RETIME: LEARNING WHEN AND HOW TO RECOVER GUI AGENTS FROM FAILURES

Anonymous authors
Paper under double-blind review

ABSTRACT

We present ReTIME, a framework for deciding when and how to recover GUI agents from failures. Recognizing an error does not establish whether recovery should begin immediately: further interaction may reveal useful evidence, but may also propagate the error and consume the remaining execution budget. In a controlled counterfactual study of 120 failed OSWorld trajectories, 78.3% contain a later checkpoint with higher recovery utility than the earliest critical error under the same recovery policy and interaction budget. ReTIME addresses this temporal variation with a stateful stopping policy that accumulates failure evidence, accounts for ongoing progress, and reserves a budget for recovery. Once intervention is triggered, a learned diagnosis model identifies the current recovery need and selects category-specific guidance for the execution agent. We also introduce ReTIME-Bench, comprising 672 checkpoints from 133 failure trajectories, to evaluate intervention timing and recovery diagnosis separately and jointly. ReTIME-8B achieves 56.31% timing balanced accuracy and 31.55% diagnosis accuracy. On OSWorld-Verified, it achieves 35.2% task success under a 50-step budget, compared with 26.8% for its backbone Qwen3-VL-8B-Instruct.

1 INTRODUCTION

GUI agents complete computer tasks through sequences of actions whose effects must be interpreted from a changing interface. Recent agents have improved visual grounding and action generation (Qin et al., 2025; Wang et al., 2025), but a single incorrect action can still divert a long execution from its goal. Recovery methods address this problem through verification, backtracking, and corrective execution (Zhang et al., 2026b; Wu et al., 2025a; Bu et al., 2026). These capabilities establish whether execution has gone wrong and provide ways to correct it. Their effectiveness also depends on the state from which correction begins.

The choice of that state creates a temporal decision. An unexpected interface response may initially be ambiguous; subsequent feedback can clarify whether an operation failed, remains pending, or affected the wrong target. Further interaction can therefore make a failure easier to diagnose and repair. However, continued execution can also introduce additional errors, overwrite useful state, or exhaust the budget needed for correction. Detecting a failure alone does not resolve this trade-off. A recovery mechanism must decide whether the evidence available now warrants intervention or whether continued execution is still useful.

We investigate this decision by varying the intervention point while holding the recovery policy and interaction budget fixed. For each failed trajectory, we replay the original actions to reconstruct candidate GUI states and launch recovery from each state. Across 120 OSWorld failure trajectories, 78.3% contain a checkpoint after the earliest critical error that yields strictly higher recovery utility, with a mean best-over-root gain of 0.17 on a unit-scale structured recovery score. Other trajectories favor immediate recovery, and the utility need not change monotonically over time (Figure 1). These results motivate treating intervention as a sequential stopping decision, whose outcome depends on both accumulated evidence and the opportunity remaining for recovery.

We introduce RETIME to make this decision during GUI-agent execution. Its timing gate maintains evidence across successive observations, allowing an isolated anomaly time for clarification while responding to persistent or decisive failures. Evidence of meaningful progress supports continued execution, and the remaining interaction budget constrains how long recovery can be deferred. The

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.

gate uses only the task and execution history available at the current step; its parameters are calibrated on separate recovery trajectories. This design connects the temporal variation observed in our study to an online intervention mechanism.

Choosing when to intervene leaves a second question: what should be repaired at the selected state? As an error propagates, the action needed to resume progress may change. ReTIME therefore diagnoses the recovery need observable at intervention time. A LoRA-adapted vision-language model assigns this need to a recovery-oriented taxonomy, and the predicted category retrieves a macro-policy that specifies the next recovery objective. The execution agent then produces concrete GUI actions from the current observation. Together, the timing gate and diagnosis model determine an intervention point and a recovery direction (Figure 2).

Evaluating this mechanism requires distinguishing a good intervention decision from a correct diagnosis. We construct RETIME-BENCH with 672 checkpoints from 133 failed trajectories generated by six source agents. Timing labels are derived from counterfactual recovery outcomes, whereas diagnosis labels describe the recovery need visible at each checkpoint. Future outcomes are used only for offline timing annotation. ReTIME-8B achieves the highest timing balanced accuracy and diagnosis accuracy among the evaluated models. In end-to-end execution on OSWorld-Verified, it achieves 35.2% task success under a 50-step budget, exceeding its backbone Qwen3-VL-8B-Instruct by 8.4%.

Our contributions are threefold:

- A controlled counterfactual study that quantifies how intervention timing affects GUI-agent recovery across source agents and recovery policies.
- ReTIME, which combines a stateful stopping policy with learned diagnosis and category-specific guidance for recovery from the current GUI state.
- ReTIME-Bench, a checkpoint-level evaluation of timing and diagnosis, together with end-to-end experiments assessing their contribution to task completion.

2 RELATED WORK

2.1 GUI AGENTS

Recent multimodal GUI agents, including SeeClick (Cheng et al., 2024), OS-Atlas (Wu et al., 2025b), Aguvis (Xu et al., 2025), UI-TARS (Qin et al., 2025), and OpenCUA (Wang et al., 2025), have advanced visual grounding, planning, and long-horizon interaction in realistic computer environments. Benchmarks such as OSWorld (Xie et al., 2024) and Windows Agent Arena (Bonatti et al., 2025) enable systematic evaluation of these capabilities, while reliable execution under failures remains challenging.

2.2 FAILURE DETECTION AND RECOVERY IN GUI AGENTS

Recent work improves GUI-agent reliability through verification, self-correction, and explicit recovery. VeriGUI (Zhang et al., 2026b) verifies action effects, BacktrackAgent (Wu et al., 2025a) detects errors and backtracks, RoTS (Bu et al., 2026) studies recovery from policy-induced errors, and VLAA-GUI (Han et al., 2026) incorporates runtime stopping and recovery. These methods primarily study failure detection or recovery after correction is required. ReTIME instead focuses on *when* intervention should occur, treating recovery timing as an adaptive decision rather than coupling it to failure recognition.

2.3 FAILURE TAXONOMIES AND RECOVERY-ORIENTED DIAGNOSIS

Prior work categorizes GUI-agent failures by causes such as perception, grounding, reasoning, and execution. RoTS (Bu et al., 2026) characterizes policy-induced errors for recovery trajectory synthesis, while CUADebug (Zhang et al., 2026a) diagnoses root causes and corrective strategies. ReTIME instead uses a checkpoint-level, recovery-oriented taxonomy that maps observable recovery needs to actionable macro-policies, directly supporting its HOW decision.

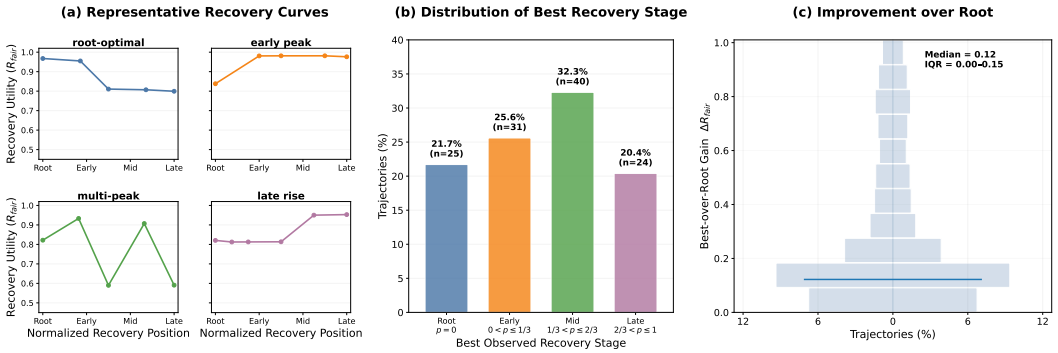


Figure 1: **Recovery utility depends on intervention timing.** (a) Representative recovery-utility trajectories exhibit heterogeneous temporal patterns. (b) Distribution of the checkpoint positions attaining the highest observed recovery utility across the 120 failure trajectories in our main counterfactual study. (c) Distribution of the best-over-root recovery gain, $\Delta U = \max_{t \in \mathcal{C}(\tau)} U(t; \pi_R) - U(t_r; \pi_R)$, where $\mathcal{C}(\tau)$ includes the root checkpoint t_r.

3 Recovery Timing in GUI Agents

Our study reveals that the root failure is frequently not the optimal recovery point: limited continued execution can improve recoverability, whereas excessive delay may degrade it. We characterize this effect through controlled counterfactual recovery across intervention checkpoints.

3.1 Problem Formulation

Consider a GUI agent executing instruction I and producing a trajectory $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T)$, where s_t and a_t denote the GUI state and action at step t. For an unsuccessful trajectory, let t_r denote the earliest critical execution error whose consequences create a recovery need. We use t_r only for offline analysis and never expose it to ReTIME at inference.

For each candidate checkpoint $t \in \mathcal{C}(\tau)$, where $\mathcal{C}(\tau) \subseteq \{t_r, \ldots, T\}$ includes t_r, we launch a recovery policy π_R. Let K_t denote the number of valid recovery rollouts at t. We define the checkpoint-level recovery utility as

$$U(t; \pi_R) = \frac{1}{K_t} \sum_{j=1}^{K_t} R_e\left(\tau_{t,j}^R\right), \tag{1}$$

where $R_e \in [0, 1]$ is a structured recovery score capturing recovery quality beyond binary task success. Its dimensions, evaluation protocol, and robustness to alternative utility formulations are detailed in Appendix A.

When multiple checkpoints attain the same utility, we define the optimal observed recovery checkpoint as the earliest maximizer:

$$t^* = \min\left(\arg\max_{t \in \mathcal{C}(\tau)} U(t; \pi_R)\right). \tag{2}$$

We then examine whether $t_r = t^*$ or a later checkpoint provides strictly higher recovery utility.

3.2 Controlled Counterfactual Recovery Study

We perform controlled counterfactual recovery over failed OSWorld trajectories. For each replayable failure, we identify t_r offline and sample 5–8 validated checkpoints from t_r onward, with denser sampling near the root. At each checkpoint, we replay the original source actions to reconstruct the GUI state and launch independent recovery rollouts under the same policy and fixed total execution budget, with prefix actions charged to that budget. The recovery agent receives only

Table 1: Recovery timing across heterogeneous source agents and recovery policies. **Root Best**: the root achieves the highest recovery utility; **Post-root Better**: at least one post-root checkpoint strictly outperforms root recovery; ΔU: best-over-root recovery gain (the theoretical maximum is 1).

Source	Recovery	# Traj.	Root Best	Post-root Better	Mean ΔU
Source-agent robustness					
OpenCUA-7B	Seed2.0 Pro	20	20.0%	**80.0%**	0.20
UI-TARS-1.5-7B	Seed2.0 Pro	20	10.0%	**90.0%**	0.15
OpenCUA-32B	Seed2.0 Pro	20	30.0%	**70.0%**	0.13
Kimi-VL-A3B-Thinking	Seed2.0 Pro	20	35.0%	**65.0%**	0.15
Claude Sonnet 4	Seed2.0 Pro	20	10.0%	**90.0%**	0.14
o3	Seed2.0 Pro	20	25.0%	**75.0%**	0.26
Overall	Seed2.0 Pro	**120**	**21.7%**	**78.3%**	**0.17**
Recovery-policy robustness on matched trajectories					
Mixed Sources	Seed2.0 Pro	50	22%	**78%**	0.16
Mixed Sources	OpenCUA-7B	50	24%	**76%**	0.15

the task instruction, current GUI observation, and execution history; root annotations and future trajectory information are never exposed. Further protocol details are provided in Appendix A.

3.3 Recovery Utility Depends on Intervention Timing

The Root Failure Is Frequently Not the Optimal Recovery Point. Figure 1 shows substantial variation in recovery utility across intervention checkpoints. Among the 120 trajectories, only 21.7% attain their highest recovery utility at the root, while 78.3% contain a post-root checkpoint that strictly outperforms root recovery. We quantify this advantage by

$$\Delta U = \max_{t \in \mathcal{C}(\tau)} U(t; \pi_R) - U(t_r; \pi_R). \tag{3}$$

The mean ΔU is 0.17, showing that failure onset and the most effective recovery point frequently differ.

A Trade-off Between Evidence Accumulation and Error Propagation. Recovery utility often improves after limited continued execution but degrades with excessive delay. Continued execution can reveal GUI feedback, action outcomes, or state inconsistencies useful for diagnosis, whereas further delay may propagate errors, alter useful state, or consume the remaining interaction budget. Some failures instead favor immediate recovery. Recovery landscapes may therefore fluctuate, plateau, or exhibit multiple peaks rather than follow a fixed temporal pattern, motivating an adaptive `CONTINUE`/`RECOVER` decision.

3.4 Robustness Across Agent Configurations

As shown in Table 1, post-root recovery advantages persist across heterogeneous source agents under a fixed recovery policy and across recovery policies on matched trajectories. The timing effect is therefore not specific to a single source–recovery pairing.

4 ReTIME: Timing-Aware Self-Recovery

Section 3 shows that recovery need and intervention timing should be treated separately. We therefore introduce ReTIME, a timing-aware self-recovering GUI agent that jointly determines **WHEN** to initiate recovery and **HOW** recovery should proceed from the current state.

4.1 Framework Overview

At each decision point, ReTIME receives the task instruction I, execution history H_t, and current GUI state s_t. As illustrated in Figure 2, the WHEN module first decides whether to Continue

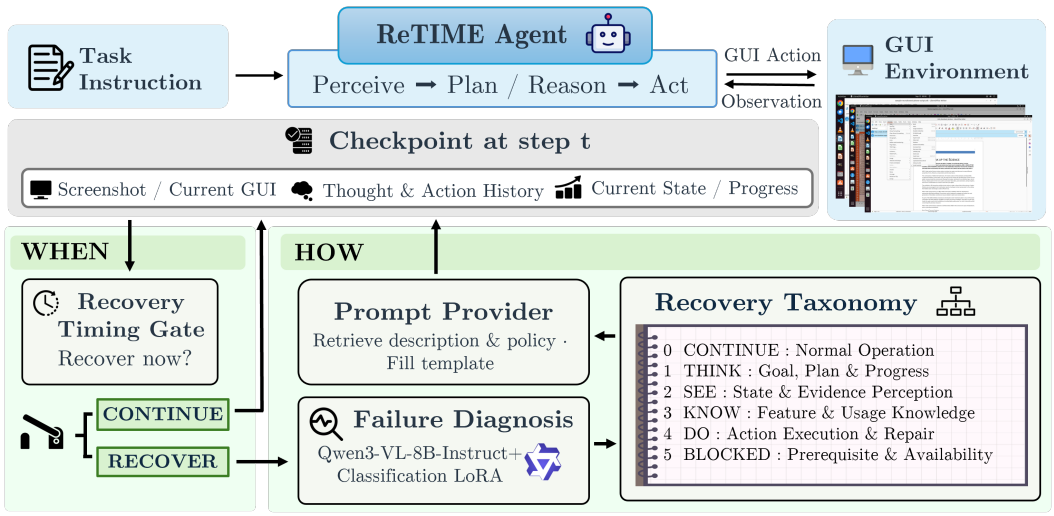


Figure 2: **Overview of the ReTIME framework.**

or RECOVER. Upon recovery, the HOW module diagnoses the current recovery need, retrieves a category-specific macro-policy, and provides corresponding guidance to the execution policy. Both modules operate solely on information available at the current checkpoint, without requiring the offline root checkpoint t_r or an external failure indicator.

4.2 WHEN: RECOVERY TIMING

The ReTIME Timing Gate is a lightweight, stateful stopping policy operating only on prefix-available information. At each step, it extracts signals z_t from the task instruction, GUI observations, and execution history, capturing both failure evidence and counter-evidence such as meaningful progress. Instantaneous evidence is aggregated as

$$F_t = \sum_k w_k z_{t,k}, \tag{4}$$

and persistent evidence is maintained through

$$E_t = \rho E_{t-1} + \max(0, F_t). \tag{5}$$

This allows isolated anomalies limited self-correction opportunity while persistent evidence accumulates.

The gate makes the stopping decision

$$g_t = G(F_t, E_t, c_t, b_t) \in \{\text{CONTINUE}, \text{RECOVER}\}, \tag{6}$$

where c_t summarizes self-correction progress and b_t denotes the remaining interaction budget. Decisive failures can trigger immediate recovery; otherwise, the decision accounts for accumulated evidence, self-correction grace, and the remaining recovery budget. The decision structure is fixed, with parameters calibrated on recovery trajectories disjoint from ReTIME-Bench and held fixed throughout evaluation. Full signal definitions and decision rules are provided in Appendix B.

4.3 HOW: RECOVERY-ORIENTED DIAGNOSIS AND RECOVERY

Once the Timing Gate returns RECOVER, ReTIME diagnoses the recovery need at the current checkpoint:

$$f_\theta(I, H_t, s_t) = \hat{y}_t, \qquad \hat{y}_t \in \mathcal{Y}, \tag{7}$$

where $\mathcal{Y}$ denotes the recovery-oriented taxonomy below. Unlike root-cause diagnosis, $\hat{y}_t$ describes what must be addressed from the current observable state and may therefore change as failure consequences evolve.

4.3.1 RECOVERY-ORIENTED TAXONOMY

We organize checkpoint-level recovery needs into a hierarchical taxonomy with 21 leaf categories under six groups: NORMAL, THINK, SEE, KNOW, DO, and BLOCKED. The taxonomy was developed by synthesizing failure categories and recovery strategies from prior GUI-agent literature, and then refined through pilot recovery experiments in which we tested alternative recovery prompts on representative failure cases. We separate failure modes when they consistently require different recovery directions, and merge distinctions that lead to similar recovery behavior.

Each recovery-oriented leaf is therefore defined not only by the observed failure pattern, but also by the recovery action it is intended to support. It maps to a category-specific macro-policy that specifies the first recovery objective from the current checkpoint while leaving low-level actions to the execution policy. Complete taxonomy definitions, annotation rules, pilot-study details, and taxonomy-to-policy mappings are provided in Appendix C.

4.3.2 RECOVERY DIAGNOSIS MODEL

We adapt Qwen3-VL-8B-Instruct with LoRA on a separate OSWorld checkpoint corpus. All ReTIME-Bench tasks are excluded before a task-UUID split into 5,190 training and 592 validation cases, and no benchmark annotations or counterfactual recovery outcomes are used for training.Given checkpoint input $x_t = (I, H_t, s_t)$ and taxonomy context $\mathcal{T}$, the model predicts

$$p_\theta(y_t \mid x_t, \mathcal{T}), \qquad y_t \in \mathcal{Y}. \tag{8}$$

Here, $\mathcal{T}$ contains the taxonomy labels and diagnostic definitions; the corresponding recovery macro-policy is retrieved only after prediction. Only information available at or before checkpoint t is used. Input construction, LoRA training, and inference details are provided in Appendix E.

4.3.3 POLICY-CONSTRAINED RECOVERY

Given diagnosis $\hat{y}_t$, ReTIME retrieves its associated recovery macro-policy $m_t = \mathcal{M}(\hat{y}_t)$. A lightweight Prompt Provider converts the diagnosis and macro-policy into recovery guidance, after which the base agent generates state-dependent actions through its normal execution process. This constrains recovery to a category-appropriate direction without prescribing a fixed low-level action sequence. Prompt construction and the complete recovery template are provided in Appendix D.

5 RETIME-BENCH

We introduce ReTIME-Bench, a checkpoint-level benchmark for jointly evaluating recovery timing and recovery-oriented diagnosis in GUI agents. Each instance is defined as

$$x_t = (I, H_t, s_t) \rightarrow \left(y_t^{\text{timing}}, y_t^{\text{diag}}\right), \tag{9}$$

where $y_t^{\text{timing}} \in \{\text{CONTINUE}, \text{RECOVER}\}$ evaluates WHEN to recover, and $y_t^{\text{diag}} \in \mathcal{Y}$ evaluates the current recovery need according to Section 4.3.1.

The two decisions are complementary: a recoverable abnormality may already be observable while recovery should not yet begin. ReTIME-Bench therefore evaluates recovery timing separately from checkpoint-level recovery diagnosis.

5.1 BENCHMARK CONSTRUCTION

ReTIME-Bench contains 672 annotated checkpoints from 133 replayable failed OSWorld trajectories generated by six source agents, with multiple post-failure checkpoints sampled per trajectory. Each checkpoint contains only prefix-available information. WHEN labels are derived from counterfactual recovery outcomes, while HOW labels follow the recovery-oriented taxonomy; future trajectory information is excluded from model inputs. The benchmark is fully held out from LoRA training and validation. Further details are provided in Appendix F.

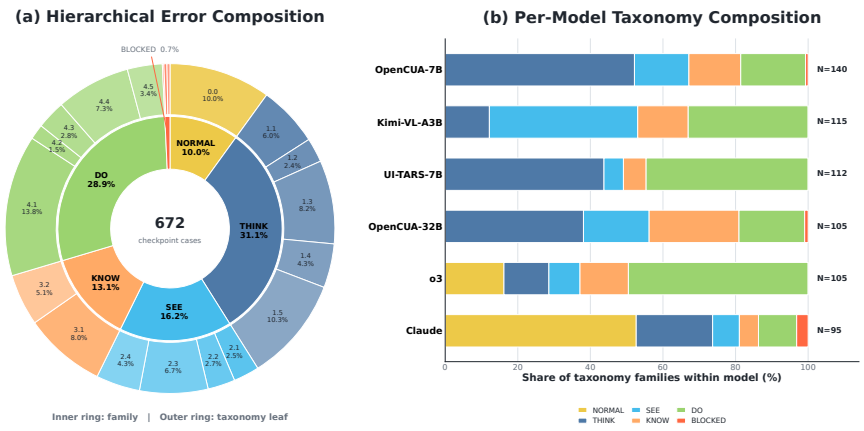


Figure 3: **Composition of ReTIME-Bench.** (a) Distribution of 672 benchmark checkpoints across taxonomy families (inner ring) and leaf categories (outer ring). (b) Taxonomy-family composition for each source agent.

5.2 Ground-Truth Construction

5.2.1 Recovery Timing

Timing supervision is derived from counterfactual recovery outcomes. For each trajectory i, repeated recovery rollouts estimate the utility $U_i(t)$ of each candidate checkpoint. We normalize utilities within each trajectory to compare relative recovery opportunities:

$$\widetilde{U}_i(t) = \frac{U_i(t) - U_i^{\min}}{U_i^{\max} - U_i^{\min} + \epsilon}. \tag{10}$$

Trajectories with negligible absolute utility variation are treated as near-flat and do not induce Continue labels.

For checkpoint t_j, we define the best future gain as

$$G_i(t_j) = \max_{k>j} \widetilde{U}_i(t_k) - \widetilde{U}_i(t_j), \tag{11}$$

and assign

$$y_{i,j}^{\text{timing}} = \begin{cases} \textsc{Continue}, & G_i(t_j) > \delta, \\ \textsc{Recover}, & G_i(t_j) \le \delta, \end{cases} \qquad \delta = 0.15. \tag{12}$$

Continue therefore indicates that waiting provides a meaningfully better recovery opportunity; otherwise, recovery is preferred at the current checkpoint. The final candidate checkpoint is labeled Recover. Additional annotation details are provided in Appendix F.1.

5.2.2 Recovery-Oriented Diagnosis

HOW labels are assigned at each checkpoint according to the recovery-oriented taxonomy in Section 4.3.1. Annotation is conducted by human experts with AI assistance. The AI system provides candidate labels and supporting evidence based only on the task instruction, GUI state, and execution history available at the checkpoint, while human experts review the evidence and determine the final label. Future trajectory information and counterfactual recovery outcomes are not used for diagnosis annotation.

Detailed annotation guidelines and quality-control procedures are provided in Appendix F.

5.3 Evaluation Protocol

ReTIME-Bench evaluates recovery timing and recovery-oriented diagnosis separately and jointly. For recovery timing, we report Timing Accuracy and Balanced Accuracy, where Balanced Accuracy

Table 2: **Main results on ReTIME-Bench.**

Model	Timing Acc.	Timing Bal. Acc.	Leaf Acc.	Leaf Macro-F1	Joint Acc.
Qwen2.5-VL-3B-Instruct	54.76%	48.89%	3.27%	2.48%	1.93%
Qwen2.5-VL-7B-Instruct	44.20%	47.21%	6.40%	5.06%	3.13%
Qwen2.5-VL-32B-Instruct	41.07%	47.92%	10.12%	3.59%	3.57%
UI-TARS-1.5-7B	47.32%	48.51%	10.27%	5.94%	4.17%
OpenCUA-7B	36.46%	50.03%	10.86%	6.61%	3.72%
Qwen3-VL-8B-Instruct	37.64%	50.17%	10.86%	6.42%	4.91%
Qwen3-VL-32B-Instruct	56.99%	50.82%	15.48%	11.89%	9.76%
Claude Sonnet 4.5	**58.93%**	52.87%	22.32%	12.21%	13.24%
GPT-5.4	58.78%	53.28%	23.96%	18.17%	12.80%
ReTIME-8B	57.59%	**56.31%**	**31.55%**	**23.95%**	**17.86%**

averages recall over the `CONTINUE` and `RECOVER` classes. For recovery-oriented diagnosis, we report leaf-level Accuracy and Macro-F1, with Macro-F1 assigning equal weight to each taxonomy leaf. We additionally report Joint Accuracy, which requires both predictions at a checkpoint to be correct.

6 Experiments

We evaluate ReTIME from three perspectives: recovery timing and diagnosis on ReTIME-Bench, end-to-end GUI task completion, and the contributions of the **WHEN** and **HOW** components.

6.1 Experimental Setup

Environment and Protocol. Experiments are conducted on OSWorld following the held-out evaluation protocol in Section 5.1. Counterfactual recovery evaluation follows Section 3.2. Unless otherwise specified, all methods use the same environment configuration, available context, and interaction budget.

Models. ReTIME is instantiated from a vision-language backbone and adapted with LoRA for recovery-oriented diagnosis. On ReTIME-Bench, we compare against open-source GUI agents and general-purpose MLLMs across multiple model scales. Implementation, training, and baseline details are provided in the Appendix.

Metrics. For recovery timing, we report Timing Accuracy and Balanced Accuracy, where the latter averages recall over the `CONTINUE` and `RECOVER` classes to account for label imbalance. For recovery-oriented diagnosis, we report leaf-level Accuracy and Macro-F1, with Macro-F1 assigning equal weight to each taxonomy leaf. Joint Accuracy requires both the timing and diagnosis predictions at a checkpoint to be correct. For end-to-end evaluation, we report task Success Rate under a fixed interaction budget.

6.2 Results on ReTIME-Bench

Table 2 reports checkpoint-level performance on recovery timing and recovery-oriented diagnosis.

ReTIME-8B achieves 57.59% Timing Accuracy, 56.31% Timing Balanced Accuracy, 31.55% Leaf Accuracy, 23.95% Leaf Macro-F1, and 17.86% Joint Accuracy. With an 8B backbone, it ranks first on the latter four metrics, outperforming Qwen3-VL-32B-Instruct, Claude Sonnet 4.5, and GPT-5.4. Compared with comparable 7B/8B baselines, ReTIME-8B improves these metrics by 6.14, 20.69, 17.34, and 12.95 percentage points, respectively, demonstrating substantially stronger recovery timing and diagnosis. “‘

Table 3: **End-to-end performance on OSWorld-Verified.** Recovery-Aware indicates whether an agent includes an explicit failure-recovery mechanism beyond its standard forward execution policy. Max Steps denotes the maximum interaction budget per task.

Agent	Recovery-Aware	Max Steps	Success Rate (%) ↑
Comparable 7B/8B Agents			
Qwen3-VL-8B-Instruct	No	50	26.8
UI-TARS-1.5-7B	No	50	27.3
OpenCUA-7B	No	50	27.9
GUI-OWL-7B	No	50	29.4
ReTIME-8B	Yes	50	**35.2**
Larger-Model References			
OpenCUA-32B	No	50	34.1
RoTS-32B	Yes	50	**47.4**

Table 4: **Component ablation for end-to-end recovery.** Timing denotes the ReTIME Timing Gate, while Diagnosis denotes recovery-oriented diagnosis with taxonomy-constrained guidance. Success Rate measures complete task success under the same interaction budget.

Method	Timing	Diagnosis	Success Rate (%) ↑
Base Recovery	×	×	26.83
Timing-only	✓	×	29.81
ReTIME	✓	✓	**35.23**

6.3 End-to-End GUI Recovery

We evaluate whether these capabilities translate into improved task completion on OSWorld-Verified under a maximum interaction budget of 50 steps.

As shown in Table 3, ReTIME-8B achieves 35.2% task success, outperforming its Qwen3-VL-8B-Instruct backbone at 26.8% by 8.4 percentage points. This result shows that adaptive recovery timing and recovery-oriented diagnosis improve complete-task execution rather than only checkpoint-level prediction. (The Qwen3-VL-8B-Instruct score is our reproduction under the same evaluation setup)

6.4 Ablation and Analysis

Component Ablation. We evaluate three operational configurations: Base Recovery, Timing-only, and the full ReTIME system. The diagnosis is invoked only after the gate emits a `RECOVER` signal and has no independently defined intervention point, so we didn't conduct a Diagnose-only experiment.

As shown in Table 4, adding the Timing Gate improves task success from 26.83% to 29.81%. Adding recovery-oriented diagnosis and taxonomy-constrained guidance after a `RECOVER` decision further improves success to 35.23%. These results show that the Timing Gate contributes independently and that diagnosis provides an additional benefit once recovery has been triggered.

7 Conclusion

We study when GUI agents should initiate recovery after execution failures. Through controlled counterfactual recovery, we show that 78.3% of the studied trajectories contain a post-root checkpoint that strictly outperforms recovery from the root. Based on this finding, we introduce **ReTIME**, which combines timing-aware intervention with recovery-oriented diagnosis, and **ReTIME-Bench**, a checkpoint-level benchmark for evaluating **WHEN** and **HOW** to recover. ReTIME-8B achieves 56.31% timing balanced accuracy and 31.55% leaf-level diagnosis accuracy on ReTIME-Bench. On OSWorld-Verified, it achieves 35.2% task success, outperforming its reproduced Qwen3-VL-8B-Instruct backbone by 8.4 percentage points under the same evaluation setup. These results demonstrate the importance of recovery timing for reliable GUI-agent execution.

AI USE STATEMENT

In this work, we used generative AI tools for several research tasks that require disclosure under the ICLR 2027 policy. Specifically, generative AI was used to help refine research hypotheses and conceptual formulations, provide feedback on research methodology and experimental design, assist with benchmark and dataset construction planning, support implementation and debugging of research code, and assist in interpreting intermediate experimental observations and results. Generative AI was not used to generate synthetic datasets, formulate or prove mathematical claims, or write mathematical proofs. Translation was not a substantive component of this work, and other required-disclosure tasks not listed above were not applicable.

Additionally, we used generative AI tools for literature discovery and organization, brainstorming, identifying related work and research gaps, suggesting paper structure, drafting and editing portions of the manuscript, formatting references, suggesting experimental parameters, and creating or editing software code.

We reviewed all AI-assisted work before incorporating it into the research workflow or manuscript. Methodological suggestions and experimental designs were evaluated and finalized by the authors; AI-assisted code was inspected, tested, and revised as needed before use; cited literature and factual claims were checked against the original sources; and AI-assisted interpretations were compared against the underlying experimental evidence. The recovery-oriented taxonomy, annotation protocol, benchmark labels, experimental settings, and final scientific conclusions were determined and validated by the authors.

We take responsibility for the final content of this work, including all text, claims, code, results, benchmark annotations, and artifacts produced with the aid of generative AI.

REFERENCES

Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, Humen Zhong, Yuanzhi Zhu, Mingkun Yang, Zhaohai Li, Jianqiang Wan, Pengfei Wang, Wei Ding, Zheren Fu, Yiheng Xu, Jiabo Ye, Xi Zhang, Tianbao Xie, Zesen Cheng, Hang Zhang, Zhibo Yang, Haiyang Xu, and Junyang Lin. Qwen2.5-VL technical report. *arXiv preprint arXiv:2502.13923*, 2025.

Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, Lawrence Keunho Jang, and Zheng Hui. Windows agent arena: Evaluating multi-modal OS agents at scale. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 4874–4910. PMLR, 2025.

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.

Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Yantao Li, Jianbing Zhang, and Zhiyong Wu. Seeclick: Harnessing GUI grounding for advanced visual GUI agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pp. 9313–9332. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.505.

Qijun Han, Haoqin Tu, Zijun Wang, Haoyue Dai, Yiyang Zhou, Nancy Lau, Alvaro A. Cardenas, Yuhui Xu, Ran Xu, Caiming Xiong, Zeyu Zheng, Huaxiu Yao, Yuyin Zhou, and Cihang Xie. VLAA-GUI: Knowing when to stop, recover, and search: A modular framework for GUI automation. *arXiv preprint arXiv:2604.21375*, 2026.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, Wanjun Zhong, Kuanye Li, Jiale Yang, Yu Miao, Woyu Lin,

Longxiang Liu, Xu Jiang, Qianli Ma, Jingyu Li, Xiaojun Xiao, Kai Cai, Chuang Li, Yaowei Zheng, Chaolin Jin, Chen Li, Xiao Zhou, Minchao Wang, Haoli Chen, Zhaojian Li, Haihua Yang, Haifeng Liu, Feng Lin, Tao Peng, Xin Liu, and Guang Shi. UI-TARS: Pioneering automated GUI interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025.

Xinyuan Wang, Bowen Wang, Dunjie Lu, Junlin Yang, Tianbao Xie, Junli Wang, Jiaqi Deng, Xiaole Guo, Yiheng Xu, Chen Henry Wu, Zhennan Shen, Zhuokai Li, Ryan Li, Xiaochuan Li, Junda Chen, Boyuan Zheng, Peihang Li, Fangyu Lei, Ruisheng Cao, Yeqiao Fu, Dongchan Shin, Martin Shin, Jiarui Hu, Yuyan Wang, Jixuan Chen, Yuxiao Ye, Danyang Zhang, Dikang Du, Hao Hu, Huarong Chen, Zaida Zhou, Yipu Wang, Heng Wang, Diyi Yang, Victor Zhong, Flood Sung, Y. Charles, Zhilin Yang, and Tao Yu. OpenCUA: Open foundations for computer-use agents. In *Advances in Neural Information Processing Systems*, 2025.

Qinzhuo Wu, Pengzhi Gao, Wei Liu, and Jian Luan. Backtrackagent: Enhancing GUI agent with error detection and backtracking mechanism. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 4250–4272. Association for Computational Linguistics, 2025a. doi: 10.18653/v1/2025.emnlp-main.212.

Zhiyong Wu, Zhenyu Wu, Fangzhi Xu, Yian Wang, Qiushi Sun, Chengyou Jia, Kanzhi Cheng, Zichen Ding, Liheng Chen, Paul Pu Liang, and Yu Qiao. OS-ATLAS: A foundation action model for generalist GUI agents. In *International Conference on Learning Representations*, 2025b.

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. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

Yiheng Xu, Zekun Wang, Junli Wang, Dunjie Lu, Tianbao Xie, Amrita Saha, Doyen Sahoo, Tao Yu, and Caiming Xiong. Aguvis: Unified pure vision agents for autonomous GUI interaction. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 69772–69805. PMLR, 2025.

Mengqi Yuan, Zilong Zhou, Xinzhuang Xiong, Weiming Wu, Jiayang Sun, Jiamin Song, Kaiqian Cui, Bowen Wang, Haoyuan Wu, Yitong Li, Dunjie Lu, Haikong Lu, Qi Zhen, Xinyuan Wang, Jiaqi Deng, Yuhao Yang, Cheng Chen, Boyuan Zheng, Alex Su, Xiao Yu, Hao Zou, Saaket Agashe, Xing Han Lu, Manpreet Kaur, Zhengyang Qi, Vincent Sunn Chen, Frederic Sala, Dayiheng Liu, Junyang Lin, Zhou Yu, Yu Su, Siva Reddy, Xin Eric Wang, Peng Qi, Tianbao Xie, and Tao Yu. OSWorld 2.0: Benchmarking computer use agents on long-horizon real-world tasks, 2026.

Weijia Zhang, Kunlun Zhu, Zeyi Liu, Yinting Chen, Tianyi Ma, Jiateng Liu, Jiaxun Zhang, Bingxuan Li, Xiangru Tang, Heng Ji, and Jiaxuan You. CUADebug: Diagnosing and repairing computer-use agent failures. *arXiv preprint arXiv:2608.02643*, 2026a.

Yuzhe Zhang, Xianwei Xue, Xingyong Wu, Mengke Chen, Chen Liu, Xinran He, Run Shao, Feiran Liu, Huanmin Xu, Qiutong Pan, and Haiwei Wang. Don't act blindly: Robust GUI automation via action-effect verification and self-correction. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 28924–28941. Association for Computational Linguistics, 2026b.

APPENDIX

A RECOVERY UTILITY AND COUNTERFACTUAL EVALUATION

This appendix specifies the evaluation protocol for the counterfactual timing analysis in Section 3.2. Recovery utility compares interventions launched from different checkpoints of the same failed source trajectory. It supports offline analysis and benchmark annotation, while the online system operates only on the execution prefix available at inference time.

A.1 Stage-Wise Recovery Evaluation

Final task success does not fully characterize recovery quality. In long-horizon computer-use tasks, two unsuccessful rollouts may preserve very different amounts of valid progress, while two successful rollouts may differ in interaction cost, state preservation, and verification quality. This motivates the fine-grained partial rewards also used by benchmarks such as OSWorld 2.0 (Yuan et al., 2026). For recovery evaluation, the distinction is especially important: a rollout may repair the original failure and restore a valid execution path even if a later, unrelated error prevents completion.

We therefore score task progress, recovery correctness, strategy redirection, useful interaction efficiency, state preservation, and verification correctness. Their aggregation yields the structured recovery score $R_e \in [0, 1]$.

All candidate checkpoints from a source trajectory share the same recovery policy, prompt, interaction budget, and rollout protocol, isolating checkpoint timing as the variable of interest.

A.2 Controlled Recovery Protocol

For each candidate checkpoint t, we reset the environment and replay the source-agent actions preceding t to reconstruct the corresponding GUI state. Recovery is then launched from the reconstructed checkpoint using the same recovery policy and interaction budget across checkpoints.

The recovery agent receives only the original task instruction, the current GUI state, and the execution history available at the checkpoint. It is never provided with the annotated root-cause checkpoint, future source-agent actions, future GUI states, recovery utilities, or outcomes of other recovery rollouts.

All counterfactual recovery rollouts use the same recovery instruction. We reproduce the exact prompt below for reproducibility:

```
You are resuming an interrupted GUI task from the current
computer state.
Original task:
{original_instruction}
The previous execution did not successfully complete the
task.  The current computer state may contain correct
progress, incorrect changes, incomplete operations, or
irrelevant actions.
Inspect the current state and continue operating the
computer until the original task is correctly completed.
You may preserve, modify, undo, or redo existing work as
needed.  Choose the actions that are most likely to complete
the task correctly while avoiding unnecessary loss of valid
progress and unnecessary operations.
Do not assume that the current state is correct.  Do not
assume that restarting is necessary.  Base your decisions on
the original task, the current GUI state, and the available
execution history.
When you believe the task is complete, verify the result
against every requirement in the original instruction before
terminating.
```

The prompt reveals that the preceding execution was unsuccessful, but not the failure type, its onset, a recovery strategy, or the relative utility of other checkpoints. The recovery policy infers these factors from the execution prefix at the intervention checkpoint.

A.3 Recovery Process and Task Progress

We first record four binary indicators describing the recovery process.

Diagnosis Correctness (DC). $\mathrm{DC} \in \{0, 1\}$ indicates whether the recovery agent correctly identifies the failure relevant to recovery at the intervention checkpoint.

Strategy Redirection (SRD). $\mathrm{SRD} \in \{0, 1\}$ indicates whether the recovery agent abandons the ineffective strategy associated with the current failure and adopts a task-relevant recovery direction.

Original Error Resolved (OER). $\mathrm{OER} \in \{0, 1\}$ indicates whether the failure motivating recovery is resolved during the recovery rollout.

Valid Path (VP). $\mathrm{VP} \in \{0, 1\}$ indicates whether the agent returns to a state from which the task remains validly executable.

For interpretability, we summarize these indicators using the Recovery Process Score

$$\mathrm{RPS} = \frac{\mathrm{DC} + \mathrm{SRD} + \mathrm{OER} + \mathrm{VP}}{4}. \tag{13}$$

RPS is retained for diagnostic analysis and is not itself used as the checkpoint-level recovery utility.

Task Progress. Let $P \in [0, 1]$ denote the fraction of the task objective correctly completed in the resulting state. Progress is evaluated relative to the task objective, rather than the amount of interaction performed. Repeated actions, irrelevant state changes, or movement away from the task objective do not increase P.

Recovery Correctness. We combine error resolution and restoration of a valid execution path as

$$C = \frac{\mathrm{OER} + \mathrm{VP}}{2}. \tag{14}$$

This term measures whether recovery repairs the execution state rather than merely producing a plausible diagnosis.

A.4 Verification, State Preservation, and Efficiency

Verification Correctness. We define

$$V_{\mathrm{correct}} \in [0, 1] \tag{15}$$

to measure whether the recovery agent correctly verifies the state resulting from its intervention. Verification is considered correct when the agent's post-recovery belief is consistent with the observable GUI state and relevant task constraints.

State Damage. We define a normalized state-damage term

$$D \in [0, 1], \tag{16}$$

where larger values indicate more severe unintended modification of the environment during recovery. Damage includes destructive edits, unintended persistent state changes, corruption of previously correct task progress, or other changes that increase the cost of subsequent recovery. Benign exploratory actions and reversible observation steps are not treated as damage.

Recovery Efficiency. Let N_{total} denote the number of recovery interaction steps, T_{total} the active recovery time, and B_s and B_t their corresponding budgets. We define

$$E_{\mathrm{step}} = \max\left(0, 1 - \frac{N_{\mathrm{total}}}{B_s}\right), \tag{17}$$

$$E_{\mathrm{time}} = \max\left(0, 1 - \frac{T_{\mathrm{total}}}{B_t}\right), \tag{18}$$

and combine them as

$$E = 0.60 E_{\mathrm{step}} + 0.40 E_{\mathrm{time}}. \tag{19}$$

To avoid rewarding short but ineffective attempts, we gate efficiency by observed recovery progress:

$$G_{\mathrm{progress}} = 0.50P + 0.50C, \tag{20}$$

$$E_{\mathrm{useful}} = E \cdot G_{\mathrm{progress}}. \tag{21}$$

Efficiency therefore contributes only in proportion to achieved task progress or recovery correctness.

Stagnation Penalty. We additionally penalize prolonged ineffective interaction. A stagnant run is detected when the same GUI action is repeated for at least three consecutive steps and the normalized mean absolute pixel difference between adjacent screenshots is at most 0.03. Let r denote the longest detected stagnant run. We define

$$L = \min\left(1, \max\left(0, \frac{r-2}{8}\right)\right). \tag{22}$$

A.5 Prompt-Assistance Control

The unified prompt states that the preceding execution was unsuccessful. This hint may be more useful at checkpoints where the source agent has not yet recognized the failure, so we apply a bounded correction based on pre-intervention evidence.

We define the Self-Recognized Intervention Signal as

$$\mathrm{SRI}(t) \in \{0, 0.5, 1\}. \tag{23}$$

$\mathrm{SRI} = 0$ indicates no goal-relevant recognition that the current strategy has failed. $\mathrm{SRI} = 0.5$ indicates explicit uncertainty, failure, stagnation, or state mismatch without a reliable strategy switch. $\mathrm{SRI} = 1$ requires explicit recognition that the current strategy or action has failed together with a concrete task-relevant strategy switch that is proposed or initiated.

SRI is determined exclusively from the most recent eight source-agent steps preceding the checkpoint. Runner exceptions, evaluator outputs, failure flags, action repetition, screenshot stagnation, or coordinate changes alone cannot establish $\mathrm{SRI} = 1$.

We define the Prompt-Assistance Factor as

$$\mathrm{PAF}(t) = 1 - \alpha\,(1 - \mathrm{SRI}(t))\,, \qquad \alpha = 0.15, \tag{24}$$

or equivalently,

$$\mathrm{PAF}(t) = 0.85 + 0.15\,\mathrm{SRI}(t). \tag{25}$$

The factor discounts the observed score by at most 15%, limiting the advantage conferred by the prompt at checkpoints with little prior failure awareness.

A.6 Structured Recovery Score and Checkpoint Utility

We aggregate the recovery dimensions into the raw structured score

$$\begin{aligned} R_{\mathrm{raw}} = {} & 0.40P + 0.25C + 0.10\mathrm{SRD} + 0.15E_{\mathrm{useful}} + 0.10V_{\mathrm{correct}} \\ & - 0.10D - 0.10L, \end{aligned} \tag{26}$$

followed by

$$R_{\mathrm{raw}} \leftarrow \mathrm{clip}(R_{\mathrm{raw}}, 0, 1). \tag{27}$$

Final task outcome is incorporated through

$$q = \begin{cases} 1.00, & \text{if the final task succeeds,} \\ 0.70, & \text{otherwise,} \end{cases} \tag{28}$$

yielding

$$R_{\mathrm{conditioned}} = qR_{\mathrm{raw}}. \tag{29}$$

The individual-rollout recovery score used throughout the main paper is

$$R_e(\tau_t^R) = R_{\mathrm{conditioned}}(\tau_t^R) \cdot \mathrm{PAF}(t). \tag{30}$$

The coefficients are fixed across tasks, source agents, recovery policies, and checkpoints. They operationalize the relative importance of complementary recovery dimensions; robustness to alternative settings is reported in Appendix A.8.

For each checkpoint t, we perform K independent recovery rollouts $\{\tau_{t,j}^R\}_{j=1}^K$. The checkpoint-level recovery utility is

$$U(t; \pi_R) = \mathbb{E}\left[R_e(\tau_t^R)\right], \tag{31}$$

which is estimated empirically as

$$\widehat{U}(t;\pi_R) = \frac{1}{K}\sum_{j=1}^{K} R_e(\tau_{t,j}^R). \tag{32}$$

We additionally retain the standard deviation across recovery replicates and the final-task success rate at each checkpoint. The recovery curves in the main paper therefore represent checkpoint-level expected recovery quality rather than the binary outcome of a single stochastic rollout.

A.7 Recovery Outcome Categories and Evaluation Consistency

For interpretability, each recovery rollout is additionally assigned to one of five outcome categories.

Full Recovery. The original error is resolved, the agent returns to a valid execution path, and the final task succeeds:

$$\mathrm{OER} = 1, \quad \mathrm{VP} = 1, \quad \mathrm{FinalSuccess} = 1. \tag{33}$$

Partial Recovery. The rollout resolves the original error or returns to a valid execution path, but does not ultimately complete the task:

$$(\mathrm{OER} = 1 \vee \mathrm{VP} = 1) \wedge \mathrm{FinalSuccess} = 0. \tag{34}$$

Failed Recovery. The rollout neither resolves the original error nor restores a valid execution path:

$$\mathrm{OER} = 0, \quad \mathrm{VP} = 0, \quad \mathrm{FinalSuccess} = 0. \tag{35}$$

Accidental Success. The final evaluator reports success, but the observed recovery behavior does not provide sufficient evidence that the original error was resolved or that the agent returned to a valid recovery direction. Such cases are recorded separately to identify discrepancies between final task outcome and the recovery process.

Uncertain. Available evidence is insufficient to reliably determine the key recovery indicators.

The categories separate repair of the original failure from eventual task completion, allowing a rollout that repairs the execution path but later fails elsewhere to be analyzed as partial recovery.

Within each source trajectory, checkpoints share the same root-failure definition, resolution criterion, budgets, prompt, policy, and rollout protocol. SRI is fixed across replicates because it is computed from the source prefix. Rollouts affected by VM, API, or evaluator failures are excluded.

The sensitivity analysis below evaluates whether the timing result persists under alternative prompt-assistance and scoring parameters.

A.8 Robustness to Recovery-Utility Parameters

We test the sensitivity of the timing result to the structured utility's parameterization through post-hoc recomputation on the fixed 120-trajectory cohort. Alternative settings reuse the same checkpoints, recovery rollouts, annotations, and validity decisions; only the scoring function changes.

For every utility setting s, checkpoint-level utility is recomputed as

$$\widehat{U}_i^{(s)}(t) = \frac{1}{K_{i,t}} \sum_{j \in \mathcal{V}_{i,t}} R_{i,t,j}^{(s)}, \tag{36}$$

where $\mathcal{V}_{i,t}$ is the frozen set of valid recovery rollouts and $K_{i,t} = |\mathcal{V}_{i,t}|$. Consistent with the definition used in the main text, the best-over-root recovery gain for trajectory i is

$$\Delta U_i^{(s)} = \max_{t \in \mathcal{C}_i} \widehat{U}_i^{(s)}(t) - \widehat{U}_i^{(s)}(t_{r,i}), \tag{37}$$

where $\mathcal{C}_i$ contains the root checkpoint $t_{r,i}$ and all evaluated post-root checkpoints. Consequently, $\Delta U_i^{(s)} \geq 0$ by construction.

A trajectory is *Post-root Better* when $\Delta U_i^{(s)} > 10^{-12}$; otherwise it is *Root Best*, including ties. The 10^{-12} tolerance suppresses floating-point artifacts and is held fixed across settings.

Table 5: Robustness of the post-root timing observation across source agents and recovery-utility parameterizations. Each entry reports the number and percentage of trajectories for which at least one post-root checkpoint strictly outperforms recovery from the root checkpoint. All settings reuse the same frozen rollouts, annotations, and validity decisions.

Source Agent	Default ($q_f = 0.70$)	Equal Weight ($q_f = 0.70$)	Full Utility ($q_f = 0.50$)	Full Utility ($q_f = 0.25$)
OpenCUA-7B	16/20 (80%)	18/20 (90%)	18/20 (90%)	18/20 (90%)
UI-TARS-1.5-7B	18/20 (90%)	17/20 (85%)	17/20 (85%)	17/20 (85%)
OpenCUA-32B	14/20 (70%)	14/20 (70%)	14/20 (70%)	14/20 (70%)
Kimi-VL-A3B-Thinking	13/20 (65%)	12/20 (60%)	12/20 (60%)	12/20 (60%)
Claude Sonnet 4	18/20 (90%)	18/20 (90%)	18/20 (90%)	18/20 (90%)
o3	15/20 (75%)	14/20 (70%)	15/20 (75%)	15/20 (75%)
Overall	**94/120 (78.3%)**	**93/120 (77.5%)**	**94/120 (78.3%)**	**94/120 (78.3%)**

Table 6: Pooled robustness results over the 120-trajectory main counterfactual-recovery cohort. Root Best includes ties.

Utility setting	q_f	Root Best	Post-root Better
Default	0.70	26/120 (21.7%)	94/120 (78.3%)
Equal Weight	0.70	27/120 (22.5%)	93/120 (77.5%)
Full Utility	0.50	26/120 (21.7%)	94/120 (78.3%)
Full Utility	0.25	26/120 (21.7%)	94/120 (78.3%)

A.8.1 Equal-Weight Positive Components

The default utility assigns different weights to task progress, recovery correctness, strategy redirection, useful efficiency, and verification correctness. To test whether the timing result is driven by this particular allocation, we assign equal weight to all five positive components:

$$R_{\text{raw}}^{\text{EW}} = 0.20P + 0.20C + 0.20\text{SRD} + 0.20E_{\text{useful}} + 0.20V_{\text{correct}} - 0.10D - 0.10L. \tag{38}$$

We then apply the same clipping, evaluator-outcome factor, and prompt-assistance factor as in the default utility:

$$R_{\text{EW}} = \text{clip}\left(R_{\text{raw}}^{\text{EW}}, 0, 1\right) \cdot q \cdot \text{PAF}. \tag{39}$$

This variant changes only the relative weights of the five positive components; penalties and checkpoint aggregation remain unchanged.

A.8.2 Sensitivity to the Evaluator-Failure Factor

To measure sensitivity to incomplete outcomes, we retain the default raw utility and prompt-assistance correction while varying the failure factor:

$$q(q_f) = \begin{cases} 1, & \text{if the final task succeeds,} \\ q_f, & \text{otherwise,} \end{cases} \qquad q_f \in \{0.25, 0.50, 0.70\}. \tag{40}$$

The corresponding rollout score is

$$R^{(q_f)} = \text{clip}\left(R_{\text{raw}}, 0, 1\right) \cdot q(q_f) \cdot \text{PAF}. \tag{41}$$

Here, $q_f = 0.70$ reproduces the default score; $q_f = 0.50$ and $q_f = 0.25$ place progressively greater emphasis on full completion.

A.8.3 Results

Table 5 reports the number of trajectories containing a post-root checkpoint that strictly outperforms the root under each utility setting. Each source-agent cohort contains 20 trajectories, for a total of 120 trajectories.

Table 6 summarizes the pooled counts.

Table 7: **Recovery-timing robustness under the default utility.** The source-agent block and matched recovery-policy block reproduce the statistics in Table 1.

Source	Recovery	# Traj.	Root Best	Post-root Better	Mean ΔU
OpenCUA-7B	Seed2.0 Pro	20	20.0%	80.0%	0.20
UI-TARS-1.5-7B	Seed2.0 Pro	20	10.0%	90.0%	0.15
OpenCUA-32B	Seed2.0 Pro	20	30.0%	70.0%	0.13
Kimi-VL-A3B-Thinking	Seed2.0 Pro	20	35.0%	65.0%	0.15
Claude Sonnet 4	Seed2.0 Pro	20	10.0%	90.0%	0.14
o3	Seed2.0 Pro	20	25.0%	75.0%	0.26
Overall	Seed2.0 Pro	**120**	**21.7%**	**78.3%**	**0.17**
Mixed Sources	Seed2.0 Pro	50	22%	78%	0.16
Mixed Sources	OpenCUA-7B	50	24%	76%	0.15

The conclusion is stable across these variants. Equal weighting changes the Post-root-Better count by one trajectory, from 94/120 (78.3%) to 93/120 (77.5%), while reducing q_f to 0.50 or 0.25 leaves it unchanged. We use the default utility because it prioritizes task progress and recovery correctness over auxiliary indicators. The sensitivity results support this priority choice without implying that its coefficients are uniquely optimal.

A.9 Source-Agent and Recovery-Policy Robustness Summary

For completeness, Table 7 reproduces the robustness statistics reported in the main text under the default structured recovery utility. Each source-agent row contains 20 failed trajectories and uses Seed2.0 Pro as the recovery policy. The matched-policy comparison uses the same 50 source trajectories for both recovery policies, so only the recovery policy changes.

B Timing Gate Details

The online Timing Gate introduced in Section 4.2 decides **WHEN** to initiate recovery, before recovery-oriented diagnosis. It is a stateful stopping policy with a fixed logical structure and data-calibrated operating parameters. Its inputs are restricted to the current execution prefix; diagnosis begins after a RECOVER decision.

B.1 Prefix-Only Timing Evidence

At step t, the Timing Gate receives the task instruction, current and historical GUI observations, preceding thoughts and actions, the current interaction step, and its recurrent gate state. A lightweight Timing Observer converts this execution prefix into a set of timing signals

$$z_t = \{z_{t,k}\}_{k=1}^{K}. \tag{42}$$

These signals provide three complementary forms of evidence.

Hard failures. Hard-failure signals represent conditions under which further execution along the current path is no longer justified. They include terminal execution errors, invalid actions, external blockers, destructive regressions of previously valid task state, and capability dead ends. Any such signal can trigger recovery immediately without temporal accumulation.

Failure and stagnation evidence. Ordinary recovery evidence captures observable indications that the current execution is failing or no longer making useful progress. This includes explicit recognition of failure, failed verification, absence of an expected action outcome, execution in an incorrect context, lack of progress, repeated actions, action oscillation, an unchanged GUI after interaction, and explicit uncertainty in the execution history.

Evidence for continued execution. The observer also extracts counter-evidence supporting continued execution, including meaningful task progress, discovery of a new verifiable execution path,

and action outcomes that are still pending. Such evidence can reduce the instantaneous tendency to recover when continued execution remains productive or informative. It cannot override a hard failure, an exhausted recovery budget, or sufficiently persistent recovery evidence.

B.2 Stateful Evidence Accumulation

A single ambiguous observation should not necessarily trigger intervention. ReTIME therefore maintains an independent gate state for each trajectory, tracking the current suspicion interval, persistence of suspicious evidence, and accumulated recovery evidence. The state is reset when a new task trajectory begins.

Given the prefix-observable signals, the instantaneous recovery evidence is computed as

$$F_t = \sum_k w_k z_{t,k}, \tag{43}$$

where positive contributions represent evidence supporting recovery and negative contributions represent evidence supporting continued execution.

The gate maintains persistent evidence through

$$E_t = \rho E_{t-1} + \max(0, F_t), \tag{44}$$

where ρ controls how strongly evidence from preceding checkpoints is retained. Meaningful progress without simultaneous failure evidence clears or substantially weakens the active suspicion state.

This stateful formulation distinguishes isolated uncertainty from persistent execution failure. Weak evidence at one checkpoint can therefore be tolerated when subsequent interaction may provide additional information or enable autonomous self-correction, whereas repeated or mutually reinforcing evidence progressively strengthens the case for intervention.

B.3 Horizon-Aware Recovery

The Timing Gate additionally accounts for the remaining interaction horizon. Let T denote the maximum interaction budget and define normalized execution progress as

$$r_t = \frac{t}{T}. \tag{45}$$

Given a soft boundary b, time pressure is defined as

$$q_t = \begin{cases} 0, & r_t \le b, \\ \dfrac{r_t - b}{1 - b}, & r_t > b. \end{cases} \tag{46}$$

Before the soft boundary, the gate allows greater opportunity for exploration and autonomous self-correction. Beyond this boundary, increasing time pressure progressively lowers the evidence requirement for recovery. This implements a soft preference for earlier intervention as the available horizon shrinks, without allowing time pressure alone to determine recovery.

ReTIME additionally reserves a fixed portion of the remaining interaction budget for corrective execution. Once the remaining budget reaches this recovery reserve, an unfinished task enters recovery rather than continuing to consume steps that may be required for correction. Together with a bounded self-correction grace period, this provides a liveness constraint that prevents the gate from issuing CONTINUE indefinitely.

B.4 Online Decision Rule

The gate evaluates recovery conditions in a fixed priority order. Decisive current failures are handled before persistent evidence, and evidence supporting continued execution is considered only after immediate, budget-critical, and persistent recovery conditions have been checked. Algorithm 1 summarizes the complete online procedure.

Algorithm 1 ReTIME Timing Gate

Require: Execution prefix x_t, gate state h_{t-1}, current step t, budget T
Extract prefix-only timing signals z_t
Compute instantaneous evidence F_t
Update accumulated evidence E_t and persistence state
Compute time pressure and the effective recovery threshold
if task is completed **then**
  return CONTINUE {No recovery intervention}
else if a hard-failure condition is observed **then**
  return RECOVER
else if current failure evidence is sufficiently corroborated **then**
  return RECOVER
else if instantaneous evidence exceeds the immediate threshold **then**
  return RECOVER
else if remaining budget reaches the recovery reserve **then**
  return RECOVER
else if persistent evidence exceeds the effective recovery threshold **then**
  return RECOVER
else if the self-correction grace period is exhausted **then**
  return RECOVER
else if meaningful progress, a new verifiable path, or a pending result is observed **then**
  return CONTINUE
else
  return CONTINUE
end if

Here, CONTINUE means that the Timing Gate does not initiate a recovery intervention; task termination itself remains handled by the base agent. Therefore, a completed task bypasses recovery rather than causing additional execution.

The gate imposes no minimum wait: a hard failure, corroborated evidence, or a strong instantaneous signal can trigger recovery at its first observable checkpoint. If a root failure leaves no observable trace, intervention occurs when its consequences appear in a later prefix.

B.5 GATE CALIBRATION

We calibrate evidence weights, persistence, thresholds, temporal grace, and budget parameters on a small set of OSWorld recovery trajectories disjoint from ReTIME-Bench. Calibration preserves the priority order in Algorithm 1. The resulting configuration is fixed across evaluation tasks and agent configurations, without test-time adaptation.

B.6 INFORMATION BOUNDARY AND RELATION TO BENCHMARK SUPERVISION

The online gate and offline timing labels use different information. The gate observes the task, GUI history, thoughts and actions, observable outcomes, remaining budget, and its recurrent state. Benchmark labels additionally use counterfactual recovery outcomes to compute $U(t)$ and $G(t)$. These future-derived quantities define offline supervision in Section 5.2.1; they are not gate inputs.

C RECOVERY-ORIENTED TAXONOMY

ReTIME's taxonomy describes the recovery need observable at a checkpoint and maps it to a category-specific macro-policy. The label may differ from the trajectory's historical root cause because the required intervention can change as execution unfolds.

Table 8: **Recovery-oriented taxonomy used by ReTIME.** Each leaf diagnosis maps to a category-specific recovery macro-policy.

Code	Recovery Category	Recovery Macro-Policy
0.0	**Normal Operation**	Continue execution and verify subsequent state.
THINK*: Goal, planning, progress, and termination*		
1.1	Goal / Target / Constraint Misunderstanding	Re-anchor the task goal and constraints.
1.2	Infeasible Strategy	Replace the current strategy with a feasible alternative.
1.3	Missing / Wrong Step Plan	Reconstruct the required steps or dependencies.
1.4	Progress Tracking Drift	Reconcile completed and pending task progress.
1.5	Termination Control Failure	Verify completion and stop or resume accordingly.
SEE*: State perception and interpretation*		
2.1	Evidence / Feedback Misinterpretation	Re-ground the decision in observable evidence.
2.2	Active Context Misperception	Re-identify the active application, window, page, or document.
2.3	Selection / Focus State Misperception	Re-establish the intended selection or input focus.
2.4	Outcome / Authoritative State Mismatch	Verify the previous action outcome and reconcile the current state.
2.5	Blocking Layer Omission	Identify and resolve the blocking interface layer.
KNOW*: Interface and usage knowledge*		
3.1	Feature Route Knowledge Failure	Discover how to access the required feature.
3.2	Control / Parameter Usage Knowledge Failure	Determine and validate the required control or input rule.
DO*: Action execution*		
4.1	Target Localization Execution Failure	Re-localize the intended interaction target.
4.2	Click / Activation Execution Failure	Retry the intended activation correctly.
4.3	Viewport / Continuous Gesture Execution Failure	Re-execute the required spatial interaction.
4.4	Input / Invocation Execution Failure	Correct the input or invocation.
4.5	Unintended State-Changing Action	Undo, contain, or compensate for the unintended change.
BLOCKED*: External or prerequisite blockers*		
5.1	Required Prerequisite Unavailable	Restore or obtain the required prerequisite.
5.2	Access / Authorization Block	Resolve the access or authorization requirement.
5.3	Transient Unavailability / Not Ready	Wait, retry, or restart within the available budget.

C.1 TAXONOMY ORGANIZATION

The taxonomy contains 21 leaf categories organized into six coarse groups: NORMAL, THINK, SEE, KNOW, DO, and BLOCKED. NORMAL contains the normal-operation leaf, while the remaining groups contain 20 recovery-oriented leaves.

The groups separate goal and planning errors (THINK), state-perception errors (SEE), missing interface knowledge (KNOW), action execution failures (DO), and external or prerequisite constraints (BLOCKED). Each leaf specifies a corresponding recovery direction.

C.2 Leaf Definitions and Annotation Rules

The leaf label is selected from the recovery need supported by the checkpoint prefix, not from a retrospective explanation of the trajectory's original mistake. The definitions below make the leaf boundaries explicit and specify when each macro-policy is applicable.

0.0 Normal Operation. The visible state and execution history are consistent with the task and do not support a recovery-specific abnormality. Assign this leaf only when the agent can continue ordinary task execution without first repairing, re-grounding, or unblocking the current state. The macro-policy is to continue and verify the next observable state.

1.1 Goal / Target / Constraint Misunderstanding. The execution is organized around an incorrect task objective, target object, or explicit task constraint. Assign this leaf when the problem is what the agent is trying to accomplish, rather than how it is navigating the interface. Recovery re-anchors the instruction, target, and constraints before further action.

1.2 Infeasible Strategy. The task objective is understood, but the current high-level approach cannot successfully reach it from the observed state. Assign this leaf when the failure is the chosen strategy itself rather than a missing interface route or a low-level action error. Recovery replaces the strategy with a feasible alternative.

1.3 Missing / Wrong Step Plan. The objective and broad strategy are viable, but required steps, dependencies, or ordering are missing or incorrect. Assign this leaf when recovery requires reconstructing what must happen next or what prerequisite step was skipped.

1.4 Progress Tracking Drift. The agent's belief about completed and pending requirements no longer matches the observable task state. Assign this leaf when the primary problem is losing track of progress, causing repeated, skipped, or misordered work. Recovery reconciles verified progress with the remaining requirements.

1.5 Termination Control Failure. The agent is about to stop before the task is complete or continues acting after the required result has already been achieved. Assign this leaf when the central recovery need is to verify completion status and either terminate or resume accordingly.

2.1 Evidence / Feedback Misinterpretation. Relevant GUI evidence is observable, but the execution history interprets that evidence incorrectly. Assign this leaf when recovery primarily requires re-reading visible feedback or action effects rather than locating a different interface context.

2.2 Active Context Misperception. The agent reasons as if a different application, window, page, tab, document, or other active context were in focus. Assign this leaf when the current context itself has been misidentified. Recovery first re-identifies the active context before resuming the task.

2.3 Selection / Focus State Misperception. The agent misidentifies which object is selected or which control currently has input focus. Assign this leaf when the surrounding context is correct but the local selection or focus state is not. Recovery re-establishes the intended selection or focus and verifies it visually.

2.4 Outcome / Authoritative State Mismatch. The agent's belief about whether a previous action succeeded, failed, or changed the task state conflicts with the authoritative state visible in the GUI. Assign this leaf when recovery requires checking the actual outcome and reconciling subsequent execution with that state.

2.5 Blocking Layer Omission. A modal, dialog, popover, overlay, or other interface layer changes what can be interacted with, but the agent fails to account for it. Assign this leaf when recognizing and resolving the blocking layer is necessary before the intended interaction can proceed.

3.1 Feature Route Knowledge Failure. The agent does not know how to reach the interface feature required by the task. Assign this leaf when the missing information concerns the navigation or access route to a feature rather than the task plan itself. Recovery discovers and verifies a valid route to the required feature.

3.2 Control / Parameter Usage Knowledge Failure. The required feature or control is available, but the agent does not know how to use its options, parameters, syntax, or interaction rule correctly. Assign this leaf when recovery requires learning or validating the control's usage rather than merely locating it.

4.1 Target Localization Execution Failure. The intended control or object is conceptually correct, but the concrete interaction target is spatially mislocalized. Assign this leaf when the agent knows what to interact with and recovery primarily requires re-localizing the target in the current GUI state.

4.2 Click / Activation Execution Failure. The intended target is identified, but the activation action does not correctly trigger it. Assign this leaf when the failure lies in executing the click or activation rather than choosing or locating the target.

4.3 Viewport / Continuous Gesture Execution Failure. A scroll, drag, swipe, resize, or other continuous spatial interaction is executed incorrectly or fails to reach the required viewport state. Assign this leaf when recovery requires re-executing that gesture while verifying its observable effect.

4.4 Input / Invocation Execution Failure. Text entry, keyboard input, command invocation, or another discrete input is executed incorrectly even though the intended operation is known. Assign this leaf when recovery requires correcting the concrete input or invocation.

4.5 Unintended State-Changing Action. An action modifies persistent task state in a way that was not intended. Assign this leaf when recovery must first undo, contain, or compensate for that change before normal task progress can safely continue.

5.1 Required Prerequisite Unavailable. A resource, file, application state, dependency, or other prerequisite needed for the next valid step is unavailable. Assign this leaf when the blocker is a missing prerequisite rather than a navigation or execution error. Recovery restores or obtains the prerequisite when possible.

5.2 Access / Authorization Block. The required action is blocked by login state, permission, authorization, or access control. Assign this leaf when progress requires satisfying that access condition before the task can continue.

5.3 Transient Unavailability / Not Ready. The required interface state or service is temporarily unavailable, loading, or not yet ready. Assign this leaf when waiting, bounded retry, or restart is appropriate and no stronger permanent blocker is supported by the evidence.

C.3 Annotation Boundaries and Ambiguity Handling

The main boundary between THINK and KNOW is whether the missing information concerns the task strategy or the interface mechanism used to realize an otherwise understood strategy. The boundary between SEE and DO is whether the agent has formed an incorrect belief about the current state or instead understands the state but executes the intended interaction incorrectly. BLOCKED is reserved for prerequisite, access, or transient conditions that prevent the intended step despite an otherwise appropriate recovery direction. NORMAL is used only when no recovery-specific condition is supported by the prefix.

When more than one leaf appears plausible, annotators select the leaf that best captures the first recovery objective required from the current checkpoint. Evidence after the checkpoint and the historical root-cause label are excluded from this decision. If the available prefix cannot support a

leaf at this specificity, the case is marked for abstention and additional review rather than forcing a label.

C.4 Pilot Refinement of the Taxonomy

The initial taxonomy was formed by synthesizing failure categories and recovery strategies from prior GUI-agent work. During pilot recovery experiments, we instantiated candidate categories as alternative recovery prompts on representative failure cases and examined whether the proposed distinction changed the first recovery objective from the current checkpoint. A distinction was retained when the compared cases consistently required different recovery directions; distinctions that led to the same practical recovery direction were merged. The resulting organization therefore emphasizes recovery action rather than retrospective root-cause granularity. This refinement produced the 21 leaves in Table 8, which are used consistently for checkpoint annotation and macro-policy retrieval.

C.5 Taxonomy Fields Used by the Prompt Provider

Each leaf entry supplies the fields consumed by the recovery template in Appendix D. The `category_definition` is the leaf definition above, and `recovery_protocol` is the corresponding macro-policy in Table 8. The `verification` field asks the executor to verify that the observable condition motivating the leaf has been resolved or reconciled. The `resume_condition` requires that this verification succeed and that a valid next task step be available. The `do_not` field prevents the executor from repeating the unsupported belief, strategy, or action that motivated the diagnosis. These fields specify recovery objectives and guards; they do not encode fixed GUI coordinates or low-level action sequences.

C.6 Checkpoint-Level Annotation

Each checkpoint is assigned a leaf-level diagnosis

$$y_t^{\mathrm{diag}} \in \mathcal{Y}, \tag{47}$$

where $\mathcal{Y}$ denotes the 21-leaf taxonomy. Labels are based on the task, GUI observations, actions and outcomes, and execution history available at the checkpoint. Different checkpoints in one trajectory may therefore receive different diagnoses.

When the available evidence is insufficient to support a leaf-level diagnosis, the sample is marked for abstention and additional review rather than forcing a taxonomy label. Abstention is not counted as an additional taxonomy leaf.

C.7 From Diagnosis to Recovery Guidance

Each leaf maps to a fixed recovery macro-policy,

$$m_t = \mathcal{M}\left(y_t^{\mathrm{diag}}\right). \tag{48}$$

which specifies an objective rather than a concrete GUI action sequence. For example, a state-perception diagnosis calls for reconciling the interface state, while an execution diagnosis may call for retrying or undoing an action.

At inference time, the Prompt Provider combines the predicted diagnosis and its macro-policy into the guidance in Appendix D; the base agent chooses the state-dependent GUI actions.

D Recovery Prompt Construction

After the Timing Gate triggers, ReTIME maps the predicted diagnosis $\hat{y}_t^{\mathrm{diag}}$ to the macro-policy in Appendix C. A fixed template combines the task instruction, diagnosis, and category-specific guidance before injection into the base agent's context.

The complete recovery prompt is reproduced below. The fields `label_code` and `label_name` are determined by the predicted diagnosis, while `category_definition`, `recovery_protocol`,

verification, resume_condition, and do_not are retrieved from the corresponding taxonomy entry.

```
You are recovering an interrupted GUI task from the current computer state.

## ORIGINAL TASK

{original_instruction}

## RECOVERY OBJECTIVE

Continue from the current state and complete the original task correctly.

Do not restart or redo existing work unless the current evidence shows that it is nec
Preserve progress that is already verified.

Before taking the next action:

1. Inspect the current screenshot and recent execution history.
2. Determine what state is currently verified, what remains uncertain, and what task
3. Apply the recovery protocol below to the current GUI state.

## RECOVERY DIAGNOSIS

Category: {label_code} | {label_name}

{category_definition}

## RECOVERY PROTOCOL

{recovery_protocol}

## VERIFY BEFORE RESUMING

{verification}

Resume normal task execution when:
{resume_condition}

## DO NOT

{do_not}

## EXECUTION RULE

Ground the recovery protocol in the current visible GUI state.

Use the diagnosis to constrain the recovery strategy, but do not assume that
any specific GUI element, coordinate, action outcome, or state is present
unless it is supported by the current screenshot or execution history.

Take the smallest safe action that advances the recovery protocol.
After each action, inspect the resulting state before deciding the next action.

Once the recovery condition is satisfied, continue the original task normally.
Before terminating, verify every requirement in the original instruction.
```

E Recovery Diagnosis Model Details

E.1 Input Construction

Each diagnosis is made after an action and before recovery. The multimodal input x_t contains the task instruction, execution history, up to two prior screenshots, the current screenshot, and the current thought and action. The textual taxonomy context $\mathcal{T}$ provides each leaf's code, name, decision rule, policy identifier, first safe action, and verification criterion. Evidence is limited to the prefix through checkpoint t, and agent thought is treated as a claim rather than an observation.

The annotation record follows schema version `3.1` with `sample`, `context`, `classification`, `evidence`, `recovery`, and `human_review` sections. The classifier generates `classification`, `evidence`, and `recovery`; sample and context fields come from the data record, and human-review fields are reserved for reviewers.

E.2 LoRA Training Corpus

The LoRA diagnosis model is not trained on ReTIME-Bench. Its supervision is constructed from the OSWorld task complement obtained after excluding the benchmark task union available when the training corpus is constructed. After class-balanced checkpoint selection and materialization, the corpus contains 5,782 post-action cases from 258 tasks and six source agents, covering all 21 taxonomy leaves. We split by task UUID into 5,190 training cases from 232 tasks and 592 validation cases from 26 tasks, with no task overlap between these two splits. Each case contains the task instruction, the thought–action prefix, up to the three most recent screenshots, and the current thought and action; the target is the current checkpoint's taxonomy leaf. ReTIME-Bench labels, future states, recovery outcomes, and counterfactual utilities are excluded from optimization. The post-construction audit finds no exact overlap between a materialized training checkpoint and a ReTIME-Bench checkpoint.

E.3 Model Adaptation

We instantiate the diagnosis model with Qwen3-VL-8B-Instruct (Bai et al., 2025). The visual encoder is frozen, while LoRA (Hu et al., 2022) is applied to the language model parameters. Given x_t and taxonomy context $\mathcal{T}$, the model predicts the three primary classifier sections

$$z_t = \{\texttt{classification}, \texttt{evidence}, \texttt{recovery}\}, \qquad y_t \in \mathcal{Y} \cup \{\perp\}. \tag{49}$$

The `classification` section records the predicted label and policy, confidence, summary, alternative label, and uncertainty. The `evidence` section contains one to four reviewable references, and `recovery` specifies the first action, its observable verification condition, and an optional `do_not` guardrail.

For non-abstaining outputs, `label_code` and `policy_id` must refer to the same taxonomy entry. The structured record exposes evidence and uncertainty for review.

E.4 Training Objective

We optimize the diagnosis model using standard autoregressive cross-entropy over the serialized structured target:

$$\mathcal{L}_{\text{diag}} = -\sum_{m=1}^{|z_t|} \log p_\theta \left(z_{t,m} \mid x_t, \mathcal{T}, z_{t,<m} \right). \tag{50}$$

Leaf-level evaluation parses `classification.label_code`; the evidence and recovery fields make the prediction auditable.

E.5 Inference

Deterministic decoding produces one schema-valid record. For a valid leaf prediction $\hat{y}_t \in \mathcal{Y}$, the recovery executor applies

$$\hat{y}_t \to \pi(\hat{y}_t). \tag{51}$$

The selected leaf determines the first safe macro-action, and the executor instantiates state-dependent low-level parameters.

When evidence is insufficient, the classifier returns `ABSTAIN_SAFE_OBSERVE`, leaves `label_code` null, and records the missing evidence. Malformed or weakly grounded outputs follow the same bounded safe-observation path before rediagnosis.

F ReTIME-Bench Construction and Annotation

ReTIME-Bench contains 672 annotated checkpoints from 133 replayable failed OSWorld trajectories generated by six source agents. Multiple post-failure checkpoints are sampled per trajectory, and every benchmark input is restricted to information available at or before the queried checkpoint. The benchmark is held out from the LoRA training and validation corpus described in Appendix E.

Each ReTIME-Bench checkpoint carries two complementary supervision signals,

$$\left(y_{i,t}^{\text{timing}}, y_{i,t}^{\text{diag}}\right), \tag{52}$$

where $y_{i,t}^{\text{timing}}$ determines **WHEN** to recover and $y_{i,t}^{\text{diag}}$ identifies **HOW** recovery should proceed.

F.1 Recovery Timing Annotation

Timing supervision is derived from counterfactual recovery outcomes. An identifiable failure does not automatically require immediate intervention: continued execution may expose a substantially better recovery opportunity.

F.1.1 Timing Label Construction

For failure trajectory i, let $\mathcal{C}_i = \{t_1, \ldots, t_K\}$ denote its ordered candidate checkpoints. We independently launch recovery rollouts from each checkpoint. Given R_j valid rollouts at checkpoint t_j, its recovery utility is estimated as

$$U_i(t_j) = \frac{1}{R_j} \sum_{r=1}^{R_j} R_e\big(\tau_{i,j,r}^{R}\big), \tag{53}$$

where R_e is the structured recovery score defined in Appendix A. We target at least three valid recovery rollouts per checkpoint.

Following Section 5.2.1, utilities are normalized within each trajectory:

$$\widetilde{U}_i(t) = \frac{U_i(t) - U_i^{\min}}{U_i^{\max} - U_i^{\min} + \epsilon}, \qquad \epsilon = 10^{-8}, \tag{54}$$

where

$$U_i^{\min} = \min_{t \in \mathcal{C}_i} U_i(t), \qquad U_i^{\max} = \max_{t \in \mathcal{C}_i} U_i(t). \tag{55}$$

Normalization is performed independently within each trajectory so that timing depends on relative recovery opportunities rather than task-specific utility scales.

For each non-final checkpoint t_j, we define the future recovery gain as

$$G_i(t_j) = \max_{k>j} \widetilde{U}_i(t_k) - \widetilde{U}_i(t_j). \tag{56}$$

The timing label is then assigned as

$$y_{i,j}^{\text{timing}} = \begin{cases} \texttt{CONTINUE}, & G_i(t_j) > \delta, \\ \texttt{RECOVER}, & G_i(t_j) \le \delta, \end{cases} \qquad \delta = 0.15. \tag{57}$$

Thus, `CONTINUE` means that a later checkpoint improves normalized utility by more than δ; `RECOVER` means that no such improvement is available. The current checkpoint need not be the global maximum.

F.1.2 Edge Cases and Quality Control

Near-flat recovery trajectories. If recovery utilities within a trajectory exhibit negligible variation,

$$U_i^{\max} - U_i^{\min} < \epsilon_{\text{flat}}, \qquad \epsilon_{\text{flat}} = 0.01, \tag{58}$$

all valid failure checkpoints are labeled `RECOVER`, since the measured landscape provides no meaningful evidence that delaying intervention yields a better recovery opportunity.

Final checkpoint. The final candidate checkpoint t_K is always labeled `RECOVER`, because no later candidate intervention point remains. This is a finite-horizon stopping rule and does not imply that t_K has the highest recovery utility.

Non-monotonic labels. Timing labels are computed independently at each checkpoint. We therefore do not enforce a monotonic transition from `CONTINUE` to `RECOVER`. For example,

$$\texttt{CONTINUE} \rightarrow \texttt{RECOVER} \rightarrow \texttt{CONTINUE} \rightarrow \texttt{RECOVER} \tag{59}$$

is valid when supported by the measured recovery utilities. This avoids assuming a unimodal recovery landscape.

Borderline checkpoints. We define the timing margin

$$M_i(t_j) = \left| G_i(t_j) - \delta \right|. \tag{60}$$

Checkpoints with $M_i(t_j) < 0.05$ are marked as *borderline* for quality control. This metadata does not modify the binary timing label.

F.1.3 Information Boundary

Counterfactual outcomes are used to construct labels, but benchmark inputs end at the queried checkpoint. Future checkpoints, actions, outcomes, utilities, and final task results remain annotation-side information.

F.2 Recovery-Oriented Diagnosis Annotation

Each checkpoint is additionally assigned a leaf-level recovery diagnosis

$$y_{i,t}^{\text{diag}} \in \mathcal{Y}, \tag{61}$$

where $\mathcal{Y}$ denotes the recovery-oriented taxonomy defined in Appendix C.

The diagnosis captures the current recovery need, which may differ across checkpoints in the same trajectory. Annotators use the task, GUI observations, execution history, actions, and observed outcomes through the queried checkpoint. Insufficient evidence triggers review rather than a forced label.

F.2.1 Diagnosis Annotation Guidelines

Annotators first identify the observable condition that prevents or threatens valid continuation from the queried checkpoint. They then apply the leaf definitions and boundary rules in Appendix C. The selected leaf must be supported by prefix evidence and must correspond to the first recovery objective needed at that state. Historical root-cause labels, future source-agent actions, future GUI states, counterfactual recovery utilities, and final outcomes are not used to choose the diagnosis label.

For ambiguous cases, annotators record the competing leaf and the missing or conflicting evidence. Cases without sufficient prefix evidence are routed to additional review rather than resolved by consulting future trajectory information. Adjudication therefore resolves label ambiguity while preserving the checkpoint information boundary.

Human–AI annotation. Diagnosis annotation is conducted by human experts with AI assistance. For each checkpoint, the AI system proposes a candidate leaf label together with supporting evidence from the prefix-available task instruction, GUI state, and execution history. Human experts review the proposed label and evidence, correct them when necessary, and determine the final diagnosis label.

The AI system is used as an annotation assistant rather than as the final decision maker. Final labels are determined and validated by human annotators. Neither the AI assistant nor the human annotators use future source-agent actions, future GUI states, counterfactual recovery utilities, or final task outcomes when assigning the diagnosis label.

Timing and diagnosis are treated as separate supervision signals. In particular,

$$y^{\text{timing}} = \texttt{CONTINUE} \quad \not\Rightarrow \quad y^{\text{diag}} = \textsc{Normal}. \tag{62}$$

A problem may already be diagnosable even when a later checkpoint offers a better intervention opportunity. Timing asks whether to recover *now*; diagnosis specifies the recovery direction.

F.3 Dataset Construction and Quality Control

ReTIME-Bench is constructed from unsuccessful GUI-agent trajectories collected from multiple source agents on OSWorld. From each selected trajectory, we sample candidate checkpoints from recovery-relevant execution regions and reconstruct the execution prefix available at each checkpoint. Each resulting sample contains the task instruction, current GUI observation, relevant execution history, and interaction-budget information available at that step.

Timing labels follow Section F.1; diagnosis labels follow Appendix C. They are constructed independently and joined at the checkpoint level.

Quality control targets borderline timing outcomes, ambiguous diagnoses, and information leakage. Borderline checkpoints are identified by the timing margin, low-evidence diagnoses receive additional review, and benchmark inputs are checked for annotation-only fields. ReTIME-Bench thereby evaluates both the intervention decision and the recovery direction at the same checkpoint.

Benchmark composition checks. The released benchmark totals 672 checkpoints from 133 failed trajectories and six source agents, matching the composition reported in the main text. Counts are checked after annotation joins so that each retained checkpoint has one valid timing label and one diagnosis record, while timing-only counterfactual information remains annotation-side metadata.

Leakage checks. The benchmark input is audited to exclude future source-agent actions, future GUI states, recovery utilities, outcomes of recovery rollouts, root-failure annotations, and other fields created only for counterfactual labeling. The LoRA corpus is constructed after excluding the ReTIME-Bench task union, and the post-construction audit described in Appendix E checks for exact checkpoint overlap.