How GUI Agents Fail: GraceBench for Execution Stability Diagnosis

Anonymous authors
Paper under double-blind review

Abstract

As GUI agents are increasingly deployed for mobile task automation, their reliability depends not only on final task outcomes, but also on whether they reach these outcomes through stable interaction trajectories. Whether in daily automation scenarios or testing workflows, unnecessary actions, detours, and execution errors can add noise to collected data, introduce irreversible side effects, or undermine trust, even when the agent eventually completes the task. It is therefore important to systematically diagnose the execution stability, including whether an agent deviates from a feasible path, enters a loop, becomes stagnant, or recovers from unstable actions. Existing benchmarks, however, primarily assess final task success or a small set of intermediate milestones, leaving step-level execution behavior unclear. To bridge this gap, we introduce GraceBench, an online evaluation framework that represents the reachable interaction space of mobile applications as UI Transition Graphs (UTGs), providing a structural reference for step-level execution diagnosis. To support reliable alignment between dynamic UI observations and graph states, GraceBench uses video-grounded trajectory evaluation over this graph-aligned representation, assessing process stability beyond final success by measuring metrics such as deviation rate, loop burden rate, and recovery ability. GraceBench contains 169 tasks spanning 10 applications and constructs 10 UTGs with 2,909 canonical GUI states and 3,116 observed transitions. Across 3,718 trajectories, GraceBench shows that outcome scores hide execution instability: GPT-5.6 Sol achieves the highest success rate (SR) but the lowest recovery ability, 51.24% of successful runs still contain a deviation or loop, and trajectories with the same success rate (SR) and completion rate (CR) can differ substantially in recovery behavior. Code will be made publicly available.

1 Introduction

Graphical user interface (GUI) agents are increasingly expected to operate real software environments on behalf of users (Zhang et al., 2025), from completing routine mobile tasks (Rawles et al., 2025) to supporting software-testing data collection workflows. In such settings, an agent's capability is not determined solely by whether it eventually reaches a target state; the execution process itself is equally important. This is especially true in GUI-based automated testing, where system performance, logs, resource usage, and behavioral signals are often collected along the execution trajectory. If an agent gets off track, repeats actions, or fails to recover, these path-dependent measurements become noisy and difficult to interpret, reproduce, and trust (Dong et al., 2021). Execution stability is therefore a key requirement for reliable GUI-agent deployment in real interactive environments.

Existing GUI-agent benchmarks primarily evaluate task completion, usually by checking whether the final state satisfies a goal condition (Rawles et al., 2025). Although terminal success is useful, it offers little visibility into the stability of the execution process. Some recent work introduces intermediate milestones, such as essential states or subtasks, to measure staged progress (Chai et al., 2026a; Chen et al., 2025), but such evaluation remains coarse-grained: it implicitly treats tasks as mostly linear and overlooks the effect of each action in a complex interactive environment. By checking only predefined semantic milestones, milestone-based evaluation cannot assess execution stability between them. As shown in Figure 1, existing evaluation methods cannot distinguish whether a successful agent follows a stable path, deviates and recovers, or falls into a repeated loop

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.

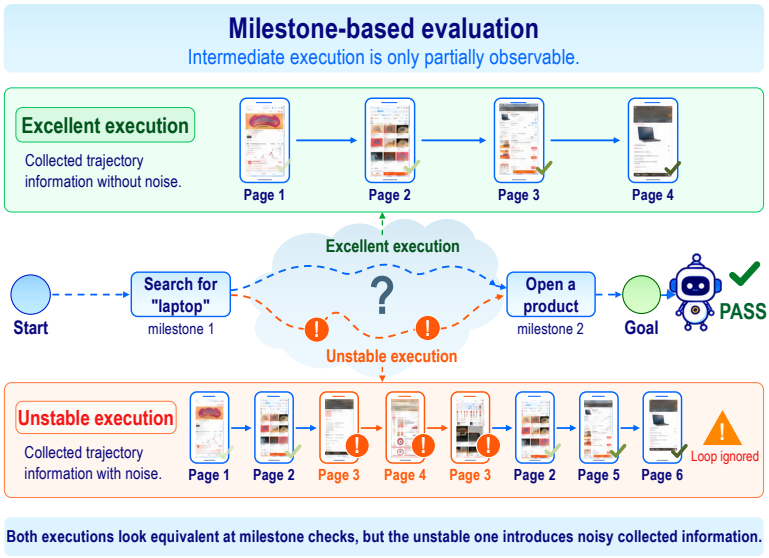

Figure 1: Limitation of existing milestone-based process evaluation. A clean execution and an unstable execution with an intermediate loop can satisfy the same semantic milestones and therefore receive the same outcome. Observing only milestone attainment hides the loop, recovery behavior, and noise introduced along the execution trajectory.

before completion. Without step-level diagnostic signals, it is difficult to systematically understand execution instability in real GUI interactions.

To address this gap, we introduce GRACEBENCH, a benchmark for diagnosing the execution stability of GUI agents in real mobile applications. GraceBench uses UTGs to represent the finite reachable interaction space of an application as a graph. In a UTG, nodes correspond to GUI states and edges correspond to executable action transitions between states (Yang et al., 2015; Wen et al., 2024). This graph serves as a coordinate system for evaluating execution: nodes indicate where the agent is, edges indicate where the agent is moving, and the surrounding graph reveals what alternative interactions are available. GraceBench defines each task on a task-specific UTG subgraph and maps the recorded execution trajectory back onto this graph for step-level diagnosis. Because dynamic mobile UIs can change during and after an action, GraceBench uses synchronized video evidence to align observations with graph states and diagnose what happened between consecutive decisions.

Based on this UTG representation, GraceBench evaluates GUI agents beyond final success. It contains 169 tasks across 10 applications and multiple domains, built on 10 UTGs with 2,909 canonical GUI states; each task's matched states and their one-hop neighborhood cover 41.09 canonical GUI states on average, providing a local structural reference for step-level diagnosis. Each task is paired with concise and detailed instruction variants, allowing us to evaluate stability under both autonomous path planning and step-level execution guidance. Beyond success and completion rates, GraceBench reports deviation rate, loop burden rate, and recovery ability to evaluate the execution stability of GUI agents.

We evaluate eleven models, including two GUI-specific models and nine general-purpose models, on all 169 tasks under two instruction conditions, yielding 3,718 trajectories. Instability drives GUI-agent failures and persists even behind successful outcomes. Across these trajectories, agents often fail through path deviation and looping, while 51.24% of successful trajectories still contain a deviation or loop. Even GPT-5.6 Sol, which achieves the highest success rate, has the lowest recovery ability on its observed recovery opportunities. Outcome-controlled comparisons further show that trajectories with the same success and completion outcomes can differ substantially in recovery behavior, indicating that final success alone can overestimate GUI-agent reliability.

Table 1: Comparison with representative GUI-agent benchmarks. *Venue* denotes the representative publication venue or preprint year; other columns summarize support for process evaluation, online execution, multi-path tasks, graph references, stability metrics, and video evidence.

Benchmark	Venue	Process Eval	Online Env.	Multi-Path	Graph Ref.	Stability Metrics	Video Ev.
AndroidWorld (Rawles et al., 2025)	ICLR 2025	✗	✓	✗	✗	✗	✗
SPA-Bench (Chen et al., 2025)	ICLR 2025	✓	✓	✗	✗	✗	✗
A3 (Chai et al., 2026a)	Findings ACL 2026	✓	✓	✓	✗	✗	✗
AndroidDaily (Sui et al., 2026)	arXiv 2026	✓	✓	✓	✗	✗	✗
ProBench (Yang et al., 2025)	AAAI 2026	✓	✓	✗	✗	✗	✗
ColorBench (Song et al., 2025)	WWW 2026	✓	✗	✓	✓	✗	✗
MobiFlow (Feng et al., 2026)	arXiv 2026	✓	✗	✓	✓	✗	✗
DAC-Bench (Zhang et al., 2026)	ACL 2026	✓	✗	✓	✗	✗	✗
GUI-RobustEval (Bu et al., 2026)	ICML 2026	✓	✓	✗	✗	✓	✗
GRACEBENCH	This work	✓	✓	✓	✓	✓	✓

Our contributions are summarized as follows:

- We introduce GRACEBENCH, a benchmark specifically designed to evaluate the execution stability of GUI agents in real mobile application environments.
- We propose a video-grounded online evaluation framework that maps recorded agent trajectories onto UI transition graphs, enabling step-level diagnosis of deviation, looping, and recovery ability.
- We systematically evaluate multiple types of GUI agents and identify shared instability patterns that are difficult to observe from final success alone, providing evidence for future diagnosis and optimization of GUI-agent capabilities.

2 RELATED WORK

2.1 MOBILE GUI AGENT BENCHMARKS

Early GUI agent benchmarks mainly rely on offline evaluation, where predicted actions are matched against a single golden trajectory step by step (Rawles et al., 2023; Li et al., 2024). While scalable, such protocols reduce GUI control to trajectory imitation, ignoring multiple valid paths and the agent's ability to recover from deviations. Recent online benchmarks, represented by AndroidWorld (Rawles et al., 2025), allow agents to interact with executable environments and evaluate task completion from the final UI state. However, final-success metrics remain coarse: they reveal whether a task is completed, but not how reliably it is executed. Process-aware benchmarks further introduce intermediate supervision. A3 (Chai et al., 2026a) evaluates the completion ratio of essential semantic states, SPA-Bench (Chen et al., 2025) verifies predefined app-level subtasks, and ProBench (Yang et al., 2025) captures process information to check key-step completion. These works move beyond outcome-only evaluation, but their signals are still mostly defined over milestones or semantic checkpoints. Consequently, they provide limited visibility into failures occurring between milestones, and cannot distinguish merely successful executions from stable executions. Our work focuses on this missing layer by evaluating GUI agents at the granularity of execution states and action transitions.

2.2 GRAPH-STRUCTURED REPRESENTATIONS FOR GUI AGENTS

Graph structures have been widely used to model GUI interaction as transitions over discrete interface states. Existing works use such representations for app exploration and knowledge construction (Sun et al., 2025; Chai et al., 2026b), task generation and pretraining (Ye et al., 2025; Wu et al., 2024), and runtime planning or recovery (Yu et al., 2026; Guo et al., 2025; Zhong et al., 2026). These methods demonstrate that graphs provide a natural abstraction for multi-step, multi-path GUI interaction. In GUI-agent evaluation, ColorBench (Song et al., 2025) traverses UI states to construct an application-level graph, while MobiFlow (Feng et al., 2026) builds a more abstract graph through trajectory fusion and state compression. Both works use graph structures to make offline evaluation closer to real interaction. However, they still overlook how dynamic changes in real environments, such as changing content and UI components, affect agent execution, whereas GraceBench aligns

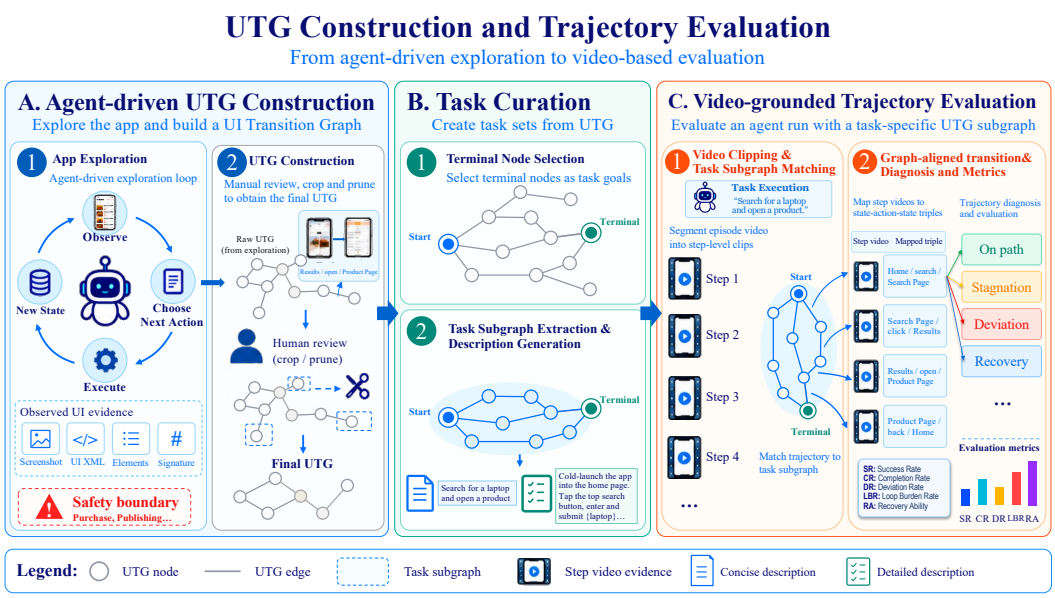

Figure 2: Overview of GraceBench task curation and evaluation. GraceBench first builds application-level UTGs through agent-driven exploration, curates task-specific subgraphs and instructions, and then evaluates online agent trajectories by mapping step-level video evidence to graph-aligned execution labels and stability metrics.

live execution traces with UI transition graphs to diagnose step-level deviation, stagnation, recovery, and execution stability.

3 GRACEBENCH

To operationalize the graph-based diagnosis motivated above, GraceBench uses task-specific UTGs as a shared reference throughout benchmark construction and evaluation. We first build application-level UTGs and curate 169 long-horizon tasks across 10 multi-functional mobile applications, each with concise and detailed instructions. Agents then execute these tasks on physical devices while GraceBench records synchronized actions, observations, and video evidence.

Figure 2 provides an overview of the resulting GraceBench pipeline: panel (a) illustrates agent-driven UTG construction, panel (b) summarizes task curation, and panel (c) shows video-grounded trajectory evaluation. We describe these components in order below.

3.1 TASK CURATION

UTG construction. For each application, we construct an application-level UTG as the structural basis for task curation and trajectory evaluation. We denote an application-level UTG as $G = (V, E)$, where V is the set of task-relevant GUI states and E is the set of directed action transitions between states. Each node $v \in V$ corresponds to a visually verifiable GUI state, and each edge $e = (v_i, a, v_j)$ indicates that executing action a at state v_i leads to state v_j.

We use an agent-assisted exploration loop to collect UI transitions. Before exploration, we set the application scope and treat high-risk operations such as payment, order submission, and authorization changes as boundary states. At each step, the agent observes the current screen through ADB, including both the screenshot and UI hierarchy, identifies interactable elements, selects a candidate action, and executes the action through ADB (Android Developers, 2026). Each observed interface transition is recorded as a state-action-state edge in the UTG. Detailed implementation of the automatic exploration loop is provided in Appendix D.

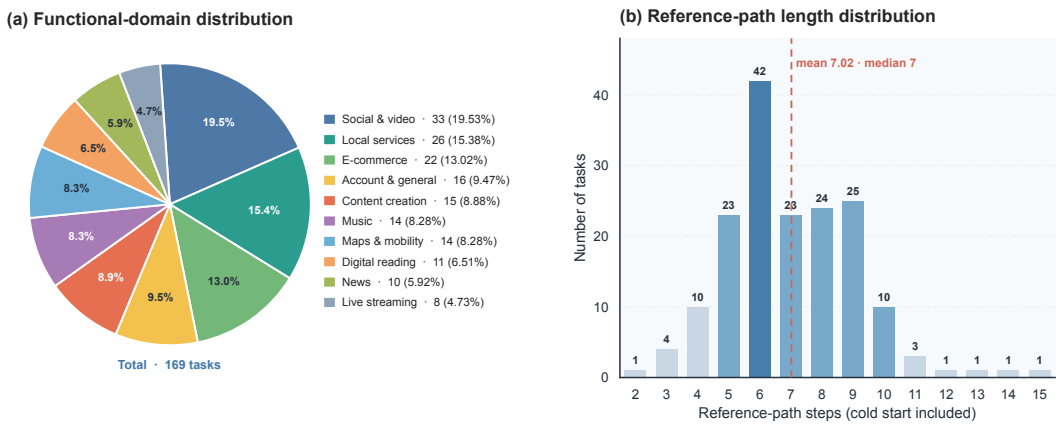

Figure 3: Distribution of the 169 GraceBench tasks. **(a)** Functional domains across 10 applications. **(b)** Reference-path length distribution.

After exploration, we clean and normalize the collected graph by deduplicating equivalent UI states, filtering unstable textual attributes, and saving a representative screenshot for each retained state. Across 10 UTGs, GraceBench normalizes 3,265 collected state observations into 2,909 canonical GUI states, together with their corresponding observed transitions. Appendix D.1 reports the per-application inventory and counting rules. All UTGs used in GraceBench are manually reviewed before task extraction to ensure state validity, transition executability, and coverage of major functional pathways.

Task-subgraph extraction. From the manually reviewed UTGs, we choose visually verifiable terminal states with stable completion semantics as task goals, excluding shallow entry or intermediate-panel states. For each task τ, we define a natural initial state v_τ^{start} and target state v_τ^{goal}, then extract a task-specific subgraph $G_\tau = (V_\tau, E_\tau) \subseteq G$ containing task-relevant states and executable transitions that make legal progress toward the goal. This subgraph includes alternative valid routes but excludes task-irrelevant detours, providing a multi-path structural reference for trajectory diagnosis and process-level metrics.

Instruction generation. For each task, GraceBench derives two instruction variants from the task subgraph: a *concise instruction* that states the user goal with necessary constraints, and a *detailed instruction* that follows one manually verified feasible path with step-level operations. This paired design varies instruction granularity while keeping the task goal fixed, separating planning ambiguity from low-level execution stability. We further parameterize query terms, categories, target entities, and selectors to make tasks extensible and reduce overfitting to fixed surface forms.

3.2 Online Execution Environment

Following ProBench (Yang et al., 2025), GraceBench evaluates agents on physical Android devices in an online loop. At each step, the agent receives the current screenshot, task instruction, and interaction history, and its textual action is parsed into ADB commands for execution until completion or budget exhaustion. Each episode is recorded as

$$T = (o_1, a_1, o_2, a_2, \ldots, o_T, a_T, o_{T+1}), \tag{1}$$

where (o_t, a_t, o_{t+1}) denotes the pre-action screen, executed action, and stabilized post-action screen used for step-level evaluation.

GraceBench uses the action space `CLICK`, `SWIPE`, `TYPE`, `ENTER`, `BACK`, `COMPLETE`, and `WAIT`; `HOME` is excluded because tasks stay within one application.

Video capture is integrated into the same execution loop. GraceBench starts a continuous screen recording before the first agent decision and stops it after the episode terminates. The recording

is implemented with `scrcpy` (Genymobile, 2026), while the runner logs the action type, action timestamp, pre-action screenshot, and post-action screenshot for each step. This ensures that the video evidence is synchronized with the agent's actual decision and device-action timeline.

Prior screencast-based GUI analysis uses video to preserve the temporal process from user operation to visual feedback (Liu et al., 2025). GraceBench instead uses video as temporal evidence for dynamic GUI effects such as loading, pop-ups, animations, and content changes, which motivates the graph-aligned diagnosis below.

3.3 Video-Grounded Trajectory Evaluation

GraceBench evaluates each execution trace by aligning observed agent behavior with the task subgraph. The evaluation pipeline contains four stages. First, the episode-level video is segmented into step-level clips based on action timestamps. These clips provide visual evidence for what happened before, during, and after each action. Second, GraceBench retrieves the task-specific subgraph from the full application UTG. The subgraph contains the task initial state, target state, required transitions, and alternative valid paths, serving as the reference structure for judging whether an observed action advances the task. Third, a video-based evaluator analyzes each local transition (o_t, a_t, o_{t+1}) using the corresponding step-level video clip and executed action. The evaluator maps the pre-action and post-action observations to graph states. For each observation, the matched graph state is defined as

$$\hat{v}_t = \begin{cases} v_i, & \text{if } o_t \text{ matches a state } v_i \in V, \\ v_{\text{out}}, & \text{otherwise,} \end{cases} \tag{2}$$

where v_{out} denotes a state outside the application-level UTG. This mapping converts each local transition into a graph-aligned transition:

$$(o_t, a_t, o_{t+1}) \mapsto (\hat{v}_t, a_t, \hat{v}_{t+1}). \tag{3}$$

Based on the aligned transition, the evaluator assigns one of four step-level labels: *on path*, *stagnation*, *deviation*, or *recovery*. A step is labeled *on path* if it follows a valid transition in G_τ, *stagnation* if it causes no meaningful state change, *deviation* if it leaves the task subgraph or enters v_{out}, and *recovery* if it returns from an off-path state to a valid state or transition in G_τ. The resulting labeled sequence is the graph-aligned diagnostic trajectory:

$$T' = [(\hat{v}_1, a_1, \hat{v}_2, \ell_1), \ldots, (\hat{v}_T, a_T, \hat{v}_{T+1}, \ell_T)]. \tag{4}$$

GraceBench uses T' to identify trajectory-level behaviors such as missing required transitions, persistent deviation, repeated no-progress actions, loops, successful recovery after deviation, and how unmatched v_{out} states should be behaviorally labeled. The step-level diagnostic labels are first generated by expert agents and then validated on a preregistered, stratified 10% sample against blinded human expert labels (Appendix E). The audit estimates agreement between generated labels and human expert judgments on sampled transitions; it does not replace every generated label in the full dataset. Figure 4 illustrates how the graph alignment localizes deviation, recovery, and loop events and connects them to the corresponding process metrics.

3.4 Evaluation Metrics

Task evaluation metrics. At the task level, we report task success and graph-aligned execution quality. (1) ***SR*** measures the fraction of trajectories marked successful in the per-trajectory evaluation summaries; these labels are preserved in all reported analyses. (2) ***CR*** measures how much of the task-required subgraph has been completed, providing a finer-grained view of progress when the final task fails. (3) ***Deviation rate (DR)*** measures how often the agent moves outside the task subgraph, capturing path instability. (4) ***Loop burden rate (LBR)*** measures how much execution is consumed by repeated no-progress behavior once looping occurs. (5) ***Recovery success rate (RSR)*** measures whether the agent can return to the task subgraph after a deviation. (6) ***Average recovery latency (ARL)*** measures how many steps successful recovery takes. (7) ***Recovery ability (RA)*** combines recovery success and recovery speed into a single robustness score. Detailed calculation rules are provided in Appendix B.

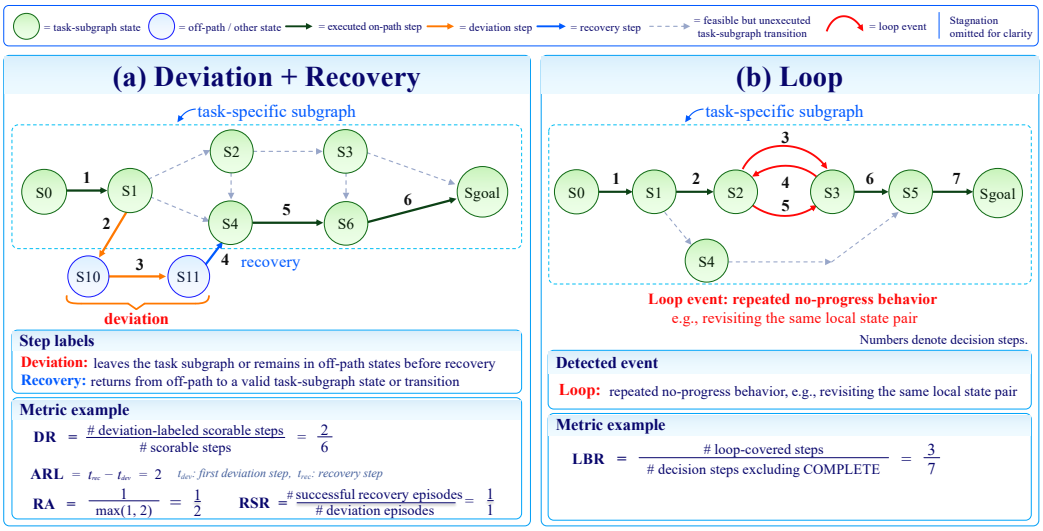

Figure 4: Trajectory-diagnosis examples on task-specific UTGs. **(a)** Steps 2–3 leave the task subgraph and step 4 recovers, providing the labels used to compute DR, ARL, RA, and RSR. **(b)** Repeated transitions between the same local state pair form a loop event and determine LBR. Dashed edges denote feasible but unexecuted task-subgraph transitions; stagnation is omitted for clarity.

4 Experiments

Our experiments address three questions: *How do GUI agents fail? Do task outcomes adequately characterize execution stability?* Finally, *Do stability metrics capture reproducible agent differences across tasks?* After the setup, each subsection answers one question using outcome-stratified statistics, outcome-controlled comparisons, or cross-task replication. Table 2 summarizes the main experimental results across agents, instruction conditions, and stability metrics.

4.1 Experimental Setup

Baseline agents. We evaluate representative screenshot-only GUI agents, including GUI-specific models and general-purpose models from the DeepSeek, Gemini, GPT, Qwen, and GLM families, using each model's official prompting and tool-calling setup when available. Each model is tested under detailed subgraph-derived instructions and concise user-level goals, with temperature set to 0 and a maximum budget of 15 actions. Table 2 contains 3,718 trajectories from eleven completed models, two instruction conditions, and 169 tasks. All trajectory-level analyses use this full eleven-model dataset; paired instruction analyses contain 1,859 model–task pairs. The two GUI-specific models use 8-bit `MLX` checkpoints (Apple, 2026) and were collected on a separate host; their evaluated records are included in every applicable analysis. In addition to SR and CR, we report DR, LBR, RSR, ARL, and RA to evaluate GUI-agent execution stability; detailed definitions and aggregation rules are provided in Appendix B.

4.2 GUI Agents Fail Through Unstable Execution

Failure combines incomplete progress with unstable execution. Table 2 shows that GPT-5.6 Sol and GLM-5.3-Flash are the strongest overall agents: GPT-5.6 Sol attains the highest SR in both instruction settings (0.6391), while GLM-5.3-Flash achieves the lowest DR in both settings (0.0743 detailed, 0.1303 concise) and the highest concise CR (0.6676). Yet strong outcomes do not imply uniformly stable execution. Even GPT-5.6 Sol has the lowest reported RA on its observed recovery opportunities under both instruction settings, showing that high final success can coexist with weak recovery after mistakes. Other agents expose different failure profiles: UI-TARS-1.5-

Table 2: Results for all 3,718 trajectories: eleven models, 169 tasks, and two instruction conditions. All rows use the supplied per-trajectory evaluation summaries, preserving the reported SR labels and process scores. Data provenance is detailed in Appendix C. Rates are decimals; ARL is measured in steps. Within each instruction block, best values are bolded and second-best values are underlined.

Detailed instruction

Type	Model	SR↑	CR↑	DR↓	LBR↓	RA↑	RSR↑	ARL↓
GUI-specific	GUI-Owl-1.5-8B-Instruct	0.1657	0.4403	0.3306	0.5559	<u>0.6603</u>	**0.8212**	1.6877
	UI-TARS-1.5-7B	0.1006	0.2956	0.5149	0.5133	0.6017	0.7888	1.9501
General open-source	DeepSeek-V4.1-Flash	0.5148	0.6099	0.1348	0.5934	0.5602	0.6987	1.8716
	GLM-5.3-Flash	<u>0.5976</u>	<u>0.7304</u>	**0.0743**	<u>0.4470</u>	0.5605	0.6857	1.4583
	Qwen3.8-Flash	0.3254	0.4474	0.1591	0.6606	0.5834	0.7819	1.7755
	Qwen3.8-Max	0.2722	0.4389	0.2111	0.6945	0.6017	0.7979	1.8400
	Qwen3.8-27B	0.2367	0.4044	0.2286	0.7412	0.5703	0.6976	1.5780
Proprietary	GPT-5.6 Sol	**0.6391**	**0.7388**	<u>0.1223</u>	**0.3870**	0.4454	0.5645	1.7143
	GPT-5.5	0.5207	0.6151	0.1855	0.4502	0.5061	0.5928	<u>1.4141</u>
	Gemini 3.5 Flash	0.3373	0.5428	0.1930	0.5278	**0.6988**	<u>0.8106</u>	**1.3689**
	Gemini 3.1 Flash Lite	0.0710	0.1799	0.3961	0.7277	0.5238	0.7269	2.2597

Concise instruction

Type	Model	SR↑	CR↑	DR↓	LBR↓	RA↑	RSR↑	ARL↓
GUI-specific	GUI-Owl-1.5-8B-Instruct	0.2367	0.4393	0.3807	0.5041	0.6440	<u>0.8296</u>	1.9366
	UI-TARS-1.5-7B	0.1538	0.3643	0.5088	<u>0.4572</u>	0.6013	0.8000	1.9740
General open-source	DeepSeek-V4.1-Flash	0.4142	0.5163	0.2128	0.6107	0.6020	0.8201	1.9388
	GLM-5.3-Flash	<u>0.6272</u>	**0.6676**	**0.1303**	**0.4289**	0.5560	0.7070	1.7027
	Qwen3.8-Flash	0.3787	0.5436	0.2142	0.5508	**0.7193**	**0.8581**	<u>1.5269</u>
	Qwen3.8-Max	0.3077	0.4035	0.2154	0.7103	<u>0.6867</u>	0.8087	**1.4523**
	Qwen3.8-27B	0.1893	0.3360	0.2907	0.7532	0.5410	0.6835	1.6158
Proprietary	GPT-5.6 Sol	**0.6391**	<u>0.6261</u>	<u>0.1856</u>	0.4746	0.4525	0.5806	1.8111
	GPT-5.5	0.4970	0.5224	0.2753	0.5154	0.4881	0.5971	1.8374
	Gemini 3.5 Flash	0.3077	0.5030	0.2462	0.5201	0.6654	0.8035	1.5971
	Gemini 3.1 Flash Lite	0.1243	0.1902	0.4650	0.7190	0.4565	0.6150	2.2734

7B is dominated by deviation (DR 0.5149 detailed, 0.5088 concise), whereas Qwen3.8-27B shows especially high loop burden (LBR 0.7412 detailed, 0.7532 concise).

Failed executions are less stable throughout the trajectory. Across all 3,718 trajectories, failures have higher DR (0.3067 vs. 0.0889 detailed; 0.3618 vs. 0.1412 concise), higher loop incidence (0.6525 vs. 0.2066; 0.6520 vs. 0.2748), higher LBR after looping begins (0.6390 vs. 0.4257; 0.6142 vs. 0.4840), and lower RA (0.5668 vs. 0.7285; 0.5918 vs. 0.6394). Thus task failure is closely associated with sustained off-path and repeated no-progress behavior; the full outcome-stratified comparison is reported in Appendix Table 4.

4.3 Stability Metrics Reveal Differences Beyond Task Outcomes

Successful outcomes still contain unstable execution. Although failed trajectories are less stable on average, instability is not confined to unsuccessful runs. Among the 1,294 successful trajectories, 51.24% (663/1,294) contain at least one deviation or loop. Under detailed instructions, 36.78% of successful trajectories contain deviations and 20.66% contain loops; under concise instructions, the corresponding rates increase to 46.11% and 27.48%. Figure 5(a) separates deviation-only, loop-only, and combined patterns within these successful trajectories. Moreover, 126 successful trajectories have zero DR but still contain a loop, showing that even an on-path execution can waste actions through repeated no-progress behavior. These results show that successful outcomes can still mask

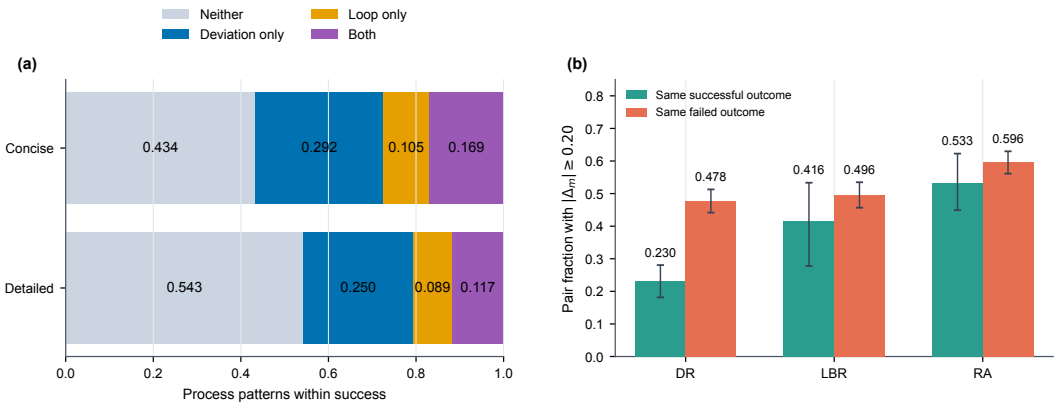

Figure 5: **Task outcomes hide execution instability.** (a) Mutually exclusive process-pattern proportions among successful trajectories. (b) Exact-CR matched pairs with process gap at least 0.2000; bars distinguish successful and failed pairs, and whiskers show task-cluster 95% intervals.

deviations and loops, so final success alone does not establish stable task execution; the outcome relevance of timely recovery after deviation is examined separately in Appendix C.4.

Equivalent task performance masks substantial stability differences. Success is only the coarsest form of outcome equivalence. We therefore compare models executing the same task under the same instruction condition, with identical SR and equal stored CR. Among eligible successful pairs, DR, LBR, and RA differ by at least 0.2000 in fractions 0.2305, 0.4158, and 0.5331, respectively. Figure 5(b) shows that these large process gaps also appear in failed pairs, with RA differing by at least 0.2000 in 0.5956 of exact-CR matched failed pairs. Matching sensitivity and partially overlapping metric correlations provide additional support for this distinction (Appendix Figure 8). Thus even matched task outcomes underdetermine both the degree and the form of execution instability.

4.4 Stability Metrics Capture Consistent Agent Differences

Model behavior profiles persist across tasks. We test whether process metrics capture reproducible model behavior by repeatedly splitting the 169 tasks into stratified halves, ranking all eleven models by each metric on each half, and measuring rank agreement with Spearman ρ. The strongest consistency appears for DR and LBR: detailed-instruction rankings reach $\rho = 0.9516$ and 0.9284, while concise-instruction rankings remain high at $\rho = 0.9228$ and 0.8381. RSR is also stable across splits ($\rho = 0.8100$ detailed; 0.8269 concise), whereas ARL is weaker ($\rho \approx 0.58$) and RA is condition-dependent. All metric–condition correlations remain significant after Bonferroni correction ($p < 0.05$); the full table and permutation details are reported in Appendix Table 9.

5 Conclusion

We introduced GraceBench, which aligns video-recorded trajectories with task-specific UI Transition Graphs to diagnose deviation, looping, and recovery during real mobile execution. Across eleven models and 3,718 trajectories, GraceBench reveals execution weaknesses hidden by SR and CR: lower-performing agents deviate and loop more often, while even the highest-SR model exhibits limited recovery on observed recovery opportunities. Successful trajectories, as well as trajectories with identical SR and CR, can still differ substantially in DR, LBR, and RA, showing that reliable GUI automation requires more than task completion alone. By providing graph-aligned evidence and process-level diagnostics, GraceBench supports future work on stronger state tracking, loop detection, and recovery policies for GUI agents.

AI USE STATEMENT

In this work, we used generative AI tools to assist benchmark construction, including drafting candidate task instructions, expanding parameterized task variants, and producing auxiliary transition annotations for GUI trajectory diagnosis. All AI-assisted outputs were treated as candidate artifacts rather than ground truth. The authors manually reviewed the retained UTGs, task definitions, task instructions, and screenshots to ensure state validity, transition executability, task feasibility, and consistency with the benchmark protocol. For step-level diagnostic labels, expert-agent outputs were not assumed to be fully human-labeled; instead, we conducted a preregistered, stratified audit in which sampled transition labels were validated against blinded human expert judgments, as described in Appendix E. The reported quantitative results are computed from the documented evaluation records and metric definitions, and generative AI tools were not used to make final scientific claims without author verification. The authors take full responsibility for the final content of this work, including text, claims, data, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

GraceBench is designed to diagnose GUI-agent execution stability in controlled mobile environments. The benchmark does not involve human-subject experiments. Because GUI agents may interact with real applications, we constrain tasks to avoid irreversible or high-risk operations such as payment, order submission, purchases, account changes, or authorization changes. During UTG construction and evaluation, such operations are treated as boundary states and are not executed. Screenshots, videos, and traces are collected from controlled devices and benchmark accounts, and released artifacts will be reviewed to remove or anonymize sensitive information where necessary. The benchmark is intended for reliability diagnosis and safety-oriented analysis rather than for enabling harmful automation.

REPRODUCIBILITY STATEMENT

To support reproducibility, we will release the benchmark task definitions, instruction variants, UTG metadata, evaluation code, metric definitions, and trace/evaluation schemas, with detailed metric calculation rules provided in the appendix. For each evaluated run, GraceBench stores action traces, pre-action and post-action screenshots, action metadata, timing information, and synchronized video evidence, and we document the evaluation environment so that trajectory judgments can be audited despite changes in mobile applications, app versions, accounts, devices, and online content. The current analysis package contains per-trajectory evaluation records for all eleven models; the two GUI-specific models were collected on a separate host, and their video and screenshot files are not included in this package.

REFERENCES

Android Developers. Android debug bridge (adb). https://developer.android.com/tools/adb, 2026. Accessed: 2026-09-26.

Apple. Mlx. https://github.com/ml-explore/mlx, 2026. Accessed: 2026-09-26.

Tianpeng Bu, Xin Liu, Qihua Chen, Hao Jiang, Shurui Li, Hongtao Duan, Lu Jiang, Lulu Hu, Bin Yang, and Minying Zhang. Recovering policy-induced errors: Benchmarking and trajectory synthesis for robust gui agents. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=iJwDylm93H. Spotlight; arXiv:2605.29447.

Yuxiang Chai, Shunye Tang, Han Xiao, Weifeng Lin, Hanhao Li, Jiayu Zhang, Liang Liu, Pengxiang Zhao, Guangyi Liu, Guozhi Wang, Shuai Ren, Rongduo Han, Haining Zhang, Siyuan Huang, and Hongsheng Li. A3: Android agent arena for mobile gui agents with essential-state procedural evaluation, 2026a. URL https://arxiv.org/abs/2501.01149. Findings of ACL 2026.

Yuxiang Chai, Han Xiao, Xinyu Fu, Jinpeng Chen, Rui Liu, and Hongsheng Li. Ui-kobe: Knowledge-oriented behavior exploration for lightweight graph-guided gui agents, 2026b. URL https://arxiv.org/abs/2605.29534.

Jingxuan Chen, Derek Yuen, Bin Xie, Yuhao Yang, Gongwei Chen, Zhihao Wu, Yixing Li, Xurui Zhou, Weiwen Liu, Shuai Wang, Kaiwen Zhou, Rui Shao, Liqiang Nie, Yasheng Wang, Jianye Hao, Jun Wang, and Kun Shao. Spa-bench: A comprehensive benchmark for smartphone agent evaluation, 2025. URL https://arxiv.org/abs/2410.15164. ICLR 2025.

Zhen Dong, Abhishek Tiwari, Xiao Liang Yu, and Abhik Roychoudhury. Flaky test detection in Android via event order exploration. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021. doi: 10.1145/3468264.3468584.

Yunfei Feng, Xi Zhao, Cheng Zhang, Dahu Feng, Daolin Cheng, Jianqi Yu, Yubin Xia, and Erhu Feng. Mobiflow: Real-world mobile agent benchmarking through trajectory fusion, 2026. URL https://arxiv.org/abs/2604.09587.

Genymobile. scrcpy. https://github.com/Genymobile/scrcpy, 2026. Accessed: 2026-09-26.

Linqiang Guo, Wei Liu, Yi Wen Heng, Tse-Hsun Chen, and Yang Wang. Agent-sama: State-aware mobile assistant, 2025. URL https://arxiv.org/abs/2505.23596.

Wei Li, William Bishop, Alice Li, Chris Rawles, Folawiyo Campbell-Ajala, Divya Tyamagundlu, and Oriana Riva. On the effects of data scale on ui control agents, 2024. URL https://arxiv.org/abs/2406.03679.

Wei Liu, Linqiang Guo, Yi Wen Heng, Chenglin Li, Tse-Hsun Chen, and Ahmed E. Hassan. Screencast-based analysis of user-perceived gui responsiveness, 2025. URL https://arxiv.org/abs/2508.01337.

Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. Android in the wild: A large-scale dataset for android device control, 2023. URL https://arxiv.org/abs/2307.10088.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy Lillicrap, and Oriana Riva. Androidworld: A dynamic benchmarking environment for autonomous agents, 2025. URL https://arxiv.org/abs/2405.14573. ICLR 2025.

Yuanyi Song, Heyuan Huang, Qiqiang Lin, Yin Zhao, Xiangmou Qu, Jun Wang, Xingyu Lou, Weiwen Liu, Zhuosheng Zhang, Jun Wang, Yong Yu, Weinan Zhang, and Zhaoxiang Wang. Colorbench: Benchmarking mobile agents with graph-structured framework for complex long-horizon tasks, 2025. URL https://arxiv.org/abs/2510.14621. WWW 2026.

Yifan Sui, Xin Huang, Hongbing Li, Fang Xu, Jiahe Lv, Haolong Yan, Yeqing Shen, Litao Liu, Zhimin Fan, Ziyang Meng, Jia Wang, Junbo Qi, Kaijun Tan, Zheng Ge, Xiangyu Zhang, Daxin Jiang, and Osamu Yoshie. Androiddaily: A verifiable benchmark for mobile gui agents on real-world closed-source applications, 2026. URL https://arxiv.org/abs/2605.27761.

Yuchen Sun, Shanhui Zhao, Tao Yu, Hao Wen, Samith Va, Mengwei Xu, Yuanchun Li, and Chongyang Zhang. Gui-xplore: Empowering generalizable gui agents with one exploration, 2025. URL https://arxiv.org/abs/2503.17709.

Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. Autodroid: Llm-powered task automation in android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, pp. 543–557, 2024. doi: 10.1145/3636534.3649379.

Qinzhuo Wu, Weikai Xu, Wei Liu, Tao Tan, Jianfeng Liu, Ang Li, Jian Luan, Bin Wang, and Shuo Shang. Mobilevlm: A vision-language model for better intra- and inter-ui understanding, 2024. URL https://arxiv.org/abs/2409.14818.

Leyang Yang, Ziwei Wang, Xiaoxuan Tang, Sheng Zhou, Dajun Chen, Wei Jiang, and Yong Li. Probench: Benchmarking gui agents with accurate process information, 2025. URL https://arxiv.org/abs/2511.09157. AAAI 2026.

Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. Static window transition graphs for android (t). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, pp. 658–668, 2015. doi: 10.1109/ASE.2015.76.

Jiabo Ye, Xi Zhang, Haiyang Xu, Haowei Liu, Junyang Wang, Zhaoqing Zhu, Ziwei Zheng, Feiyu Gao, Junjie Cao, Zhengxi Lu, Jitong Liao, Qi Zheng, Fei Huang, Jingren Zhou, and Ming Yan. Mobile-agent-v3: Fundamental agents for gui automation, 2025. URL https://arxiv.org/abs/2508.15144.

Mingxian Yu, Siqi Luo, and Xu Chen. Graphpilot: Gui task automation with one-step llm reasoning powered by knowledge graph, 2026. URL https://arxiv.org/abs/2601.17418.

Chaoyun Zhang, Shilin He, Jiaxu Qian, Bowen Li, Liqun Li, Si Qin, Yu Kang, Minghua Ma, Guyue Liu, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. Large language model-brained GUI agents: A survey, 2025. URL https://arxiv.org/abs/2411.18279.

Yuqing Zhang, Honghui Sheng, Xueyu Hu, Shengyu Zhang, and Fei Wu. Dac-bench: A decision-aware benchmark for compositional mobile gui tasks. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 44578–44598, 2026. doi: 10.18653/v1/2026.acl-long.2064.

Hongbin Zhong, Fazle Faisal, Luis Franca, Tanakorn Leesatapornwongsa, Adriana Szekeres, Kexin Rong, and Suman Nath. Actionengine: From reactive to programmatic gui agents via state machine memory, 2026. URL https://arxiv.org/abs/2602.20502.

A TASK-TO-DOMAIN MAPPING

Table 3: Task-to-domain mapping used in Figure 3. Tasks are grouped by their primary user goal. Parenthesized names denote scenario-specific subgraphs within an application; the six Douyin subgraphs remain components of one Douyin application-level UTG.

Task domain	Application/scenario assignment	Tasks	Share
Social & Video Entertainment	TikTok; Xiaohongshu; Hongguo Short Drama	33	19.53%
Local Services	Douyin (Group Buying); Dianping	26	15.38%
E-commerce	Douyin (Mall); Douyin Mall	22	13.02%
Account & General Services	Douyin (Home Sidebar; Profile)	16	9.47%
Content Creation	Douyin (Content Creation)	15	8.88%
Music	Qishui Music	14	8.28%
Maps & Mobility	Amap (Gaode Map)	14	8.28%
Digital Reading	Fanqie Novel	11	6.51%
News & Information	Toutiao	10	5.92%
Live Streaming	Douyin (Live)	8	4.73%
Total	**10 applications / 10 application-level UTGs**	**169**	**100.00%**

B METRIC CALCULATION DETAILS

This appendix gives the detailed calculation rules for the metrics used in GraceBench. Let $G_\tau = (V_\tau, E_\tau)$ be the task subgraph for task τ, and let $E_{\text{req}} \subseteq E_\tau$ be the set of required task transitions. After evaluating an execution trace, GraceBench identifies the set of missing required edges $M_\tau \subseteq E_{\text{req}}$, and the number of completed required edges is

$$C^\tau_{\text{req}} = |E_{\text{req}}| - |M_\tau|. \tag{5}$$

Let $\mathcal{I}_\tau$ denote the scorable action steps in the aligned trajectory, excluding `WAIT` and `COMPLETE`.

Success Rate. Success rate is the proportion of runs marked successful in the frozen per-trajectory evaluation summaries. For each task,

$$SR_\tau = \mathbf{1}[\text{the evaluation summary reports task success}], \tag{6}$$

The task-success labels are taken directly from the supplied summaries and preserved throughout aggregation and post-hoc analysis. The benchmark-level success rate is

$$SR = \frac{1}{N}\sum_{\tau=1}^{N} SR_\tau. \tag{7}$$

Completion Rate. Completion rate measures the fraction of task-required transitions completed by the agent:

$$CR_\tau = \frac{C^\tau_{\text{req}}}{|E_{\text{req}}|}. \tag{8}$$

We report macro-averaged CR over all evaluated tasks.

Deviation Rate. Deviation rate measures the fraction of scorable steps labeled as $deviation$:

$$DR_\tau = \frac{|\{t \in \mathcal{I}_\tau : \ell_t = deviation\}|}{|\mathcal{I}_\tau|}. \tag{9}$$

We report macro-averaged DR over all evaluated tasks.

Loop Burden Rate. Let L_τ be the set of steps covered by detected loop events in task τ. Let $\mathcal{D}_\tau$ denote all decision steps except `COMPLETE`; unlike $\mathcal{I}_\tau$, it includes `WAIT`, since repeated waiting can consume the interaction budget. For a trajectory with at least one loop, the loop step rate is

$$LBR_\tau = \frac{|L_\tau|}{|\mathcal{D}_\tau|}. \tag{10}$$

In the main results, loop burden is aggregated only over trajectories where a loop occurs:

$$LBR = \frac{\sum_{\tau:|L_\tau|>0} |L_\tau|}{\sum_{\tau:|L_\tau|>0} |\mathcal{D}_\tau|}. \tag{11}$$

Recovery Success Rate. A recovery opportunity begins when the agent enters an off-path state. For task τ, let N^τ_{dev} be the number of deviation episodes and N^τ_{succ} be the number of episodes that successfully return to the task subgraph. When $N^\tau_{\text{dev}} > 0$,

$$RSR_\tau = \frac{N^\tau_{\text{succ}}}{N^\tau_{\text{dev}}}. \tag{12}$$

Tasks without recovery opportunities are excluded from aggregate recovery analysis. The reported RSR pools successful episodes and opportunities across applicable tasks, rather than averaging their individual ratios.

Average Recovery Latency. For each successful recovery episode i, recovery latency is the number of steps from the first deviation step to the step where the trajectory returns to the task subgraph:

$$\text{latency}_i = t^{\text{rec}}_i - t^{\text{dev}}_i. \tag{13}$$

ARL is averaged over successful recovery episodes:

$$ARL = \frac{1}{N_{\text{succ}}} \sum_{i=1}^{N_{\text{succ}}} \text{latency}_i. \tag{14}$$

Recovery Ability. Recovery ability combines recovery success and recovery speed. For each recovery opportunity i, the contribution is

$$c_i = \begin{cases} \frac{1}{\max(1,\text{latency}_i)}, & \text{if recovered,} \\ 0, & \text{otherwise.} \end{cases} \tag{15}$$

For a task with $N^\tau_{\text{dev}} > 0$,

$$RA_\tau = \frac{1}{N^\tau_{\text{dev}}} \sum_{i=1}^{N^\tau_{\text{dev}}} c_i. \tag{16}$$

As with RSR, tasks without recovery opportunities are treated as not applicable rather than assigned a perfect recovery score. The reported RA pools the contributions c_i across all episodes and divides by the total recovery-opportunity count.

C Post-hoc Diagnostic Ablations

Dataset scope and reproducibility. Every trajectory-level analysis in the main text and this appendix starts from the same 3,718 records: eleven models, 169 tasks, and two instruction conditions. Outcome and metric-availability restrictions define the explicitly reported supports within this full dataset. The completed reruns of 78 error-ended and 10 invalid-startup cases are included under the same reconciliation rules used for the main analysis. The existing nine-model summaries contribute 3,042 records. The imported UI-TARS and GUI-Owl bundle contributes the remaining 676 records. All task identities, instructions, process metrics, count denominators, and reported SR labels are retained from the supplied per-trajectory summaries. No labels are replaced using separate terminal reviews. The imported package does not contain video or screenshot bytes, so this analysis reuses existing evaluations and does not perform new visual adjudication. The remainder of this appendix provides the supporting analyses referenced in Section 4, including outcome-stratified instability, instruction effects, outcome-controlled comparisons, cross-task model consistency, recovery relevance, and predictive-cost ablations.

Table 4: Process instability by final task outcome across all 3,718 trajectories. Gaps are unsuccessful minus successful. DR is averaged over all trajectories; loop incidence is the fraction of trajectories containing a loop; LBR is computed only over trajectories in which a loop occurs; RA is computed only over trajectories with recovery opportunities.

	Detailed instruction			Concise instruction		
Metric	Success	Failure	Gap	Success	Failure	Gap
DR	0.0889	0.3067	+0.2178	0.1412	0.3618	+0.2206
Loop incidence	0.2066	0.6525	+0.4459	0.2748	0.6520	+0.3772
LBR	0.4257	0.6390	+0.2133	0.4840	0.6142	+0.1302
RA	0.7285	0.5668	-0.1618	0.6394	0.5918	-0.0476

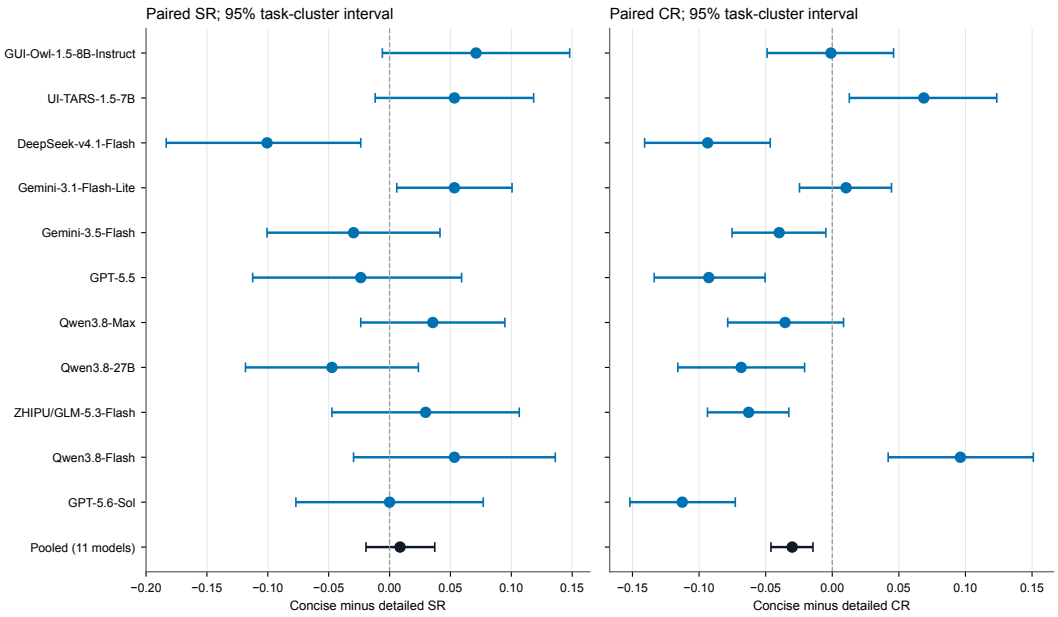

Figure 6: **Instruction effects across all eleven models.** Concise-minus-detailed SR and CR, with task-cluster 95% intervals. Each model has 169 pairs; the pooled estimate has 1,859 pairs. All 88 replacement runs are selected before pairing.

Outcome-stratified process instability. Table 4 provides the full comparison summarized in Section 4.2.

C.1 Instruction Granularity

Detailed guidance improves path coverage and reduces deviation. Across 1,859 matched model–task pairs, detailed and concise instructions yield 639 and 655 successes. The concise-minus-detailed SR difference is 0.0086 (95% interval [-0.0194, 0.0371]), whereas the CR difference is -0.0301 (interval [-0.0460, -0.0145]) and the DR difference is 0.0522 (interval [0.0332, 0.0719]). Thus detailed instructions improve required-path coverage in this cohort, while evidence for an agent-success difference remains inconclusive. Final-state success and prescribed-path coverage are different estimands: a task can reach its target without completing every reference transition. Figure 6 shows that effects vary by model, and Table 5 preserves every reported model's direction.

Estimands and pairing. All paired differences are concise minus detailed. CR and DR compare task-level scores; opportunity-dependent LBR and RA are compared only where both members are applicable, and those paired means differ from the pooled denominator-weighted leaderboard estimands.

Table 5: Concise-minus-detailed differences from the displayed four-decimal aggregates in Table 2. Paired intervals in Figure 6 use the complete per-trajectory records for all eleven models.

Model	ΔSR	ΔCR	ΔDR	ΔLBR	ΔRA
UI-TARS-1.5-7B	+0.0532	+0.0687	-0.0061	-0.0561	-0.0004
GUI-Owl-1.5-8B-Instruct	+0.0710	-0.0010	+0.0501	-0.0518	-0.0163
DeepSeek-V4.1-Flash	-0.1006	-0.0936	+0.0780	+0.0173	+0.0418
Qwen3.8-Max	+0.0355	-0.0354	+0.0043	+0.0158	+0.0850
Qwen3.8-27B	-0.0474	-0.0684	+0.0621	+0.0120	-0.0293
GLM-5.3-Flash	+0.0296	-0.0628	+0.0560	-0.0181	-0.0045
Qwen3.8-Flash	+0.0533	+0.0962	+0.0551	-0.1098	+0.1359
GPT-5.6 Sol	+0.0000	-0.1127	+0.0633	+0.0876	+0.0071
GPT-5.5	-0.0237	-0.0927	+0.0898	+0.0652	-0.0180
Gemini 3.1 Flash Lite	+0.0533	+0.0103	+0.0689	-0.0087	-0.0673
Gemini 3.5 Flash	-0.0296	-0.0398	+0.0532	-0.0077	-0.0334

Table 6: Paired process changes when both descriptions succeed. Opportunity-dependent comparisons use their own common support.

Measure	Pairs	Mean change	95% task-cluster interval
DR	426	0.0467	[0.0236, 0.0713]
Loop incidence	426	0.0423	[-0.0072, 0.0922]
LBR	34	0.0339	[-0.0653, 0.1232]
RA	96	-0.0911	[-0.1717, -0.0133]
CR	426	-0.0591	[-0.0879, -0.0325]

Among 1,859 pairs, 426 succeed in both conditions, 229 only in concise, 213 only in detailed, and 991 in neither. The both-success comparisons cover 123 tasks. Bootstrap samples draw all 169 task IDs with replacement, 10,000 times (seed 20260923), retaining all eligible records for each sampled task. The intervals use the 2.5th and 97.5th percentiles and condition on this model set; they do not estimate run-to-run device variation.

Composition and reference-length sensitivity. We repeat the pooled paired estimates after excluding each model and each scenario in turn. The following ranges summarize point estimates, not uncertainty intervals.

All 26 composition checks retain a negative CR interval and a positive DR interval; all SR intervals cross zero. We also split tasks by reference path length at six steps, without choosing the cut to maximize an observed effect. Shorter paths (at most six steps) have 880 pairs and a CR change of -0.0078 with interval [-0.0329, 0.0167]. Longer paths have 979 pairs and a CR change of -0.0502 with interval [-0.0701, -0.0305].

The longer-path interval favors detailed coverage, whereas the shorter-path interval crosses zero; the difference between subgroup effects is not itself tested. Scenario estimates are exploratory, have heterogeneous sample sizes, and are not selected as independent confirmatory tests.

C.2 Outcome-Controlled Stability Comparisons

Outcome-controlled pair construction and sensitivity. For each task and condition, let k of the eleven models succeed. This stratum contributes $\binom{k}{2} + \binom{11-k}{2}$ within-outcome unordered model pairs. We require a CR gap at most 0.00005 for exact stored-precision matching and require both values of each compared process metric. We sweep CR tolerances 0.00005, 0.05, and 0.10, and process-gap thresholds 0.10, 0.20, and 0.30. The resulting 54 outcome–metric–threshold settings use the same source records. For the primary threshold, task-cluster intervals retain all dependent model pairs from each sampled task.

A stricter descriptive search holds the other two metric gaps at most 0.0500 while requiring the target gap at least 0.2000 and exact CR. It yields 6 DR pairs, 16 LBR pairs, and 7 RA pairs. This demonstrates that individual metrics can distinguish some otherwise similar records, without establishing causal isolation. The common-support correlation analysis includes 1,447 trajectories on 169 tasks; using the same support avoids conflating metric association with different missingness patterns.

Table 7: Leave-one-group-out sensitivity of concise-minus-detailed effects.

Excluded unit	SR range	CR range	DR range
One model	[0.0024, 0.0195]	[-0.0427, -0.0218]	[0.0485, 0.0581]
One scenario	[-0.0057, 0.0183]	[-0.0365, -0.0153]	[0.0345, 0.0598]

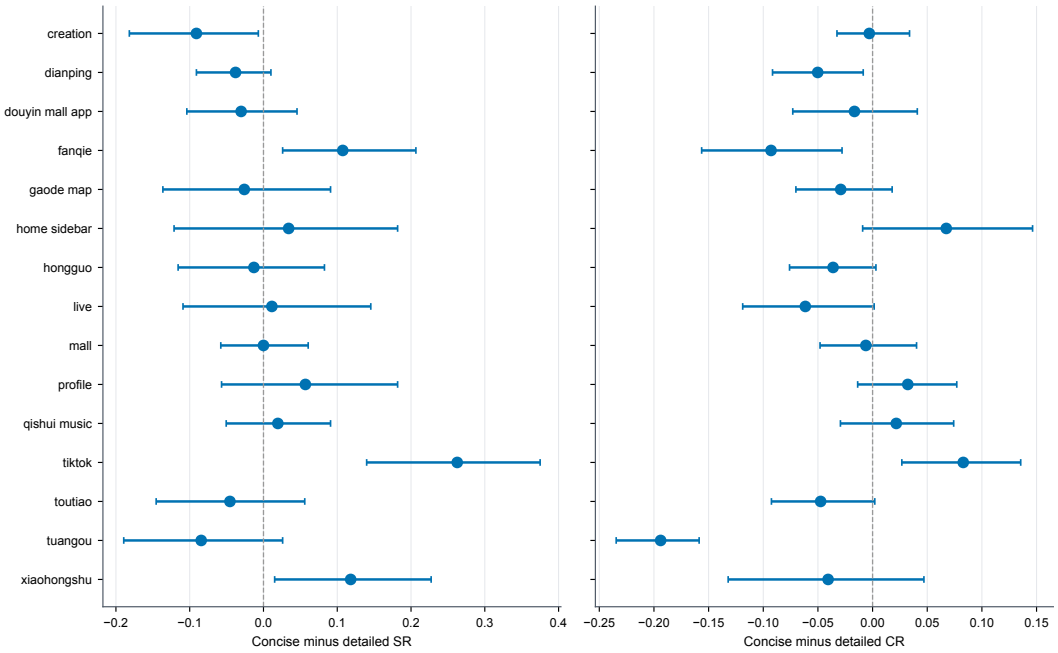

Figure 7: **Scenario-specific paired instruction effects.** All eleven models are included. Intervals resample task clusters; small scenarios have correspondingly less precise estimates. These comparisons are exploratory and unadjusted for multiplicity.

C.3 Cross-task Model Consistency

Table 9 uses all 3,718 records without selecting on SR or CR. We generate 1,000 partitions with seed 20260926, stratifying by scenario and reference length (at most six versus more than six steps); all models and both instructions for a task remain in the same subset. Each subset independently aggregates DR over tasks, LBR over loop-containing trajectories, RSR and RA over recovery opportunities, and ARL over successful recoveries. Unavailable scores are not replaced by zero. The reported ranges describe sensitivity to task partitioning, not uncertainty from independent repeated experiments. For the random-ranking baseline, we hold all split-specific rankings fixed and apply one shared random permutation of subset-B model identities across all 1,000 partitions, preserving their dependence. The one-sided permutation p-value is the fraction of 200,000 permutations (seed 20260927) whose mean cross-subset Spearman correlation is at least the observed value. All ten metric–condition tests remain significant after Bonferroni correction. Restricting RA to episodes initiated by `off_path` yields mean rank correlations of 0.4388 and 0.7760 under detailed and concise instructions, respectively, reinforcing the dependence of recovery rankings on opportunity composition.

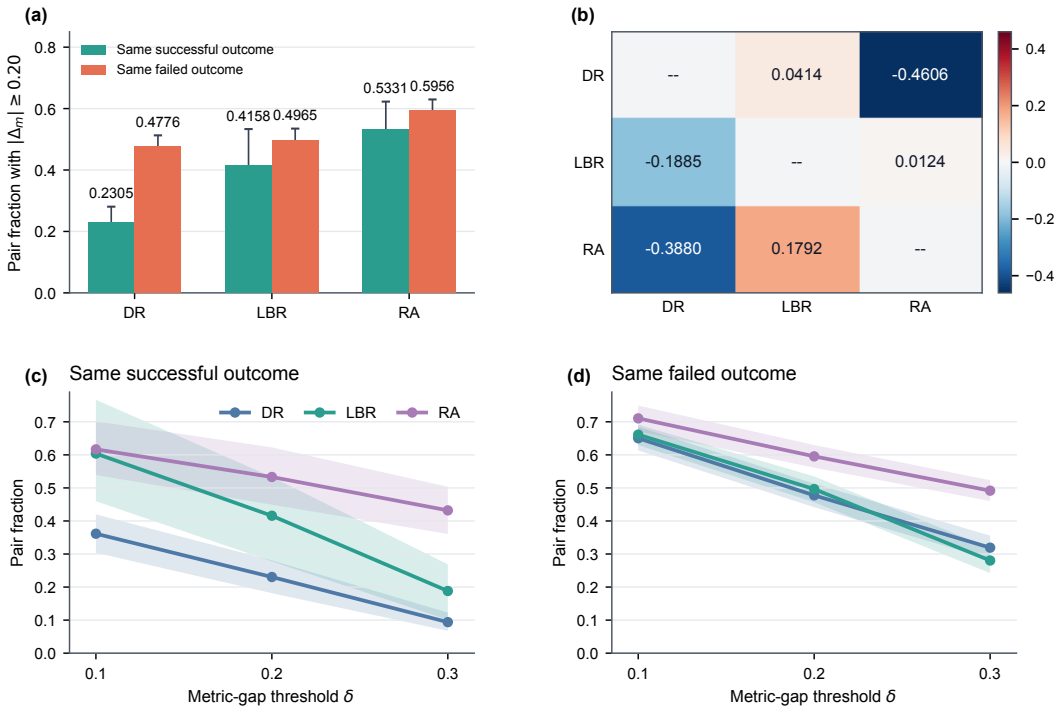

Figure 8: **Outcome-controlled process diagnostics on all 3,718 trajectories.** (a) Exact-CR matched pairs with process gap at least 0.2000; bars distinguish successful and failed pairs and whiskers are task-cluster 95% intervals. (b) Upper triangle: common-support Spearman correlation; lower triangle: residual correlation controlling task, model, condition, SR, and CR. (c–d) Threshold sensitivity for exact-CR matched pairs; shaded bands show task-cluster 95% bootstrap intervals.

Table 8: Exact-CR matched pairs with process gap at least 0.2000. Pair counts are not independent sample sizes; intervals cluster by task.

Outcome	Metric	Large / eligible	Fraction	95% interval
Success	DR	410/1779	0.2305	[0.1815, 0.2806]
Success	LBR	42/101	0.4158	[0.2778, 0.5333]
Success	RA	153/287	0.5331	[0.4492, 0.6227]
Failure	DR	926/1939	0.4776	[0.4417, 0.5128]
Failure	LBR	425/856	0.4965	[0.4567, 0.5348]
Failure	RA	860/1444	0.5956	[0.5613, 0.6298]

Table 9: **Cross-task reproducibility of model profiles.** Mean Spearman ρ between eleven-model rankings on disjoint task subsets over 1,000 stratified 84/85-task partitions (higher is more reproducible); brackets show the central 95% of partitions, not confidence intervals. One-sided p-values use 200,000 model-identity permutations; all remain significant after Bonferroni correction over ten tests.

Metric	Detailed ρ	Permutation p	Concise ρ	Permutation p
DR	0.9516 [0.8636, 0.9909]	< 0.0001	0.9228 [0.8182, 0.9727]	< 0.0001
LBR	0.9284 [0.8182, 0.9909]	< 0.0001	0.8381 [0.6727, 0.9636]	0.0001
RSR	0.8100 [0.6182, 0.9364]	0.0002	0.8269 [0.7091, 0.9364]	0.0002
ARL	0.5816 [0.2455, 0.8727]	0.0042	0.5880 [0.1998, 0.8636]	0.0039
RA	0.5983 [0.3000, 0.8818]	0.0029	0.8508 [0.6361, 0.9636]	< 0.0001

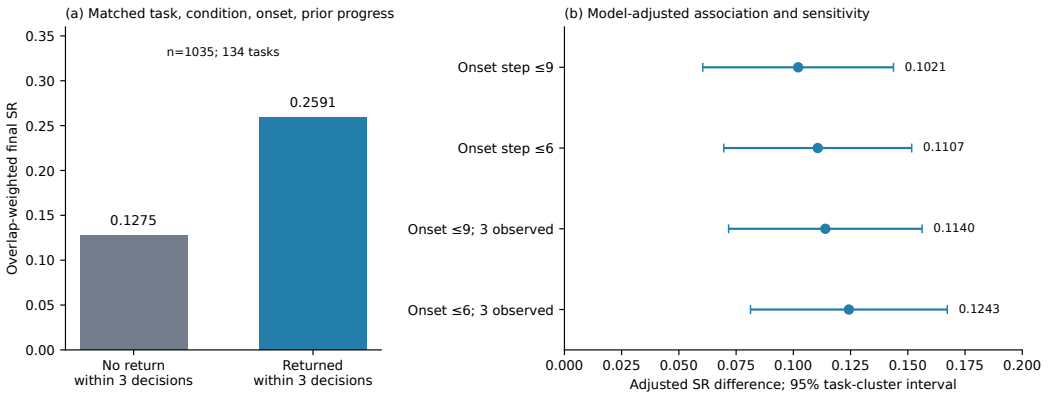

Figure 9: **Timely path return is associated with task success.** (a) Overlap-weighted SR after matching task, instruction condition, deviation-onset bin, and prior-progress bin. (b) Adjusted SR differences under onset and observation-window restrictions; points control task–condition and model fixed effects, onset bin, and prior progress, and whiskers are 95% task-cluster intervals.

C.4 Outcome Relevance of Timely Recovery

Matched recovery-window analysis. We test whether a labeled recovery event is associated with eventual task completion after an actual path deviation. For each trajectory, we take the first `off_path` episode whose zero-indexed onset is at most step 9 and observe the next three decisions, excluding terminal-mismatch starts and isolated recovery labels. This produces 2,230 eligible trajectories over all 169 tasks. We match trajectories by task, instruction condition, three-step onset bin, and prior-progress bin, retain strata containing both a return and a non-return, and apply overlap weights. Among the resulting 1,035 trajectories over 134 tasks, the weighted SR is 0.2591 for a return within three decisions and 0.1275 without one, a difference of 0.1316 with a 95% task-cluster interval of [0.0800, 0.1829].

Adjusted association and sensitivity. On all 2,230 eligible trajectories, a linear probability analysis controlling task–condition and model fixed effects, onset bin, and prior-progress bin estimates an SR difference of 0.1021 [0.0604, 0.1438]. As Figure 9 shows, the estimate remains positive when requiring all three subsequent decisions to be observed or restricting onset to step 6 or earlier. This consistency connects timely path return to a consequential task outcome, while the observational design leaves naturally occurring error difficulty uncontrolled.

Interpretation boundary. Return is neither sufficient for task success nor an isolated validation of the RA latency weighting: among 1,301 trajectories that recovered from every observed `off_path` opportunity, only 408 eventually succeeded (SR=0.3136). After controlling whether a return occurred, a 0.1 increase in the local reciprocal-latency score is associated with an SR difference of 0.0047 [-0.0039, 0.0133]. The experiment therefore supports the outcome relevance of timely return, but not a causal effect, an optimal latency function, or a common-error-start ranking of general recovery competence.

C.5 Grouped Predictive Ablation

The availability-aware baseline B contains SR, CR, model, instruction condition, scenario, expected task length, graph depth, loop occurrence, and recovery-opportunity occurrence. The full model is $F = B \cup \{\mathrm{DR}, \mathrm{LBR}, \mathrm{RA}\}$. We fit ridge regression with fixed $\alpha = 10$ to $\log(1 + \mathrm{duration})$ and executed steps. All 3,718 records have valid target metadata; no success-only selection is applied. Five-fold cross-validation is repeated over three fixed task-level splits. All models and both conditions for a task stay in one fold, and categorical encoding and standardization are fitted only on training folds. An unavailable LBR or RA is encoded as zero only for regression, with availability indicators included on both sides of each comparison. Intervals resample fixed cross-fitted loss

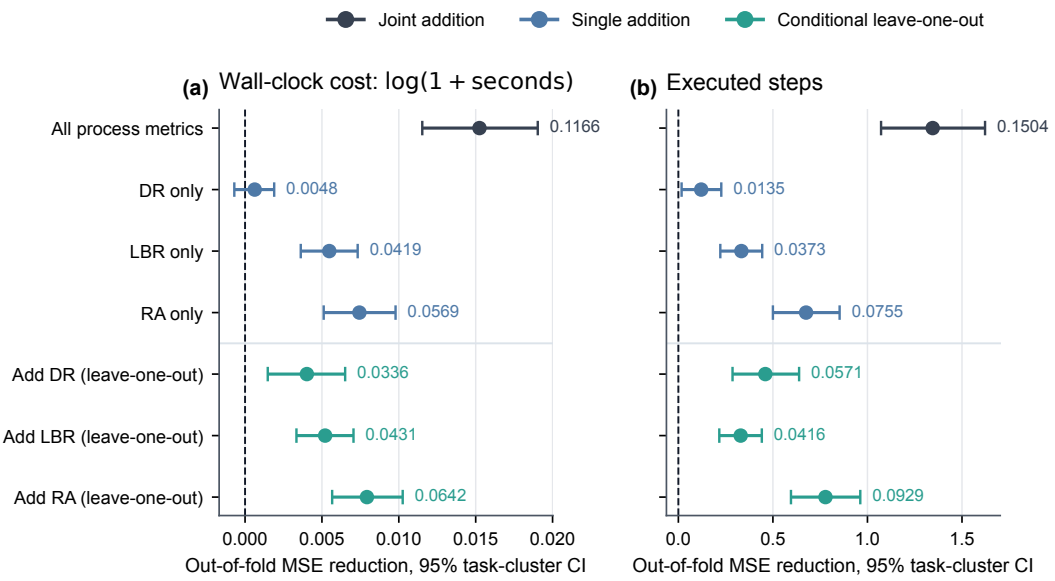

Figure 10: **Grouped predictive ablation on all 3,718 records.** Points show baseline MSE minus augmented-model MSE; positive values favor the augmentation. Whiskers are 95% task-cluster intervals. Labels give relative MSE reduction as decimals. Blue rows add one metric to the availability-aware baseline; green rows add an omitted metric back to the other two.

differences by task, without refitting during bootstrap. They quantify variation over tasks for the fitted procedure, rather than training-instability uncertainty. Adding all three process metrics reduces cross-validated MSE relative to the availability-aware baseline by 0.1504 for steps and 0.1166 for log duration.

Table 10: Conditional leave-one-out gains. Relative gains are decimals; absolute MSE units differ between targets.

Target	Added metric	Relative gain	MSE reduction	95% interval
Log duration	DR	0.0336	0.0040	[0.0015, 0.0065]
Log duration	LBR	0.0431	0.0052	[0.0033, 0.0071]
Log duration	RA	0.0642	0.0079	[0.0057, 0.0103]
Steps	DR	0.0571	0.4599	[0.2865, 0.6385]
Steps	LBR	0.0416	0.3296	[0.2165, 0.4413]
Steps	RA	0.0929	0.7780	[0.5955, 0.9621]

All six leave-one-out intervals lie above zero. The standalone DR duration addition instead has MSE reduction 0.0006 with interval [-0.0007, 0.0019]. This negative result distinguishes standalone utility from conditional complementarity and is retained.

Interpretation and remaining limitation. Equal agent success outcomes and path coverage do not determine process quality; matched comparisons, threshold sensitivity, common-support associations, and conditional cost gains provide complementary evidence. These analyses alone do not validate annotation accuracy, prove statistical independence, or identify uniquely optimal metric definitions; Appendix E separately audits annotation accuracy on a stratified visual-evidence sample. Duration can reflect inference speed, device, and collection period; step counts are mathematically related to DR/LBR denominators. RA is conditional on naturally occurring recovery opportunities and their difficulty, remaining budget, and error mixture. All bootstrap comparisons are exploratory and unadjusted for multiple testing. The imported GUI-model package contains evaluated records and timing metadata but excludes media bytes; the present analyses rely on the frozen reported labels.

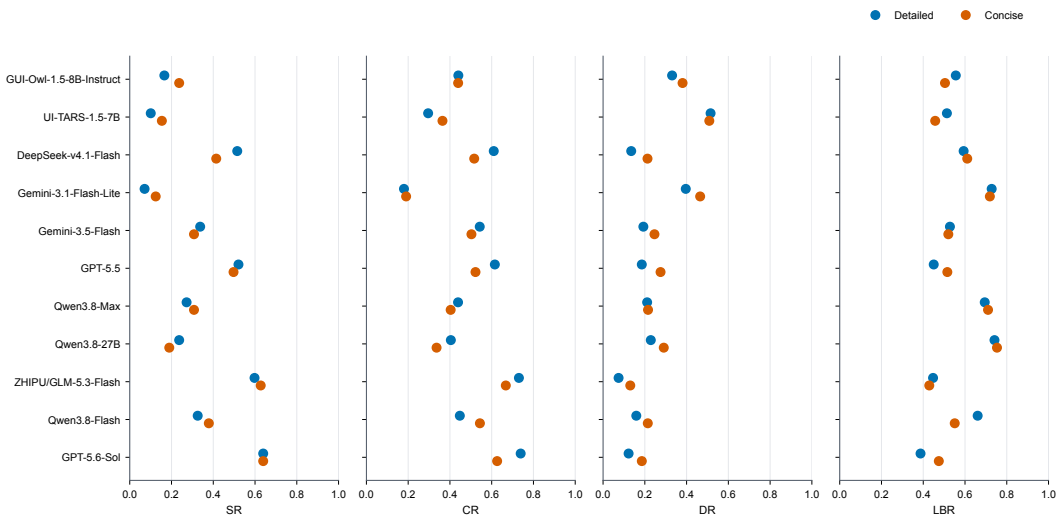

Figure 11: **All eleven models in the full dataset.** Detailed and concise SR, CR, DR, and LBR recomputed from the same 3,718 per-trajectory records used throughout the experiments.

D AUTOMATIC UTG EXPLORATION DETAILS

Agent–harness decomposition. Our explorer is not a fixed-path crawler. It separates semantic decisions from deterministic device operations: an agent identifies the current GUI state, selects business-relevant actions, and assigns stable state names, while an external harness performs evidence capture, action execution, graph serialization, risk gating, and consistency checks. This design allows the same execution protocol to cover heterogeneous applications without encoding app-specific navigation paths in the runtime. Before exploration, we perform a short reconnaissance pass and define an application-specific value spine, such as search–product–cart for shopping, feed–playback–comments for media, or book detail–reader–table of contents for reading applications. Low-value settings branches are expanded only after the value spine has been covered.

Observation and evidence capture. The default observation waits 2 seconds for the interface to stabilize and then records three synchronized evidence sources through ADB (Android Developers, 2026): (i) a lossless PNG screenshot using `exec-out screencap -p`; (ii) the focused package and Activity from `dumpsys window`; and (iii) a UI hierarchy from `uiautomator dump`, with a compressed-dump fallback. For asynchronous transitions such as opening a reader or a media detail page, the agent requests a second observation with an extended wait rather than treating an intermediate loading frame as a state. Each observation is stored as a JSON record together with its screenshot and XML paths, timestamp, visible text, candidate controls, scrollable regions, and hashes. The files are indexed as `step_XXXX_<label>` so every graph update can be traced to the original evidence.

State signatures and matching. The XML parser removes invalid regions smaller than 6×6 pixels and deduplicates controls by label and center coordinates quantized to eight-pixel cells. At most 60 tap candidates, 24 stable controls, and 12 scrollable regions are retained per observation. Personalized strings containing long numbers, prices, popularity counts, usernames, or other volatile values are excluded from the stable-control set. The harness computes a 16-hex-character screenshot hash from the PNG and a structural hash from

$$(\text{Activity},\ \text{node count},\ \text{clickable count},\ \text{first 18 stable controls}).$$

These SHA-1-based hashes are deterministic identity hints, not security primitives. The agent first supplies a stable semantic state name; when several nodes share that normalized name, the harness resolves the anchor using exact screenshot hash ($+160$), exact structural hash ($+120$), matching Activity ($+100$), pending status ($+80$), and existing evidence/children. An Activity conflict incurs a

Algorithm 1 Evidence-grounded construction of an application-level UTG

Require: Android device d, safety policy R, value-spine targets H
Ensure: Reviewed UTG $G = (V, E)$ with state/action evidence

Initialize G, pending queue Q, and current state v
$o \leftarrow \textsc{Observe}(d, 2\text{ s})$
while $\neg\textsc{Stop}(G, Q, H)$ **do**
  $C_{\text{tap}}, C_{\text{swipe}} \leftarrow \textsc{ExtractCandidates}(o.\text{xml})$
  $\hat{v} \leftarrow \textsc{AgentClassify}(o, C_{\text{tap}}, C_{\text{swipe}})$
  $v \leftarrow \textsc{MatchOrCreate}(G, \hat{v}, o)$; $\textsc{AttachEvidence}(v, o)$
  $A \leftarrow \textsc{AgentPropose}(v, C_{\text{tap}}, C_{\text{swipe}}, H)$
  for all $a \in A$ **do**
    if $\textsc{ViolatesPolicy}(a, R)$ **then**
      $\textsc{AddBoundary}(G, v, a, \text{DEFER})$
    else if $\neg\textsc{MountGuard}(v, a)$ **then**
      $\textsc{Block}(a, \text{NEEDS-REVIEW})$
    else
      $\textsc{AttachPendingEdge}(G, v, a)$; $\textsc{Push}(Q, a)$
    end if
  end for
  $Q \leftarrow \textsc{RankPending}(Q)$
  if $\textsc{NoEligibleAction}(Q)$ **then**
    break
  end if
  $a^\star \leftarrow \textsc{AgentSelect}(Q)$; $\textsc{ExecuteOne}(d, a^\star)$
  $o' \leftarrow \textsc{Observe}(d, 2\text{ s})$
  if $\textsc{Transient}(o')$ **then**
    $o' \leftarrow \textsc{Observe}(d, \text{extended wait})$
  end if
  $v' \leftarrow \textsc{AgentVerify}(o', a^\star)$
  $\textsc{ResolveTransition}(G, v, a^\star, v', o')$; $\textsc{Reconcile}(G)$
  $o \leftarrow o'$; $v \leftarrow v'$
end while
return $\textsc{ManualReviewAndDeduplicate}(G)$

-40 penalty. The final merge decision also checks modal presence, scroll position, selected tab, and semantic role, preventing visually different sheets or scrolled detail states from being merged solely because they share an Activity.

Candidate generation and prioritization. For an XML control c, the initial tap score is

$$s_{\text{tap}}(c) = 8\mathbb{I}_{\text{clickable}} + 5\mathbb{I}_{\text{labeled}} + 6\sum_{k \in K_H} \mathbb{I}[k \in \text{label}(c)] - 12\mathbb{I}_{\text{risk}}, \tag{17}$$

where K_H contains high-value concepts such as search, playback, comments, filters, lists, reading, and scrolling. The parser also converts every `scrollable=true` region into directional swipe candidates. Vertical regions produce up/down actions, horizontal regions produce left/right actions, and reader pages receive page-turn candidates when XML does not expose a scrollable container. Swipes are therefore first-class graph edges rather than fallback actions.

The pending-action queue ranks unresolved edges by

$$\begin{aligned} s_{\text{plan}}(a) = 10n_H(a) + w_{\text{priority}} + 18\mathbb{I}_{\text{swipe}} + 4\mathbb{I}_{\text{up/left}} \\ - 100\mathbb{I}_{\text{risk}} - 80\mathbb{I}_{\text{needs review}} - 15\mathbb{I}_{\text{recollect}}, \end{aligned} \tag{18}$$

with $w_{\text{priority}} \in \{30, 20, 5, -10\}$ for critical, high, normal, and low priorities. The queue is advisory: the agent selects one action after checking current visual evidence and business value. The harness then executes exactly one tap, swipe, back, or text-input action. A swipe lasts 420 ms and is constrained to the observed scrollable bounds when available; otherwise it uses a screen-relative fallback region.

Graph update and online mount guard. In the serialized tree projection, each conceptual UTG edge is represented as state $\rightarrow$ action record $\rightarrow$ state; in the graph definition, the action remains an edge attribute. Action records store `gesture`, `direction`, `bounds`, `priority`,

`status`, and `risk`; state and action records both retain screenshot, XML, Activity, structural hash, and observation path. Before a pending edge enters the normal queue, a lightweight mount guard checks the current parent, proposed action, target state, and sibling context. For example, player-only actions such as comments or episode selection cannot be mounted directly under a search/ranking list, and closing one overlay should not transition directly to an unrelated overlay. A failed check writes `mount_check.status=needs_review`, sets `blocked=true`, marks the target `needs_reobserve=true`, and removes it from the normal pending queue. This local guard runs online; a full-tree audit is reserved for completed value-spine milestones and final delivery.

Safety and termination. The execution gate uses both explicit metadata and lexical redline categories covering payment, order submission, recharge, withdrawal, authentication, authorization, external sharing, publishing, and account-asset mutations. A risky entry may be represented as a boundary edge with `risk=true` and `status=defer`, but the side-effecting confirmation is not executed. Search submission and filter application are explicitly treated as safe exceptions. After each verified transition, reconciliation marks an action `done` only when every non-risk target has evidence or explored children; blocked, risky, and deferred edges remain unresolved by design. Exploration stops when the application value spine has been covered and the remaining queue contains only risky, blocked, duplicate, or low-value branches. Finally, we manually review state names and evidence, deduplicate equivalent states, verify return transitions, and synchronize the canonical graph with its visualization copy.

D.1 UTG Structural Coverage

Table 11 reports an auditable inventory of the final normalized UTGs. Canonicalization maps 3,265 collected state observations to 2,909 canonical GUI states by merging viewport-only or dynamic-content duplicates while preserving task-relevant differences in modal state, selected tab, scroll position, and semantic role. An observed transition is a unique $(\text{source state}, \text{action label}, \text{target state})$ triple for which both endpoints have canonical-state mappings. We exclude 83 unresolved target observations from the transition counts rather than treating them as verified graph edges. A branch state has at least two distinct outgoing action–target pairs.

Table 11: **Structural coverage of the ten application-level UTGs.** States are canonical GUI states; transitions are distinct observed action-conditioned edges; branch states expose at least two distinct outgoing action–target pairs. Task steps are computed from the 169 curated reference paths and include the cold start as one action.

Application	Tasks	States	Transitions	Branch states	Mean / max steps
Douyin	63	827	763	91	6.60 / 11
Dianping	12	206	237	29	9.25 / 10
Douyin Mall	12	268	311	41	6.42 / 12
Fanqie Novel	11	312	361	38	8.00 / 15
Amap	14	171	189	33	7.14 / 9
Hongguo	14	370	399	47	8.93 / 11
Qishui Music	14	145	162	13	6.21 / 9
TikTok	9	163	177	19	7.33 / 13
Toutiao	10	137	148	24	6.20 / 8
Xiaohongshu	10	310	369	39	5.40 / 9
Total	**169**	**2,909**	**3,116**	**374**	**7.02 / 15**

The inventory confirms that every benchmark application is represented by a nontrivial reviewed graph and that the task suite is supported by 3,116 observed transitions and 374 decision-bearing branch states. Together with the per-task path lengths, these statistics support structural coverage adequacy within GraceBench's declared task and safety scope. They do not imply exhaustive coverage of all functionality exposed by the underlying applications.

Task-level state support. We further measure how much local graph structure supports each modeled task. For task τ, let S_τ be the union of existing canonical GUI states matched to its curated expert path. Where multiple route witnesses exist, we retain the nonredundant routes that satisfy the

instruction's explicit entry constraints. Action-only endpoints and task states without a corresponding canonical state do not create synthetic nodes. We define the undirected one-hop closure over the UTG transitions as

$$N_1(S_\tau) = S_\tau \cup \{v \in V : \exists u \in S_\tau,\ (u, a, v) \in E \ \vee \ (v, a, u) \in E\}. \tag{19}$$

This closure includes the matched task states themselves and every distinct predecessor or successor connected by one modeled action. It measures the local state context available around a task route, rather than asserting that every neighboring state belongs to a legal completion path.

Table 12: **Canonical states supporting each task subgraph.** Statistics cover all 169 tasks. Confidence intervals are nonparametric task-bootstrap intervals with 10,000 resamples and seed 20260925.

Measure	Mean	95% CI	Median	IQR	Range
Core task states	5.91	[5.62, 6.18]	6	[5, 7]	[1, 11]
Additional one-hop states	35.19	[32.88, 37.66]	34	[24, 43]	[6, 102]
One-hop closure	41.09	[38.62, 43.69]	40	[30, 49]	[7, 112]
Per-task expansion factor	7.20×	[6.82, 7.59]	7.14×	[5.33, 8.80]	[2.80, 15.50]

The matched core of a task contains 5.91 canonical states on average, while including directly connected one-hop context increases this to 41.09 states and adds 35.19 distinct states. The mean per-task expansion is 7.20×; under this definition, each task is represented by its matched path states plus a larger set of directly adjacent states. This statistic quantifies local structural support and does not claim exhaustive application coverage.

E Human Expert Audit of Generated Transition Labels

Transition sampling and review protocol. We froze a visual-validation population of 3,042 trajectories containing 34,767 transitions with locally verified MP4 evidence. The 676 imported aggregate-only records without media bytes were excluded. With seed 20260926, we sampled by execution model, instruction condition, and scenario, retaining at least one trajectory per nonempty stratum. The resulting 313 trajectories contain 3,487 transitions, or 0.1003 of the eligible transition population. Reviewers saw the task, expert path, primitive action, and a five-frame ordered action-aligned storyboard (pre-action, onset, post-action, response, and settled frames), while model identity, condition, automatic labels, and terminal outcomes remained hidden.

Expert agents generated the initial diagnostic labels for exact expert-edge state matches and Deviation, Recovery, and Loop events. Human experts then independently reviewed the sampled transition storyboards under blinded conditions and produced reference binary labels for each diagnostic field. After the primary reviews were frozen, a human expert adjudicated field-level disagreements; all 3,487 transitions were resolved. Human experts served as auditors and annotators rather than experimental subjects, so this audit is not a human-subject experiment.

Cohen's κ is computed between the expert-agent generated binary labels and the human-adjudicated reference labels. For each diagnostic field, the expert-agent output and the human-adjudicated reference each provide a binary label for every one of the 3,487 sampled transitions. The adjudicated labels are used as the human reference to measure agreement between generated and human expert labels.

Table 13: **Agreement between generated and human expert labels.** Cohen's κ compares expert-agent generated binary labels with human-adjudicated reference labels over the 3,487 sampled transitions.

Field	Cohen's κ
State match	0.7980
Deviation	0.7952
Recovery	0.7663
Loop	0.8656