3.6-27B	General	70.3	N/R	N/R	32/116	27.59
Qwen3.8-27B	General	81.9	N/R	N/R	63/116	54.31
Gemma-4-31B-IT (M3A)	General	N/P	N/R	N/R	56/116	48.28
Gemini Flash 3.0 (Qwen3-VL prompt)	General	N/P	48/116	41.38	57/116	49.14
Gemini Flash 3.5 (Qwen3-VL prompt)	General	N/P	N/R	N/R	76/116	65.52
Gemini Flash 3.8	General	N/P	N/R	N/R	72/116	62.07
Gemini Pro 3.1	General	N/P	N/R	N/R	76/116	65.52
GUI-Owl-7B	Expert	66.4	70/116	60.34	71/116	61.21
GUI-Owl-32B	Expert	N/P	72/116	62.07	75/116	64.66
GUI-Owl-1.5-2B	Expert	67.9	77/116	66.38	78/116	67.24
GUI-Owl-1.5-4B	Expert	69.8	82/116	70.69	74/116	63.79
GUI-Owl-1.5-8B	Expert	69.0	81/116	69.83	77/116	66.38
GUI-Owl-1.5-32B	Expert	69.4	88/116	75.86	77/116	66.38
MAI-UI-2B	Expert	49.1	51/116	43.97	44/116	37.93
MAI-UI-8B	Expert	70.7	71/116	61.21	60/116	51.72
UI-TARS-1.0-7B-SFT	Expert	33.0	29/116	25.00	31/116	26.72
UI-TARS-1.5-7B	Expert	N/P	28/116	24.14	25/116	21.55
Gemini Flash 3.5 Computer Use	Expert	N/P	61/116	52.59	45/116	38.79
Mobile-Agent-v3-7B	Agent	N/P	N/R	N/R	78/116	67.24
Mobile-Agent-v3-32B	Agent	73.3	79/116	68.10	76/116	65.52
MobileRun (Gemini Flash 3.0)	Agent	N/P	N/R	N/R	95/116	81.90
FinalRun (FinalRun, 2026)	Agent	97.4	N/R	N/R	83/116	71.55
Qwen3-VL-4B-Thinking	Reasoning	52.0	N/R	N/R	34/116	29.31
Qwen3-VL-8B-Thinking	Reasoning	50.0	N/R	N/R	42/116	36.21
Qwen3-VL-32B-Thinking	Reasoning	63.7	N/R	N/R	34/116	29.31
GUI-Owl-1.5-8B-Thinking	Reasoning	71.6	84/116	72.41	63/116	54.31
GUI-Owl-1.5-32B-Thinking	Reasoning	N/P	N/R	N/R	82/116	70.69

N/P: not published—the developer has not reported a score on the 116-task AndroidWorld set. N/R: not run—no run was performed under that condition.

C ANDROIDFLUX: ERROR RECOVERY BENCHMARK

C.1 HUMAN VERIFIED SET

Six source models each attempted 112 AndroidWorld tasks, yielding 672 trajectories in total. The 112 tasks were obtained by excluding four of the original 116 tasks due to environment issues that prevented reliable execution or evaluation. All 672 trajectories were then manually reviewed step by step.

We retained tasks for which at least one of the six source models produced a trajectory containing an error at a non-boundary step; trajectories whose only error occurred at the first or final step were excluded. From the eligible trajectories, we selected one source trajectory per task while balancing the source-model distribution as evenly as possible, resulting in 80 human-verified cases. Table 11 reports the resulting source-model distribution.

We then attempted to reconstruct each selected error state by replaying the recorded action sequence. For 65 cases, action replay successfully reproduced the annotated error state and was used directly for evaluation. For the remaining 15 cases, replaying the actions alone did not reproduce the intended error state, so we instead restore the corresponding state from an emulator snapshot. Details of the replay and snapshot procedure are provided in Appendix C.5.1. Table 21 reports the distribution of first-error categories and the corresponding category-wise recovery results.

C.2 $t{-}5$, $t{-}3$, $t{-}1$, $t{+}1^{+}$, AND $t{+}1^{-}$

We construct two complementary extensions of the first-error checkpoint t. The pre-error conditions $t{-}n$, for $n \in \{1, 3, 5\}$, measure how recovery changes as the target policy is given more opportunity to act before the annotated error. For each condition, we replay the source trajectory through step $t_{\text{err}} - n$ and truncate the model-visible history at the same point; the target policy then completes the task autonomously. Thus, the environment state and the primed history always correspond to the same source prefix. Some errors occur too early to admit a requested offset. When $t_{\text{err}} < n$, we retain the task but map that condition to a clean start, with neither replayed actions nor primed history. This keeps all conditions paired over the same 65 replay cases, but it also means that the aggregate $t{-}3$ and $t{-}5$ scores mix true offset starts with clean starts. We therefore report both the 65-case aggregate and, where relevant, the score restricted to feasible cases. Table 12 quantifies this distinction: all cases admit $t{-}1$, whereas only 60.0% and 32.3% admit $t{-}3$ and $t{-}5$, respectively.

The post-error conditions probe the downstream effect of one candidate recovery action. For a given task, both members of a pair begin at the same restored error state t. We execute either the first action from a recovery rollout that eventually succeeds, denoted $t{+}1^{+}$, or the first action from one that eventually fails, denoted $t{+}1^{-}$; after this injected action, the evaluated policy acts autonomously from the resulting state. The superscripts therefore describe the outcome of the rollout that supplied the action, rather than a claim that the single action is sufficient for success or failure.

Candidate actions are collected from the 27 models with accepted t-condition rollouts. We retain a task only when both a successful-rollout action and a failed-rollout action are available, producing 52 matched tasks (104 actions); the remaining 13 tasks have no successful recovery in the candidate pool. Before selection, actions are validated against the AndroidFlux action grammar, and syntactically valid calls that cannot be executed in AndroidWorld are discarded. To limit dependence on any single policy, we process tasks with the fewest candidates first and choose, on each side, the eligible contributor used least often so far. Table 13 lists the agents whose actions are selected for the resulting matched subset and their counts on each side of the pair.

C.3 LLM VERIFIED SET

Table 14 reports the source-model distribution. Coverage is deliberately unbalanced: the detector–verifier pipeline surfaces more usable first errors from weaker or more verbose policies, so a few sources dominate while the tail supplies breadth.

Table 10: AndroidWorld clean-start success and human-verified recovery at the first-error checkpoint. *AndroidWorld* is the Docker run over the 116 AndroidWorld tasks in Table 9, given for reference. *Scratch* is the same 116-task run restricted to the 80 recovery tasks; t_{all} replays the recorded prefix to the first-error checkpoint over the same 80 cases.

		AndroidWorld (116)		Scratch (80)		t_{all} (80)	
Model	Type	Correct	SR (%)	Correct	SR (%)	Correct	SR (%)
Qwen3-VL-2B-Instruct	General	33/116	28.45	13/80	16.25	12/80	15.00
Qwen3-VL-4B-Instruct	General	51/116	43.97	21/80	26.25	21/80	26.25
Qwen3-VL-8B-Instruct	General	63/116	54.31	33/80	41.25	23/80	28.75
Qwen3-VL-30B-A3B-Instruct	General	27/116	23.28	11/80	13.75	21/80	26.25
Qwen3-VL-32B-Instruct	General	59/116	50.86	29/80	36.25	24/80	30.00
Qwen3.6-27B (think on)	General	32/116	27.59	14/80	17.50	25/80	31.25
Qwen3.8-27B (think on)	General	63/116	54.31	36/80	45.00	26/80	32.50
Gemma-4-31B-IT (M3A)	General	56/116	48.28	30/80	37.50	34/80	42.50
Gemini Flash 3.0 (Qwen3-VL prompt)	General 57/116	49.14	32/80	40.00	22/80	27.50	
Gemini Flash 3.5 (Qwen3-VL prompt)	General	76/116	65.52	45/80	56.25	34/80	42.50
Gemini Flash 3.8	General	72/116	62.07	41/80	51.25	43/80	53.75
Gemini Pro 3.1	General	76/116	65.52	44/80	55.00	40/80	50.00
GPT-5.6 Luna	General	–	–	–	–	27/80	33.75
GPT-5.6 Terra	General	–	–	–	–	36/80	45.00
GUI-Owl-7B	Expert	71/116	61.21	37/80	46.25	24/80	30.00
GUI-Owl-32B	Expert	75/116	64.66	43/80	53.75	21/80	26.25
GUI-Owl-1.5-2B	Expert	78/116	67.24	46/80	57.50	32/80	40.00
GUI-Owl-1.5-4B	Expert	74/116	63.79	40/80	50.00	27/80	33.75
GUI-Owl-1.5-8B	Expert	77/116	66.38	45/80	56.25	32/80	40.00
GUI-Owl-1.5-32B	Expert	77/116	66.38	45/80	56.25	26/80	32.50
MAI-UI-2B	Expert	44/116	37.93	17/80	21.25	17/80	21.25
MAI-UI-8B	Expert	60/116	51.72	30/80	37.50	28/80	35.00
UI-TARS-1.0-7B-SFT	Expert	31/116	26.72	13/80	16.25	13/80	16.25
UI-TARS-1.5-7B	Expert	25/116	21.55	10/80	12.50	19/80	23.75
Gemini Flash 3.5 Computer Use	Expert	45/116	38.79	31/80	38.75	28/80	35.00
Mobile-Agent-v3-7B	Agent	78/116	67.24	47/80	58.75	31/80	38.75
Mobile-Agent-v3-32B	Agent	76/116	65.52	46/80	57.50	34/80	42.50
MobileRun (Gemini Flash 3.0)	Agent	95/116	81.90	63/80	78.75	54/80	67.50
FinalRun (M3A + app knowledge)	Agent	83/116	71.55	53/80	66.25	33/80	43.75
Qwen3-VL-4B-Thinking	Reasoning	34/116	29.31	10/80	12.50	10/80	12.50
Qwen3-VL-8B-Thinking	Reasoning	42/116	36.21	18/80	22.50	13/80	16.25
Qwen3-VL-32B-Thinking	Reasoning	34/116	29.31	9/80	11.25	14/80	17.50
GUI-Owl-1.5-8B-Thinking	Reasoning	63/116	54.31	33/80	41.25	24/80	30.00
GUI-Owl-1.5-32B-Thinking	Reasoning	82/116	70.69	50/80	62.50	30/80	37.50

Table 11: Source-model distribution of the human-verified set. Replay cases are restored by replaying the recorded action prefix; snapshot cases resume from a saved emulator snapshot.

Source model	Replay (n=65)	Snapshot (n=15)	Total (n=80)
Gemini Flash (Qwen3-VL prompt)	17 (26.15%)	5 (33.33%)	22 (27.50%)
Qwen3-VL-2B	10 (15.38%)	4 (26.67%)	14 (17.50%)
GUI-Owl-32B	10 (15.38%)	1 (6.67%)	11 (13.75%)
Mobile-Agent-v3-32B	10 (15.38%)	1 (6.67%)	11 (13.75%)
MAI-UI-8B	9 (13.85%)	2 (13.33%)	11 (13.75%)
Qwen3-VL-4B	9 (13.85%)	2 (13.33%)	11 (13.75%)
Total	65	15	80

Table 12: Construction of the pre-error conditions on the 65 replay-based, human-verified cases. A clipped case is evaluated from a clean start because the source trajectory does not contain the requested pre-error offset.

Condition	Offset n	Feasible	Clipped	Feasible share
$t-1$	1	65	0	100.0%
$t-3$	3	39	26	60.0%
$t-5$	5	21	44	32.3%

Table 13: Recovery-action source distribution in the matched $t{+}1^{+}/t{+}1^{-}$ subset. Counts identify the models that authored the selected actions, not the models that generated the original error trajectories.

Recovery-action source	+	−	Total	Recovery-action source	+	−	Total
MobileRun (Gemini Flash 3.0)	9	1	10	GUI-Owl-1.5-32B	0	4	4
Gemini Flash 3.5 Computer Use	3	1	4	GUI-Owl-1.5-8B (FinalRun app info)	0	4	4
GUI-Owl-1.5-2B	3	1	4	Gemini Flash 3.0 Computer Use	0	4	4
Gemma-4-31B-IT (M3A)	3	1	4	MAI-UI-2B	1	2	3
GUI-Owl-1.5-8B	3	1	4	MAI-UI-8B	2	1	3
GUI-Owl-1.5-8B-Thinking	4	0	4	Qwen3-VL-32B-Instruct	1	2	3
Gemini Flash 3.5	3	1	4	Qwen3-VL-8B-Instruct	2	1	3
Mobile-Agent-v3-32B	2	2	4	Qwen3-VL-30B-A3B-Instruct	1	2	3
Gemini Flash 3.0	3	1	4	Qwen3-VL-8B-Thinking	1	2	3
GUI-Owl-32B	3	1	4	UI-TARS-1.5-7B	1	2	3
GUI-Owl-7B	1	3	4	Qwen3-VL-2B-Instruct	0	3	3
GUI-Owl-1.5-4B	2	2	4	Qwen3-VL-4B-Instruct	0	3	3
Mobile-Agent-v3-7B	1	3	4	UI-TARS-1.0-7B-SFT	0	3	3
Qwen3-VL-4B-Thinking	3	1	4				
Total (27 models)	**52**	**52**	**104**				

Table 14: Source-model distribution of the LLM-verified set (n=312 over 104 tasks, 21 source models).

Source model	Cases	Source model	Cases
Gemini Flash (Qwen3-VL prompt)	72 (23.08%)	GUI-Owl-1.5-2B	5 (1.60%)
GUI-Owl-32B	49 (15.71%)	Mobile-Agent-v3-7B	4 (1.28%)
Gemma-4-31B-it-0716 (M3A)	49 (15.71%)	Qwen3-VL-4B	4 (1.28%)
Gemma-4-31B (M3A)	42 (13.46%)	Qwen3-VL-4B-thinking	3 (0.96%)
Qwen3-VL-2B	24 (7.69%)	GUI-Owl-1.5-32B	3 (0.96%)
GUI-Owl-7B	17 (5.45%)	GUI-Owl-1.5-4B	2 (0.64%)
UI-TARS-1.0-7B-SFT	12 (3.85%)	MAI-UI-8B	2 (0.64%)
GUI-Owl-1.5-8B (Think)	7 (2.24%)	Mobile-Agent-v3-32B	1 (0.32%)
Qwen3-VL-32B-Instruct	7 (2.24%)	GUI-Owl-1.5-8B	1 (0.32%)
MAI-UI-2B	6 (1.92%)	Qwen3-VL-8B-thinking	1 (0.32%)
		Qwen3-VL-30B-A3B-Instruct	1 (0.32%)
		Total	312 (100%)

Table 15: LLM-verified set (312 cases), full table. *Scratch* projects the model's 116-task AndroidWorld clean-start run onto these 312 cases; t replays the recorded prefix to the first-error checkpoint. *AndroidWorld* and *s104* are the same 116-task run and its projection onto the 104 distinct tasks this set covers, given for reference.

Model	Type	Scratch (312)	SR (%)	t (312)	SR (%)	AndroidWorld (116)	s104
Qwen3-VL-2B-Instruct	General	69/312	22.12	77/312	24.68	33/116	23/104
Qwen3-VL-4B-Instruct	General	120/312	38.46	101/312	32.37	51/116	40/104
Qwen3-VL-8B-Instruct	General	153/312	49.04	92/312	29.49	63/116	51/104
Qwen3-VL-32B-Instruct	General	141/312	45.19	106/312	33.97	59/116	47/104
Gemma-4-31B-IT (M3A, think off)	General	135/312	43.27	139/312	44.55	56/116	45/104
Gemma-4-31B-IT (M3A, think on)	General	–	–	148/312	47.44	–	–
Gemini Flash 3.0 (Qwen3-VL prompt, think on)	General	144/312	46.15	124/312	39.74	–	–
Gemini 3.8 Flash	General	183/312	58.65	192/312	61.54	72/116	61/104
Gemini 3.1 Pro	General	192/312	61.54	182/312	58.33	76/116	64/104
Qwen3.6-27B (think on)	General	72/312	23.08	105/312	33.65	32/116	24/104
Qwen3.8-27B (think off)	General	105/312	33.65	88/312	28.21	46/116	35/104
Qwen3.8-27B (think on)	General	156/312	50.00	118/312	37.82	62/116	52/104
GPT-5.6 Luna	General	–	–	99/312	31.73	–	–
GPT-5.6 Terra	General	–	–	133/312	42.63	–	–
GUI-Owl-7B	Expert	177/312	56.73	85/312	27.24	71/116	59/104
GUI-Owl-32B	Expert	189/312	60.58	88/312	28.21	75/116	63/104
GUI-Owl-1.5-8B	Expert	195/312	62.50	138/312	44.23	77/116	65/104
GUI-Owl-1.5-32B	Expert	195/312	62.50	113/312	36.22	77/116	65/104
MAI-UI-8B	Expert	144/312	46.15	99/312	31.73	60/116	48/104
UI-TARS-1.0-7B-SFT	Expert	60/312	19.23	42/312	13.46	31/116	20/104
Gemini Flash 3.5 Computer Use	Expert	126/312	40.38	125/312	40.06	45/116	42/104
Mobile-Agent-v3-7B	Agent	198/312	63.46	141/312	45.19	78/116	66/104
Mobile-Agent-v3-32B	Agent	192/312	61.54	144/312	46.15	76/116	64/104
MobileRun (Gemini Flash 3.0)	Agent	249/312	79.81	209/312	66.99	95/116	83/104
MobileRun (Gemini Pro 3.1)	Agent	–	–	215/312	68.91	–	–
FinalRun (M3A + app knowledge)	Agent	213/312	68.27	122/312	39.10	83/116	71/104
Qwen3-VL-32B-Thinking	Reasoning	81/312	25.96	66/312	21.15	34/116	27/104
GUI-Owl-1.5-32B-Thinking	Reasoning	210/312	67.31	139/312	44.55	82/116	70/104
GUI-Owl-1.5-8B-Thinking	Reasoning	159/312	50.96	125/312	40.06	63/116	53/104
Qwen3-VL-8B-Thinking	Reasoning	93/312	29.81	87/312	27.88	42/116	31/104
Qwen3-VL-4B-Thinking	Reasoning	72/312	23.08	68/312	21.79	34/116	24/104

C.4 CROSS-AGENT RECOVERY MATRIX

Agent \ Traj. Source	Mobile-Agent-v3-32B	Gemini Flash 3.0	GUI-Owl-32B	MAI-UI-8B	Qwen3-VL-2B	Qwen3-VL-4B	Overall
Source agents							
Mobile-Agent-v3-32B	30.0%	82.4%	30.0%	33.3%	10.0%	33.3%	41.5%
Gemini Flash 3.0	10.0%	52.9%	10.0%	33.3%	0.0%	22.2%	24.6%
GUI-Owl-32B	20.0%	41.2%	10.0%	22.2%	0.0%	22.2%	21.5%
MAI-UI-8B	20.0%	52.9%	30.0%	33.3%	0.0%	33.3%	30.8%
Qwen3-VL-2B	0.0%	41.2%	10.0%	0.0%	0.0%	0.0%	12.3%
Qwen3-VL-4B	10.0%	47.1%	20.0%	33.3%	0.0%	0.0%	21.5%
By class (Avg.)							
General	28.8%	61.0%	23.8%	37.5%	10.0%	34.7%	35.6%
Expert	20.0%	47.1%	26.7%	29.6%	13.3%	37.0%	30.8%
Agentic framework	36.7%	68.6%	36.7%	51.9%	16.7%	55.6%	46.7%

Table 16: Error recovery evaluation matrix at condition t on the human-verified set. Values are recovery success rates (%). The upper block is the square source-to-source matrix. The lower block averages evaluated agents that are not trajectory sources, grouped by class as in Tables 3 and 4.

The matrix shows that recovery is governed strongly by which agent produced the error. Self-recovery is rarely the strongest entry in its own column: only Mobile-Agent-v3-32B recovers its own errors better than the other source agents, and Qwen3-VL-2B and Qwen3-VL-4B recover none of theirs, suggesting that the states on which an agent first fails are not necessarily the states it is best equipped to repair. Recovery also tracks the composition of first-error categories in Table 21. The three sources with the largest Execution share, Gemini Flash 3.0, MAI-UI-8B, and Qwen3-VL-4B, are the three most recoverable under every agent class, with Gemini Flash 3.0 the most recoverable throughout. The remaining sources, Mobile-Agent-v3-32B, GUI-Owl-32B, and Qwen3-VL-2B, have at most 27.3% Execution and at least 45.5% Direction errors, and are the three hardest. An Execution error leaves the plan intact, so the target can often repair it by redoing the step, whereas a Direction error first requires recognizing that the route itself is wrong. Qwen3-VL-2B is the hardest source under every class, combining a Direction-heavy profile (50.0%) with the highest Stagnation rate in the set (28.6%). Agentic frameworks recover best from every source, for example reaching 55.6% on Qwen3-VL-4B against 36.5% and 33.3% for general and expert agents. Overall, the matrix indicates that recoverability depends jointly on the source policy and the type of state induced by its first error. Further details on the error-category distributions are provided in Appendix C.7.

C.5 REPRODUCIBILITY

Every configuration in Tables 17 and 18 is run four times under an identical setup: same Docker image, same emulator snapshot and pinned app versions, same task list, and fixed seeds. This still does not make the pipeline deterministic. The emulator renders asynchronously, so observations may be captured mid-transition before an animation, network call, or background service has settled. Residual device state drifts with execution order—caches, permission dialogs, and recency-ordered surfaces such as the launcher and file pickers differ across nominally identical initial states—and clock-dependent verifiers are sensitive to elapsed time. Batched serving further makes floating-point reduction order depend on batch composition, so decoding is not bitwise reproducible even when greedy.

Table 17: Repeated Docker runs on AndroidWorld (116 tasks). Cells report the number of successful tasks with the corresponding success rate in parentheses.

	Successful tasks (SR %)				Aggregate	
Model	Run 1	Run 2	Run 3	Run 4	Mean SR (%)	Range (pp)
Qwen3-VL-4B-Instruct	46 (39.66)	46 (39.66)	46 (39.66)	47 (40.52)	39.88	0.86
Qwen3-VL-8B-Instruct	66 (56.90)	61 (52.59)	60 (51.72)	63 (54.31)	53.88	5.17
MAI-UI-8B	60 (51.72)	60 (51.72)	60 (51.72)	60 (51.72)	51.72	0.00
GUI-Owl-7B	73 (62.93)	71 (61.21)	68 (58.62)	71 (61.21)	60.99	4.31
GUI-Owl-1.5-4B	74 (63.79)	74 (63.79)	74 (63.79)	74 (63.79)	63.79	0.00
GUI-Owl-1.5-8B	77 (66.38)	74 (63.79)	76 (65.52)	79 (68.10)	65.95	4.31

C.5.1 REPLAY AND SNAPSHOT

The 80 human-verified cases are divided into 65 replay-based and 15 snapshot-based cases according to whether the annotated error state can be reproduced by action replay. For the 65 replay-based cases, we re-execute in the emulator the source model's recorded action sequence through the annotated first-error step t. A case is considered replayable when this procedure reaches the same erroneous state observed in the original trajectory. Once that state is reached, the recovering policy takes over and continues the task from the restored state. Its inherited history is constructed from the source trajectory metadata and rendered in the target policy's native format, as detailed in Appendix C.8.

For the remaining 15 cases, replaying the recorded actions from the task's initial state does not reproduce the original error state. A primary source of such mismatch is residual environment state from preceding tasks. For example, when the original AndroidWorld tasks are executed sequentially, previously used applications may change their positions in recency-ordered interfaces; replaying an individual task in isolation does not reproduce this preceding interaction history and can therefore lead to a different state despite executing the same actions. For these cases, we manually reconstruct the emulator to match the original error state and save it as a VM snapshot, allowing evaluation to resume directly from that state. The inherited history remains based on the original source trajectory, with only the minimal state-dependent details responsible for the replay mismatch adjusted to remain consistent with the reconstructed state. The recovering policy then receives this reconstructed state and history under the same evaluation protocol as in the replay-based cases.

C.5.2 RECOVERY TRAJECTORY GENERATION

Given a source trajectory τ^{src} with annotated first error t, a recovery trajectory is generated by instantiating the Resume operator of Eq. (3) in the live emulator at $k = t$, the checkpoint at which the error has been committed under an otherwise correct prefix. The environment is first restored to $\hat{s}_t$ by re-executing the recorded action prefix $a_{0:t}$ under the same task seed; for the 15 human-verified trajectories whose replay does not reach the error state (Appendix C.5.1), the manually reconstructed emulator VM snapshot is loaded instead. The target policy π then receives the reconstructed context h_t^{π}, built by re-rendering the schema-neutral step records of Appendix C.8 into π's native prompt format, together with the live observation of the restored screen. Nothing in that context marks the prefix as replayed, erroneous, or authored by another model: from the target's perspective the episode is simply in progress, and steps 0 through t read as its own past actions. The target is therefore never told that a recovery is required—noticing the broken state is part of what AndroidFlux measures.

From step $t+1$ the target acts autonomously over the action space of Table 8, and actions are executed on the device exactly as in a clean-start rollout. The episode terminates when the target emits a terminal action (`status` or `answer`) or exhausts the remaining step budget $H - t$, and the terminal state is scored by the task's programmatic verifier r. A trajectory is thus credited only when the target both repairs the effect of a_t and completes the remainder of the task; partial repair without completion is scored as failure. When $\pi = \pi_{\text{src}}$ the run measures self-recovery, and otherwise the target inherits a failure and a history authored by a different model—the cross-agent condition of Table 16. Because the restored state and the rendered prefix are both fixed before any target is run, the same broken state is handed identically to every agent in the roster, so differences in success rate are attributable to the recovering policy alone.

Each (task, target) pair is run four times under the configuration of Appendix C. Table 18 reports per-run recovery counts for both the 65 replay-restored cases and the additional 15 snapshot cases. Run-to-run spread on the replay-restored subset stays within 2–5 recovered tasks per model, comparable in absolute terms to the variation of clean-start AndroidWorld rollouts under the same setup (Table 17), so resuming from a restored state does not add instability beyond what the environment itself contributes. Recovery rates broken down by first-error category are reported in Appendix C.7.

Table 18: Repeated Docker runs on the AndroidFlux human-verified set (80 tasks: 65 action-replay, 15 snapshot). Runs report the number of tasks recovered after resuming at the first-error step t; mean columns report the mean count with the corresponding rate in parentheses.

	Action-replay (65)					Snapshot (15)					Overall (80)	
Model	R1	R2	R3	R4	Mean (%)	R1	R2	R3	R4	Mean (%)	Mean (%)	Range (pp)
Qwen3-VL-4B-Instruct	12	14	15	13	13.50 (20.77)	8	8	8	7	7.75 (51.67)	21.25 (26.56)	3.75
Qwen3-VL-8B-Instruct	13	15	14	18	15.00 (23.08)	6	7	3	6	5.50 (36.67)	20.50 (25.63)	8.75
MAI-UI-8B	16	15	18	17	16.50 (25.38)	7	7	8	7	7.25 (48.33)	23.75 (29.69)	5.00
GUI-Owl-7B	20	14	18	17	17.25 (26.54)	6	5	8	3	5.50 (36.67)	22.75 (28.44)	8.75
GUI-Owl-1.5-4B	19	20	20	20	19.75 (30.38)	8	6	5	7	6.50 (43.33)	24.25 (30.31)	2.50
GUI-Owl-1.5-8B	24	25	25	24	24.50 (37.69)	8	8	9	8	8.25 (55.00)	32.75 (40.94)	2.5

C.6 Rewinding vs. Injection by Failure Timing

Episodes and models. The analysis uses the 312 episodes of the LLM-verified set. The 18 evaluated agents are listed in Table 19. GPT-5.6 Luna and Terra, Gemma-4-31B-IT (M3A) and Mobile-Agent-v3-32B are not included. A run counts as a success when its score is positive.

Gains and contrast. The unit is a (model, episode) pair, and the outcome of a run is 1 if it succeeds and 0 otherwise. For each pair, the rewinding gain is the outcome under condition $t{-}1$ minus the outcome under condition t, and the injection gain is the outcome under $t{+}1^{+}$ minus the outcome under $t{+}1^{-}$; each is therefore $+1$, 0 or -1. Averaged over the pairs in a bucket, a gain is the difference in success rate between its two conditions, reported in percentage points. Within a bucket, Δ is the injection gain minus the rewinding gain. The headline contrast is $\Delta_{\geq 7} - \Delta_{\leq 3}$: Δ in the latest bucket (≥ 7 steps before the error) minus Δ in the earliest bucket (≤ 3 steps), written "latest $-$ earliest".

Failure timing. The number of steps before the error is the index of the first erroneous step in the recorded source trajectory, i.e. how many steps the source agent had executed before it went wrong. Episodes are split into ≤ 3, 4–6 and ≥ 7 steps. These boundaries were fixed in an earlier analysis of a subset of the data and were not tuned on the present results.

Coverage of the two gains. Rewinding is measured on all 312 episodes (116, 100 and 96 per bucket; the n in Table 5), i.e. 2,088, 1,800 and 1,728 model–episode pairs. The two $t{+}1$ arms each cover 208 episodes and both arms are available for 196 of them, so injection is averaged over 196 episodes (77, 64 and 55 per bucket; 1,386, 1,152 and 990 pairs).

Uncertainty. Every model attempts every episode, so the pairs within an episode are correlated. We therefore resample *episodes*: the episodes of the earliest and the latest bucket are drawn with replacement, all pairs of a drawn episode are kept, and the contrast is recomputed. The 95% interval is

Table 19: Per-model Δ (injection $-$ rewinding, pp) by steps before the error, and the latest $-$ earliest contrast. 17 of the 18 models are positive.

Model	Type	≤ 3	4–6	≥ 7	Latest $-$ earliest
Qwen3-VL-2B-Instruct	General	$+0.02$	$+14.62$	$+3.37$	$+3.35$
Qwen3-VL-4B-Instruct	General	-13.38	$+6.75$	$+5.97$	$+19.34$
Qwen3-VL-8B-Instruct	General	-9.48	-6.88	$+10.64$	$+20.13$
Qwen3-VL-32B-Instruct	General	-7.34	-3.06	-16.65	-9.30
Qwen3.8-27B (think on)	General	-17.67	$+5.81$	$+7.52$	$+25.19$
Gemini Flash 3.0 (think on)	General	-5.62	$+5.81$	-2.86	$+2.76$
Gemini Flash 3.8	General	$+1.72$	$+5.12$	$+11.70$	$+9.98$
Gemini 3.1 Pro	General	$+0.01$	$+13.19$	$+7.52$	$+7.51$
GUI-Owl-7B	Expert	-12.05	-21.56	$+8.05$	$+20.10$
GUI-Owl-32B	Expert	-17.25	-24.81	-0.02	$+17.23$
GUI-Owl-1.5-8B	Expert	-9.92	$+1.12$	$+11.16$	$+21.07$
MAI-UI-8B	Expert	-8.21	-16.44	$+1.80$	$+10.01$
UI-TARS-1.0-7B-SFT	Expert	-2.99	$+2.94$	-0.80	$+2.19$
Gemini Flash 3.5 Computer Use	Expert	-16.82	-14.62	-13.54	$+3.27$
Mobile-Agent-v3-7B	Agent	-20.31	$+12.31$	$+15.08$	$+35.38$
MobileRun (Gemini Flash 3.0)	Agent	-12.95	-2.12	-8.31	$+4.64$
FinalRun (M3A + app knowledge)	Agent	-9.92	$+3.69$	$+5.19$	$+15.11$
GUI-Owl-1.5-8B-Thinking	Reasoning	-13.79	$+16.06$	-1.82	$+11.97$
All 18 (pooled)		-9.77	-0.11	$+2.44$	$+12.22$

the percentile interval, $[+4.1, +20.8]$. A one-way random-effects intra-class correlation with episodes as clusters is 0.090 for rewinding and 0.111 for injection, with 18 pairs per episode. The design effects are 2.54 and 2.89, so the effective sample is 39% and 35% of the nominal pairs. We do not report a bootstrap p for the headline contrast: it depends on only 6 to 13 of the 6,000 resamples, so it changes with the random seed, while the interval changes by less than 0.25pp.

Robustness checks. Table 20 lists the checks cited in Section 6.4. Per model, the contrast is positive for 17 of 18 models; the exception is Qwen3-VL-32B-Instruct (-9.30pp). A two-sided sign test gives $p = 0.0001$. Difficulty quartiles split episodes by their mean success at t over the 18 models (Q1 hardest). For trajectory length, we cut at the 33rd and 67th percentiles of the recorded source trajectory length; because many episodes have the same length, the three groups have 156, 54 and 102 episodes. Longer trajectories tend to have more steps before the error, so this check asks whether trajectory length alone explains the contrast. For the random split, we shuffle the 312 episodes once and split them into two halves of 156. Each group uses the same episode bootstrap with 4,000 resamples; p is two-sided.

Table 20: Latest $-$ earliest contrast (pp) within groups of episodes. All nine are positive.

Check	Group	Episodes	Contrast	95% CI	p
Difficulty	Q1 (hardest)	99	$+12.69$	$[+3.2, +22.8]$	0.010
	Q2	68	$+7.86$	$[-9.5, +25.2]$	0.380
	Q3	69	$+14.84$	$[-4.1, +36.5]$	0.143
	Q4 (easiest)	76	$+20.39$	$[+4.1, +37.3]$	0.011
Trajectory length	≤ 10 steps	156	$+21.03$	$[+2.5, +41.4]$	0.023
	11–16 steps	54	$+18.38$	$[+0.9, +36.2]$	0.038
	≥ 17 steps	102	$+3.29$	$[-9.9, +18.0]$	0.671
Random split	Half A	156	$+12.66$	$[+1.2, +26.3]$	0.029
	Half B	156	$+11.49$	$[+0.5, +23.3]$	0.044

C.7 First Error and Recovery Across Error Categories

We categorize first errors by *behavior* rather than by widget or app type. Each step is viewed as a triple of (intended goal direction, chosen action, observed screen transition), and the three classes

are distinguished by which element of this triple is broken. Classes are checked in a fixed priority order — stagnation → direction → execution — so that a repeated action is labeled as stagnation without further inspecting its direction or realization. Figure 7 shows, for each class, the screen immediately before and after a representative first-error step (a: stagnation, b: direction, c: execution), and Tables 21 and 22 summarize the per-category counts for the two verified sets.

Stagnation. The agent re-issues an action that has already failed on an essentially unchanged screen, or oscillates between states, so that the trajectory makes no visible progress (Fig. 7a). The failure lies not in *what* the agent intends but in the fact that it repeats the attempt without adapting. A first unsuccessful attempt is not counted: the error step is the point at which repetition is established (typically the second or later identical attempt), and actions with intrinsically subtle visual effects (e.g., text editing, keyboard input) are excluded from the "no screen change" criterion. Observed sub-patterns include identical re-taps on the same element, repeated scrolling over an unchanged list, re-entering an already-failed route, and acting on a nonexistent or already-achieved UI state.

Direction. The action executes successfully, but it moves the trajectory *away* from the goal (Fig. 7b): the agent commits to a visibly wrong target, screen, date, or app, regresses by dismissing progress it has already achieved, abandons a route on which the required affordance is already exposed, or ignores a visible contradiction. Local execution is correct; the choice is not. Exploratory route-opening, reversible entries into a wrong menu, and app switches for source verification are protected as valid — a direction error is assigned only when a wrong target or state is committed, or when the agent leaves a required screen without justification. This class subsumes goal-regressive, goal-irrelevant, goal-direction, and unhandled system-error behaviors in our finer-grained taxonomy.

Execution. The agent targets the right element with the right kind of action, but the action's *local effect* is wrong or absent (Fig. 7c): the committed text, value, or selection differs from the intent, the tap lands on a neighboring control, or a launch/save/toggle produces no state change. The plan is correct; the realization is not. The judgment unit is strictly the current step's intent versus its immediate outcome — an error is assigned when the immediately following screenshot shows an incorrect committed value, but not merely because the overall task remains unfinished, and stale UI or a missing toast after a correctly targeted commit is not presumed to be a failure.

Boundary between stagnation and execution. "No screen change" appears in both classes; the distinction is repetition. The *first* task-critical action that produces no effect (e.g., an app launch or toggle that fails) is an execution error, whereas re-attempting an action that has already failed is stagnation. Human-verified labels follow this convention consistently.

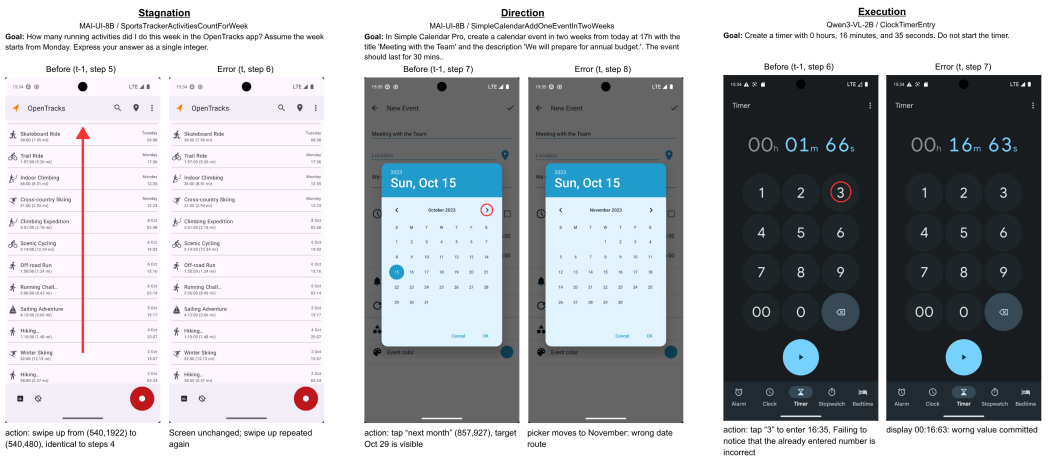
Distributions across verified sets. Tables 21 and 22 break down the first-error class of every verified trajectory by the source model that produced it: the first row of each table gives the overall class distribution of the set, and the remaining rows show, for each source model, how many of its error trajectories fall into each class. In the human-verified set, direction errors are the largest class overall (43.75%), and all three classes occur for every source model; the balance shifts by model, with direction dominating for GUI-Owl-32B and Qwen3-VL-2B, while MAI-UI-8B and Qwen3-VL-4B fail most often at execution. In the LLM-verified set, stagnation is the largest class overall (47.11%) and execution the smallest (17.31%), but this aggregate hides substantial per-model variation: GUI-Owl-32B and the two Gemma-4-31B runs, which together contribute 140 of the 312 trajectories, show a direction-dominant profile (45–53%) similar to the human-verified set, whereas the 72 Gemini Flash trajectories are labeled as stagnation in 76.39% of cases. Two caveats apply when comparing the two sets: the LLM-verified labels are obtained by collapsing the detector's five-way `pred_error_type` predictions into the three classes above rather than by human judgment, and the model composition of the two sets differs, so the per-model rows rather than the overall rows are the appropriate unit of comparison.

Table 21: First-error class distribution of the human-verified set (n=80), overall and by source model. Percentages are within each row.

Source model	Cases	Stagnation	Direction	Execution
All	80	18 (22.50%)	35 (43.75%)	27 (33.75%)
Gemini Flash (Qwen3VL Prompt)	22	4 (18.18%)	9 (40.91%)	9 (40.91%)
Qwen3-VL-2B	14	4 (28.57%)	7 (50.00%)	3 (21.43%)
GUI-Owl-32B	11	3 (27.27%)	6 (54.55%)	2 (18.18%)
Mobile-Agent-v3-32B	11	3 (27.27%)	5 (45.45%)	3 (27.27%)
MAI-UI-8B	11	3 (27.27%)	3 (27.27%)	5 (45.45%)
Qwen3-VL-4B	11	1 (9.09%)	5 (45.45%)	5 (45.45%)

Table 22: First-error class distribution of the LLM-verified set (n=312, 21 source models), overall and by source model. Classes are the detector's `pred_error_type` folded to the three-way taxonomy (stagnation→Stagnation; goal_regression, goal_irrelevant, goal_direction and system_error→Direction; incomplete_execution→Execution). Percentages are within each row.

Source model	Cases	Stagnation	Direction	Execution
All	312	147 (47.11%)	111 (35.58%)	54 (17.31%)
Gemini Flash (Qwen3VL Prompt)	72	55 (76.39%)	8 (11.11%)	9 (12.50%)
GUI-Owl-32B	49	10 (20.41%)	25 (51.02%)	14 (28.57%)
Gemma-4-31B-it-0716 (M3A)	49	12 (24.49%)	26 (53.06%)	11 (22.45%)
Gemma-4-31B (M3A)	42	10 (23.81%)	19 (45.24%)	13 (30.95%)
Qwen3-VL-2B	24	18 (75.00%)	6 (25.00%)	0 (0.00%)
GUI-Owl-7B	17	9 (52.94%)	5 (29.41%)	3 (17.65%)
UI-TARS-1.0-7B-SFT	12	9 (75.00%)	3 (25.00%)	0 (0.00%)
GUI-Owl-1.5-8B (Think)	7	2 (28.57%)	3 (42.86%)	2 (28.57%)
Qwen3-VL-32B-Instruct	7	6 (85.71%)	1 (14.29%)	0 (0.00%)
MAI-UI-2B	6	1 (16.67%)	5 (83.33%)	0 (0.00%)
GUI-Owl-1.5-2B	5	1 (20.00%)	3 (60.00%)	1 (20.00%)
Mobile-Agent-v3-7B	4	2 (50.00%)	2 (50.00%)	0 (0.00%)
Qwen3-VL-4B	4	2 (50.00%)	2 (50.00%)	0 (0.00%)
Qwen3-VL-4B-thinking	3	3 (100.00%)	0 (0.00%)	0 (0.00%)
GUI-Owl-1.5-32B	3	2 (66.67%)	1 (33.33%)	0 (0.00%)
GUI-Owl-1.5-4B	2	1 (50.00%)	1 (50.00%)	0 (0.00%)
MAI-UI-8B	2	1 (50.00%)	1 (50.00%)	0 (0.00%)
Mobile-Agent-v3-32B	1	0 (0.00%)	0 (0.00%)	1 (100.00%)
GUI-Owl-1.5-8B	1	1 (100.00%)	0 (0.00%)	0 (0.00%)
Qwen3-VL-8B-thinking	1	1 (100.00%)	0 (0.00%)	0 (0.00%)
Qwen3-VL-30B-A3B-Instruct	1	1 (100.00%)	0 (0.00%)	0 (0.00%)



(a) Stagnation (b) Direction (c) Execution

Figure 7: One human-verified first-error example per class (s_{t-1}: screen before the action; s_t: screen where the error is observed). (a) Stagnation: an identical swipe is repeated on an unchanged list. (b) Direction: the action succeeds but leaves the goal route (October 29 was visible; the agent advances to November). (c) Execution: the correct control is targeted but the wrong value is committed (00:16:63 instead of 00:16:35).

C.8 GENERALIZED HISTORY AND CONTEXT

Table 23: The schema-neutral step record. Provenance: *Copied* verbatim from the source step, *Norm.* deterministically re-encoded, *Synth.* generated when the source emits nothing usable, *Annot.* from the detector–verifier labels. Only *Synth.* fields reach a recovering agent; copied text is for auditing.

Group	Field	Prov.	Content
`step`	`memory_step_idx`	Norm.	Position in the replay prefix, contiguous from 0
	`source_step_idx`	Copied	Index in the original source trajectory
	`goal`	Copied	Task instruction c in force at this step
`screenshot`	`before_screenshot`	Copied	Observation o_i before the action
	`after_screenshot`	Copied	Observation o_{i+1} after the action
`action`	`parsed_action`	Norm.	Canonical tool call, coordinates rescaled to $[0, 1]$
	`json_action_type`	Norm.	Canonical type (`click`, `swipe`, `type`, ...)
	`execution_success`	Copied	Whether the emulator executed the action
	`action_raw`	Copied	Source model's action string, unedited
`thought`	`synthetic_thought`	Synth.	Imperative restatement of local intent
	`thought_raw`	Copied	Source model's reasoning u_i, unedited
`summary`	`summary_short`	Synth.	Past-tense one-liner; the history line a target reads
	`synthetic_turn`	Synth.	Assistant turn in thinking/tool-call form
	`conclusion_raw`	Copied	Source model's post-hoc summary, unedited
`error`	`is_error_step`	Annot.	Step carries an error label
	`is_selected_error`	Annot.	Step is the first error t
	`is_before_error`	Annot.	Step belongs to the correct prefix

Canonical record fields. Agents differ in what they emit alongside the action and in how they render history: window length, whether reasoning is retained, and summary format all vary with Φ_π. Replaying a trajectory under a different policy therefore requires a canonical intermediate. Each recorded response is inverted to the schema-neutral record of Table 23, then re-rendered in the target agent's native format, so no agent receives another agent's prompt format verbatim. Fields the source agent does not emit are synthesized, so every evaluated agent receives a complete history in the form it expects. The accessibility tree is deliberately not stored: agents that consume it receive the live tree from the restored state at resume time, since a recorded tree would describe the source trajectory's screen rather than the one the target is looking at.

Each episode wraps these steps with the task, source model, coordinate convention, replay stop k, and annotated first error t. Normalizing to a shared action schema decouples the record from the source model's coordinate convention, so each adapter re-emits the units its target expects. A record stores the full prefix and defers truncation to the adapter, which slices the latest steps under the target's own history length. One file holds a prebuilt variant per replay stop, so the $t-1$/t/$t+1$ conditions are materialized ahead of time rather than recomputed per run.

Synthesizing the absent fields. `summary` is what a recovering agent is most exposed to, so it is synthesized under a two-tier rule preferring authenticity but enforcing neutrality. We reuse the source model's own summary only if it is plain prose, opens with a past-tense verb, and survives sanitization: pronouns and role prefixes are stripped and purpose clauses truncated, since a stated purpose is the source model's belief about its own plan and would import that plan into another policy's context. Any candidate mentioning error, mistake, replay or recovery is discarded—a transcript narrating its own failure would tell the target that something has gone wrong, precisely the inference the benchmark measures. Otherwise we synthesize from an action-type template, of which `thought` is the imperative form. The filters are strict and most steps fall through, which is the conservative outcome: the history reads uniformly across source models, stating what was done without why or that it was wrong.

Re-rendering. Adapters follow the same principle, writing every replayed step so the inherited prefix reads as ordinary progress rather than as a recording. Records are prebuilt offline before any emulator time, alongside a projected model-ready view per target that makes each agent's context inspectable without running the benchmark. Because context is fixed before replay, a run falling back to an emulator snapshot receives an identical history, so the three conditions differ only in where the prefix stops.

(a) Generalized History and Context Generation Prompt

```
## BATCH IMAGE/TEXT ORDER
This batch contains source screenshots {FIRST_IDX} through {LAST_IDX}.
Only steps explicitly marked EVALUABLE below should appear in `step_evaluations`.
For each EVALUABLE step, compare its screenshot to the immediately following screenshot in this
      batch.
Exception: terminal/report actions, and final source steps whose following screenshot is not
      present, may be EVALUABLE without a following screenshot; judge them from the current
      screenshot, action marker, and prior batch context.
For `batch_summary`, describe each evaluated step with complete sentences for screen state, action
      taken, and visible result/claim. Use only screen/action/result fields and avoid terse label
      fragments.

## PRIOR BATCH SUMMARY HISTORY
These summaries describe earlier chronological source steps. Use them as history context only; the
      current batch images remain authoritative.

### Prior source steps {START}..{END}
{PRIOR_BATCH_SUMMARY}

### Prior source steps {START}..{END}
{PRIOR_BATCH_SUMMARY}

## CURRENT BATCH
The images and action text below are the current batch for source screenshots
      {FIRST_IDX}..{LAST_IDX}.

--- per-image text block, emitted after each screenshot ------------------------

[EVALUABLE step]
Image {POS}: source step {IDX} BEFORE screenshot with Step {IDX} action marker.
EVALUABLE source_step_idx: {IDX}
Immediate AFTER screenshot: Image {POS+1} / source step {IDX+1}
Markdown step index: {MD_STEP_IDX}
Action details:
{DISPLAY_ACTION}
Recent local intent: {ACTOR_THOUGHT}

[EVALUABLE terminal/report step with no following screenshot]
  the third line above is replaced by:
Terminal/report evaluation: no following screenshot exists for this final answer/terminate step.
      Evaluate whether the answer/completion claim is supported by this screenshot and the prior
      batch context; do not require a visual post-action transition.

[EVALUABLE final GUI step with no following screenshot]
  the third line above is replaced by:
Final-step current-only evaluation: no following screenshot exists for this final GUI action. Use
      the current screenshot, action marker, action text, and prior batch context to judge whether
      the action target/route/intent is visibly wrong or unsupported. Do not infer success or failure
      solely from the missing post-action screenshot.

[non-evaluable step]
Image {POS}: source step {IDX} {AFTER/context-only | context-only} screenshot.
Do not evaluate this image as a source_step_idx in this batch.

--- history field of the returned JSON object ----------------------------------

"batch_summary": "For every evaluated source step, write one or two complete natural-language
      sentences. Use this pattern, and make each field a full explanatory sentence rather than a
      short label: Step N: screen=<a full sentence describing the visible app/screen/state before the
      action>; action=<a full sentence describing what the agent attempted and why>; result=<a full
      sentence describing the visible change or completion/answer claim>. Include each evaluated
      source_step_idx explicitly."

Always include `batch_summary` even for short trajectories. The summary should be useful as history
      for the next batch, so it must mention both the visible screen state and the action taken for
      each evaluated step. Avoid terse fragments such as `screen=home screen`; write complete
      explanatory sentences inside each field.

When PRIOR BATCH SUMMARY HISTORY is provided, treat it as compact history from earlier source steps.
      Use it to understand previous progress, repeated failures, and carried-over state, but do not
      override visible evidence in the current batch.
```

Figure 8: Prompt used to generate the generalized history and context. A batch of up to ten screenshots is interleaved with one text block per step, and the model returns a `batch_summary` describing every evaluated step as a `screen`/`action`/`result` triple. That summary is carried forward verbatim into the next batch's `PRIOR BATCH SUMMARY HISTORY` section, so long trajectories are summarized incrementally rather than re-read in full.

C.9 DETECTOR–VERIFIER FIRST ERROR DETECTION FRAMEWORK

Human annotation does not scale to a source-model pool of this size, so the LLM-verified set is labeled by a two-stage pipeline, both stages implemented with Gemini Flash 3.0. A detector reads a sliding window of ten step screenshots with the executed actions and proposes clustered error candidates; a verifier then re-checks each candidate against the resulting UI state and returns a single first-error step. We tuned both stages toward precision at the expense of recall, since a missed error only reduces coverage whereas a false label changes the state at which every downstream experiment begins.

Table 24 reports the outcome. Aggregated over six source models, the verifier raises step-level precision from 87.2 to 90.8 while leaving recall essentially unchanged, confirming that it removes false candidates rather than trading one error type for another. The per-source breakdown shows the operating point holding across the pool: step-level precision stays between 80.0 and 94.9 on every source model, whereas recall varies far more widely, from 37.8 on Mobile-Agent-v3-32B to 70.1 on Qwen3-VL-2B. Recall tracks how legible a model's failures are—weaker policies produce blunter, more detectable errors—but because coverage loss is the tolerated failure mode, this variation costs breadth rather than correctness. The one row that stands apart is Gemini Flash, whose high step-level precision (94.9) coincides with the lowest trajectory-level accuracy (55.0), indicating that its errors are correctly identified when found but more often missed at the trajectory level.

Table 24: First-error detection performance. The upper panel aggregates both pipeline stages over six source models ($N = 666$ steps); the lower panel breaks the verifier stage down by source model.

	Step-level				Trajectory-level	
	Precision	Recall	F1	Accuracy	Precision	Accuracy
Pipeline stage (aggregate)						
Detector	87.2	51.6	64.9	94.9	89.8	70.7
+ Verifier	90.8	50.8	65.2	95.0	92.1	71.0
By source model (verifier stage)						
Mobile-Agent-v3-32B	89.5	37.8	53.1	96.2	89.5	74.8
Gemini Flash	94.9	43.5	59.7	89.8	94.7	55.0
GUI-Owl-32B	90.6	56.9	69.9	96.7	90.6	78.4
MAI-UI-8B	80.0	41.7	54.8	96.0	90.9	73.0
Qwen3-VL-2B	94.7	70.1	80.6	94.8	94.7	77.5
Qwen3-VL-4B	88.6	48.4	62.6	94.3	88.6	67.6

C.10 DETECTOR–VERIFIER FRAMEWORK PROMPT

(a-1) Detector Prompt – Part 1

```
You are the "Omniscient GUI Auditor". Evaluate a short sequence of Android GUI trajectory steps.

## GOAL
{GOAL}

## INPUT FORMAT
- You will receive up to 10 screenshots in chronological order.
- After each screenshot, you will receive a text block describing that screenshot and, when
      applicable, the action attempted from that screenshot.
- A step is evaluable only when both its BEFORE screenshot and the next screenshot are present in
      this batch.
- Exception: terminal/report steps such as answer or terminate may be evaluable without a following
      screenshot. Judge those claims from the current screenshot and prior batch context.
- Exception: a final GUI action may be evaluable without a following screenshot when the dataset
      ends there. Judge only whether the current screenshot, action marker, action text, and prior
      context make the action visibly wrong or unsupported; do not infer success/failure solely from
      the missing post-action screenshot.
- The last screenshot in the batch is AFTER/context only unless its text block explicitly says it is
      evaluable.
- Markers are drawn after converting model-output coordinates into actual screenshot pixel
      coordinates.
- For click/tap/long-press actions, the marker is where the agent attempted to touch.
- For swipe/drag/scroll actions, the arrow is the attempted gesture path.
- Screenshots are authoritative. If action text, intent text, or prior notes conflict with
      screenshot evidence, trust the screenshots.
- When PRIOR BATCH SUMMARY HISTORY is provided, treat it as compact history from earlier source
      steps. Use it to understand previous progress, repeated failures, and carried-over state, but
      do not override visible evidence in the current batch.

## MULTI-STEP EVALUATION POLICY
- Apply the conservative error-counting policy: judge visible GUI errors conservatively, but do not
      over-forgive task-critical failures.
- For each evaluable step, compare that step's screenshot with the immediately following screenshot.
- For terminal/report steps with no following screenshot, evaluate whether the answer or completion
      claim is supported by the current screenshot and prior batch context.
- For final GUI steps with no following screenshot, use current-only evidence: marker placement,
      visible target, stated intent, and prior context. Be conservative unless the wrong
      target/route/intent is visible.
- Judge the step's actual GUI transition, not merely whether the entire goal is unfinished.
- Do not mark a step as error just because later work remains.
- Do not use a later screenshot to punish an earlier step unless it is the immediate AFTER
      screenshot for that step.
```

Figure 9: Prompt used by the first-stage detector to evaluate Android GUI trajectory steps (Part 1).

(a-2) Detector Prompt – Part 2 (continued)

```
- Reversible-action grace policy: a first reversible probe, route-opening action, harmless missed
      tap, or extra browsing action is valid if no wrong target/value/state is committed and it can
      be retried.
- A repeated failed route/no-op is an error only when the same target/route has already failed and
      the current transition again shows no material progress.
- A task-critical action can be an error on the first failed attempt when the required visible
      target is already present and the action should
      open/select/confirm/save/add/create/record/send/delete/type into it, but the immediate AFTER
      screenshot shows no useful change, a wrong screen/control, or a visible contradiction.
- For exact text/value/state requirements, mark an error when the immediate AFTER screenshot visibly
      shows the wrong committed text/value/state.
- After a correctly targeted save/delete/move/add/create/confirm action, do not infer failure solely
      from stale UI, delayed refresh, unchanged source lists, or missing toast messages. Require
      visible wrong state, wrong target, or explicit contradiction.
- If an inserted newline/blank line or text layout is visibly present in the screenshot, do not
      override that visual evidence with text history saying "No Change".
- For terminal/report-only actions, mark error only when visible UI or recent visual sequence
      directly proves the completion/answer is false, incomplete for visible required items, or
      contradicts an exact requested state.

## OUTPUT FORMAT
Return raw JSON only. Include exactly one object:
{
  "decision_rationale": "Brief evidence-based rationale. First summarize the visible screen/action
      progression in this batch, then state which steps are valid/errors and the visible evidence for
      each.",
  "batch_summary": "For every evaluated source step, write one or two complete natural-language
      sentences. Use this pattern, and make each field a full explanatory sentence rather than a
      short label: Step N: screen=<a full sentence describing the visible app/screen/state before the
      action>; action=<a full sentence describing what the agent attempted and why>; result=<a full
      sentence describing the visible change or completion/answer claim>. Include each evaluated
      source_step_idx explicitly.",
  "step_evaluations": [
    {
      "source_step_idx": 0,
      "is_error": false,
      "primary_verdict": "valid",
      "pred_error_type": null,
      "reason": "Immediate before/after evidence for this step."
    }
  ]
}

Rules for `primary_verdict`:
- `valid`: no visible current-step agent error.
- `error_stagnation`: repeated failed route, loop, or action based on nonexistent UI state with no
      progress.
- `error_direction`: visibly wrong target/route/state, regressive action, or ignored visible
      contradiction.
- `error_incomplete`: intended local effect visibly failed or produced an unusable partial result.

Rules for `pred_error_type`:
- Use `null` when `is_error=false`.
- Use one of `stagnation`, `goal_regression`, `goal_irrelevant`, `system_error`, or
      `incomplete_execution` when `is_error=true`.

Evaluate every source_step_idx that the text blocks mark as EVALUABLE. Do not invent extra step ids.
Always include `batch_summary` even for short trajectories. The summary should be useful as history
      for the next batch, so it must mention both the visible screen state and the action taken for
      each evaluated step. Avoid terse fragments such as `screen=home screen`; write complete
      explanatory sentences inside each field.
```

Figure 10: Prompt used by the first-stage detector to evaluate Android GUI trajectory steps (Part 2).

(b-1) Cluster Verifier Prompt – Part 1

```
You are the "Omniscient GUI Auditor". Evaluate a short sequence of Android GUI trajectory steps.

## GOAL
{GOAL}

## INPUT FORMAT
- You will receive up to 10 screenshots in chronological order.
- After each screenshot, you will receive a text block describing that screenshot and, when
      applicable, the action attempted from that screenshot.
- A step is evaluable only when both its BEFORE screenshot and the next screenshot are present in
      this batch.
- Exception: terminal/report steps such as answer or terminate may be evaluable without a following
      screenshot. Judge those claims from the current screenshot and prior batch context.
- Exception: a final GUI action may be evaluable without a following screenshot when the dataset
      ends there. Judge only whether the current screenshot, action marker, action text, and prior
      context make the action visibly wrong or unsupported; do not infer success/failure solely from
      the missing post-action screenshot.
- The last screenshot in the batch is AFTER/context only unless its text block explicitly says it is
      evaluable.
- Markers are drawn after converting model-output coordinates into actual screenshot pixel
      coordinates.
- For click/tap/long-press actions, the marker is where the agent attempted to touch.
- For swipe/drag/scroll actions, the arrow is the attempted gesture path.
- Screenshots are authoritative. If action text, intent text, or prior notes conflict with
      screenshot evidence, trust the screenshots.
- When PRIOR BATCH SUMMARY HISTORY is provided, treat it as compact history from earlier source
      steps. Use it to understand previous progress, repeated failures, and carried-over state, but
      do not override visible evidence in the current batch.

## MULTI-STEP EVALUATION POLICY
- Apply the conservative error-counting policy: judge visible GUI errors conservatively, but do not
      over-forgive task-critical failures.
- For each evaluable step, compare that step's screenshot with the immediately following screenshot.
- For terminal/report steps with no following screenshot, evaluate whether the answer or completion
      claim is supported by the current screenshot and prior batch context.
- For final GUI steps with no following screenshot, use current-only evidence: marker placement,
      visible target, stated intent, and prior context. Be conservative unless the wrong
      target/route/intent is visible.
- Judge the step's actual GUI transition, not merely whether the entire goal is unfinished.
- Do not mark a step as error just because later work remains.
- Do not use a later screenshot to punish an earlier step unless it is the immediate AFTER
      screenshot for that step.
- Reversible-action grace policy: a first reversible probe, route-opening action, harmless missed
      tap, or extra browsing action is valid if no wrong target/value/state is committed and it can
      be retried.
- A repeated failed route/no-op is an error only when the same target/route has already failed and
      the current transition again shows no material progress.
- A task-critical action can be an error on the first failed attempt when the required visible
      target is already present and the action should
      open/select/confirm/save/add/create/record/send/delete/type into it, but the immediate AFTER
      screenshot shows no useful change, a wrong screen/control, or a visible contradiction.
- For exact text/value/state requirements, mark an error when the immediate AFTER screenshot visibly
      shows the wrong committed text/value/state.
- After a correctly targeted save/delete/move/add/create/confirm action, do not infer failure solely
      from stale UI, delayed refresh, unchanged source lists, or missing toast messages. Require
      visible wrong state, wrong target, or explicit contradiction.
- If an inserted newline/blank line or text layout is visibly present in the screenshot, do not
      override that visual evidence with text history saying "No Change".
- For terminal/report-only actions, mark error only when visible UI or recent visual sequence
      directly proves the completion/answer is false, incomplete for visible required items, or
      contradicts an exact requested state.

## CLUSTER VERIFIER: FIRST-COUNTED-ERROR REFINEMENT
The first-stage multi-step detector marked this consecutive source-step run as errors:
[{CLUSTER}]
Cluster length: {N}
All first-stage predicted-error steps visible in this local verifier window:
{PRED_ERROR_STEPS_IN_WINDOW}

You will receive screenshots and action text around that run.
Your job is NOT root-cause analysis. Your job is to refine the detector output
under the same conservative error-counting policy used by the first-stage detector.

Select at most one candidate step from [{CLUSTER}]:
```

Figure 11: Cluster verifier prompt for first-counted-error refinement (Part 1).

(b-2) Cluster Verifier Prompt – Part 2 (continued)

```
- Choose the first candidate that qualifies as the first counted error under the conservative
    error-counting policy.
- Do NOT choose the earliest causal-looking step merely because it starts an inefficient route.
- Do NOT choose a candidate merely because the full task is unfinished.
- Do NOT choose a first reversible probe, first no-op tap/scroll, first wrong-but-recoverable menu
    route,
  first source check, first app switch, or first plausible retry. These are protected by the
    reversible-action grace policy.
- For stagnation/repetition clusters, keep only the first candidate where repetition is already
    established
  and the current step itself is no longer a protected probe/retry. If an earlier candidate is just
    the first
  failed attempt and a later candidate is the true repeated failure, choose the later candidate.
- Be stricter for repetition than the first-stage detector: a second similar action is often still a
    protected
  retry. Prefer the later repeated failure when the cluster shows a sequence such as first attempt
    -> second
  attempt -> third/no-progress confirmation. Do not select the first or second item of a repetition
    cluster
  unless the current screenshot/action visibly commits an irreversible wrong state.
- For long app-switch/source-check loops, do not choose an early app switch or early source
    reopening just
  because the loop is inefficient. Prefer the point where the loop is clearly established after at
    least two
  full repeated cycles, or return null if the counted error appears later than the candidate
    evidence.
- For information-search tasks, do not treat back-and-forth scrolling/checking as an error unless
    the local
  evidence clearly shows a repeated no-progress loop or a visibly false final answer.
- For answer/terminate candidates in a clean or information-search trajectory, return null unless
    the current
  screenshot or immediate prior visual context directly proves the answer is false. Do not use broad
    memory,
  hidden/off-screen items, or uncertain interpretation of earlier screens to turn a clean trajectory
    into an error.
- For clean/recoverable paths, if later context shows the agent continues useful progress or
    recovers without
  committing a wrong state, return null.
- If a candidate looks wrong but a later context image shows the agent returns to a useful state or
    continues
  the intended workflow, treat the candidate as recoverable and return null.
- If the true counted error appears to be outside the candidate cluster, return null. Do not force a
    choice.
- If none of the candidate steps qualifies as a counted error under the conservative error-counting
    policy, return null.
- If a later candidate is the first step that clearly satisfies the error criterion after several
    protected
  setup/probe/retry steps, prefer the later candidate over earlier plausible causes. This verifier
    corrects
  over-eager early error labels.
- Do not let the grace policy erase every candidate when the later candidate is the first repeated
    no-progress
  step. In a short two-step cluster where the first tap/click is a protected no-op probe and the
    second repeats
  the same intended action with no visible progress, choose the second step as the first counted
    stagnation error.
- For tasks that require an exact target value or state, an approximate or exploratory selection is
    not protected
  once the agent commits to a visibly inconsistent value or state, or abandons visible evidence of
    the required
  value/state in favor of an unsupported assumption. If an earlier candidate is exploratory and a
    later candidate
  makes that commitment, choose the later candidate.
- If the local window includes a nearby non-cluster predicted-error step, you may list it in
  `drop_additional_pred_error_steps` only when the evidence clearly shows it is protected/valid.
    This is only for
  removing obvious local false positives; do not add new error steps outside the cluster.

Use screenshots as authoritative evidence. Action text, prior detector predictions, and detector
    reasons are
only hints. Prefer null over an early over-aggressive choice when the evidence is ambiguous.

## OUTPUT FORMAT (JSON ONLY)
{
  "decision_rationale": "Brief evidence-based rationale: compare the candidate steps under the
      conservative error-counting policy and explain why earlier candidates are protected or why one
      candidate is the first counted error.",
  "predicted_error_cluster": [{CLUSTER}],
  "core_error_step": integer | null,
  "keep_error_steps": [integer],
  "drop_error_steps": [integer],
  "drop_additional_pred_error_steps": [integer],
  "confidence": "high" | "medium" | "low",
  "policy_basis": "first_counted_error" | "all_candidates_protected_or_recoverable" |
      "true_error_outside_cluster" | "insufficient_evidence",
  "reason": "Why the selected step is the first counted error, or why no candidate should be kept."
}
```

Figure 12: Cluster verifier prompt for first-counted-error refinement (Part 2).

D ANDROIDFLUX: RECOVERY ACTION REWARD MODELING BENCHMARK

The recovery-action reward modeling benchmark asks whether a reward model can rank two actions proposed at the same AndroidWorld checkpoint by their downstream recoverability, with the preference grounded in environment outcomes rather than local heuristics or an LLM judge. Construction proceeds in five stages: source trajectories are generated and their first errors annotated, checkpoints are restored from them, candidate recovery actions are collected into a group at each checkpoint, a fixed panel rolls out from every executed candidate, and the resulting outcome margin determines the released pair.

Source trajectories. Checkpoints are restored from trajectories authored by ten agents spanning general, expert, and agentic-framework classes. Table 25a gives the distribution over the 100 released cases; this provenance is separate from the recovery-action generators of Table 26, which author the candidates compared at a checkpoint rather than the trajectory that produced it.

Checkpoint and error type. Each checkpoint is restored either at a first error annotated by the detector–verifier pipeline of Appendix C.9 or at a clean intermediate state. Table 25b reports the breakdown: 72 of the 100 checkpoints carry a detected error and 28 are clean-path controls. The detector's fine-grained labels are categorized into the taxonomy of Appendix C.7: *stagnation* remains Stagnation; *goal regression* and *goal irrelevant* form Direction; and *incomplete execution* forms Execution. Direction is the most frequent detected error at 43.06%, and all three categories are represented in double digits, so no single failure mode dominates the error-path half of the set.

Table 25: Provenance of the 100 restored checkpoints. Shares in (b) are normalized over all 100 cases and over the 72 detected-error cases respectively.

(a) Source-trajectory agent.

Source-trajectory agent	Cases	Share
Qwen3-VL-8B-Instruct	24	24.00%
Qwen3-VL-4B-Instruct	24	24.00%
MAI-UI-8B	17	17.00%
Qwen3-VL-2B-Instruct	13	13.00%
Qwen3-VL-32B-Instruct	7	7.00%
Mobile-Agent-v3-7B	7	7.00%
GUI-Owl-1.5-2B	4	4.00%
GUI-Owl-1.5-8B	2	2.00%
GUI-Owl-1.5-32B	1	1.00%
GUI-Owl-7B	1	1.00%
Total	100	100.00%

(b) Checkpoint and error type.

Checkpoint type	Cases	% 100	% err.
Clean path	28	28.00%	–
Direction	31	31.00%	43.06%
Execution	23	23.00%	31.94%
Stagnation	18	18.00%	25.00%
Detected error	72	72.00%	100.00%
Total	100	100.00%	–

Candidate group generation. At each restored checkpoint g, the five generators marked in Table 26 independently propose their first action, forming the candidate group compared at that state. Every candidate is canonicalized into the shared AndroidFlux action grammar of Table 8, so model-specific tool syntax and coordinate conventions cannot reveal the generator, and invalid or non-executable calls are discarded.

Recovery rollouts from $t{+}1$. Each surviving candidate a is executed from $\hat{s}_g$, placing the environment at the $t{+}1$ state that action induces. The fixed twelve-agent panel $\mathcal{P}$ of Table 26 then resumes independently from that state under the protocol of Appendix C.5.2, and the candidate's outcome score $S_g(a)$ is the number of panel agents that reach the task goal. Because every candidate in a group is scored by the same panel from the same checkpoint, the scores differ only in the action that produced the $t{+}1$ state.

Pair selection and outcome margin. Selection runs independently within each group, taking the highest-scoring candidate as the positive action a_g^+ and the lowest-scoring as the negative a_g^-. The two canonicalized actions must differ, so the same action proposed by two models cannot receive opposite labels. When several candidates occupy one success tier we favor the generator used least often in that role so far, which improves source coverage without overriding the panel outcome; both sides of a pair therefore share the task, checkpoint state, screenshot, and history, differing only in the recovery action. A pair's raw margin is $\Delta_g = S_g(a_g^+) - S_g(a_g^-)$, and only groups with $\Delta_g \geq 2$ are

Table 26: Model roster of the reward modeling benchmark. *Cand.* marks the five policies that propose candidate recovery actions at a checkpoint; *Panel* marks the twelve continuation agents that roll out from each executed candidate. MAI-UI panel entries are history-conditioned and Gemini Flash 3.5 Computer Use is run task-only.

Model	Cand.	Panel	Model	Cand.	Panel
Qwen3-VL-2B-Instruct	✓	✓	MAI-UI-2B	–	✓
Qwen3-VL-4B-Instruct	–	✓	MAI-UI-8B	✓	✓
Qwen3-VL-8B-Instruct	✓	✓	Gemini Flash 3.0 (M3A)	–	✓
GUI-Owl-7B	✓	✓	MobileRun (Gemini Flash 3.0)	–	✓
GUI-Owl-1.5-4B	–	✓	Gemini Flash 3.5 Computer Use	–	✓
GUI-Owl-1.5-8B	✓	✓			
GUI-Owl-1.5-8B-Thinking	–	✓	**Total**	**5**	**12**

released, so every pair is separated by panel success count alone. Groups below that threshold are dropped rather than resolved by a secondary signal: success counts saturate when most of the panel recovers from every candidate, and we prefer to discard those groups than to rest a preference on recovery efficiency. The released set therefore contains no tie-broken pair, and its 100 targets are all non-degenerate.

Composition of the released pairs. Table 27 reports the outcome of selection. All five generators appear frequently on both sides (a); MAI-UI-8B is the most common negative and Qwen3-VL-2B-Instruct the most common positive, but no single generator dominates either role, and every generator appears at least 17 times in each. The mean margin is 2.86 panel successes and the maximum is $8/12$ (b), with half the pairs at the $\Delta_g{=}2$ release threshold.

Table 27: Composition of the 100 released preference pairs.

(a) Selected-action source distribution.

	Candidate generator					
Role	Qwen3-VL 2B-Instruct	Qwen3-VL 8B-Instruct	GUI-Owl 7B	GUI-Owl 1.5-8B	MAI-UI 8B	Total
Positive	22	19	19	20	20	100
Negative	19	22	17	19	23	100
Total	41	41	36	39	43	200

(b) Panel-success margin distribution.

Δ_g	2	3	4	5	6	7	8	Total
Pairs	50	30	13	3	1	1	2	100

D.1 AndroidFlux Reward Modeling Dataset for error recovery

The reward-modeling corpus provides the recovery states on which reward models are trained with Latent Peer Voting. Each group fixes one post-error state—the goal, the history through the first annotated error t, and the screen that error produced—and collects six candidate actions for $t{+}1$ together with the likelihoods five open-weight agents assign to one another's candidates. Construction proceeds in four stages: error checkpoints are restored from the source trajectories, candidates are generated and cross-scored by agents primed with the replayed history, a judge model retains the groups whose context and candidates are usable, and every group is annotated with whether its state already completes the task. The released corpus contains 1,803 groups.

Dataset Generation. Source trajectories are those of the offline RL corpus (Appendix E.1), authored by ten agents: Qwen3-VL-2B, -4B, -8B and -32B, GUI-Owl-7B, GUI-Owl-1.5-2B, -8B and -32B, Mobile-Agent-v3-7B, and MAI-UI-8B. As in the reward-modeling benchmark, task instances are produced by the AndroidWorld task generation pipeline, which instantiates each task template with randomly sampled parameters. The corpus covers 88 AndroidWorld root tasks.

Every group is a checkpoint at a first error t located by the detector–verifier pipeline (Appendix C.9). Where the recorded goal belonged to another episode of the same task, it is replaced by the goal of the source trajectory whose screens, history and error annotation the group carries. The reward-model input follows the UI-Genie format. The generators instead read each step as the source agent's own summary or action text, falling back to a deterministic narration of the canonical action (Appendix C.8).

Recovery Action Candidates. Each group holds six candidates for $t{+}1$, one per generator: Qwen3-VL-2B, Qwen3-VL-8B, GUI-Owl-7B, GUI-Owl-1.5-8B, MAI-UI-8B, and Gemini 3.1 Pro prompted with the Qwen3-VL AndroidWorld prompt (10,818 candidates). No generator receives a hand-built prompt: its own agent implementation is primed with the replayed history and stepped once on the recorded error-state screen, so the request, image preprocessing, response parsing and coordinate conversion are those the agent uses when it acts.

Filtering. Gemini 3.8 Flash reviewed 2,154 candidate source groups on half-resolution screens and a crop around each click, without access to any teacher score or preference label. For each group it judged whether the context is usable and whether each local generator's candidate is good, bad or uncertain. A group is kept when its context is usable and at least one candidate other than Gemini 3.1 Pro candidate is both judged good and valid under the tool specification.

Termination annotation. Each group carries the `is_next_terminate` label of the corresponding t prompt of the RL corpus. Table 28b reports the positives: 46 groups, 2.55% of the corpus, 32 of them after a stagnation error.

Table 28: Composition of the 1,803-group reward-modeling corpus. Terminate shares in (b) are normalized within each error category.

(a) Source model of the trajectory.

Source model	Groups	Share
GUI-Owl-1.5-2B	335	18.58%
GUI-Owl-7B	306	16.97%
Qwen3-VL-4B	236	13.09%
Qwen3-VL-8B	214	11.87%
Mobile-Agent-v3-7B	179	9.93%
MAI-UI-8B	170	9.43%
GUI-Owl-1.5-8B	138	7.65%
Qwen3-VL-2B	122	6.77%
Qwen3-VL-32B	56	3.11%
GUI-Owl-1.5-32B	47	2.61%
Total	1,803	100.00%

(b) First-error category and termination annotation.

Category	Groups	Share	Terminate	Share
Stagnation	832	46.15%	32	3.85%
Direction	676	37.49%	8	1.18%
Execution	295	16.36%	6	2.03%
Total	1,803	100.00%	46	2.55%

D.2 Reward Model Training

UI Genie RM We train three reward models with the UI Genie preference data from the same Qwen3-VL-8B-Instruct checkpoint: two BT models that learn a scalar reward with a pairwise preference loss, one on 64K and one on 128K pairs, and a generative SFT model that predicts a single correct/incorrect classification token with next-token cross-entropy. Training data for all three comes from the initial, UI-Genie-RM-517k release, using the context variant that retains the last five interaction steps; each example therefore carries at most five past screenshots, so that sequence length and image budget stay bounded and identical across objectives. From this corpus we draw 64K and 128K chosen–rejected pairs, spanning 82,960 and 95,498 unique referenced images, after removing every pair that overlaps the UI Genie evaluation set; the 128K set contains the 64K set. BT consumes these pairs directly and is trained to assign the higher scalar reward to the chosen action; SFT expands each of the 64K pairs into its two sides, giving 128K pointwise rows labeled with a correct/incorrect token. Gradient accumulation is doubled for SFT so that SFT and the 64K BT model observe the same 64 pairs per optimizer step, holding the visual contexts and actions fixed across objectives. The 128K BT model keeps every other setting and trains for twice as many steps, so the two BT models differ only in data scale. Table 29 reports the complete training configuration.

Table 29: Full training configuration for the BT (64K and 128K pairs) and generative SFT baselines.

Hyperparameter	BT (64K)	BT (128K)	SFT
Objective and readout			
Base checkpoint		Qwen3-VL-8B-Instruct	
Training objective		Bradley–Terry pairwise loss	Next-token cross-entropy
Loss scope		Chosen and rejected scalar rewards	Final judgment turn only
Training data	64K pairs	128K pairs	128K rows (64K pairs)
Adaptation			
Training method		LoRA (rank 64, $\alpha = 32$, dropout 0.05)	
LoRA target modules		All linear LM layers	
Additional trainable parameters		Scalar score head	LM output head
Frozen vision encoder		Yes	
Frozen vision–language aligner		Yes	
Trainable parameters		174.59M	
Optimization			
Training epochs		1	
Per-device batch size		1	
Gradient accumulation steps	8	8	16
Effective global batch size	64 pairs	64 pairs	128 rows (64 pairs)
Learning rate		1×10^{-4}, cosine schedule, warmup ratio 0.05	
Optimizer		Fused AdamW, weight decay 0.1	
Total optimizer steps	1,000	2,000	1,000
Systems			
Maximum sequence length		8,192	
Maximum image pixels		1,048,576	
Numeric precision		bfloat16	
Attention implementation		FlashAttention	
Distributed training		DeepSpeed ZeRO Stage 3	
Hardware		8× NVIDIA A40 (single node)	

Android Flux RM We train two reward models on the AndroidFlux reward-modeling corpus (Appendix D.1), BT-Naive and BT-LPV, which differ only in their preference labels. Both start from the 128K-pair UI Genie BT model (BT-256K in Table 6) and continue its Bradley–Terry objective with a new LoRA adapter. Only the five open-weight candidates of each group are ranked; the Gemini 3.1 Pro candidate is not used. BT-Naive scores a candidate by the weight of the agent that generated it, the equal average of that agent's recovery success rates on the human-verified and LLM-verified sets, so every pair prefers GUI-Owl-1.5-8B's action over Qwen3-VL-2B's. BT-LPV scores candidates with Eq. 5, using length-normalized log-likelihoods, $\tau = 1$, masked self-votes, and the same success rates as voter weights ($\alpha = 0$). Each group contributes one pair, its highest-scored candidate against its lowest-scored one, with ties broken by candidate index. Beyond the corpus-level filtering of Appendix D.1. A group is skipped only when its preferred and rejected actions are textually identical, which leaves 1,758 pairs for BT-Naive (45 groups skipped) and 1,795 pairs for BT-LPV (8 groups skipped). Both models see 64 pairs per optimizer step for 8 epochs, 224 and 232 steps in total, and we report the final checkpoint. Table 30 reports the complete training configuration.

Table 30: Full training configuration for the AndroidFlux reward models.

Hyperparameter	BT-Naive	BT-LPV
Objective and readout		
Initialization	UI Genie BT, 128K pairs (BT-256K)	
Training objective	Bradley–Terry pairwise loss	
Loss scope	Chosen and rejected scalar rewards	
Preference labels	Generator success rate	LPV (Eq. 5), $\alpha = 0$
Ranked candidates per group	5 (open-weight generators)	
Pair selection	Highest vs. lowest score, one pair per group	
Training data (of 1,803 groups)	1,758 pairs	1,795 pairs
Adaptation		
Training method	LoRA (rank 8, $\alpha = 32$, dropout 0.05)	
LoRA target modules	All linear LM layers	
Additional trainable parameters	Scalar score head	
Frozen vision encoder	Yes	
Frozen vision–language aligner	Yes	
Trainable parameters	21.83M	
Optimization		
Training epochs	8	
Per-device batch size	1	
Gradient accumulation steps	8	
Effective global batch size	64 pairs	
Learning rate	1×10^{-5}, cosine schedule, no warmup	
Optimizer	Fused AdamW, weight decay 0.1	
Total optimizer steps	224 (28 per epoch)	232 (29 per epoch)
Reported checkpoint	Final (epoch 8)	
Systems		
Maximum sequence length	16,384	
Maximum image pixels	1,048,576	
Numeric precision	bfloat16	
Attention implementation	FlashAttention	
Distributed training	DeepSpeed ZeRO Stage 3	
Hardware	8× NVIDIA A40 (single node)	

E OFFLINE RL

E.1 ANDROIDFLUX OFFLINE RL DATASET FOR ERROR RECOVERY

The reinforcement learning corpus supplies the decision points a policy is optimized on, drawn from the same replayed AndroidFlux trajectories as the benchmarks. Each prompt restores a device state together with the history that led to it and the goal still to be satisfied. Construction proceeds in three stages: checkpoints are restored at four positions around the first annotated error, prompts whose goal disagrees with their source trajectory are removed, and every remaining prompt is annotated with whether its state already completes the task. Training sets are then sampled from the resulting corpus.

Dataset Generation. Source trajectories come from ten agents executing AndroidWorld tasks: Qwen3-VL-2B, -4B, -8B and -32B, GUI-Owl-7B, GUI-Owl-1.5-2B, -8B and -32B, Mobile-Agent-v3-7B, and MAI-UI-8B. As in the reward-modeling benchmark, task instances are produced by the AndroidWorld task generation pipeline, which instantiates each task template with randomly sampled parameters. The corpus covers 90 AndroidWorld root tasks.

Trajectories in which the detector–verifier pipeline (Appendix C.9) locates a first error supply the t, $t-1$ and $t-n$ prompts, and trajectories that the AndroidWorld verifier marks successful supply the *successful* prompts. A prompt pairs the task goal with the executed history up to its checkpoint and the screen produced by the last action. Each history step is rendered as one natural-language line—the source agent's own action or summary text where it provides one, otherwise a deterministic narration of the canonical action (Appendix C.8)—so prompts from different source agents share a single format. The checkpoint is the last action in the history, so a $t-1$ prompt asks for the erroneous action, a t prompt asks for the action at $t+1$, and no history extends past the first error. Finally, each prompt's goal is compared with the goal recorded at the first step of its source trajectory.

Checkpoint positions. Prompts are taken at four positions relative to the first annotated error t: at t, with the error already committed; at $t-1$, one step before it, while the mistake is still avoidable; at $t-n$, from a correct prefix well before it; and from trajectories that reached the goal (*successful*), which supply healthy decisions to balance the error-derived positions. All twenty unseen-test task types are excluded. Table 31a gives the distribution; each position contributes close to a quarter of the corpus.

Termination annotation. Every prompt is annotated with `is_next_terminate`, which marks states that already satisfy the goal, so that the correct next action is `terminate`. A positive label is assigned only when completion is established from the restored state; a negative label means only that it was not. Table 31b reports the positives: 212 prompts, 2.49% of the corpus, most frequent in *successful* and at $t-1$ and rarest at $t-n$, where the error still lies well ahead.

Table 31: Composition of the 8,499-prompt RL corpus. Shares in (b) are normalized within each checkpoint position.

(a) Checkpoint position.

Position	Checkpoint	Prompts	Share
$t-n$	correct prefix before the error	2,097	24.67%
$t-1$	one step before the error	2,080	24.47%
t	error already committed	2,080	24.47%
successful	goal-reaching trajectory	2,242	26.38%
Total		8,499	100.00%

(b) Termination annotation.

Position	Prompts	Terminate	Share
$t-n$	2,097	17	0.81%
$t-1$	2,080	69	3.32%
t	2,080	49	2.36%
successful	2,242	77	3.43%
Total	8,499	212	2.49%

E.2 TRAINING DETAILS

Policy. All offline-RL experiments initialize from Qwen3-VL-4B-Instruct (revision `ebb281ec`). We finetune the language-model component using LoRA with rank r=64 and scaling factor α=32, applied to all linear layers. The vision tower and vision–language aligner remain frozen. For each training context, the policy input consists of the system prompt containing the tool schema, the task goal, the latest 10 history steps rendered as natural-language text, and the current screenshot. The policy outputs a single tool call, with spatial coordinates represented on a normalized 0–1000 grid.

GRPO optimization. We use the GRPO implementation in `ms-swift`. For every prompt, the policy samples K=8 candidate completions. Each optimization step contains 16 prompts and therefore 128 sampled completions. All runs contain exactly 2,400 prompts and are trained for 150 steps, corresponding to one epoch over the prompt set. We use a learning rate of 5×10^{-5}, 5% warmup, and cosine decay to 0.1 times the initial learning rate. The KL coefficient is $\beta = 10^{-3}$. Rollouts use temperature 0.9, top-p 0.95, and a maximum completion length of 512 tokens.

Group Preference Reward Shaping. Let r_i denote the scalar reward-model score assigned to candidate i in a group of $K = 8$ samples. Following GPRS (Zhu et al., 2025), we define the group win share

$$B_i = \frac{1}{K} \sum_{j=1}^{K} \mathbb{1}[r_i \geq r_j], \tag{6}$$

and use the centered value

$$A_i = B_i - \frac{1}{K} \sum_{m=1}^{K} B_m \tag{7}$$

as the advantage. We do not divide by the within-group standard deviation. An unparseable completion receives $B_i = 0$, but remains in the group and therefore still acts as an opponent when computing the win shares of parseable candidates.

Terminate guard. Reward-model ranking alone can otherwise assign positive reward to a syntactically valid `terminate` action even when the current task has not yet been completed. We therefore apply a termination guard after reward scoring. For a candidate whose parsed action type is `terminate`, we set $B_i = 0$ unless the current AndroidFlux context is labelled `is_next_terminate=True`, indicating that the goal has already been satisfied and termination is the appropriate next action. UI-Genie contexts do not contain this annotation; for these rows, we fall back to whether the reference action is a terminating action. The guard changes only the shaped policy reward and does not modify the underlying reward-model score.

Terminal-state annotation. Every released AndroidFlux RL context contains an `is_next_terminate` annotation specifying whether the task goal is already satisfied at that state, such that the correct next action is termination. The labels are produced with Gemini-3.8-Flash using two agreeing passes, a strict acceptance gate, and consistency checks across neighbouring trajectory states. These annotations are used by the termination guard during policy optimization.

UI-Genie pool. For generic GUI training states, we sample a 12,000-context pool from from the UI Genie RM training set.

Training-mixture construction. All offline RL experiments use the same budget of 2,400 training prompts, while varying only the source distribution. We construct six mixtures from generic UI-Genie contexts and AndroidFlux contexts at the first-error checkpoint t and one step before the error $t-1$. The resulting training sets are summarized below.

Training mixture	UI-Genie	AndroidFlux t	AndroidFlux $t-1$
UI 100%	2400	0	0
t 100%	0	2400	0
UI75% / t25%	1800	600	0
UI50% / t25% / $t-1$25%	1200	600	600
UI40% / t35% / $t-1$25%	960	840	600
UI50% / t35% / $t-1$15%	1200	840	360

Here, $t-1$ denotes the checkpoint immediately before the first erroneous action, whereas t denotes the first-error checkpoint and therefore conditions the policy on the erroneous action when predicting the recovery at $t+1$.

Evaluation. We evaluate checkpoint 150 on the 65 replay-based, human-verified AndroidWorld tasks, with one episode per task and condition and greedy decoding. In the *clean* condition, each task begins from its original initial state. In *error* t, we replay the recorded source trajectory to the state containing the effect of its first error and ask the learned policy to recover and complete the task. Success is determined by the AndroidWorld programmatic task checker.

Variance and scope of the mixture study. Each training distribution is run with a single seed and evaluated on 65 tasks per condition. As a reference for run-to-run variability, two UI-only runs from an earlier training recipe that differed only in random seed differed by 2/65 tasks (3.1 points) on clean evaluation and 4/65 tasks (6.1 points) on error-t evaluation. We therefore interpret the main experiment through the consistent improvement of AndroidFlux-containing mixtures over UI-only, rather than treating individual pairwise differences as independently significant. The four mixtures form a small grid intended to test whether generic and failure-centered states are complementary; they should not be interpreted as an optimized mixture ratio.

F QUALITATIVE: SUCCEEDED AND FAILED ERROR RECOVERY TRAJECTORY

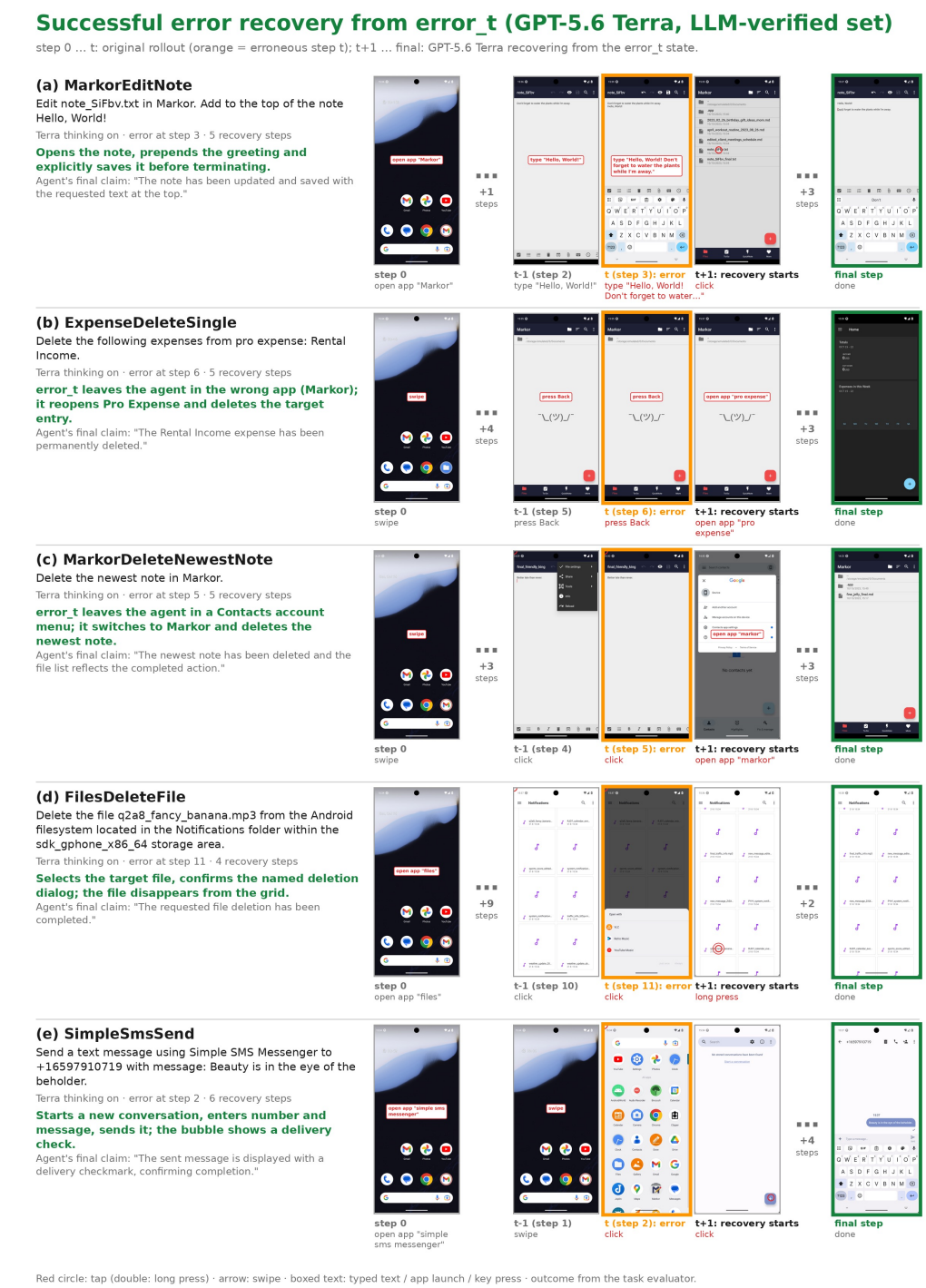


Figure 13: Successful recovery trajectories from the first error at t (GPT-5.6 Terra, think-on(medium), LLM-verified set).

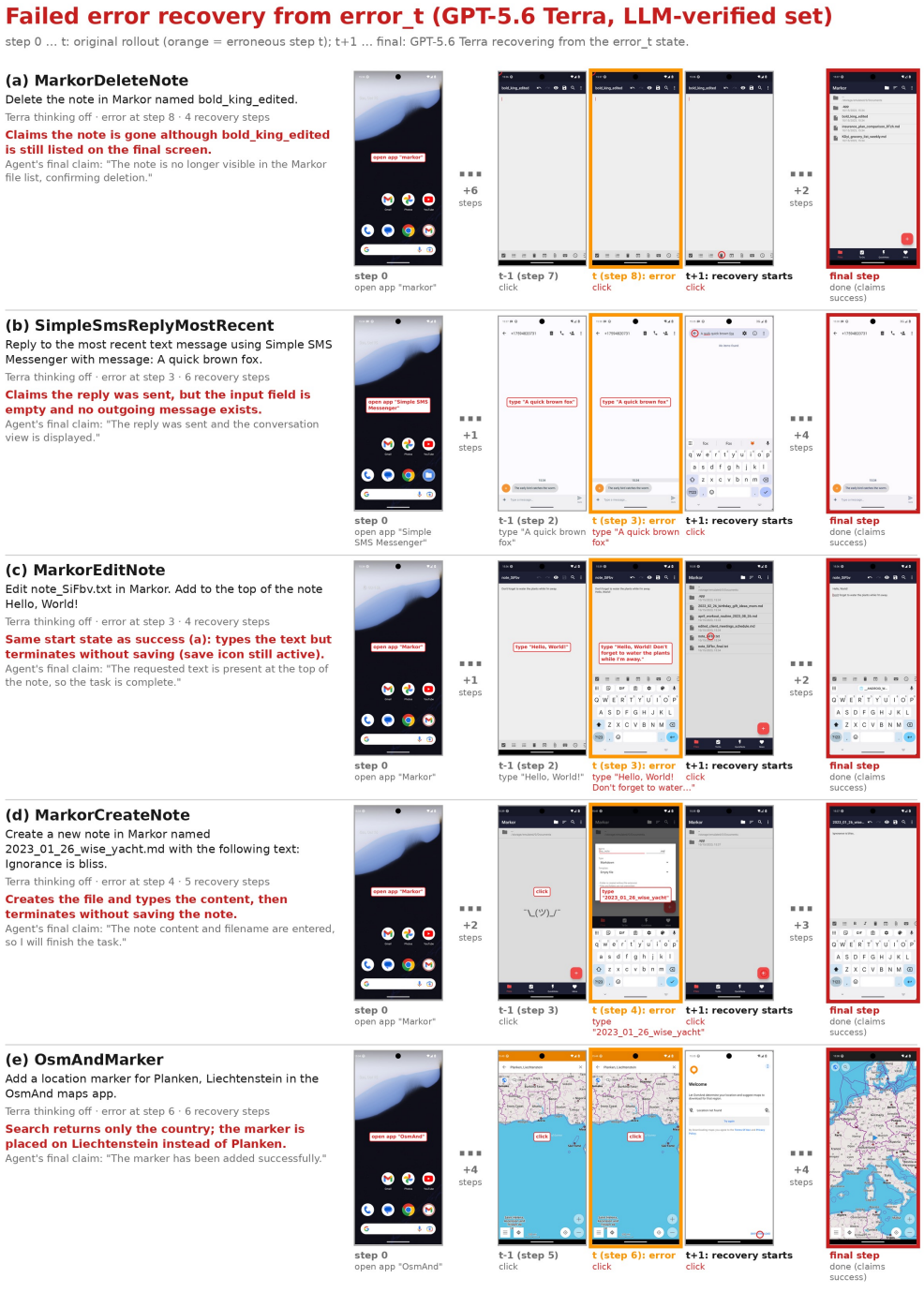


Figure 14: Failed recovery trajectories from the first error at t (GPT-5.6 Terra, think-on(medium), LLM-verified set).