GUITestScape: Towards Open-set Evaluation on Exploratory GUI Testing

Anonymous ACL submission

Abstract

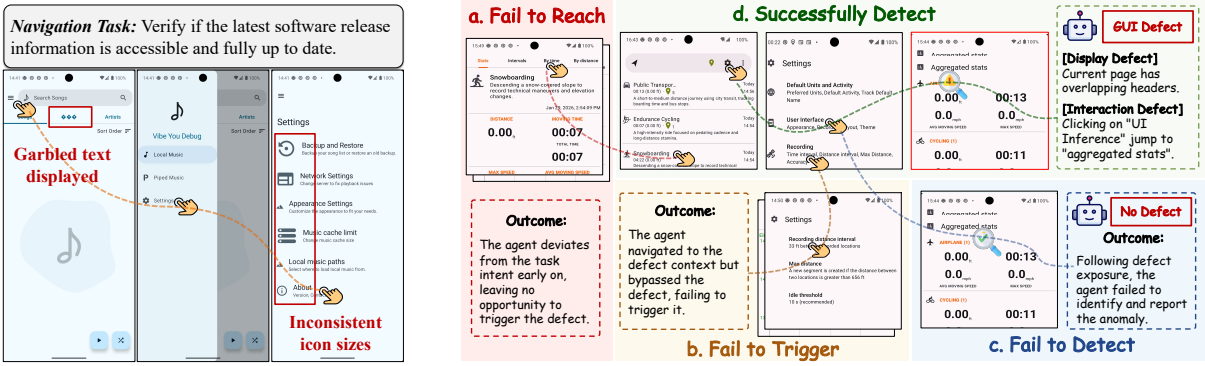
Exploratory GUI testing is a particularly demanding setting for MLLM agents: without predefined test scripts, an agent must autonomously navigate an application and discover defects through its own interaction. However, current evaluation falls short on two fronts. First, existing benchmarks focus almost exclusively on interaction defects, leaving display defects outside the evaluation frame. Second, evaluation protocols are bound to predefined defect annotations, collapsing the testing process into a single end-state judgment that conflates qualitatively distinct failure modes. To address these challenges, we present **GUITestScape**, an interactive benchmark covering 61 real-world Android applications and 508 preset defects spanning interaction and display types, and introduce **GUIJudge**, an open-set evaluator that decomposes an agent's testing trajectory into independently diagnosable capabilities. Experimental results demonstrate that GUIJudge achieves reliable process-aware evaluation beyond predefined annotations, substantially outperforming all baselines. Benchmarking on GUITestScape further reveals that detection remains the critical bottleneck for existing models across both defect types, and that integrating GUIJudge's verifiers into existing agents significantly boosts their detection performance without retraining.

1 Introduction

Multimodal Large Language Models have substantially strengthened GUI agents, extending their reach from single-step grounding to long-horizon tasks (Qin et al., 2025; Rawles et al.; Xie et al., 2024a) and practical applications such as software quality assurance (Zhao et al., 2024). Among these, exploratory GUI testing stands out as particularly demanding: without predefined test scripts, an agent must autonomously navigate an application and discover defects through its own interaction (Gao et al., 2026a). Yet current evaluation falls short of measuring these capabilities faithfully.

We identify two fundamental challenges: (1) **Display-defect blindness**. Current benchmarks (Gao et al., 2026a; Ahmed et al., 2026) focus almost exclusively on interaction defects, overlooking display defects such as rendering errors, missing elements, and layout misalignment (Liu et al., 2020, 2022). These defects often have the most immediate impact on user experience, yet detecting them demands open-ended visual anomaly detection, a capability that interaction-oriented evaluation never probes. (2) **Closed-set evaluation**. Existing protocols verify agent reports against predefined defect annotations, which introduces two limitations. First, the evaluation cannot scale to unlabeled defects, since any anomaly outside the annotated set has no reference for verification. Second, scoring collapses the entire testing process into a single end-state judgment (He et al., 2024), conflating qualitatively distinct failure modes. An agent may never reach the defect scenario, may reach it but fail to trigger the defect, or may trigger the anomaly yet fail to recognize it. Consequently, agents with very different capability profiles are flattened to indistinguishable scores. Figure 1 illustrates these challenges with two examples: a realistic interaction case where display defects appear repeatedly during testing, and an exploratory testing process where agents exhibit different failure modes before successful defect detection.

To address these challenges, we present **GUITestScape**, an interactive benchmark covering 61 real-world Android applications and 508 preset defects spanning interaction and display types. Each defect is instantiated into a structured evaluation case along three dimensions: a resettable scenario that ensures run-to-run consistency, a state-level test basis that captures app-specific trigger criteria, and multiple navigation tasks that simulate how testers might approach the same defect through dif-



(a) Display defects in one test case.

(b) Different outcomes in exploratory GUI testing.

Figure 1: Motivating examples for GUITestScape and GUIJudge. (a) A realistic testing process exposes multiple display defects, such as garbled text and inconsistent icon sizes. (b) The same exploratory testing process may lead to different outcomes, including failures at Reach, Trigger, and Detect, as well as successful defect detection.

ferent interaction patterns. We further introduce **GUIJudge**, an open-set evaluator that decomposes an agent's testing trajectory into three independently diagnosable capabilities: *Reaching*, *Triggering*, and *Detection*, disentangling failure modes that prior end-state scoring conflates.

Experimental results demonstrate that GUIJudge achieves reliable process-aware evaluation, outperforming all baseline evaluators with 97.12% Recall at the Detect stage. GUITestScape further reveals that detection remains the critical bottleneck for existing models, with the best-performing model reaching only 40.55% Recall. This bottleneck can be effectively addressed by integrating GUIJudge's verifiers into existing agents without retraining. Our contributions include:

- We present **GUITestScape**, an interactive benchmark covering 61 real-world Android applications and 508 preset defects across both interaction and display types.
- We propose a **three-stage capability decomposition** that disentangles exploratory GUI testing into Reaching, Triggering, and Detection as independently diagnosable dimensions.
- We introduce **GUIJudge**, an open-set evaluator that enables process-aware evaluation beyond predefined defect annotations.

2 Related Works

2.1 Automated GUI Testing

Automated GUI testing has long relied on random, heuristic, or UI-guided exploration. Tools such as Monkey (Choudhary et al., 2015) and DroidBot (Li et al., 2017) can efficiently traverse application interfaces, but their testing logic is mainly driven by event generation and UI structure rather than app intent or defect semantics. Recent work has introduced MLLM-based agents into GUI testing. ScenGen (Yu et al., 2026), VisionDroid (Liu et al., 2024), GUITester (Gao et al., 2026b), and SpecOps (Ahmed et al., 2026) move testing toward scenario-guided, vision-driven, or defect-oriented exploration. GUITestScape follows this direction by providing a unified benchmark for evaluating GUI agents in exploratory GUI testing scenarios.

2.2 Benchmarks for GUI Testing

Existing GUI benchmarks such as AndroidWorld (Rawles et al., 2025) and OSWorld (Xie et al., 2024b) mainly evaluate task completion. GUI testing benchmarks instead target GUI testing capabilities: GUI Testing Arena (Zhao et al., 2024) structures evaluation around test intent generation, task execution, and defect detection; GUITester (Gao et al., 2026a) introduces GUITestBench for exploratory defect discovery; and ScenGen (Yu et al., 2026) emphasizes scenario-based test construction. Compared with these benchmarks, GUITestScape provides a more comprehensive benchmark that covers both interaction and display defects.

2.3 GUI Trajectory Evaluation

Evaluating exploratory GUI trajectories is difficult because valid trigger paths are not unique and defect evidence may appear only in short trajectory intervals. Rule-based methods, including those used in GUITester (Gao et al., 2026b) and WebArena (Zhou et al., 2024), rely on pre-

defined checks or golden-path matching, making them reliable in closed-set settings but difficult to generalize. LLM-as-Judge methods have been studied in related agent evaluation settings such as Mind2Web (Deng et al., 2023), AgentRewardBench (Lù et al., 2025), and Beyond Binary Rewards (Damani et al., 2025). However, holistic judging over long trajectories provides limited process diagnosis. GUIJudge instead localizes defect-relevant segments and verifies defect evidence with verifiers designed for different evidence forms, enabling process-aware evaluation of the testing process.

3 GUITestScape

This section presents GUITestScape, an interactive benchmark for evaluating agents in exploratory GUI testing. Figure 2 provides an overview of its construction pipeline. We first describe the collection and categorization of real-world defects, covering both display and interaction types (§3.1). We then detail the per-defect composition that constitutes each evaluation case, comprising a resettable scenario, a state-level test basis and multiple navigation tasks (§3.2). Finally, we report the scale and distribution of GUITestScape (§3.3).

3.1 Defect Collection and Categorization

We collect GUI defects from public issue trackers and bug reports of real-world Android applications. Building on the defect taxonomy of GTArena (Zhao et al., 2024), we organize the collected defects into two categories, display defects and interaction defects.

Display Defects are visual abnormalities embedded in the rendered interface itself, present regardless of whether the user interacts with the screen. Detecting them requires open-ended visual anomaly perception, since no behavioral cue signals their presence and the agent must judge the screen on visual evidence alone. We further divide them into two fault modes. (1) *Content Rendering* defects (CR) concern the abnormal presentation of textual or visual content, such as broken icons or garbled text. (2) *Element Layout* defects (EL) concern abnormalities in spatial organization, such as overlapping elements or alignment errors.

Interaction Defects are anomalies that surface from the interplay between an agent's actions and the application's responses, and detecting them requires reasoning about the logical consistency of the resulting interaction trajectory rather than visual evidence alone. We adopt three fault modes from GUITester (Gao et al., 2026a). (1) *Operation No Response* (ONR), where an interaction yields no observable feedback. (2) *Unexpected Task Result* (UTR), where the outcome of a multi-step operation deviates from its intended effect. (3) *Navigation Logic Error* (NLE), where flawed transitions lead to incorrect navigation flow.

3.2 Benchmark Composition

A benchmark requires a dynamic case in which an agent can actually encounter, trigger, and judge a defect. We instantiate each defect into such a case along three dimensions, each addressing a distinct requirement of evaluation: (1) a resettable scenario (§3.2.1) that restores the same starting environment before every run; (2) test basis (§3.2.2) that provides the criteria for judging whether a defect is genuinely triggered; and (3) multiple navigation tasks (§3.2.3) that simulate the diverse interaction patterns of real testers

3.2.1 Resettable Scenario

Whether a defect remains reproducible depends on the application's runtime context (Romano et al., 2021), and uncontrolled drift in this context makes evaluation unstable across runs. We therefore wrap each defect in a resettable evaluation scenario that fixes its prerequisites for triggering. Beyond the target application, we specify the defect-relevant initial conditions under which testing begins, including pre-populated application data and required system-side settings. These conditions are restored before every evaluation, so that different agents are tested from an identical starting point when probing the same defect. Consequently, evaluation results reflect the agent's testing competence rather than environmental contingency.

3.2.2 State-level Test Basis

Exploratory GUI testing conducted by human testers does not rely solely on generic interaction experience. Instead, testers interpret interaction results according to a test basis (Schieferdecker, 2025), which captures the design specifications and business logic of the target application. For example, when a feedback form returns the user to the home screen after submission, a tester familiar with the app's design rationale recognizes this as the intended streamlined flow, whereas the same transition in an app that should keep the user on

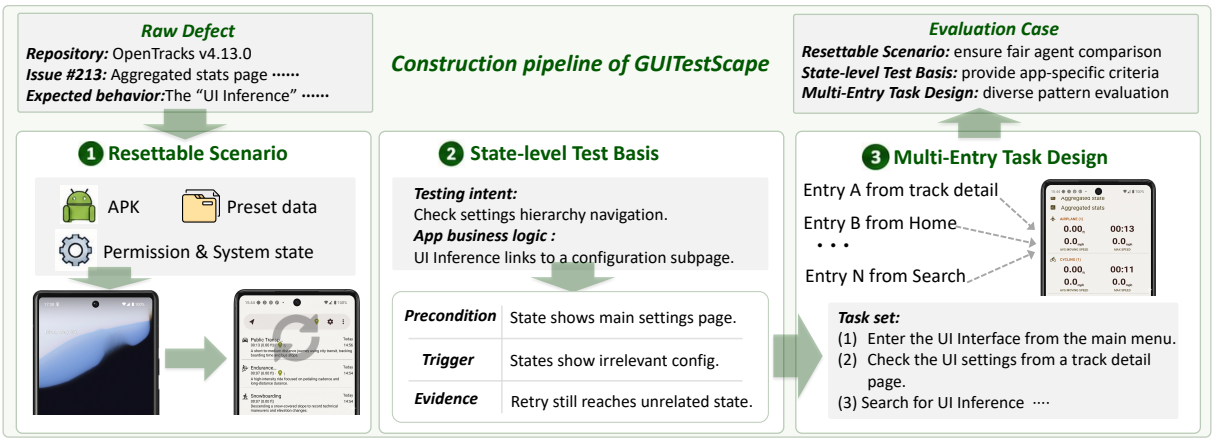


Figure 2: Construction pipeline of GUITestScape. Each real-world defect is instantiated into an evaluation case with a resettable scenario, a state-level test basis, and a multi-entry task set.

the editable page would be flagged as a navigation logic error. Without access to such app-specific semantics, an agent has no consistent ground for judging across applications.

For each defect in GUITestScape, we provide state-level test basis based on the target application's business logic and testing intent. It specifies which states demonstrate that the necessary preconditions have been established, which states indicate that the defect has been triggered, and which states provide verifiable evidence of the resulting anomaly. Rather than prescribing a unique action trajectory, the state-level test basis keeps navigation paths open and only fixes the state-level checkpoints that any valid trigger process must traverse. This design provides the app-specific test basis that generic interaction priors cannot cover, while remaining permissive enough to accommodate diverse interaction patterns.

3.2.3 Multi-Entry Task Design

Human testers exploring an application rarely approach a defect through a single fixed path; they typically vary entry points and interaction sequences to rule out coincidental triggers and to expose the defect under different operational conditions. To preserve the exploratory nature of testing, GUITestScape pairs each defect with a set of navigation tasks derived from its state-level test basis. Specifically, we leverage the entry conditions, result-relevant states, and inspection points encoded in the test basis to instantiate tasks that approach the same defect-relevant context from different functional entry points and under varying levels of task guidance. The defect therefore remains the evaluation target, while the path leading to it becomes a controlled source of variation, exposing whether an agent's testing competence generalizes across interaction patterns rather than being tied to any single predefined route.

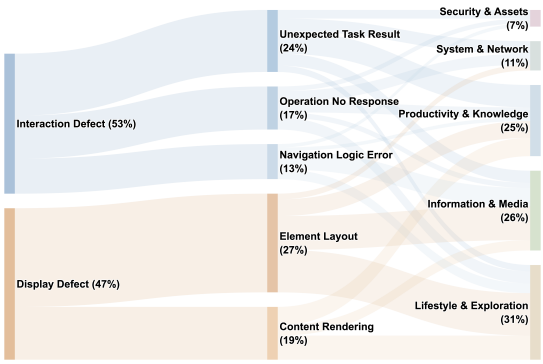


Figure 3: Hierarchical distribution of preset defects across defect types, fault modes, and application categories in GUITestScape.

3.3 Benchmark Statistics

GUITestScape collects 508 preset defects from 61 real-world Android applications across 5 major categories. We consolidate these defects into 82 resettable scenarios, annotate each with state-level test basis, and expand them into 384 navigation tasks. Figure 3 shows the distribution across defect types and application categories, with defects nearly evenly split between interaction (53%) and display (47%) types.

4 GUIJudge

Building on the evaluation cases provided by GUITestScape, we introduce GUIJudge, our evaluator

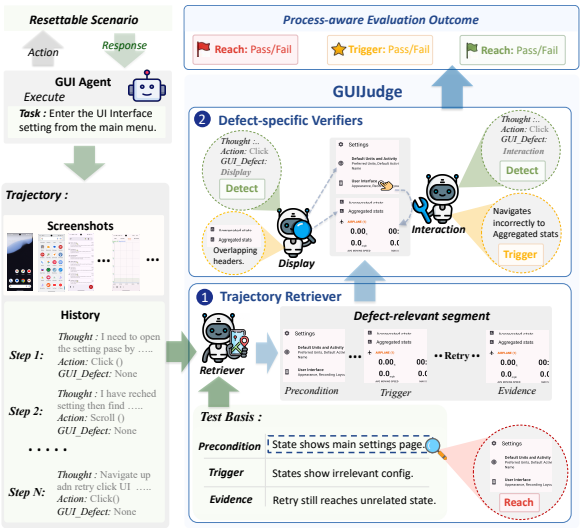


Figure 4: GUIJudge Workflow. Given an agent trajectory and the state-level test basis of the corresponding evaluation case, GUIJudge first retrieves defect-relevant segments and then applies two defect-specific verifiers to produce process-aware judgments across three dimensions: Reach, Trigger, and Detect.

for exploratory GUI testing. We first define three evaluation dimensions that decompose defect discovery into diagnosable capabilities (§4.1), then detail the evaluation pipeline of GUIJudge, including a trajectory retriever and two defect-specific verifiers (§4.2). Figure 4 provides an overview of the GUIJudge workflow.

4.1 Evaluation Targets

Defect discovery unfolds in three stages: an agent must first reach the defect's location, then trigger the defect through interaction, and finally recognize the resulting anomaly. A breakdown at any stage halts this progression and leaves the defect unreported. Yet existing end-state evaluation collapses the entire progression into a single verdict, masking the capability gap that actually caused the failure. GUIJudge therefore evaluates each stage as an independent dimension.

- ***Reaching*** evaluates whether an agent correctly interprets the testing task and navigates to the precondition state required for defect exposure.
- ***Triggering*** evaluates whether an agent, having reached the defect-relevant context, performs the interaction that surfaces the defect. This dimension applies only to interaction defects.
- ***Detection*** evaluates whether an agent, once the defect is exposed, recognizes the anomaly and reports it as a GUI defect.

4.2 Two-Stage Evaluation Pipeline

Evaluating an agent's testing trajectory faces two practical obstacles. First, exploratory trajectories interleave defect-relevant moments with extensive irrelevant navigation, making end-to-end judgment prone to evidence dilution. Second, display and interaction defects manifest through fundamentally different forms of evidence, precluding a single unified verification procedure. These two challenges motivate the design of the trajectory retriever (§4.2.1) and the defect-specific verifiers (§4.2.2).

4.2.1 Trajectory Retriever

To reduce trajectory noise caused by defect-irrelevant steps, GUIJudge first retrieves defect-relevant segments from the full trajectory. The retriever performs the Reach judgment based on the state-level test basis of the corresponding evaluation case. Specifically, it localizes segments corresponding to the precondition states required for defect exposure. If such segments are identified, GUIJudge considers the agent to have reached the target context; otherwise, the defect discovery process fails before exposure. The retrieved segments are not treated as final defect judgments. Instead, they are passed to defect-specific verifiers, which further judge whether the defect is triggered and whether the agent reports the anomaly correctly.

4.2.2 Defect-specific Verifiers

Since display and interaction defects rely on different evidence forms, GUIJudge applies defect-specific verifiers to the retrieved defect-relevant segments. For display defects, the verifier examines GUI states for visual anomalies, such as garbled text, broken images, overlapping elements, or alignment errors. For interaction defects, the verifier examines action-state transitions and checks whether the observed response is consistent with the task expectation and the app-specific business logic. The verifiers judge *Trigger* by determining whether the target defect is actually exposed, and judge *Detect* by checking whether the agent's report is consistent with the verified defect evidence.

5 Experiments

We conduct comprehensive experiments to answer the following research questions:

- **RQ1**: Does GUIJudge yield valid and reliable judgments of GUI testing outcomes?

Method	Stage	Display Defects				Interaction Defects						Overall	
		CR		EL		NLE		ONR		UTR		Overall	
		Recall↑	F1↑	Recall↑	F1↑	Recall↑	F1↑	Recall↑	F1↑	Recall↑	F1↑	Recall↑	F1↑
GUIJudge (Process-Aware Performance)													
GUIJudge	Reach	100.00	94.34	100.00	100.00	100.00	98.25	100.00	100.00	100.00	97.03	100.00	97.87
GUIJudge	Trigger	100.00	94.34	100.00	100.00	100.00	96.77	100.00	100.00	100.00	94.92	100.00	97.12
GUIJudge	Detect	**100.00**	**89.36**	**96.00**	**92.31**	**100.00**	**96.77**	**100.00**	**100.00**	**92.31**	**94.12**	**97.12**	**93.95**
Ablation of GUIJudge Design													
w/o Retrieval	Detect	85.71	72.39	70.67	66.69	75.56	78.02	94.12	90.39	65.38	69.58	76.92	74.23
w/o Typed Verifiers	Detect	95.24	90.91	84.00	89.37	80.00	85.71	82.35	90.32	65.38	79.07	80.77	87.05
Baseline Evaluators (Detect Only)													
GUI-Tester	Detect	–	–	–	–	86.67	89.66	62.51	76.92	72.58	78.26	73.21	81.19
Rule-based	Detect	–	–	–	–	86.67	92.68	68.75	81.48	60.13	71.43	69.64	80.41
Claude-4.6	Detect	66.67	71.67	14.67	21.94	86.67	78.79	92.16	83.15	62.82	62.41	60.26	64.14
GPT-5.4	Detect	53.97	61.24	24.12	35.91	77.78	76.91	60.78	67.91	64.13	66.51	53.85	61.88
Gemini-3.1 Flash	Detect	84.13	80.33	45.33	55.73	82.22	70.51	82.35	74.42	56.41	58.62	67.31	67.53

Table 1: Comparison of evaluation performance on the manually annotated evaluation set. The table reports GUIJudge's process-aware performance, two ablations without trajectory retriever and without type-specific verifiers, and comparisons with baseline evaluators that only support final-outcome detect evaluation. In the Detect block, **bold** marks the best result in each column, and underline marks the second-best. "–" indicates that the corresponding evaluator does not support that defect type.

- **RQ2**: What is the performance landscape of existing models on the exploratory GUI testing?
- **RQ3**: Can GUIJudge's defect verifiers improve the agent testing capability?

Accordingly, we first describe the experimental setup (§5.1), and then organize the experiments into three parts: GUIJudge validation (§5.2), model benchmarking on GUITestScape (§5.3), and verifier integration analysis (§5.4).

5.1 Experiment Setup

In this section, we describe the experimental settings and evaluation metrics used throughout our experiments.

RQ1: GUIJudge Validation. To quantitatively assess the effectiveness of GUIJudge's design and the reliability of its judgments, we construct a manually annotated evaluation set covering 91.57% of the defect scenarios in the benchmark. We compare GUIJudge against several representative evaluator baselines, including the hybrid evaluator from GUITester (Gao et al., 2026a), a rule-based evaluator, and three LLM-as-Judge models: GPT-5.4, Claude-4.6-Sonnet, and Gemini-3.1-Flash. We further conduct two ablation studies to validate the design choices in GUIJudge's pipeline, examining the respective contributions of trajectory retriever and type-specific verifiers.

RQ2: Model Benchmarking on GUITestScape. We benchmark a diverse set of existing models on GUITestScape using GUIJudge as a unified evaluator. The evaluated models span two categories: general-purpose vision-language models (General Models), including GPT-5.4, Gemini-3.1-Pro, Qwen-3-VL-Plus, and Seed-1.8; and GUI-specialized agent models (Agent Models), including UI-TARS-1.5-7B, GUI-Owl-1.5 (8B/32B), and MAI-UI-8B. This setup enables a systematic comparison of exploratory GUI testing capability across different fault modes and model categories.

RQ3: Verifier Integration Analysis. We integrate GUIJudge's defect-specific verifiers into the workflows of UI-TARS-1.5-7B, Seed-1.8, and Gemini-3.1-Pro. This allows the agents to retain their existing navigation capabilities while receiving additional support for defect recognition during execution. The resulting performance is then evaluated on GUITestScape to assess the practical benefit of verifier integration.

Evaluation Metrics. Across all experiments, we adopt Recall and F1 as the primary metrics. Recall directly reflects a model's defect discovery capability in exploratory GUI testing, while F1 guards against overestimation arising from overly aggressive anomaly reporting.

5.2 RQ1

Overall Performance. As shown in Table 1, GUIJudge achieves 97.87% and 97.12% F1 scores at both the Reach and Trigger dimensions. At the Detect dimension, GUIJudge improves overall Recall from 73.21% to 97.12% (+23.91%) and overall

Model	Stage	Display Defects				Interaction Defects						Overall	
		CR		EL		NLE		ONR		UTR		Overall	
		Recall↑	F1↑	Recall↑	F1↑	Recall↑	F1↑	Recall↑	F1↑	Recall↑	F1↑	Recall↑	F1↑
General Models													
GPT-5.4	Detect	23.91	33.85	5.23	8.29	19.67	24.74	17.33	21.49	16.67	20.34	15.75	21.48
Gemini-3.1-Pro	Detect	46.74	59.72	30.23	42.98	**47.54**	**55.24**	**41.33**	**46.62**	**27.78**	33.89	36.42	46.19
Seed-1.8	Detect	**53.26**	**65.33**	**44.19**	**56.30**	40.98	48.08	34.67	45.22	**27.78**	**35.71**	**40.55**	**51.05**
Qwen-3-VL-Plus	Detect	28.26	42.62	22.09	33.63	27.87	37.36	28.00	33.87	14.81	20.51	23.23	32.82
Agent Models													
UI-TARS-1.5-7B	Detect	5.43	9.71	1.74	3.28	29.51	37.89	8.00	10.34	5.56	7.74	7.48	11.66
GUI-Owl-1.5-8B	Detect	1.09	2.13	1.74	3.39	1.64	2.99	0.00	0.00	0.93	1.75	1.18	2.26
GUI-Owl-1.5-32B	Detect	9.78	17.65	3.49	6.63	4.92	8.70	2.67	4.82	0.00	0.00	3.94	7.30
MAI-UI-8B	Detect	1.09	2.15	1.16	2.29	8.20	14.29	9.33	15.38	3.39	6.06	4.13	7.62

Table 2: Pass@1 Detect-stage performance of general models and specialized agent models on GUITestScape. **Bold** marks the best result in each column, and underline marks the second best.

F1 from 81.19% to 93.95% (+12.76%) over the strongest baseline evaluator. These results demonstrate that GUIJudge not only reliably retrieves defect-relevant trajectory segments, but also produces accurate judgments on top of them, effectively reformulating final-only evaluation into a staged, process-aware capability assessment.

Effect of Retrieval. Removing trajectory retrieval reduces overall Detect Recall from 97.12% to 76.92%. The performance drop is smallest on ONR but largest on UTR. This is because ONR defects are typically triggered by a single action under relatively fixed verification conditions, making their evidence straightforward to localize; UTR defects, by contrast, often require cross-state comparison and delayed outcome verification, resulting in more dispersed and harder-to-localize evidence. These results indicate that evaluation reliability is fundamentally constrained by evidence localization quality, confirming that trajectory retrieval is essential for stable evaluation and constitutes a key source of GUIJudge's performance gains.

Analysis Across Defect Types. At the Detect stage, GUIJudge achieves the best performance across all defect types. By contrast, GUITester and the rule-based evaluator cannot handle display defects, while LLM-as-Judge methods exhibit unstable performance across defect categories. The ablation further reveals that replacing type-specific verifiers with a unified verifier reduces overall F1 from 93.95% to 87.05%, while also increasing performance variance across defect types. This degradation stems from the fundamentally different evidence requirements of the two defect categories: display defects are assessed through visual anomalies in interface states, whereas interaction defects are assessed through the consistency between action outcomes and task expectations. By tailoring each verifier to its corresponding evidence form, GUIJudge achieves more stable and reliable evaluation across diverse defect types.

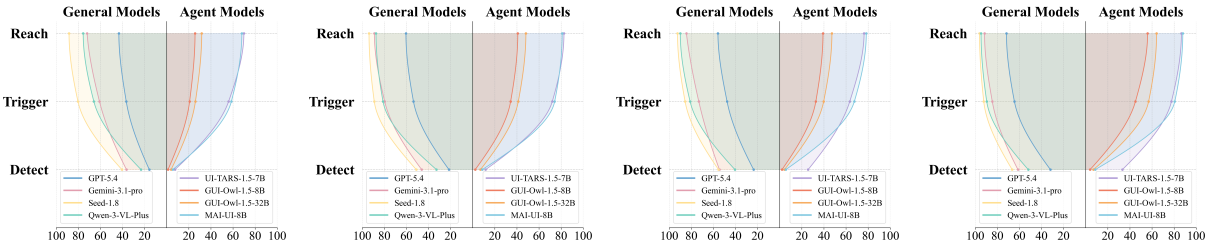
ANSWER TO RQ1

GUIJudge enables reliable process-aware evaluation, and stable evaluation requires both trajectory retrieval and type-specific verification.

5.3 RQ2

Overall Performance. Figure 5 shows that the evaluated models are generally able to trigger defects under task guidance. In particular, Seed achieves overall F1 scores of 93.93% and 89.18% at the Reach and Trigger stages, respectively. However, performance drops sharply for most models from Trigger to Detect. For example, the F1 score of UI-TARS decreases from 72.45% to 28.08%. This suggests that, in exploratory testing scenarios, the core bottleneck arises after the defect is exposed, when models fail to accurately recognize anomaly evidence and make judgments accordingly. Therefore, future research on GUI Test Agents should focus on further improving anomaly awareness, outcome verification, and defect attribution.

General Models vs. Agent Models. As shown in Figure 5 and Table 2, general models exhibit considerably smaller performance degradation from Trigger to Detect than agent models. For instance, Seed-1.8's F1 score drops from 89.18% to 64.33%, whereas UI-TARS declines more sharply from 72.45% to 28.08%. This contrast suggests that the primary gap between the two groups lies in how



(a) Overall Recall, Pass@1 (b) Overall F1, Pass@1 (c) Overall Recall, Pass@3 (d) Overall F1, Pass@3

Figure 5: Overall Recall and F1 trends across Reach, Trigger, and Detect for general models and agent models on GUITestScape under Pass@1 and Pass@3. Detailed results are provided in Tables 4 and 5 in the appendix.

they interpret abnormal states exposed during task execution. Agent models are largely optimized for task-intent following, which enables strong Reach and Trigger performance even at smaller model scales; however, once an anomaly is exposed, they tend to overlook it rather than treating it as evidence for further judgment. General models, by contrast, are more oriented toward holistic interface understanding, which makes them better equipped for state interpretation and anomaly judgment after defect exposure.

ANSWER TO RQ2

(1) General models outperform specialized GUI agent models in exploratory GUI testing, with Seed-1.8 achieving the best overall Detect performance among all evaluated models. (2) Current models struggle with anomaly recognition after defect exposure, especially when the judgment requires structural layout expectations or app specific outcome verification.

Analysis Across Fault Modes. Nearly all models perform better on CR than on EL, and also better on NLE and ONR than on UTR. CR is usually supported by directly visible evidence, such as garbled text, whereas EL further depends on whether the model can infer a reasonable page layout, which may vary substantially across apps. Similarly, NLE and ONR can often be judged from a single state transition, while UTR requires checking across states whether the expected outcome occurs under the app specific business logic. These results suggest that current models mainly lack two capabilities: (1) judging interface structural plausibility and (2) verifying delayed task outcomes under app specific business logic.

5.4 RQ3

Table 3 shows that integrating GUIJudge's defect-specific verifiers substantially improves exploratory GUI testing performance across all evaluated agents. Most notably, UI-TARS-1.5-7B improves its Recall from 7.48% to 57.02% and its F1 from 11.66% to 69.07%. Since the verifiers are incorporated without retraining the underlying interaction policy, these gains indicate that a dedicated defect verification module can effectively compensate for agent models' inherent weakness in anomaly recognition and defect judgment after defect exposure.

Agent Setting	Overall	
	Recall↑	F1↑
w/ GUIJudge Verifier		
UI-TARS-1.5-7B	57.02 (+49.54)	69.07 (+57.41)
Gemini-3.1-Pro	<u>61.27</u> (+24.85)	<u>71.07</u> (+24.88)
Seed-1.8	**70.97** (+30.42)	**79.48** (+28.43)

Table 3: Detect stage overall performance after integrating GUIJudge Verifier into each agent's workflow. Numbers in parentheses denote absolute gains over the corresponding unassisted baseline. **Bold** marks the best result, and <u>underline</u> marks the second-best.

ANSWER TO RQ3

Integrating GUIJudge's verifiers consistently improves Detect performance across all agents, confirming that defect recognition can be effectively decoupled from navigation capability and addressed through modular integration.

6 Conclusion

This paper identifies two core challenges in exploratory GUI testing evaluation: insufficient coverage of display defects and closed-set final-outcome evaluation. We introduce GUITestScape, the first interactive benchmark that jointly covers display and interaction defects, and propose GUIJudge, which enables open-set, process-aware evaluation. Experiments demonstrate the reliability of GUIJudge and further reveal that the main bottlenecks of current models in exploratory GUI testing lie in judging interface structural plausibility and verifying delayed outcomes.

Limitations

(1) **Extraction of App-specific Test Basis.** Compared with closed-set evaluation, GUITestScape substantially reduces construction costs. However, reliable defect judgment requires app-specific standards about expected behavior and interface structure. In our benchmark, these standards are provided through resettable scenarios and state-level test bases, which support reproducible and judgeable evaluation, but their construction still partly relies on human effort.

(2) **Boundary in Evaluating Test Design.** GUITestScape evaluates the execution stage of exploratory GUI testing: given a navigation task, whether an agent can reach, trigger, and detect defects during interaction. It does not yet evaluate upstream test design capabilities, such as generating testing tasks, designing boundary conditions, or planning exploration strategies for broader coverage.

(3) **Broader Platform and Workflow Coverage.** GUITestScape currently focuses on mobile Android applications. Although it covers both display and interaction defects, it does not yet cover web, desktop, or more complex cross-application workflows. These settings may involve different interaction patterns, environmental dependencies, and defect manifestations.

Future work will focus on these directions: (1) automatically extracting app-specific test bases from application documents, historical interaction traces, and normal page styles, (2) extending evaluation from test execution to test design, including testing task generation and boundary condition design, and (3) expanding GUITestScape to broader GUI platforms and more complex workflows.

References

Syed Yusuf Ahmed, Shiwei Feng, Chanwoo Bae, and Calix Barrus Xiangyu Zhang. 2026. Specops: A fully automated ai agent testing framework in real-world gui environments. *arXiv preprint arXiv:2603.10268*.

Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for android: Are we there yet? (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440.

Mehul Damani, Isha Puri, Stewart Slocum, Idan Shenfeld, Leshem Choshen, Yoon Kim, and Jacob Andreas. 2025. Beyond binary rewards: Training lms to reason about their uncertainty. *Preprint*, arXiv:2507.16806.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. 2023. Mind2web: Towards a generalist agent for the web. *Preprint*, arXiv:2306.06070.

Yifei Gao, Jiang Wu, Xiaoyi Chen, Yifan Yang, Zhe Cui, Tianyi Ma, Jiaming Zhang, and Jitao Sang. 2026a. Guitester: Enabling gui agents for exploratory defect discovery. *Preprint*, arXiv:2601.04500.

Yifei Gao, Jiang Wu, Xiaoyi Chen, Yifan Yang, Zhe Cui, Tianyi Ma, Jiaming Zhang, and Jitao Sang. 2026b. Guitester: Enabling gui agents for exploratory defect discovery. *arXiv preprint arXiv:2601.04500*.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. 2024. Webvoyager: Building an end-to-end web agent with large multimodal models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6864–6890.

Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight ui-guided test input generator for android. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, page 23–26. IEEE Press.

Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2020. Owl eyes: Spotting ui display issues via visual understanding. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, pages 398–409.

Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2022. Nighthawk: Fully automated localizing ui display issues via visual understanding. *IEEE Transactions on Software Engineering*, 49(1):403–418.

Zhe Liu, Cheng Li, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Yawen Wang, Jun Hu, and Qing Wang. 2024. Seeing is believing: Vision-driven non-crash functional bug detection for mobile apps. *Preprint*, arXiv:2407.03037.

Xing Han Lù, Amirhossein Kazemnejad, Nicholas Meade, Arkil Patel, Dongchan Shin, Alejandra Zambrano, Karolina Stańczak, Peter Shaw, Christopher J. Pal, and Siva Reddy. 2025. Agentrewardbench: Evaluating automatic evaluations of web agent trajectories. *Preprint*, arXiv:2504.08942.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, and 1 others. 2025. Ui-tars: Pioneering automated gui interaction with native agents, 2025. *URL https://arxiv.org/abs/2501.12326*.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice

Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy Lillicrap, and Oriana Riva. 2025. Androidworld: A dynamic benchmarking environment for autonomous agents. *Preprint*, arXiv:2405.14573.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, and 1 others. Androidworld: A dynamic benchmarking environment for autonomous agents, 2024. *URL https://arxiv. org/abs/2405.14573*.

Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. 2021. An empirical analysis of ui-based flaky tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1585–1597. IEEE.

Ina K. Schieferdecker. 2025. Navigating the growing field of research on ai for software testing – the taxonomy for ai-augmented software testing and an ontology-driven literature survey. *Preprint*, arXiv:2506.14640.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, and 1 others. 2024a. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. 2024b. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Preprint*, arXiv:2404.07972.

Shengcheng Yu, Yuchen Ling, Chunrong Fang, Quan Zhou, Yi Zhao, Chunyang Chen, Shaomin Zhu, and Zhenyu Chen. 2026. Scenario-guided llm-based mobile app gui testing. *Preprint*, arXiv:2506.05079.

Kangjia Zhao, Jiahui Song, Leigang Sha, Haozhan Shen, Zhi Chen, Tiancheng Zhao, Xiubo Liang, and Jianwei Yin. 2024. Gui testing arena: A unified benchmark for advancing autonomous gui testing agent. *Preprint*, arXiv:2412.18426.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pages 15585–15606.

A GUIJudge

A.1 Workflow of GUIJudge

Algorithm 1 presents the complete workflow of GUIJudge. For each evaluation record $d_i = (g_i, p_i, [s_i, e_i], K_i, c_i)$ comprising the navigation task g_i, the trajectory pointer p_i, the scenario window $[s_i, e_i]$, the ordered state-level checkpoints K_i, and the defect category c_i GUIJudge coordinates three modules: **Trajectory Retrieval**, the **Display Verifier**, and the **Interaction Verifier**. All defect-type verdicts are accumulated across records into $\mathcal{E}_{disp}$ and $\mathcal{E}_{int}$.

Trajectory Retrieval. The Trajectory Retrieval module localizes the trajectory segment relevant to the target defect by matching K_i against the candidate observations O_i inside the scenario window. When K_i contains a single key state, it scans O_i in temporal order and accepts the first observation matched by the multimodal judge f_θ. Otherwise, it sequentially aligns the first $|K_i|-1$ anchors with O_i and, only when this prefix is fully covered, searches backward from the trajectory tail for the terminal key state. The matched set m_i and the flag trigger_i together resolve the *Reach* and *Trigger* targets defined in §4.1.

Display Verifier. When the target defect is a display defect ($c_i \in \mathcal{C}_{disp}$), the Display Verifier inspects each matched observation $o \in m_i$ as a standalone GUI state and outputs its per-state defect report r. Any flagged defect is merged into $\mathcal{E}_{disp}$, realizing the *Detect* target for static visual anomalies.

Interaction Verifier. When the target defect is an interaction defect ($c_i \in \mathcal{C}_{int}$), verification proceeds only if the anchors have been fully recovered ($\text{trigger}_i = \texttt{True}$); otherwise the record is skipped. The module then extracts the transitions Δ_i inside the matched anchor span and verifies each transition (o_t, a_t, o_{t+1}) conditioned on the task metadata $\mathcal{M}_{g_i}$, the anchors K_i, and a rolling interaction history H. Detected defects are merged into $\mathcal{E}_{int}$, and H is updated with the executed action and its observed effect so that subsequent transitions are judged with full context.

The loop terminates once every record in $\mathcal{D}$ has been processed, and GUIJudge returns the aggregated defect set $\mathcal{E}_{disp} \cup \mathcal{E}_{int}$.

A.2 Prompt

Each module of GUIJudge is driven by a dedicated prompt. The prompt used by the Trajectory Re-

Algorithm 1: GUIJudge Workflow.

Input: evaluation records $\mathcal{D}$, agent trajectories $\mathcal{T}$, task metadata $\mathcal{M}$
Output: detected GUI defects $\mathcal{E} = \mathcal{E}_{disp} \cup \mathcal{E}_{int}$
Init: $\mathcal{E}_{disp} \leftarrow \emptyset$, $\mathcal{E}_{int} \leftarrow \emptyset$;

foreach *record* $d_i \in \mathcal{D}$ **do**
  $(g_i, p_i, [s_i, e_i], K_i, c_i) \leftarrow d_i$;
  if d_i *invalid* ***or*** $p_i \notin \mathcal{T}$ **then continue**;
  $\tau_i \leftarrow$ trajectory at p_i; // observation-action sequence
  // trajectory retrieval
  $O_i \leftarrow$ observations of τ_i in $[s_i, e_i]$; // candidate states
  $m_i \leftarrow \emptyset$; $j \leftarrow 1$; // matched set; scan pointer
  if $|K_i| = 1$ **then**
    foreach $o \in O_i$ *in temporal order* **do**
      if $f_\theta(o,\ K_i[1])$ **then** $m_i \leftarrow \{o\}$; **break**;
  else
    for $q \leftarrow 1$ **to** $|K_i| - 1$ **do**
      while $j \leq |O_i| \wedge \neg f_\theta(O_i[j],\ K_i[q])$ **do** $j \leftarrow j{+}1$;
      if $j > |O_i|$ **then break**;
      $m_i \leftarrow m_i \cup \{O_i[j]\}$; $j \leftarrow j + 1$;
    if $|m_i| = |K_i| - 1$ **then**
      for $k \leftarrow |O_i|, \ldots, j$ **do**
        if $f_\theta(O_i[k],\ K_i[|K_i|])$ **then** $m_i \leftarrow m_i \cup \{O_i[k]\}$; **break**;
  $\texttt{trigger}_i \leftarrow (|m_i| = |K_i|)$;
  // display defect verification
  if $c_i \in \mathcal{C}_{disp}$ **then**
    foreach *matched observation* $o \in m_i$ **do**
      $r \leftarrow f_\theta(o)$; // judge rendering / layout
      if r.has_defect **then** $\mathcal{E}_{disp} \leftarrow \mathcal{E}_{disp} \cup r$.defects;
  // interaction defect verification
  if $c_i \in \mathcal{C}_{int}$ **then**
    if $\neg\,\texttt{trigger}_i$ **then continue**;
    $\Delta_i \leftarrow$ TRANSITIONS$(\tau_i,\ m_i)$; // steps within the matched defect-relevant span
    $H \leftarrow \emptyset$; // interaction history
    foreach *transition* $(o_t, a_t, o_{t+1}) \in \Delta_i$ **do**
      $r_t \leftarrow f_\theta(\mathcal{M}_{g_i},\ K_i,\ H,\ (o_t, a_t, o_{t+1}))$;
      if r_t.has_defect **then** $\mathcal{E}_{int} \leftarrow \mathcal{E}_{int} \cup r_t$.defects;
      $H \leftarrow H \cup \{(a_t,\ r_t.\texttt{effect})\}$;
return $\mathcal{E}_{disp} \cup \mathcal{E}_{int}$;

trieval module is shown in Table 6, the prompt used by the Display Verifier is shown in Table 7, and the prompt used by the Interaction Verifier is shown in Table 8.

B Additional Experimental Results

B.1 Process-Level Metric: Pass@1

Table 4 reports stage-wise Pass@1 results on GUI-TestScape, covering the Reach, Trigger, and Detect stages across five defect subtypes (CR, EL, NLE, ONR, UTR) for both general models and specialized agent models.

B.2 Process-Level Metric: Pass@3

Table 5 reports stage-wise Pass@3 results on GUI-TestScape, covering the Reach, Trigger, and Detect stages across five defect subtypes (CR, EL, NLE, ONR, UTR) for both general-purpose VL models and specialized GUI agent models.

C More Cases

C.1 Display Defect, Element Layout

Figure 6 shows an Element Layout defect on a music application. The task asks the agent to open the settings page through the navigation drawer. Once the settings list is reached, the icons for the *Appearance Settings* and *Local music paths* entries are noticeably larger than the icons of the other rows, breaking the uniform icon-size and text-start alignment that the rest of the list maintains. Because the anomaly is fully visible from a single rendered state, GUIJudge's Display Verifier identifies the defect without any interaction-level reasoning.

C.2 Display Defect, Content Rendering

Figure 7 shows a Content Rendering defect on a recipe application. The agent is asked to open the settings page and then tap *Preferred category*. The resulting dialog lists three options, but the third option renders as a row of garbled "tofu"/placeholder squares rather than legible text, indicating a font or character-encoding failure for that entry. The defect is exposed by a single GUI state, so the Display Verifier can identify the display defect based on a single screenshot.

C.3 Interaction Defect, Operation No Response

Figure 8 shows an Operation No Response defect on a tasks application. The agent opens the search field, types "Exercise", and taps the keyboard search button. The expected post-state is a task list filtered to entries matching the query, but the post-state is identical to the pre-state the full unfiltered list (*Submit Status Report*, *Attend Staff Meeting*, etc.) is still shown. Because the agent's action is correct ("hit" on the search key is true) yet the interface produces no observable effect, the Interaction Verifier classifies the transition as an Operation No Response defect.

Model	Type	Display Defects				Interaction Defects						Overall	
		CR		EL		NLE		ONR		UTR		Overall	
		Recall↑	F1↑	Recall↑	F1↑	Recall↑	F1↑	Recall↑	F1↑	Recall↑	F1↑	Recall↑	F1↑
Reach Stage													
GPT-5.4	General	44.57	61.65	34.30	51.08	52.46	68.82	42.67	59.81	51.85	68.29	43.31	60.44
Gemini-3.1-pro	General	80.43	89.16	83.14	90.79	77.05	87.04	78.67	88.06	75.93	86.32	79.72	88.72
Seed-1.8	General	**89.13**	**94.25**	**90.12**	**94.80**	**85.25**	**92.04**	**86.67**	**92.86**	**87.96**	**93.60**	**88.39**	**93.83**
Qwen3-VL-Plus	General	73.91	85.00	84.88	91.82	80.33	89.09	73.33	84.62	77.78	87.50	79.13	88.35
UI-TARS-1.5-7B	Agent	52.17	68.57	76.74	86.84	67.21	80.39	61.33	76.03	73.15	84.49	68.11	81.03
GUI-Owl-1.5-8B	Agent	19.78	33.03	31.98	48.46	21.31	35.14	21.33	35.16	25.93	41.18	25.64	40.82
GUI-Owl-1.5-32B	Agent	40.22	57.36	37.21	54.24	32.79	49.38	21.33	35.16	22.22	36.36	31.69	48.13
MAI-UI-8B	Agent	66.30	79.74	66.86	80.14	77.05	87.04	62.67	77.05	79.63	88.66	70.08	82.41
Trigger Stage													
GPT-5.4	General	44.57	61.65	34.30	51.08	39.34	56.47	34.67	51.49	33.33	50.00	36.61	53.60
Gemini-3.1-pro	General	80.43	89.16	83.14	90.79	57.38	72.92	54.67	70.69	39.81	56.95	66.14	79.62
Seed-1.8	General	**89.13**	**94.25**	**90.12**	**94.80**	**80.33**	**89.09**	**69.33**	**81.89**	**64.81**	**78.65**	**80.31**	**89.08**
Qwen3-VL-Plus	General	73.91	85.00	84.88	91.82	63.93	78.00	56.00	71.79	48.15	65.00	68.31	81.17
UI-TARS-1.5-7B	Agent	52.17	68.57	76.74	86.84	45.90	62.92	30.67	46.94	44.44	61.54	54.92	70.90
GUI-Owl-1.5-8B	Agent	19.78	33.03	29.65	45.74	21.31	35.14	16.00	27.59	10.19	18.49	20.71	34.31
GUI-Owl-1.5-32B	Agent	40.22	57.36	37.21	54.24	29.51	45.57	10.67	19.28	4.63	8.85	25.98	41.25
MAI-UI-8B	Agent	66.30	79.74	66.86	80.14	60.66	75.51	48.00	64.86	49.07	65.84	59.45	74.57
Detect Stage													
GPT-5.4	General	23.91	33.85	5.23	8.29	19.67	24.74	17.33	21.49	16.67	20.34	15.75	21.48
Gemini-3.1-pro	General	46.74	59.72	30.23	42.98	**47.54**	**55.24**	**41.33**	**46.62**	27.78	33.89	36.42	46.19
Seed-1.8	General	**53.26**	**65.33**	**44.19**	**56.30**	40.98	48.08	34.67	45.22	**27.78**	**35.71**	**40.55**	**51.05**
Qwen3-VL-Plus	General	28.26	42.62	22.09	33.63	27.87	37.36	28.00	33.87	14.81	20.51	23.23	32.82
UI-TARS-1.5-7B	Agent	5.43	9.71	1.74	3.28	29.51	37.89	8.00	10.34	5.56	7.74	7.48	11.66
GUI-Owl-1.5-8B	Agent	1.09	2.13	1.74	3.39	1.64	2.99	0.00	0.00	0.93	1.75	1.18	2.26
GUI-Owl-1.5-32B	Agent	9.78	17.65	3.49	6.63	4.92	8.70	2.67	4.82	0.00	0.00	3.94	7.30
MAI-UI-8B	Agent	1.09	2.15	1.16	2.29	8.20	14.29	9.33	15.38	3.39	6.06	4.13	7.62

Table 4: Stage-wise Pass@1 performance on GUITestScape across Reach, Trigger, and Detect, broken down by defect subtype (CR, EL, NLE, ONR, UTR) for general models and specialized agent models. **Bold** marks the best result in each column within each stage, and underline marks the second-best.

C.4 Interaction Defect, Navigation Logic Error

Figure 9 shows a Navigation Logic Error on a social-image application. The task asks the agent to inspect the *Report Pin* reason list and then dismiss it by tapping the "X" button in the top-left corner. The expected behavior is to close only the report dialog and return to the previous pin-detail context; instead, the application returned to the main interface, performing a full navigation reset rather than a dialog dismissal. The Interaction Verifier flags this abnormal navigation path as a Navigation Logic Error.

C.5 Interaction Defect, Unexpected Task Result

Figure 10 shows an Unexpected Task Result defect on a stock-information application. The agent is asked to switch the *News* tab into Chinese by tapping the language button at the top-right and selecting "Simplified Chinese". After the selection, the language indicator visibly updates from *EN* to *ZH*, suggesting the preference was accepted, yet the actual interface content news headlines, navigation tabs, and body text remains entirely in English. The post-state contradicts the task intent, so the Interaction Verifier classifies the case as an Unexpected Task Result.

Model	Type	Display Defects				Interaction Defects						Overall	
		CR		EL		NLE		ONR		UTR		Overall	
		Recall↑	F1↑	Recall↑	F1↑	Recall↑	F1↑	Recall↑	F1↑	Recall↑	F1↑	Recall↑	F1↑
Reach Stage													
GPT-5.4	General	58.70	73.98	47.67	64.54	64.75	78.57	63.89	77.99	56.00	68.19	55.91	71.72
Gemini-3.1-pro	General	86.96	93.02	86.63	92.83	83.61	91.07	82.67	90.51	80.56	89.23	84.45	91.57
Seed-1.8	General	**92.39**	**96.05**	**93.02**	**96.39**	**88.52**	**93.91**	**90.67**	**95.10**	**95.37**	**97.63**	**92.52**	**96.11**
Qwen3-VL-Plus	General	89.13	94.25	91.86	95.76	85.25	92.04	88.00	93.62	91.67	95.65	89.96	94.72
UI-TARS-1.5-7B	Agent	61.96	76.51	83.72	91.14	73.77	84.91	66.67	80.00	85.19	92.00	76.38	86.61
GUI-Owl-1.5-8B	Agent	32.61	49.18	45.93	62.96	36.07	53.01	34.67	51.49	40.74	57.89	39.17	56.02
GUI-Owl-1.5-32B	Agent	55.43	71.83	51.16	67.69	45.90	62.92	34.67	51.49	37.04	54.05	47.24	64.16
MAI-UI-8B	Agent	77.17	87.12	76.16	86.47	81.97	90.09	69.33	81.89	87.04	93.07	78.35	87.86
Trigger Stage													
GPT-5.4	General	58.70	73.98	47.67	64.54	50.82	67.39	45.33	62.39	43.52	60.65	47.64	64.52
Gemini-3.1-pro	General	86.96	93.02	86.63	92.83	67.21	80.39	64.00	78.05	50.00	66.67	73.23	84.55
Seed-1.8	General	**92.39**	**96.05**	**93.02**	**96.39**	**80.33**	**89.09**	**77.33**	**87.22**	**76.85**	**86.91**	**85.63**	**92.26**
Qwen3-VL-Plus	General	89.13	94.25	91.86	95.76	73.77	84.91	73.33	84.62	66.67	80.00	81.10	89.57
UI-TARS-1.5-7B	Agent	61.96	76.51	83.72	91.14	50.82	67.39	40.00	57.14	55.56	71.43	63.39	77.59
GUI-Owl-1.5-8B	Agent	32.61	49.18	43.02	60.33	31.15	47.46	26.67	42.11	18.52	31.25	32.68	49.26
GUI-Owl-1.5-32B	Agent	55.43	71.83	51.16	67.69	39.34	56.47	18.67	31.46	13.89	24.39	39.76	56.92
MAI-UI-8B	Agent	77.17	87.12	76.16	86.47	62.30	76.77	52.00	68.42	59.26	74.42	67.52	80.61
Detect Stage													
GPT-5.4	General	34.78	45.71	11.05	17.34	30.33	37.76	28.00	33.87	27.78	33.33	23.62	31.95
Gemini-3.1-pro	General	**66.67**	**76.03**	41.10	55.81	77.50	72.09	**90.91**	75.47	**73.58**	63.93	61.36	66.46
Seed-1.8	General	58.02	69.63	**66.67**	**76.63**	**79.59**	**82.11**	77.78	**80.77**	69.88	**75.32**	**68.59**	**76.37**
Qwen3-VL-Plus	General	39.19	54.21	36.60	51.38	70.45	73.81	77.55	72.38	57.14	61.54	49.74	60.25
UI-TARS-1.5-7B	Agent	14.58	22.95	9.77	17.11	58.06	59.02	66.67	43.84	46.43	47.27	27.40	35.01
GUI-Owl-1.5-8B	Agent	1.63	3.18	2.33	4.46	2.46	4.60	1.33	2.48	1.39	2.63	2.05	3.92
GUI-Owl-1.5-32B	Agent	10.33	18.24	4.65	8.89	6.56	11.65	4.00	7.41	1.39	2.63	4.25	7.92
MAI-UI-8B	Agent	1.69	3.33	1.64	3.23	8.33	14.63	16.22	24.49	5.56	9.84	4.47	8.24

Table 5: Stage-wise Pass@3 performance on GUITestScape across Reach, Trigger, and Detect, broken down by defect subtype (CR, EL, NLE, ONR, UTR) for general models and specialized agent models. **Bold** marks the best result in each column within each stage, and underline marks the second-best.

You are an advanced Android GUI trajectory retrieval expert. Your task is **not** to judge whether a defect occurred, but to determine whether the current screenshot has already reached/presented the given key page state.

Page Key Point: `{current_kp}`

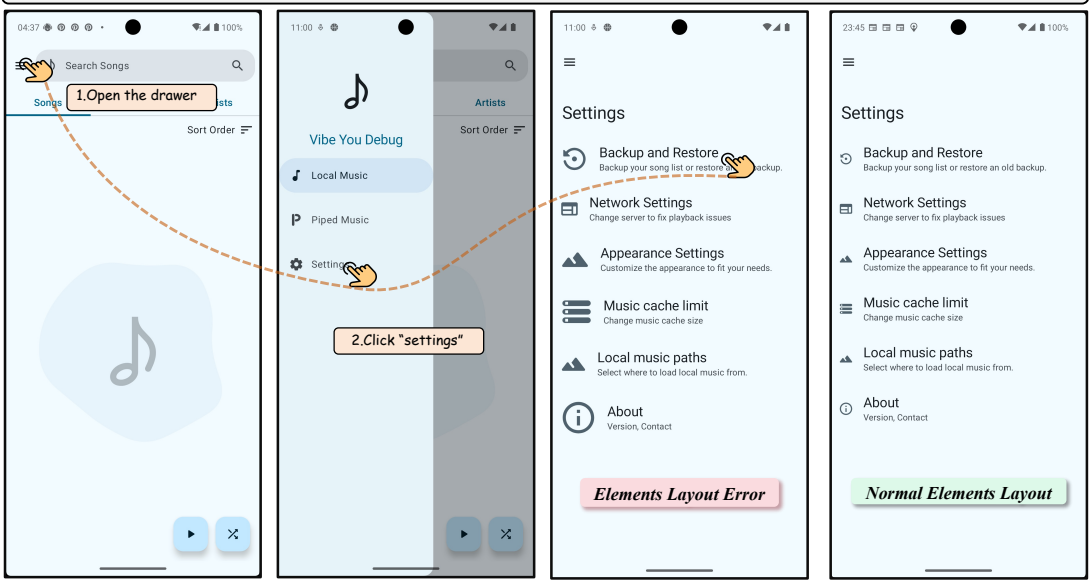
- Perform **page state matching only**: check whether the page, pop-ups, menus, lists, main titles, core controls, and visible content in the current screenshot are semantically consistent with the key point.
- Exact text match is **not** required. As long as the page type, main UI structure, key controls, or key content basically match the description, judge `matched=true`.
- If the key point describes a pop-up, menu, settings page, detail page, list page, or editing page, judge `matched=true` whenever that core interface is currently visible even with minor occlusions, toasts, differing selection states, scroll positions, or language/format details.
- If the screenshot is the same core page before/after this key state and the system is already within the relevant functional flow, also judge `matched=true`.
- Only judge `matched=false` when the current screenshot clearly belongs to an unrelated app, unrelated page, error page, or splash screen, or is missing the core page/control/content described in the key point.

Output: Strict JSON only, must include `matched` (boolean) and `reason`:

```
{
"matched": true/false,
"reason": "..."
}
```

Table 6: Retrieval Prompt

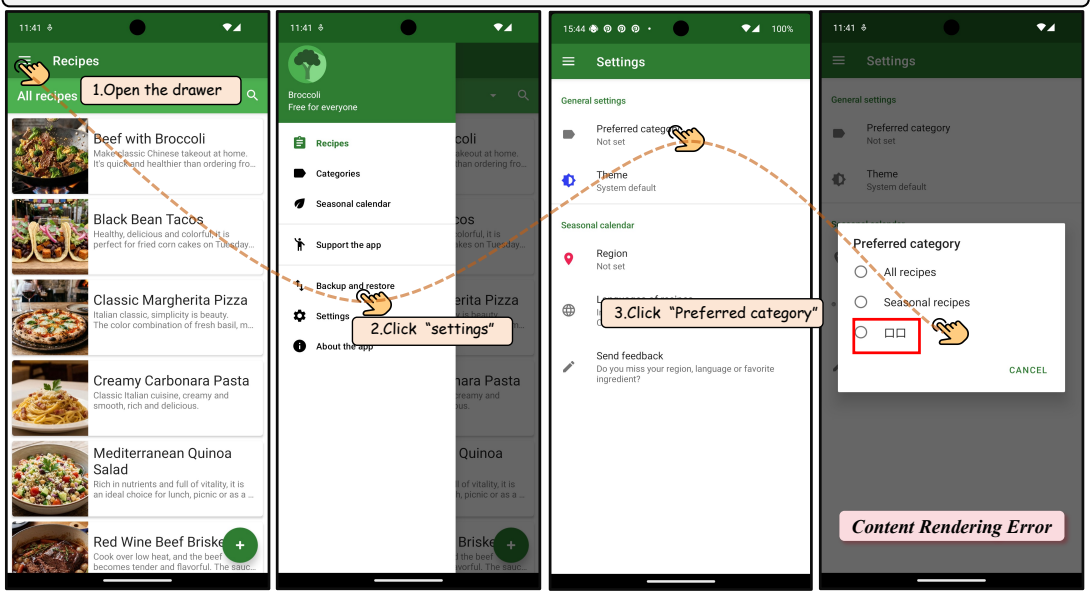
Task: Open the navigation drawer, then access the settings interface through the navigation bar to view all available setting options.
Defect : The icons on the settings page are misaligned and fail to maintain consistent positioning with their corresponding text labels.



Judgement: The settings menu exhibits distinct layout inconsistencies within its list-item components. The leading icons for each setting option vary significantly in size and aspect ratio (the "Music cache limit" icon appears horizontally stretched, while the "Appearance Settings" icon is much smaller). This irregular text alignment disrupts the uniform layout of the local module, resulting in a jagged and unpolished visual experience.

Figure 6: Example of DD-EL (Element Layout) defect. The task requires opening the settings page via the navigation drawer. After reaching the settings list, the icons of the *Appearance Settings* and *Local music paths* entries appear noticeably larger than the icons of the other rows, breaking the uniform icon-size and text-start alignment that the rest of the list maintains and exposing an element-layout anomaly.

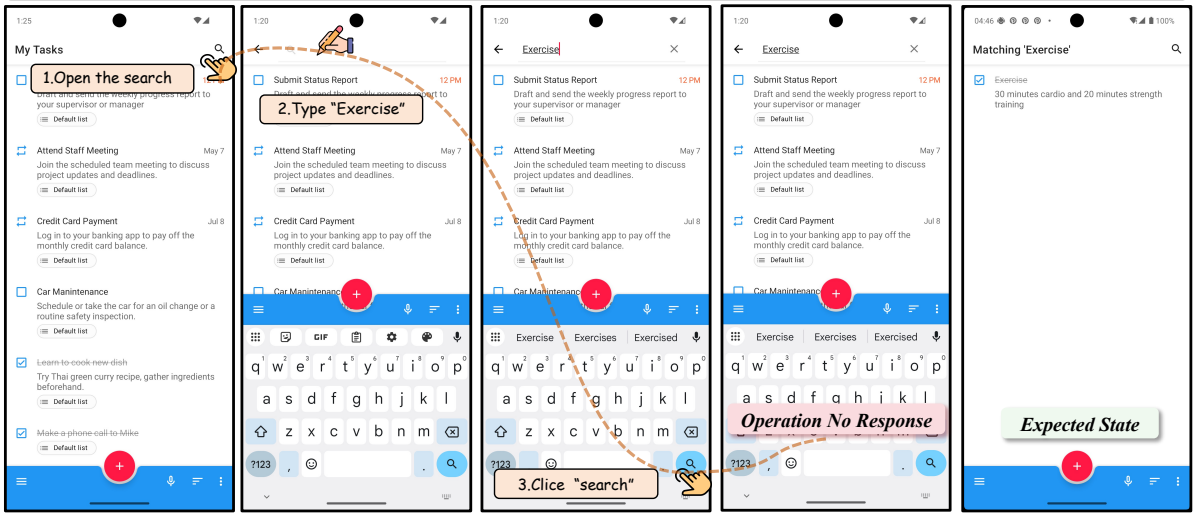
Task: Open the navigation drawer, go to the settings page, then click on 'Preferred category' to view the available category options.
Defect : The third option in the selection dialog displays as garbled square characters, failing to render the intended text correctly.



Judgement: The third option in the 'Preferred category' dialog displays two square placeholder blocks instead of readable text, indicating that the app's font does not support the required characters or there is an underlying text encoding issue affecting this specific list item.

Figure 7: Example of DD-CR (Content Rendering) defect. The task requires opening the settings page and then tapping *Preferred category*. The resulting dialog lists three options, but the third option renders as a row of garbled placeholder squares instead of legible text, indicating a font or character-encoding failure for that entry.

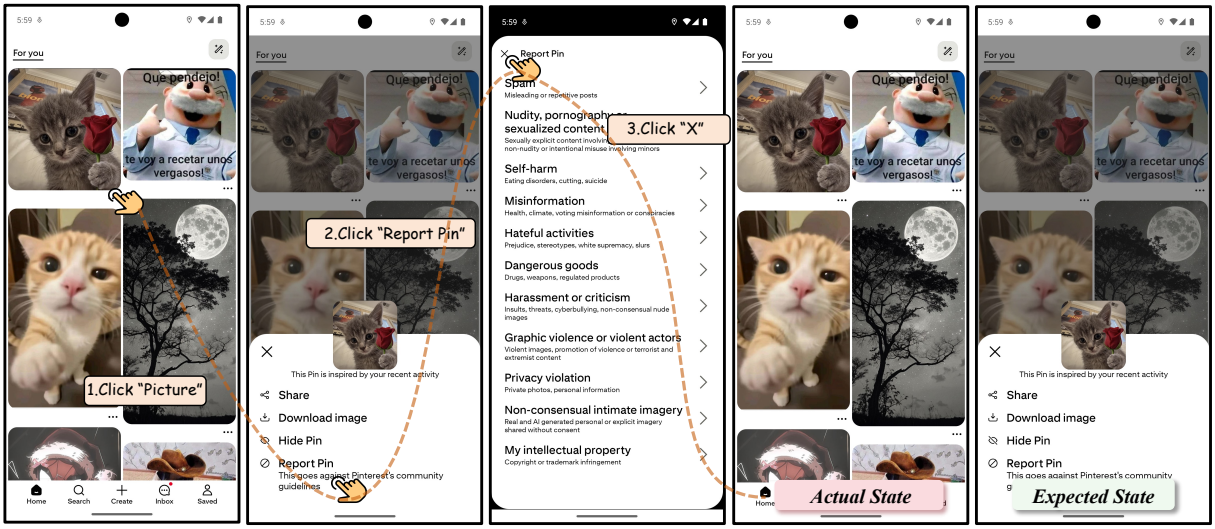
Task: Click Search, type 'Exercise', then press Enter to start the search.
Defect : After pressing search, the task list should filter to show only tasks matching 'Exercise'. Instead, the post-state shows the same unfiltered list



Judgement: Agent correctly hit the keyboard search button (magnifying glass) with Hit=True. After pressing search, the task list should filter to show only tasks matching 'Exercise'. Instead, the post-state shows the exact same unfiltered list (Submit Status Report, Attend Staff Meeting, Credit Card Payment, Car Maintenance) with no change whatsoever. The search function failed to filter results even after explicitly triggering the search action, confirming the app's search functionality is broken.

Figure 8: Example of ID-ONR (Operation No Response) defect. The task requires opening the search field, typing "Exercise", and tapping the keyboard search button to filter the list. After the search action, the task list remains identical to the pre-state with the full unfiltered list still shown, demonstrating that the search interaction produces no observable feedback or state change despite a correct hit on the search key.

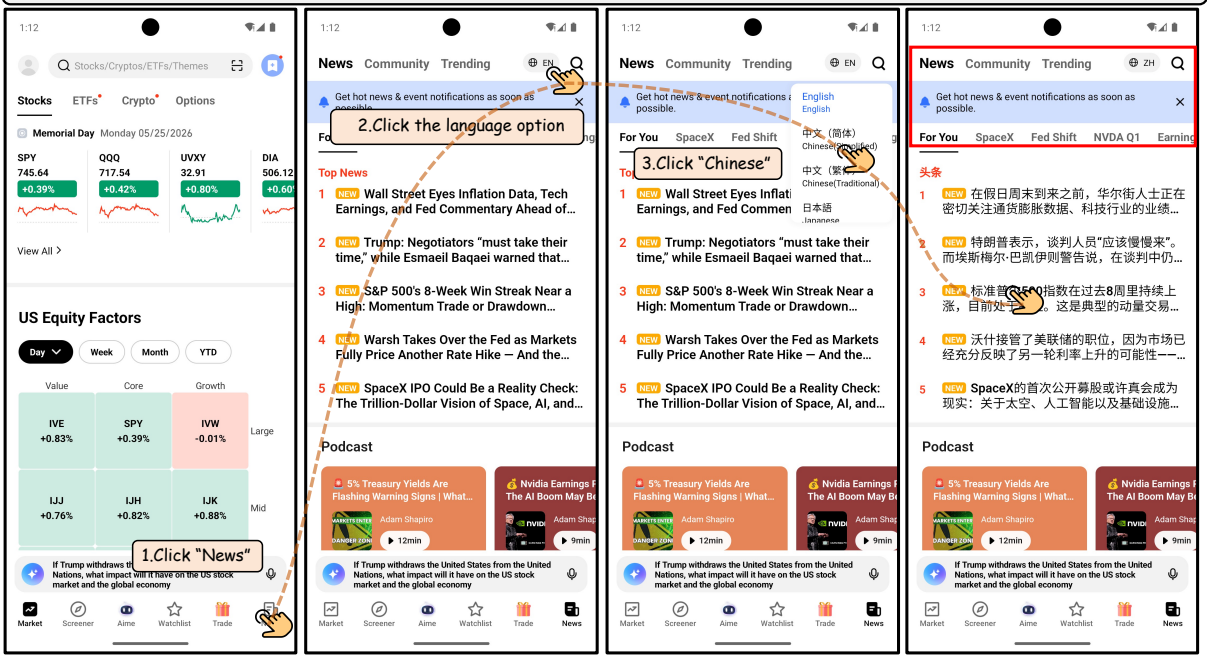
Task: Confirm if there is an option for Japanese law on the 'Report Pin' page, and then help me click the 'X' in the top-left corner to exit."
Defect : 'Report Pin' Reason Selection Page Returned Abnormally, jumping back to the home feed instead of the previous screen.



Judgement:"Clicking the 'X' close button on the 'Report Pin' dialog should dismiss the dialog and return the user to the pin detail page or the previous context where the report was initiated. Instead, the app navigated all the way back to the main 'For you' feed home screen. This is an abnormal return path — closing a report dialog should not cause a full navigation reset to the home feed."

Figure 9: Example of ID-NLE (Navigation Logic Error) defect. The task requires inspecting the *Report Pin* reason list and then dismissing the dialog by tapping the "X" button in the top-left corner. After tapping the close button, the app navigates all the way back to the main interface, performing an abnormal navigation reset rather than a dialog dismissal.

Task: I cannot understand English news; please help me switch to the 'News' tab and click the top right to change the language to Chinese.
Defect : The language switch in the APP is not complete, leaving all news content and UI text still in English.



Judgement:"The user clicked the 'Chinese (Simplified)' option and the selection was successful; the language button indeed changed from 'EN' to 'ZH', but the interface content did not switch to Chinese at all. All news headlines (such as 'Dow Jumps 1,300 Points', 'Stocks Rally As Oil Plunges'), navigation tabs (Market, Screener, News, etc.), and page text remained in English. This is a typical incomplete language-switching defect: the system recorded the language preference but did not actually apply it to the interface content, which does not meet the expected result of 'switching to Chinese' as required by the task."

Figure 10: Example of ID-UTR (Unexpected Task Result) defect. The task requires switching the *News* tab into Chinese by tapping the language button at the top-right and selecting "Simplified Chinese". After the selection, the language indicator updates from *EN* to *ZH*, yet the actual interface content remains entirely in English, contradicting the task intent of switching to Chinese.

You are a strict, evidence-driven Android GUI **Display Defects Detector**. You will receive one app screenshot (set-of-mark format, with red bold bounding boxes marking interactive controls) and an XML summary of the page. Your task is to determine, based on visible content in the screenshot aided by the XML, whether display defects (DD) exist.
Rules

- Only inspect what is **already visible** in the screenshot, assisted by the XML. No speculation.
- Output `has_defect=true` only if there is a clear, describable, and locatable visible anomaly belonging to one of the two defect types below; otherwise output `false`.

Defect Types

1. `DD.ContentRendering`
   - Images or icons that are missing, broken, blank, or failed to load.
   - Garbled text, abnormal characters, or placeholder values such as `null` / `None` / `undefined`.
   - Text that is occluded, clipped, or truncated, making it incomplete or unreadable.
   - The page structure clearly implies content should be present, but an obvious content gap or rendering failure is visible.
2. `DD.ElementLayout`
   - Similar elements are obviously missing, misaligned, abnormally sized, or abnormally positioned.
   - Elements overlap, drift, escape their container, or are only partially visible.
   - Within the same local module, elements that should share a left edge, center axis, or icon column are clearly not on the same line.
   - Duplicate or extraneous elements, or anomalous blank slots, disrupt the page structure.
   - An icon element is noticeably smaller or larger than its peers.

Evidence Requirements: `evidence` contains only directly visible facts, comparing the anomalous object with similar objects or local boundaries. `location_hint` gives an approximate position, e.g., *top-left*, *header*, *list item #3*, *dialog*, *bottom-right*.

Output: Strict JSON only, no Markdown or code fences:

```
{
"has_defect": true/false,
"defects": [{
"type": "None | DD.ContentRendering | DD.ElementLayout",
"evidence": ["..."],
"location_hint": "...",
"reason": "..."
}]
}
```

Table 7: Display Defect Detection Prompt

Role. You are an Android GUI interaction defect verification expert executing a *step-wise verifier*: each round judges only whether the interaction result of the **current step** constitutes an interaction defect.
You will receive: (1) candidate tasks; (2) `defect.desc` provided by eval; (3) actual observations of historical steps; (4) the current step to verify including its thought, action, target, hit, pre-state, and post-state; (5) two screenshots: `pre` and `post`.
Core Task. Based on all of the above, judge whether the result of the current step reflects an interaction defect. Prioritize answering:

1. Did the current step actually execute the interaction the user/agent intended?
2. Is the post-state consistent with the step's goal and task requirements?
3. If not, is this an **app** interaction defect rather than agent misoperation, missed hit, or task misunderstanding?

The eval-provided `defect.desc` is an auxiliary clue only and must not override direct evidence from the current step.
Allowed Defect Types

1. `ID.OperationNoResponse` — A reasonable interaction was performed, but the interface gave no expected feedback or the result that should follow did not appear.
2. `ID.NavigationLogicError` — The current step caused the app to jump to a wrong/unrelated page or incorrect navigation path, or back/forward navigation logic is clearly abnormal.
3. `ID.UnexpectedTaskResult` — The current step produced an incorrect result, wrong state, incorrect content, erroneous pop-up, missing result, unupdated state, incomplete setting toggle, or output inconsistent with task constraints.

The **step** field must be filled with the current step number if a defect is found, or 0 otherwise.
Candidate tasks: `{tasks}`. Eval defect description: `{defect_desc}`. Historical observations: `{history_context}`. Current step: `{action_desc}`.

Output: Strict JSON only, no Markdown. Return `has_defect=true` only when evidence supports an app defect; otherwise `has_defect=false` with `type=None`:

```
{
"has_defect": true/false,
"defect": {
"type": "ID.OperationNoResponse | ID.NavigationLogicError | ID.UnexpectedTaskResult |
None",
"step": 0,
"reason": "...",
"effect": "..."
}
}
```

Table 8: Interaction Defect Verification Prompt