GUI-HARVEST: SELF-IMPROVING GUI AGENTS THROUGH EVIDENCE-DRIVEN HARNESS EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

The executable harness surrounding a GUI model determines how observations are assembled, actions are executed, and verification, recovery, and termination are controlled. Compared with harness optimization for non-GUI agents, automatically optimizing this harness poses three coupled challenges: reconciling model intent with observed visual effects, diagnosing failures under variable execution outcomes, and identifying recurrent failure patterns across tasks and translating them into reusable runtime changes. We introduce GUI-HARVEST, an automatic harness optimizer that enables self-improving GUI agents with frozen backbone models. First, to ground diagnosis in observed action effects, it aligns model outputs and executed actions with before-and-after screenshots, producing findings tied to specific interface transitions. Second, to account for execution variability, it treats repeated executions of the same task as a joint evidence unit, using within-task comparisons to locate outcome-relevant behavioral differences. Third, to derive reusable interventions from task-local evidence, it consolidates verified findings across tasks into recurring failure patterns, maps them to bounded source-code edits with predictions recorded before evaluation, and checks the predicted behavioral effects alongside task performance through repeated execution. Experiments on OSWorld-Verified show consistent held-out gains across six general-purpose open, GUI-specialized open, and proprietary backbone models; Qwen3-VL-32B-Instruct gains 12.33 points on the full suite. Frozen-harness transfer improves GPT-5 by 13.87 percentage points on WindowsAgentArena at 50 steps without further optimization. With the same backbone and initial harness, GUI-HARVEST outperforms Self-Harness and Meta-Harness, suggesting that GUI-specific diagnosis and validation help produce harness improvements that generalize to unseen tasks.

1 INTRODUCTION

GUI agents rely on an executable *runtime harness* to construct observations and context, execute actions, and control verification, recovery, and termination. Agent S and Agent S2/S3 demonstrate how these runtime choices affect computer-use performance (Agashe et al., 2025a;b; Gonzalez-Pumariega et al., 2026b). Harness adaptation enables self-improving GUI agents whose own executions guide reusable changes to the runtime while model weights remain fixed. Building on methods that retain improvements from execution experience (Shinn et al., 2023; Zelikman et al., 2024), automatic harness optimizers such as Meta-Harness and Self-Harness revise prompts and code using textual traces and feedback in non-GUI domains, including mathematical reasoning and software engineering (Lee et al., 2026; Zhang et al., 2026a). GUI harness optimization poses three coupled challenges that these generic methods do not directly address.

First, *model intent must be reconciled with observed visual effects*. A textual trace may claim success while the screen shows an unresolved state: clicking Save, for example, does not complete the task if a dialog remains open. **Second**, *failures must be diagnosed under variable execution outcomes*. The same agent can succeed or fail on repeated executions of the same task, even under deterministic decoding (Gonzalez-Pumariega et al., 2026a). In our baseline runs, 11.8–20.4% of Full tasks produce both zero and positive scores across three executions, depending on the backbone (Appendix Table 8). A single failed trajectory is therefore an incomplete account of the agent's behavior; other runs may reveal successful alternatives and where execution diverged. **Third**, *task-specific findings must yield reusable harness changes*. A patch motivated by one task need not help elsewhere. Similar failures

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

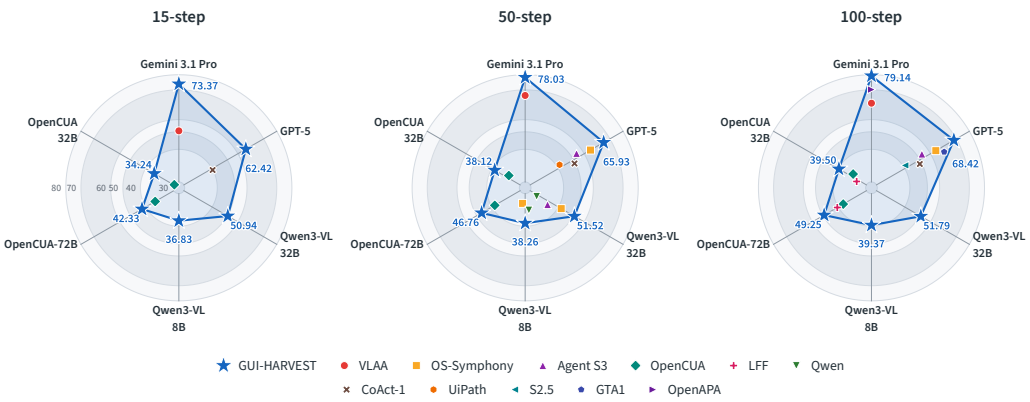

Figure 1: OSWorld-Verified results for six backbones at 15, 50, and 100 maximum environment steps. Axes fix the backbone; markers compare harnesses and agent frameworks at each budget. GUI-HARVEST uses only 15-step Search rollouts for optimization and remains frozen for evaluation, achieving the highest score in every model–budget comparison shown.

across tasks must be linked to a shared runtime mechanism and translated into edits that generalize. Such edits must also be checked against the shared failure pattern: higher aggregate scores alone do not show that the recurring failure was corrected.

To address the challenges above, we introduce GUI-HARVEST (**H**arness **A**daptation from **R**epeated **V**isual **E**xecutions and **S**tructured **T**rajectories), which automatically optimizes harnesses for GUI agents with frozen backbones through three corresponding design choices (Figure 2). **First**, to ground diagnosis in visual effects, the Evidence Analyst aligns model outputs and executed actions with before-and-after screenshots. The resulting findings identify discrepancies between intent and observed state changes and retain links to supporting text and images. **Second**, to account for execution variability, it compares repeated executions of the same task as a joint evidence unit, using successful paths when available to identify outcome-relevant behavioral differences. **Third**, to derive reusable interventions, the Cross-task Clusterer groups verified findings into *recurring failure patterns*. The Harness Engineer maps these patterns to bounded source-code edits and records predictions of observable behavioral effects of these edits before evaluation. The Validator promotes edits only when repeated executions pass score checks on search and held-out validation tasks and behavioral checks of the predictions. An update ledger records decisions to guide subsequent proposals. The test set remains sealed until optimization ends.

We evaluate GUI-HARVEST on OSWorld-Verified across six backbones spanning general-purpose open, GUI-specialized open, and proprietary models. At 15 steps, Test scores improve by 1.49–10.16 percentage points across all six frozen backbones. Qwen3-VL-32B-Instruct achieves the largest full-suite gain, reaching 50.94% (+12.33 points; Table 1). Harnesses optimized at 15 steps improve further at 50- and 100-step evaluation budgets, with Gemini 3.1 Pro reaching 79.14% at 100 steps without further optimization (Figure 1). Frozen-harness transfer to WindowsAgentArena (WAA) at 50 steps raises Qwen3-VL-32B-Instruct and GPT-5 scores by 6.47 and 13.87 points, respectively, *without WAA optimization* (Table 11). On Qwen3-VL-32B-Instruct, GUI-HARVEST also outperforms Self-Harness and Meta-Harness from the same initial harness, suggesting that GUI-specific diagnosis and validation help produce harness improvements that generalize better to unseen tasks. Its final harnesses also outperform LFF's released harnesses on both OpenCUA backbones (Table 2).

Our contributions are threefold:

- **Evidence-driven self-improvement for GUI agents.** We formulate harness adaptation as evidence-driven improvement of an executable runtime around a frozen GUI model.
- **Multimodal diagnosis and intervention validation.** We propose GUI-HARVEST to convert repeated multimodal execution evidence into reusable code changes validated by score and behavior checks.
- **Evaluation across backbones and environments.** We demonstrate Test gains across six backbones, performance at larger step budgets, and frozen-harness transfer beyond OSWorld.

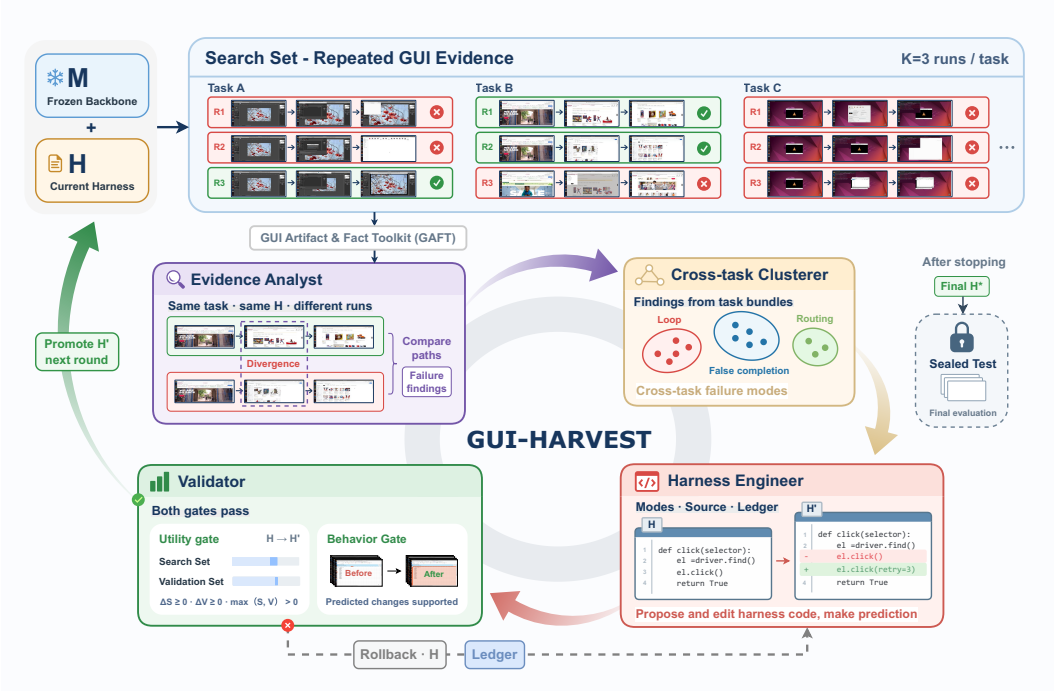

Figure 2: GUI-HARVEST self-improvement loop. A frozen backbone and the current harness produce repeated search executions ($K = 3$), which support task-local diagnosis, cross-task clustering, and source-code edits with explicit behavioral predictions. The Validator combines score and behavior checks to promote or roll back each edit; the test set remains sealed until optimization stops.

2 Problem Setting

Let M be a frozen GUI-capable model and $H_0 \in \mathcal{H}$ an executable harness. A harness contains the editable program around the model: prompts and context construction, memory, action and tool interfaces, control flow, verification, recovery, and termination. The optimizer cannot modify model weights, benchmark tasks, evaluators, environment infrastructure, or resource limits. A capability manifest specifies which harness files may be edited and where new modules may be created.

Before optimization, tasks are divided into search $\mathcal{S}$, validation $\mathcal{V}$, and sealed test $\mathcal{T}$ by a domain-stratified random split with fixed per-domain proportions. Search trajectories supply optimization evidence. Validation trajectories are hidden from the optimizer and contribute only aggregate scores. The sealed test set is evaluated after optimization stops. An optimization round is the search performed at a fixed harness state H_t. The optimizer may evaluate up to P L1-valid candidate edits in that round. Only promotion sets H_{t+1} and starts the next round; exhausting all P attempts terminates optimization at H_t. We cap the process at B rounds. The optimizer returns $\mathcal{O}(M, H_0, \mathcal{S}, \mathcal{V}, B, P) \to H^*$. For a task x_i, one run under harness H is $\tau_{i,r}(H) = \tau(M, H, x_i; \xi_{i,r})$, where $\xi_{i,r}$ includes model sampling, rendered observations, interface timing, and application responses. If $R(\tau) \in [0, 1]$ is the benchmark's raw evaluator score, the repeated mean on a task set D is

$$\widehat{J}_{D,K}(H) = \frac{1}{|D|} \sum_{i \in D} \frac{1}{K} \sum_{r=1}^{K} R(\tau_{i,r}(H)) \,. \tag{1}$$

3 GUI-HARVEST

In this paper, a *task* is a user instruction to achieve a specified outcome in one or more desktop applications, together with an initial environment state and an evaluator that scores the resulting state. GUI-HARVEST improves the executable *harness* around a frozen *backbone*. As shown in Figure 2, repeated task executions supply evidence for four optimizer roles: the Evidence Analyst, Cross-task

Clusterer, Harness Engineer, and Validator. Together, they connect observed failures to reusable code changes and test whether those changes have their intended effects.

3.1 THE SELF-IMPROVEMENT LOOP

Starting from H_0, we collect K *rollouts* (complete task executions) per search and validation task, reusing runs already available for the current harness. Search provides detailed execution evidence; validation exposes only aggregate scores. Each round diagnoses failures under H_t, groups recurring behaviors, and proposes and evaluates an edited harness $\widetilde{H}_t$. *Promotion* adopts a candidate that passes the performance and behavior checks; *rollback* restores H_t after rejection. An *update ledger*, a record of edits, predictions, scores, and decisions, guides subsequent proposals. This closes a self-improvement loop in which accepted harness changes reshape subsequent executions, supplying evidence for further adaptation.

After promotion, we analyze the candidate's search rollouts and rebuild the behavior modes. After rejection, we retain the current harness's verified findings and modes, and the engineer proposes a materially different edit. Appendix Algorithm 1 summarizes this loop; Appendix G specifies the role interfaces, and Appendix H gives the formal record definitions and promotion rule.

A candidate that passes the L0/L1 code checks (Section 3.5) and enters GUI evaluation consumes one attempt in the current round. Promotion advances to the next round and opens a fresh attempt budget; if all $P = 5$ attempts at a harness state are rejected, optimization stops. The main protocol permits at most $B = 10$ rounds. L0/L1 repairs made before a valid candidate enters GUI evaluation consume neither budget. After stopping, we freeze H^* for sealed-test evaluation; the test set remains unavailable during optimization.

3.2 EVIDENCE ANALYST: DIAGNOSIS FROM REPEATED EXECUTIONS

The Evidence Analyst compares repeated runs of the same task to produce *findings*: evidence-supported descriptions of problematic behavior in a specific run. This *task-local* analysis keeps the task and harness fixed; successful runs, when available, provide alternative paths to completion.

Task-round bundles. Each task is run K times ($K = 3$ in the main setting) from *clean environment snapshots* that restore its initial state. Runs share the model, harness behavior hash, action limit, and decoding settings. Each run records a *multimodal trajectory*—text, planned and executed actions, screenshots, and tool outputs at each step—along with its benchmark score and runtime and termination metadata. A *task-round bundle* groups these runs with a Task Card containing the instruction, domain, related applications, feasibility annotation, setup steps, and any published hint, together with a *Harness Card* summarizing the runtime and its observable outputs.

A bundle enters diagnosis when at least one valid run scores zero. The analyst diagnoses these zero-score runs, using the remaining runs as comparison evidence. All valid scores, including partial scores, contribute to Equation 1.

Organizing execution evidence. We use the GUI Artifact and Fact Toolkit (GAFT) as supporting infrastructure to align text, actions, and before-and-after screenshots by execution step, retaining links to tool outputs, final scores, and evidence files. It computes *mechanical facts*, such as repeated actions and pixel changes; the analyst interprets their relevance to failure. A visual index supports retrieval of complete traces and full-resolution screenshots.

Diagnosing failures. One budget-limited analyst examines each bundle independently. It compares the *terminal application state* with the task objective, then locates where runs diverge and checks whether intended actions, executed actions, and subsequent screens agree. A finding identifies a decisive step in a failed run, describes the observed behavior and machine outcome, and links the diagnosis to run/step evidence. A *deterministic verifier* checks the harness version, cited run and step, quotation locality, and mechanically testable trajectory facts before retaining the finding. These checks establish evidence consistency, while the behavioral interpretation remains the analyst's assessment.

3.3 CROSS-TASK CLUSTERER: RECURRING BEHAVIOR MODES

The Cross-task Clusterer groups verified findings from independent tasks into *behavior modes*: recurring patterns that suggest shared harness changes. To keep this induction grounded without imposing a fixed mode taxonomy, each finding is first assigned a deterministic outcome category from its decisive action and observed machine result. Within each category, the clusterer uses the finding's mechanism and evidence-supported counterfactual to discover open-vocabulary modes shared by at least two tasks. The categories constrain the comparison space; they do not prescribe mode names or code patches.

Each mode records an observable mechanism, a membership test, a behavioral target, its member findings, and independent-task support. Member identifiers retain the link to task-local evidence. Modes are stored as versioned round artifacts and regenerated from the updated harness's executions after promotion.

3.4 HARNESS ENGINEER: SOURCE-AWARE INTERVENTIONS

The Harness Engineer translates behavior modes into harness code changes. Its *source-aware* session allows code inspection, editing, and testing using the modes, representative evidence, Harness Card, and relevant rejected attempts from the ledger. Within the capability manifest, the engineer selects a small compatible set of modes, locates the relevant runtime code, applies a bounded *patch*, and runs local tests.

Before GUI evaluation, the engineer records an update plan linking the target modes and evidence to the *source surface* (code region), proposed edit, observable predictions, and search tasks used to check them. The plan also specifies *invariants*: interfaces or behaviors that must remain intact. Recording predictions before evaluation makes the edit testable: a completion-check patch might predict that the agent resolves an open save dialog before declaring success.

3.5 VALIDATOR: UTILITY AND BEHAVIORAL CHECKS

The Validator checks task performance and predicted behavioral changes. Two deterministic code checks first screen $\widetilde{H}_t$: L0 checks edit permissions, diff limits, syntax, and static rules; L1 checks imports, interface compatibility, unit tests, and a *smoke test*. Passing candidates are run K times on every search and validation task under the fixed execution-cost constraint. This batch supplies scores and post-edit bundles for the checks below; promoted candidates also supply the next round's search evidence.

Utility hard gate. Using the utility in Equation 1, let Δ_D denote the candidate's mean score minus the current harness's mean score on split $D \in \{\mathcal{S}, \mathcal{V}\}$. The hard gate requires no decline on either split and a strict gain on at least one:

$$\mathrm{HardGate}_{\mathcal{S},\mathcal{V}} = (\Delta_{\mathcal{S}} \geq 0) \wedge (\Delta_{\mathcal{V}} \geq 0) \wedge \big(\max(\Delta_{\mathcal{S}}, \Delta_{\mathcal{V}}) > 0\big). \tag{2}$$

Both splits contribute scores, but only search executions are available for diagnosis and behavioral validation.

Behavioral soft gate. This required check tests whether behavior changed as predicted. For each targeted search task, a separate validator compares the frozen prediction and pre-edit task-local finding with the post-edit K-run trajectories and available screen evidence. It returns *supported* if the evidence agrees with the prediction, *contradicted* if it conflicts, or *inconclusive* if insufficient, citing verifiable runs and steps. A deterministic aggregator applies a fixed rule across task verdicts: each prediction needs at least one *determinate* verdict (supported or contradicted) and more supporting than contradicting verdicts. Each validator's visual context is restricted to one task bundle.

The candidate is promoted exactly when it passes L0, L1, the utility hard gate, and the behavioral soft gate. Promotion adopts the candidate, updates the Harness Card, and retains new search bundles. Rejection restores H_t and its verified evidence and modes. The ledger records the patch, predictions, score changes, and verdicts. The loop repeats until a stopping condition in Section 3.1 is met.

Table 1: Paired 15-step results on Test and Full. Parentheses show gains over the corresponding initial harness.

	Initial H_0		GUI-HARVEST H^*	
Backbone	Test	Full	Test	Full
Qwen3-VL-8B	28.79	29.49	**34.84** (+6.05)	**36.83** (+7.34)
Qwen3-VL-32B	43.02	38.61	**53.18** (+10.16)	**50.94** (+12.33)
OpenCUA-32B	32.82	30.71	**34.31** (+1.49)	**34.24** (+3.53)
OpenCUA-72B	41.25	37.65	**44.18** (+2.93)	**42.33** (+4.68)
Gemini 3.1 Pro	66.42	66.46	**72.63** (+6.20)	**73.37** (+6.91)
GPT-5	56.35	55.20	**61.86** (+5.51)	**62.42** (+7.22)

Table 2: Harness-optimizer comparison at 15 steps. Experimental protocols are detailed in Appendix G.8.

General harness optimizers (GUI-adapted search)		
Method	Test	Full
Self-Harness	45.02 (+2.00)	42.08 (+3.47)
Meta-Harness	47.40 (+4.38)	45.10 (+6.49)
GUI-HARVEST	**53.18** (+10.16)	**50.94** (+12.33)
GUI failure-driven optimizers (final harnesses)		
Backbone	LFF	GUI-HARVEST
OpenCUA-32B	31.83 / 32.12	**34.31 / 34.24**
OpenCUA-72B	41.29 / 39.50	**44.18 / 42.33**

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Benchmarks and metrics. Following common OSWorld evaluation practice, we use OSWorld-Verified with 361 tasks after excluding eight Google Drive tasks (Xie et al., 2024; Han et al., 2026). A fixed, domain-stratified random split assigns 80 tasks to Search, 80 to Validation, and 201 to sealed Test while preserving each domain's proportion. Only Search trajectories supply optimization evidence; Validation exposes aggregate scores, and Test is used only after harness optimization. We report mean raw evaluator scores (Eq. 1) on each split and the full 361-task suite, comparing the initial harness H_0 with the selected harness H^*.

Backbones and harnesses. We evaluate Qwen3-VL-8B/32B-Instruct, OpenCUA-32B/72B, Gemini 3.1 Pro, and GPT-5, with target-model weights and decoding settings frozen (Bai et al., 2025; Wang et al., 2025). Qwen3-VL and proprietary backbones start from Agent S3 with its default grounding model (UI-TARS-1.5-7B) and behavior best-of-N disabled ($N = 1$) (Gonzalez-Pumariega et al., 2026b); OpenCUA models use their official coordinate-action runtime. All four optimizer roles in our algorithm use Claude Sonnet 5 with the same role prompts across backbones.

Optimization and evaluation. Harness search uses a 15-step limit and $K = 3$ independent runs per task. It allows at most $B = 10$ rounds, with up to $P = 5$ evaluated candidate edits per round; optimization stops when all five attempts at one harness state fail promotion. Selected harnesses are frozen for OSWorld evaluation at 15, 50, and 100 steps. Qwen3-VL-32B-Instruct and GPT-5 also transfer to WindowsAgentArena at 50 steps without WAA optimization (Bonatti et al., 2025). Ablations on Qwen3-VL-32B-Instruct remove visual evidence, cross-task clustering, repeated optimizer evidence, or behavioral validation, with all selected harnesses evaluated using $K = 3$. Published comparisons retain their reported runtimes and budgets. Appendix G provides more implementation details and comparison protocols.

4.2 HARNESS ADAPTATION AND CROSS-BENCHMARK TRANSFER

Harness adaptation across backbones. Under the unified 15-step, $K = 3$ protocol, GUI-HARVEST improves Test and Full scores for all six frozen backbones (Table 1). Qwen3-VL-32B-Instruct achieves the largest gains, reaching 53.18% on Test (+10.16 percentage points) and 50.94% on Full (+12.33 points). These held-out gains show that harness adaptation generalizes beyond the optimization tasks across both general-purpose and GUI-specialized backbones. Appendix Table 4 provides complete split-level results. Figure 3 traces the self-improvement process, showing Search and Validation gains accumulating over successive promoted edits and variation in adaptation effort across backbones.

Comparison with harness optimizers. Table 2 shows that, from the same initial Qwen3-VL-32B-Instruct/Agent S3 harness, **GUI-HARVEST gains 10.16 Test points, versus 2.00 for Self-Harness and 4.38 for Meta-Harness** (Zhang et al., 2026a; Lee et al., 2026). With the optimizer model and ten-round cap also shared (Appendix G.8), this advantage suggests that GUI-specific diagnosis and validation yield more generalizable edits than the evaluated generic search procedures. GUI-HARVEST also outperforms LFF (Sun et al., 2026) on both OpenCUA backbones; on OpenCUA-

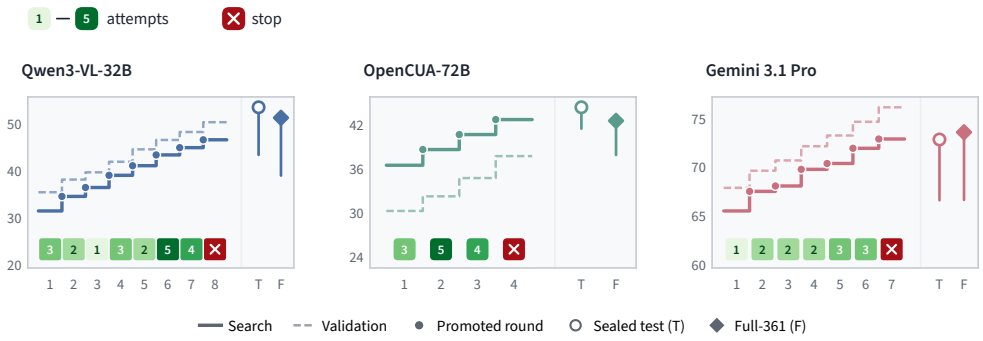

Figure 3: Harness evolution for Qwen3-VL-32B, OpenCUA-72B, and Gemini 3.1 Pro. Solid and dashed curves show Search and Validation scores, respectively; filled circles mark promoted harnesses, and the rightmost markers show Test (T) and Full (F) scores after optimization stops. The numbered tiles report candidate-edit attempts in each optimization round, and the red cross marks the terminal round. Results for the other three backbones are plotted in Appendix Figure 8.

72B, Test and Full scores for GUI-HARVEST reach 44.18% and 42.33%, versus 41.29% and 39.50% for LFF. For LFF, we evaluate the released intervention patches under the same protocol, as the optimizer implementation and complete search configuration are not publicly available; this comparison therefore focuses on the performance of the resulting harnesses.

Transfer to WindowsAgentArena. We additionally transfer the frozen Qwen3-VL-32B-Instruct and GPT-5 harnesses to WAA at a 50-step budget, without using WAA trajectories for optimization. Qwen3-VL-32B-Instruct improves from 38.21% to 44.68% (+6.47), while GPT-5 improves from 50.88% to 64.75% (+13.87). With only platform adapters changed, these gains suggest that runtime improvements learned on OSWorld remain useful under a different platform and task distribution. Appendix Tables 11 and 12 report the paired transfer results and six-domain benchmark context, respectively.

4.3 Matched-Backbone OSWorld Comparison

Figure 1 compares the harnesses optimized by GUI-HARVEST against existing widely adopted GUI models and harnesses, with backbones and step budgets matched. The selected harnesses achieve **the highest score in every comparison shown, spanning six backbones and 15-, 50-, and 100-step budgets.** These advantages extend across general open, GUI-specialized, and proprietary backbones and persist at larger evaluation budgets: all GUI-HARVEST harnesses are optimized at 15 steps and then frozen.

The largest gains include **22.61 percentage points over CoAct-1 with GPT-5** (62.42% versus 39.81%) and **21.68 points over VLAA-GUI with Gemini 3.1 Pro** (73.37% versus 51.69%) at 15 steps. At 50 steps, the selected Qwen3-VL-32B-Instruct harness reaches 51.52%, exceeding the Qwen baseline (32.60%) by 18.92 points. Figure 5 provides more detailed comparisons, and Appendix B reports domain-level results and source details.

Adaptation headroom depends on more than model capability: Qwen and frontier backbones gain 6.91–12.33 Full points over their initial harnesses, versus 3.53–4.68 for OpenCUA (Table 1). OpenCUA is post-trained around its native coordinate-bearing action protocol; its smaller gains are consistent with lower adaptability to added tools and control mechanisms. Increasing the budget from 15 to 100 steps, however, adds only 2.54 and 0.85 points for Qwen3-VL-8B and 32B, versus 5.77 and 6.00 for Gemini 3.1 Pro and GPT-5 (Figure 1). Frontier models thus benefit more from additional interaction, while Qwen's limited gains suggest persistent long-horizon execution bottlenecks (Appendix Table 9).

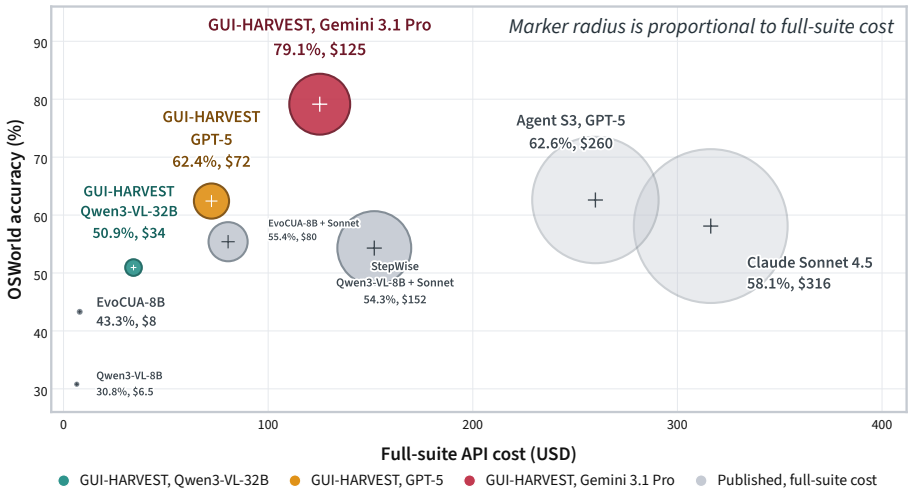

Figure 4: OSWorld accuracy versus full-suite API cost for systems with available cost records. We show the 15-step GUI-HARVEST points for Qwen3-VL-32B-Instruct and GPT-5 and the 100-step point for Gemini 3.1 Pro. Published points retain the models, harnesses, budgets, protocols, and per-task costs reported by Wei et al. (2026); Gonzalez-Pumariega et al. (2026b); full-suite costs are reconstructed as detailed in Appendix F.

Table 3: Component ablations and optimization traces on Qwen3-VL-32B-Instruct. Test and Full gains are measured against the initial harness. Each numbered trace cell denotes a promoted round; its value and shade encode the number of edit attempts required for promotion. X marks termination. All selected harnesses use the same $K = 3$ final evaluation protocol.

	Final harness			Optimization trace (1 – 5 attempts, X stop, unused)									
Variant	**Test**	**Full**	**Rounds**	1	2	3	4	5	6	7	8	9	10
w/o visual evidence	44.54 (+1.52)	40.69 (+2.08)	4	4	4	5	X						
w/o Cross-task Clusterer	45.66 (+2.64)	42.13 (+3.52)	4	4	5	4	X						
Single-run optimizer evidence ($K = 1$)	47.44 (+4.42)	43.12 (+4.51)	5	3	5	2	4	X					
Hard gate only	49.36 (+6.34)	48.08 (+9.47)	9	2	1	2	1	2	3	1	4	X	
Full GUI-HARVEST	**53.18 (+10.16)**	**50.94 (+12.33)**	**8**	3	2	1	3	2	5	4	X		

4.4 Accuracy–Cost Operating Points

Figure 4 compares source-reported operating points, not a shared-budget experiment: our GPT-5 point uses 15 steps, Agent S3/GPT-5 uses 100, and the other published points retain their source protocols. With GPT-5, our 15-step harness **nearly matches Agent S3's accuracy (62.4% versus 62.6%) at approximately 72% lower full-suite API cost** ($72 versus $260). With Gemini 3.1 Pro, our 100-step harness reaches 79.1% at $125, **exceeding Claude Sonnet 4.5 (58.1% at $316) by 21.0 percentage points while costing approximately 60% less.** Qwen3-VL-32B-Instruct offers a lower-cost operating point of 50.9% at $34. A plausible explanation for our cost reductions is that the optimized harness reduces repeated failed actions and unnecessary continuation, which both consume API calls and impede task completion. Correcting these behaviors can therefore improve accuracy and inference efficiency together. More detailed results are reported in Appendix F.

4.5 Component Ablations

All four ablations underperform the complete optimization loop (Table 3), supporting the usefulness of each component. Full GUI-HARVEST reaches 53.18% on Test and 50.94% on Full. Removing visual evidence produces the largest drop, to 44.54% and 40.69%, highlighting its value for GUI failure diagnosis. Removing the Cross-task Clusterer yields 45.66% and 42.13%, supporting cross-task aggregation to guide reusable harness changes. Using one optimization rollout per task ($K = 1$)

remains 5.74 and 7.82 points below the full method, respectively, demonstrating the benefit of repeated execution evidence.

The hard-gate-only variant evaluates nine rounds yet selects a lower-scoring harness than full GUI-HARVEST, which terminates in round eight. Behavioral checks therefore complement aggregate score changes when selecting an intervention. Appendix Figure 7 further shows that $K = 3$ gives the strongest final harness in the measured sweep (50.9%), while $K = 5$ increases rollout cost without improving the result (49.7%).

4.6 Insights into Harness Adaptation

Appendix Tables 9 and 10 show that **useful interventions are backbone dependent.** Qwen3-VL combines code routing with action recovery and completion checks; frontier models mainly benefit from routing, persistence, feasibility, and completion-policy changes. OpenCUA adaptations remain close to its native coordinate-action contract, emphasizing action normalization, file finalization, and terminal semantics. Similar failure symptoms therefore need not admit a universal patch.

Appendix Table 9 highlights **remaining failures that are difficult to detect from screenshots and execution logs.** Early edits address failures with explicit runtime signals, such as repeated actions, invalid calls, or unsaved files. Remaining errors increasingly involve successfully executed actions targeting the wrong object, route, or evaluator-relevant state. These actions may appear valid in screenshots and execution logs despite failing the task, leaving the harness without a clear local signal to trigger recovery or guide a corrective action. This makes further harness-only correction harder.

5 Related Work

GUI models and runtime frameworks. OSWorld provides executable desktop tasks with programmatic evaluation (Xie et al., 2024). Agent S/S2/S3 develop hierarchical and flat workers, experience retrieval, specialized grounding, and coding actions (Agashe et al., 2025a;b; Gonzalez-Pumariega et al., 2026b); CoAct-1 routes between GUI and programmatic execution, and OS-Symphony coordinates planning, grounding, search, coding, and memory (Song et al., 2026; Yang et al., 2026a). VLAA-GUI and related systems add completion verification, loop recovery, multimodal memory, and state access (Han et al., 2026; Zhang et al., 2026d; Zeng et al., 2026; Yang et al., 2026b). Together they define a rich but evaluation-time-static harness design space. We study whether execution-driven adaptation can find a better member of that space for a given frozen model.

GUI trajectory diagnosis and improvement. GUI errors unfold across state–action–state transitions. OSWorld analyzes perception, grounding, planning, knowledge, and environment failures; CUADebug localizes root causes from screenshots and actions; AdaMAST induces adaptive trace taxonomies; and RoTS identifies policy errors for recovery synthesis (Xie et al., 2024; Zhang et al., 2026b; Cemri et al., 2026; Bu et al., 2026). A complementary reliability study uses repeated OSWorld executions to separate environment stochasticity, instruction ambiguity, and planning variability, establishing repeated execution as a richer account of an agent–task pair (Gonzalez-Pumariega et al., 2026a). LFF provides an early GUI-domain attempt to turn failed executions into runtime code changes. Its paper describes providing an LLM with the task instruction, action history, and thought process, then deriving four families of interventions that improve OpenCUA-72B from 42.3% to 48.9% (Sun et al., 2026). GUI-HARVEST incorporates the screenshot sequence into the diagnostic evidence and closes the loop from repeated task-local analysis to cross-task pattern induction, source editing, and post-edit behavioral validation.

We defer discussions of additional related work on self-improving agents to Appendix A.

6 Conclusion

GUI-HARVEST turns repeated multimodal execution evidence into reusable harness changes, validated by task performance and predicted behavior. On OSWorld-Verified, all six frozen backbones improve on held-out tasks, with Qwen3-VL-32B-Instruct reaching 50.94% on the full suite at 15 steps (+12.33 percentage points). It outperforms Self-Harness and Meta-Harness, and its frozen GPT-5 harness gains 13.87 points on WindowsAgentArena at 50 steps. These results support harness adaptation as a mechanism for self-improving GUI agents with frozen backbones.

AI USE STATEMENT

We used generative AI tools to polish the writing and assist with research code implementation. The authors reviewed the resulting text and code and take responsibility for the final paper.

REPRODUCIBILITY STATEMENT

Section 3 and Algorithm 1 describe the GUI-HARVEST optimization procedure, with formal definitions in Appendix H. Section 4.1 specifies the backbones, benchmarks, data splits, and evaluation protocol. Appendix G details the initial harnesses, optimization budgets, baseline adaptations, and leakage controls. Appendices F and J document the inference-cost accounting and representative role prompts, respectively. Our source code is provided in the supplementary material to support reproduction of the main experiments.

REFERENCES

Saaket Agashe, Jiuzhou Han, Shuyu Gan, Jiachen Yang, Ang Li, and Xin Eric Wang. Agent S: An open agentic framework that uses computers like a human. In *International Conference on Learning Representations*, 2025a. URL https://arxiv.org/abs/2410.08164.

Saaket Agashe, Kyle Wong, Vincent Tu, Jiachen Yang, Ang Li, and Xin Eric Wang. Agent S2: A compositional generalist-specialist framework for computer use agents. In *Conference on Language Modeling*, 2025b. URL https://openreview.net/forum?id=zg5is4GJ3R.

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, et al. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2507.19457.

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, et al. Qwen3-VL technical report. *arXiv preprint arXiv:2511.21631*, 2025. URL https://arxiv.org/abs/2511.21631.

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. URL https://proceedings.mlr.press/v267/bonatti25a.html.

Tianpeng Bu, Xin Liu, Qihua Chen, Hao Jiang, Shurui Li, Hongtao Duan, Lu Jiang, Lulu Hu, Bin Yang, and Minying Zhang. Recovering policy-induced errors: Benchmarking and trajectory synthesis for robust GUI agents. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2605.29447.

Mert Cemri, Andrei Cojocaru, Melissa Pan, Shu Liu, Shubham Agarwal, Alexander Krentsel, Jay Tang, Kannan Ramchandran, Joseph E. Gonzalez, Matei Zaharia, Alex Dimakis, and Ion Stoica. Fantastic adaptive taxonomies and how to use them. *arXiv preprint arXiv:2607.16387*, 2026. URL https://arxiv.org/abs/2607.16387.

Mengzhuo Chen, Junjie Wang, Zhe Liu, Yawen Wang, Haiming Zheng, and Qing Wang. From failed trajectories to reliable LLM agents: Diagnosing and repairing harness flaws. *arXiv preprint arXiv:2606.06324*, 2026. URL https://arxiv.org/abs/2606.06324.

Gonzalo Gonzalez-Pumariega, Saaket Agashe, Jiachen Yang, Ang Li, and Xin Eric Wang. On the reliability of computer use agents. *arXiv preprint arXiv:2604.17849*, 2026a. URL https://arxiv.org/abs/2604.17849.

Gonzalo Gonzalez-Pumariega, Vincent Tu, Chih-Lun Lee, Jiachen Yang, Ang Li, and Xin Eric Wang. Scaling agents for computer use. *Transactions on Machine Learning Research*, 2026b. URL https://arxiv.org/abs/2510.02250. Originally released as arXiv:2510.02250.

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. URL https://arxiv.org/abs/2604.21375.

Wen Jiang, Mingmin Chu, Yimeng Tian, Qianxin Zhang, Haofei Yang, Rui Yang, Yang Liu, Tao Lv, and Fangming Li. HarnessEvolve: Learning from reference trajectories for reliable agent self-evolution. *arXiv preprint arXiv:2609.00829*, 2026. URL https://arxiv.org/abs/2609.00829.

Yoonho Lee, Joseph Boen, and Chelsea Finn. Feedback descent: Open-ended text optimization via pairwise comparison. *arXiv preprint arXiv:2511.07919*, 2025. URL https://arxiv.org/abs/2511.07919.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026. URL https://arxiv.org/abs/2603.28052.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, and Yu-Gang Jiang. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 2026. URL https://arxiv.org/abs/2604.25850.

Xiaotian Luo, Dizhan Xue, Fengxingyu Wang, Chuanrui Hu, and Yafeng Deng. HarnessBank: Semantic gene-bank search with gated verification for agent-harness self-evolution. *arXiv preprint arXiv:2607.13683*, 2026. URL https://arxiv.org/abs/2607.13683.

OSWorld Team. OSWorld-Verified Results. Official benchmark leaderboard, 2026. URL https://osworld-v1.xlang.ai/. Accessed 2026-09-23.

Sungho Park, Wonjoong Kim, Rongyuan Tan, Jue Zhang, Wook-Shin Han, Pengfei Gao, Chanyoung Park, Yongqiang Yao, Rao Fu, Elsie Nallipogu, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. AutoSaddler: Automatic harness optimization with durable updates from agent execution traces. *arXiv preprint arXiv:2608.23041*, 2026. URL https://arxiv.org/abs/2608.23041.

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 7957–7968, 2023. doi: 10.18653/v1/2023.emnlp-main.494. URL https://aclanthology.org/2023.emnlp-main.494/.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, et al. UI-TARS: Pioneering automated GUI interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025. URL https://arxiv.org/abs/2501.12326.

Shuai Shao, Kangning Zhang, Qingyao Li, Shijian Wang, Hao Wang, Wenxiang Jiao, Yuan Lu, Yi Guo, Weiwen Liu, and Weinan Zhang. Harness-R1: Learning to edit executable runtime harnesses from agent failure trajectories. *arXiv preprint arXiv:2608.02276*, 2026. URL https://arxiv.org/abs/2608.02276.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652, 2023. doi: 10.52202/075280-0377. URL https://arxiv.org/abs/2303.11366.

Linxin Song, Yutong Dai, Viraj Prabhu, Jieyu Zhang, Taiwei Shi, Li Li, Junnan Li, Silvio Savarese, Zeyuan Chen, Jieyu Zhao, Ran Xu, and Caiming Xiong. CoAct-1: Computer-using multi-agent system with coding actions. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=l1MQVgIKEU.

Xueqiao Sun, Xiaohan Wang, Ludwig Schmidt, Serena Yeung-Levy, and Yuhui Zhang. Learning from failure: Inference-time self-improvement for computer-use agents. *arXiv preprint arXiv:2606.31270*, 2026. URL https://arxiv.org/abs/2606.31270.

Xinyuan Wang, Bowen Wang, Dunjie Lu, Junlin Yang, Tianbao Xie, Junli Wang, Jiaqi Deng, Xiaole Guo, Yiheng Xu, Chen Wu, et al. OpenCUA: Open foundations for computer-use agents. In *Advances in Neural Information Processing Systems*, volume 38, pp. 139756–139806, 2025. doi: 10.52202/085713-4669. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/cc7ae529e945226b0d52ea4ac478c4f3-Abstract-Conference.html.

Jinbiao Wei, Kangqi Ni, Yilun Zhao, Guo Gan, and Arman Cohan. Step-level optimization for efficient computer-use agents, 2026. URL https://arxiv.org/abs/2604.27151.

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, pp. 52040–52094, 2024. URL https://arxiv.org/abs/2404.07972.

Jinghan Xu, Yikai Zhang, Aili Chen, Weiyuan Li, Jiaqing Liang, and Deqing Yang. Verify smarter, evolve further: Efficient harness evolution through behavior-aware verification. *arXiv preprint arXiv:2608.27311*, 2026. URL https://arxiv.org/abs/2608.27311.

Bowen Yang, Kaiming Jin, Zhenyu Wu, Zhaoyang Liu, Qiushi Sun, Zehao Li, JingJing Xie, Zhoumianze Liu, Fangzhi Xu, Kanzhi Cheng, Yian Wang, Qingyun Li, Yu Qiao, Zun Wang, and Zichen Ding. OS-Symphony: A holistic framework for robust and generalist computer-using agents. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 22300–22330, 2026a. doi: 10.18653/v1/2026.acl-long.1021. URL https://aclanthology.org/2026.acl-long.1021/.

Yan Yang, Xiangru Jian, Ziyang Luo, Zirui Zhao, Yutong Dai, Ziji Shi, Hanshu Yan, Jun Hao Liew, Silvio Savarese, and Junnan Li. StateAct: Program state, before pixels, for long-horizon computer-use agents. *arXiv preprint arXiv:2607.22798*, 2026b. URL https://arxiv.org/abs/2607.22798.

Yan Yang, Dongxu Li, Yutong Dai, Yuhao Yang, Ziyang Luo, Zirui Zhao, Zhiyuan Hu, Junzhe Huang, Amrita Saha, Zeyuan Chen, et al. GTA1: GUI test-time scaling agent. In *International Conference on Learning Representations*, 2026c. URL https://arxiv.org/abs/2507.05791.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative AI by backpropagating language model feedback. *Nature*, 639(8055):609–616, 2025. doi: 10.1038/s41586-025-08661-4.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (STOP): Recursively self-improving code generation. In *Proceedings of the First Conference on Language Modeling*, 2024. URL https://arxiv.org/abs/2310.02304.

Ziyun Zeng, Hang Hua, Bocheng Zou, Mu Cai, Rogerio Feris, and Jiebo Luo. MementoGUI: Learning agentic multimodal memory control for long-horizon GUI agents. *arXiv preprint arXiv:2605.18652*, 2026. URL https://arxiv.org/abs/2605.18652.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-harness: Harnesses that improve themselves. *arXiv preprint arXiv:2606.09498*, 2026a. URL https://arxiv.org/abs/2606.09498.

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*, 2026b. URL https://arxiv.org/abs/2608.02643.

Yifan Zhang, Yutong Dai, Juntao Tan, Luyu Yang, Rishi Mullur, Thai Hoang, Zhiyuan Hu, James Zhu, Phil Mui, Silvio Savarese, Ran Xu, and Zeyuan Chen. DarwinX: Evolving agent harnesses through natural selection. *arXiv preprint arXiv:2608.07545*, 2026c. URL https://arxiv.org/abs/2608.07545.

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, 2026d. URL https://arxiv.org/abs/2604.05477.

A Additional Related Work

Self-improving agents and automatic harness optimization. Self-improving agent systems retain execution feedback in forms that influence later behavior. Reflexion stores linguistic feedback in episodic memory, while STOP improves an executable scaffolding program (Shinn et al., 2023; Zelikman et al., 2024). TextGrad, ProTeGi, GEPA, and Feedback Descent optimize textual components from language feedback (Yuksekgonul et al., 2025; Pryzant et al., 2023; Agrawal et al., 2026; Lee et al., 2025). Meta-Harness and Self-Harness expand the editable object to agent source code and retain changes using execution results. Meta-Harness evaluates classification, mathematical reasoning, and terminal coding, while Self-Harness evaluates Terminal-Bench, SWE-bench, and AppWorld (Lee et al., 2026; Zhang et al., 2026a). Their optimizer-facing evidence consists predominantly of serialized source, language/tool traces, and scores. Concurrent systems study repair, durable update archives, learned editors, and behavior-aware validation (Lin et al., 2026; Chen et al., 2026; Park et al., 2026; Luo et al., 2026; Zhang et al., 2026c; Shao et al., 2026; Xu et al., 2026; Jiang et al., 2026). GUI-HARVEST specializes this search to GUI interaction, evolving the executable harness from same-task repeated visual evidence and selecting updates through held-out utility and predicted-behavior validation.

B Extended OSWorld Comparisons

B.1 Complete split-level adaptation results

Table 4 reports the Search and Validation results used during optimization alongside the Test and Full operating points summarized in the main paper.

Table 4: Complete paired 15-step results before and after harness optimization. Search and Validation guide optimization, whereas Test remains sealed until the final harness is selected. Full reports all 361 OSWorld-Verified tasks. Parentheses show absolute percentage-point gains over the corresponding initial harness.

	Initial harness H_0				GUI-HARVEST H^*			
Backbone	Search	Val.	Test	Full	Search	Val.	Test	Full
Qwen3-VL-8B-Instruct	31.01	29.72	28.79	29.49	41.43 (+10.41)	37.22 (+7.50)	**34.84** (+6.05)	**36.83** (+7.34)
Qwen3-VL-32B-Instruct	31.07	35.06	43.02	38.61	46.25 (+15.18)	50.00 (+14.94)	**53.18** (+10.16)	**50.94** (+12.33)
OpenCUA-32B	27.40	28.75	32.82	30.71	33.69 (+6.29)	34.63 (+5.88)	**34.31** (+1.49)	**34.24** (+3.53)
OpenCUA-72B	36.25	30.00	41.25	37.65	42.50 (+6.25)	37.50 (+7.50)	**44.18** (+2.93)	**42.33** (+4.68)
Gemini 3.1 Pro	65.33	67.69	66.42	66.46	72.68 (+7.35)	75.92 (+8.23)	**72.63** (+6.20)	**73.37** (+6.91)
GPT-5	52.50	55.00	56.35	55.20	66.25 (+13.75)	60.00 (+5.00)	**61.86** (+5.51)	**62.42** (+7.22)

B.2 Benchmark visualization and domain-level results

Tables 5– 7 give the corresponding domain-level results and a broader accounting of public runs involving our target backbones. Published rows retain their original model, runtime, step budget, and aggregation protocol; domain entries remain blank when the source reports only an overall score.

Table 5: OSWorld-Verified comparison with general open-weight backbones. Public rows retain their original runtime and evaluation budget; GUI-HARVEST is optimized with 15-step Search rollouts and frozen before evaluation at all three budgets (Xie et al., 2024; Yang et al., 2026a; Bai et al., 2025; OSWorld Team, 2026). "–" denotes an unreported domain score; bold marks the best reported value within each step-budget block.

Method / model	Step	OS	Office	Daily	Prof.	Multi	Avg.
Max 15 Steps							
GUI-HARVEST / Qwen3-VL-8B-Instruct	15	58.33	32.47	38.64	67.35	19.16	36.83
GUI-HARVEST / Qwen3-VL-32B-Instruct	15	**83.33**	**48.68**	**57.62**	**75.51**	**26.88**	**50.94**
Max 50 Steps							
Qwen / Qwen3-VL-8B-Instruct (Yang et al., 2026a)	50	–	–	–	–	–	33.90
OS-Symphony / Qwen3-VL-8B-Instruct (Yang et al., 2026a)	50	–	–	–	–	–	33.90
Qwen / Qwen3-VL-32B-Instruct (Yang et al., 2026a)	50	–	–	–	–	–	32.60
Agent S3 / Qwen3-VL-32B-Instruct (Yang et al., 2026a)	50	50.00	36.67	50.62	61.22	21.96	40.11

Continued on next page.

Table 5 continued.

Method / model	Step	OS	Office	Daily	Prof.	Multi	Avg.
Qwen / Qwen3-VL-32B-Thinking (Yang et al., 2026a)	50	–	–	–	–	–	41.00
OS-Symphony / Qwen3-VL-32B-Instruct (Yang et al., 2026a)	50	58.33	40.94	53.54	75.10	31.24	46.86
GUI-HARVEST / Qwen3-VL-8B-Instruct	50	54.17	32.47	45.61	59.18	24.26	38.26
GUI-HARVEST / Qwen3-VL-32B-Instruct	50	**75.00**	**46.18**	**60.08**	**75.51**	**32.35**	**51.52**
Max 100 Steps							
Qwen / Qwen2.5-VL-32B-Instruct (OSWorld Team, 2026)	100	–	–	–	–	–	3.88
Qwen / Qwen2.5-VL-72B-Instruct (OSWorld Team, 2026)	100	–	–	–	–	–	5.00
GUI-HARVEST / Qwen3-VL-8B-Instruct	100	50.00	33.33	48.17	55.10	28.57	39.37
GUI-HARVEST / Qwen3-VL-32B-Instruct	100	**70.83**	**46.18**	**61.37**	**75.51**	**33.42**	**51.79**

Table 6: OSWorld-Verified comparison with GUI-specialized open models. OpenCUA rows use the official coordinate-action runtime; LFF and GUI-HARVEST modify that runtime family (Qin et al., 2025; Wang et al., 2025; Sun et al., 2026; Yang et al., 2026a; OSWorld Team, 2026). "–" denotes an unreported domain score; bold marks the best reported value within each step-budget block.

Method / model	Step	OS	Office	Daily	Prof.	Multi	Avg.
Max 15 Steps							
UI-TARS-1.5-7B (Song et al., 2026)	15	34.78	27.19	27.99	61.45	5.38	25.76
OpenCUA / OpenCUA-32B (Wang et al., 2025)	15	–	–	–	–	–	29.71
OpenCUA / OpenCUA-72B (Wang et al., 2025)	15	–	–	–	–	–	39.03
GUI-HARVEST / OpenCUA-32B	15	**58.33**	29.05	44.76	65.31	9.36	34.24
GUI-HARVEST / OpenCUA-72B	15	45.83	**41.01**	**48.67**	**75.51**	**20.28**	**42.33**
Max 50 Steps							
UI-TARS-1.5-7B (Song et al., 2026)	50	25.00	26.51	31.41	48.91	9.77	25.08
OpenCUA / OpenCUA-32B (Wang et al., 2025)	50	–	–	–	–	–	34.20
OpenCUA / OpenCUA-72B (Wang et al., 2025)	50	–	–	–	–	–	44.89
GUI-HARVEST / OpenCUA-32B	50	66.67	31.62	**51.18**	65.31	13.66	38.12
GUI-HARVEST / OpenCUA-72B	50	**70.83**	**47.85**	46.10	**71.43**	**26.72**	**46.76**
Max 100 Steps							
UI-TARS-1.5-7B (Song et al., 2026)	100	29.17	25.01	31.07	46.99	8.80	25.41
OpenCUA / OpenCUA-32B (Wang et al., 2025)	100	–	–	–	–	–	34.88
LFF / OpenCUA-32B (Sun et al., 2026)	100	–	–	–	–	–	38.20
DeepMiner-Mano-7B (Yang et al., 2026a)	100	50.00	39.28	44.87	73.47	17.20	40.15
UI-TARS (Yang et al., 2026a)	100	41.67	50.42	55.69	51.02	14.66	41.85
OpenCUA / OpenCUA-72B (three-run mean) (Wang et al., 2025)	100	–	–	–	–	–	44.99
OpenCUA / OpenCUA-72B (leaderboard run) (Han et al., 2026)	100	61.13	44.73	49.95	72.58	22.16	44.91
LFF / OpenCUA-72B (Sun et al., 2026)	100	–	–	–	–	–	48.9 ± 1.2
UI-TARS-2 (Yang et al., 2026a)	100	41.67	61.11	**62.12**	61.22	34.13	53.10
DeepMiner-Mano-72B (Yang et al., 2026a)	100	66.67	**63.22**	52.51	**83.67**	24.41	**53.91**
GUI-HARVEST / OpenCUA-32B	100	**70.83**	33.33	51.18	65.31	15.81	39.50
GUI-HARVEST / OpenCUA-72B	100	**70.83**	50.41	44.82	71.43	**34.25**	49.25

Table 7: OSWorld-Verified comparison with proprietary frontier backbones and systems. Each public row preserves its original framework and step budget and provides literature-level benchmark context (Agashe et al., 2025b; Song et al., 2026; Yang et al., 2026c; Gonzalez-Pumariega et al., 2026b; Yang et al., 2026a; Han et al., 2026; OSWorld Team, 2026). "–" denotes an unreported domain score; bold marks the best reported value within each step-budget block.

Method / model	Step	OS	Office	Daily	Prof.	Multi	Avg.
Max 15 Steps							
OpenAI o3 (Song et al., 2026)	15	37.50	1.45	8.02	12.29	11.82	9.09
OpenAI CUA / GPT-4o (Song et al., 2026)	15	45.83	22.17	37.65	41.22	10.75	26.01
Jedi-7B w/ GPT-4o (OSWorld Team, 2026)	15	–	–	–	–	–	26.80
Agent S2.5 / OpenAI o3 (Song et al., 2026)	15	70.83	42.85	44.61	57.10	17.82	38.98
CoAct-1 / GPT-5 (Song et al., 2026)	15	66.67	47.18	42.30	47.74	23.82	39.81
VLAA-GUI / Gemini 3 Flash (Han et al., 2026)	15	79.20	29.92	54.14	57.13	34.00	43.15
VLAA-GUI / Gemini 3.1 Pro (Han et al., 2026)	15	83.30	52.99	54.33	61.22	34.70	51.69
VLAA-GUI / Claude Sonnet 4.6 (Han et al., 2026)	15	83.30	69.72	58.72	57.13	**60.20**	64.13
VLAA-GUI / Claude Opus 4.6 (Han et al., 2026)	15	83.30	60.65	66.38	79.59	55.90	64.75
GUI-HARVEST / Gemini 3.1 Pro	15	**87.50**	**81.73**	**67.71**	**87.76**	56.36	**73.37**
GUI-HARVEST / GPT-5	15	83.33	65.78	61.09	73.47	48.10	62.42
Max 50 Steps							
OpenAI o3 (Song et al., 2026)	50	37.50	11.50	19.78	30.10	11.82	17.17

Continued on next page.

Table 7 continued.

Method / model	Step	OS	Office	Daily	Prof.	Multi	Avg.
OpenAI CUA / GPT-4o (Song et al., 2026)	50	70.83	23.56	38.43	52.09	15.86	31.19
Claude 3.7 Sonnet (OSWorld Team, 2026)	50	–	–	–	–	–	35.80
Agent S3 / GPT-5-Mini (Yang et al., 2026a)	50	62.50	54.62	46.67	44.90	37.04	47.58
UiPath Screen Agent / GPT-5 (Yang et al., 2026a)	50	73.91	49.52	62.12	71.43	37.30	53.69
Agent S2.5 / OpenAI o3 (Song et al., 2026)	50	75.00	52.81	55.80	75.42	39.53	54.21
CoAct-1 / GPT-5 (Yang et al., 2026a)	50	70.83	60.65	54.09	69.39	42.37	56.39
OS-Symphony / GPT-5-Mini (Yang et al., 2026a)	50	73.68	58.17	61.39	75.00	47.37	58.05
Claude Sonnet 4.5 (OSWorld Team, 2026)	50	–	–	–	–	–	58.08
Agent S3 / GPT-5 (Gonzalez-Pumariega et al., 2026b)	50	–	–	–	–	–	61.10
VLAA-GUI / Gemini 3 Flash (Han et al., 2026)	50	83.30	69.19	62.75	62.49	51.00	63.14
OS-Symphony / GPT-5 (Yang et al., 2026a)	50	75.00	64.85	61.19	69.23	54.86	63.61
UiPath Screen Agent / Claude Opus 4.5 (OSWorld Team, 2026)	50	–	–	–	–	–	64.40
VLAA-GUI / Gemini 3.1 Pro (Han et al., 2026)	50	83.30	71.51	67.32	67.35	55.90	66.80
VLAA-GUI / Claude Sonnet 4.6 (Han et al., 2026)	50	83.30	79.23	69.24	57.13	**66.70**	71.11
VLAA-GUI / Claude Opus 4.6 (Han et al., 2026)	50	83.30	78.60	72.60	83.67	61.30	73.85
GUI-HARVEST / Gemini 3.1 Pro	50	**91.67**	**86.00**	**72.84**	**87.76**	63.73	**78.03**
GUI-HARVEST / GPT-5	50	87.50	69.77	64.93	73.47	52.40	65.93
Max 100 Steps							
OpenAI o3 (Song et al., 2026)	100	62.50	17.23	26.29	38.79	16.53	23.00
Jedi-7B w/ GPT-4o (OSWorld Team, 2026)	100	–	–	–	–	–	29.30
Qwen3-VL API / Qwen3-VL-Flash (OSWorld Team, 2026)	100	–	–	–	–	–	41.57
Agent S2.5 / GPT-5 (Yang et al., 2026c)	100	–	–	–	–	–	58.40
CoAct-1 / GPT-5 (Han et al., 2026)	100	75.00	62.93	57.94	71.43	47.87	59.93
Seed / Seed-1.8 (Han et al., 2026)	100	66.67	68.80	67.05	71.43	42.38	61.87
Agent S3 / GPT-5 (Yang et al., 2026a)	100	77.50	66.46	61.23	69.80	51.37	62.63
Claude Sonnet 4.5 (Han et al., 2026)	100	70.83	72.59	61.35	63.27	49.54	62.84
GTA1 / GPT-5 (Han et al., 2026)	100	79.17	63.91	62.56	79.59	50.91	63.41
OS-Symphony / GPT-5 (Yang et al., 2026a)	100	79.17	65.73	67.76	69.23	57.98	65.84
UiPath Screen Agent / Claude Opus 4.5 (Han et al., 2026)	100	70.83	74.13	68.33	73.47	52.97	67.14
Agent S3 / Claude Opus 4.5 (Han et al., 2026)	100	75.00	76.06	67.51	59.18	59.00	67.46
VLAA-GUI / Gemini 3 Flash (Han et al., 2026)	100	**91.70**	64.90	74.85	67.33	63.40	68.77
VLAA-GUI / Claude Sonnet 4.6 (Han et al., 2026)	100	83.30	79.24	69.25	57.14	**68.80**	71.67
VLAA-GUI / Gemini 3.1 Pro (Han et al., 2026)	100	83.30	76.60	73.76	73.47	62.90	72.47
HIPPO / Claude Opus 4.5 (Han et al., 2026)	100	87.50	74.27	69.27	**95.92**	64.31	74.49
VLAA-GUI / Claude Opus 4.6 (Han et al., 2026)	100	**91.70**	82.87	**75.17**	83.67	65.60	77.45
OpenAPA / Gemini 3.1 Pro (OSWorld Team, 2026)	100	–	–	–	–	–	78.34
GUI-HARVEST / Gemini 3.1 Pro	100	91.67	**86.86**	74.12	87.76	65.88	**79.14**
GUI-HARVEST / GPT-5	100	87.50	73.18	66.22	75.51	55.63	68.42

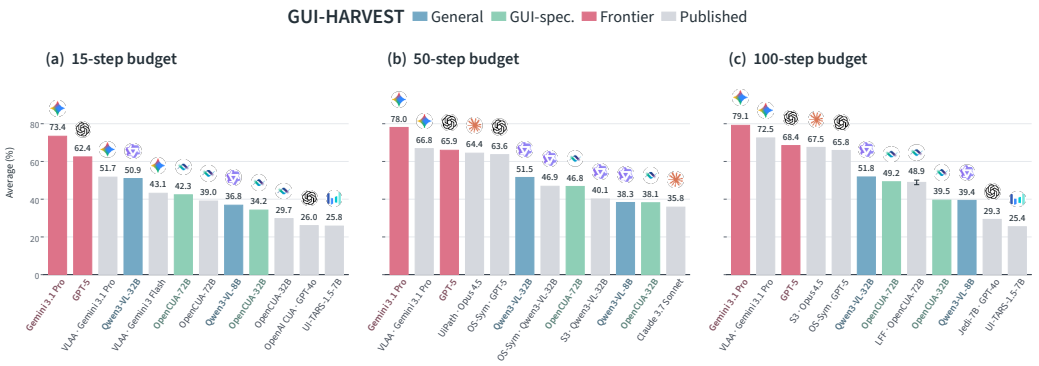

Figure 5: Selected OSWorld-Verified operating points at 15, 50, and 100 maximum environment steps. Colored bars show frozen GUI-HARVEST harnesses optimized with 15-step Search rollouts; gray bars show published systems in their reported model and runtime configurations. This visualization summarizes the domain-level tables above; published values are drawn from the corresponding papers and the official OSWorld leaderboard (Song et al., 2026; Wang et al., 2025; Sun et al., 2026; Yang et al., 2026a; Han et al., 2026; OSWorld Team, 2026).

C Repeated Execution and Repeat-Count Ablation

GUI execution is not reproducible at the level of byte-identical observations or deterministic state transitions. Across clean resets, initial screenshots already differ through clocks, tray state, cursor animation, notifications, dynamic web content, antialiasing, and compositor timing. During execution, rendering latency, focus, loading state, retained selections, and unrelated dialogs can change the result of the same intended action. Because each new screenshot conditions the next model decision and GUI actions target pixels, small early differences can redirect subsequent actions and accumulate into a different terminal state (Figure 6).

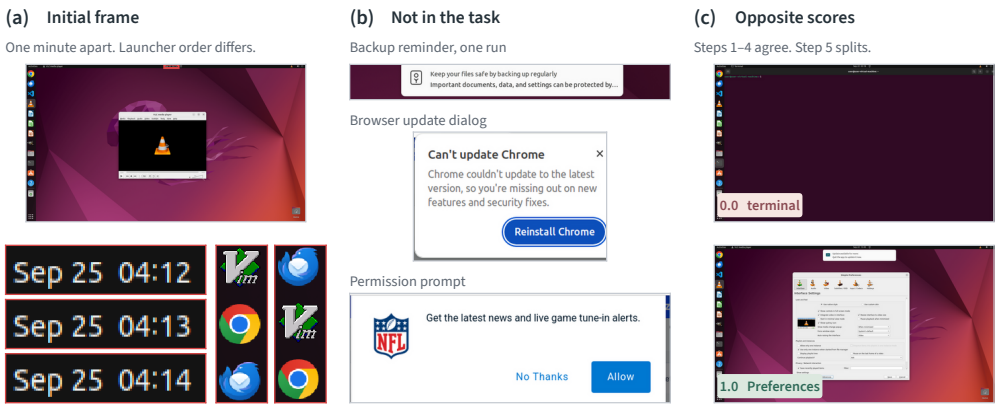

Figure 6: Execution variability in desktop GUI tasks. (a) Clean resets one minute apart already differ in the initial frame. (b) Task-irrelevant system and web events enter the observation stream during execution. (c) Two runs share the first four steps but diverge at step five and receive opposite scores. These perturbations enter the multimodal feedback loop rather than remaining isolated pixel noise.

The endpoint can therefore flip for the same task, model, and harness. A failed run alone does not establish that the task is unreachable, while a successful run does not erase an informative failure path. A K-run task bundle exposes both paths when they coexist, allowing the analyst to compare their divergence instead of treating one noise-amplified branch as a stable mechanism. This motivation agrees with recent computer-use reliability results showing that outcome variation persists under deterministic decoding (Gonzalez-Pumariega et al., 2026a). Table 8 measures its prevalence in the six model–harness configurations used by the optimizer.

Table 8: Task-level flip rates under each baseline harness over $K = 3$ Full-361 executions. A task flips when its repeated runs contain both a zero score and a positive score.

	Qwen3-VL-32B-Instruct	Qwen3-VL-8B-Instruct	Gemini 3.1 Pro	GPT-5	OpenCUA-72B	OpenCUA-32B
Flip rate (%)	20.4	16.6	12.7	12.4	15.6	11.8

Depending on the backbone, 11.8–20.4% of tasks flip between zero and positive score across three executions. The highest measured rates occur for Qwen3-VL-32B-Instruct (20.4%) and Qwen3-VL-8B-Instruct (16.6%); Gemini 3.1 Pro and GPT-5 still flip on 12.7% and 12.4% of tasks. Task-level bundles expose both paths when they coexist and support comparisons at their divergence.

C.1 Choosing the Repeat Count

The sweep separates evaluation precision, diagnostic coverage, and optimization outcome. Increasing K from 1 to 3 lowers the standard error from 1.61 to 0.93 points and reveals that 20.4% of tasks contain both zero- and positive-score outcomes. The selected harness improves from 43.1% at $K = 1$

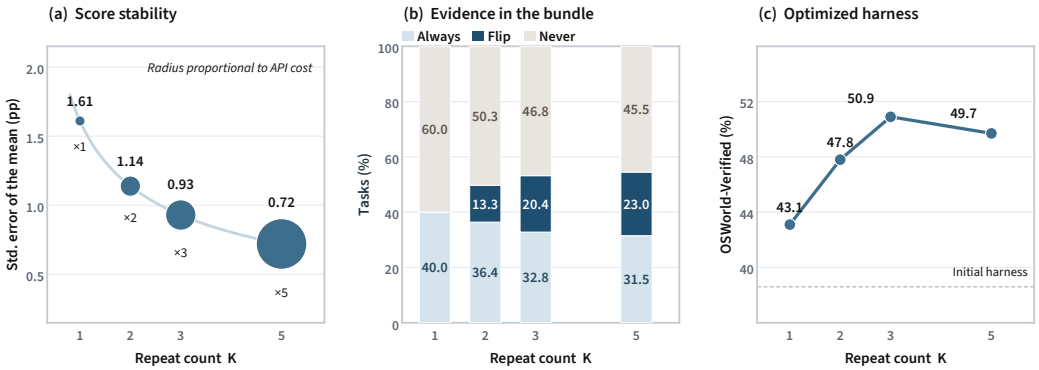

Figure 7: Effect of the repeat count K for Qwen3-VL-32B-Instruct. (a) Standard error of the Full mean decreases with K, while rollout API cost scales approximately linearly. (b) Task-level flip rate as K varies, with always-positive and never-positive shares shown for context. (c) Full score of the harness selected by each setting; the dashed line is the initial harness.

to 50.9% at $K = 3$. Increasing to $K = 5$ further lowers the standard error and exposes more mixed-outcome tasks, but raises rollout cost to $5\times$ and selects a slightly weaker 49.7% harness. We therefore use $K = 3$ throughout the main experiments.

D Model-Specific Findings and Harness Evolution

D.1 Behavior profiles across optimization

The optimizer does not converge to one shared patch set. Table 9 connects representative task-level behavior changes to the executable runtime mechanisms selected for each backbone. We report patterns that can be checked under both the initial and final harness, while retaining overlapping counts when one task exhibits more than one behavior.

Table 9: Representative verified behavioral-pattern changes and the corresponding runtime adaptations for each backbone. Counts denote Search-set tasks matching each pattern under the initial and final harnesses ($H_0 \rightarrow H^*$). Patterns are not mutually exclusive, so the counts do not form a partition of failures.

Backbone	Verified pattern changes ($H_0 \rightarrow H^*$)	Selected H^* mechanisms	Residual bottleneck
General-purpose open models			
Qwen3-VL-8B-Instruct	redundant action loop $42 \rightarrow 27$; inefficient execution strategy $33 \rightarrow 19$; infeasibility misjudgment $10 \rightarrow 3$	loop rejection and fresh restart; code routing; budget verdict; save nudge	after a blocked action, the model may choose another ineffective route or emit `done()` again without correcting the state
Qwen3-VL-32B-Instruct	inefficient execution strategy $36 \rightarrow 17$; premature completion $28 \rightarrow 15$; redundant action loop $11 \rightarrow 6$	application-aware code routing; route ledger; save gate; completion adjudication; code receipt	wrong targets or routes despite executed actions; incomplete work after code execution
GUI-specialized models			
OpenCUA-32B	zero-duration drag $33 \rightarrow 28$; unsupported mouse API $3 \rightarrow 0$; infeasibility misjudgment $3 \rightarrow 0$	drag-duration normalization; `computer.*`-to-`pyautogui.*` mapping; infeasibility remapping	correctly executed actions can still select the wrong target or drag distance; no alternative execution path is introduced
OpenCUA-72B	unpersisted LibreOffice final state $24 \rightarrow 18$; dropped terminal action $4 \rightarrow 0$; infeasibility misjudgment $5 \rightarrow 3$	pre-completion save; preservation of the last-step `DONE`; text-grounded infeasibility handling	ineffective or wrong GUI actions and visually plausible false completion
Proprietary frontier models			
Gemini 3.1 Pro	premature completion $20 \rightarrow 10$; tool–application state mismatch $13 \rightarrow 5$; inefficient execution strategy $11 \rightarrow 4$	code routing; Office-file normalization; dry-run validation; constrained infeasibility criterion	stale application views or semantically incorrect code output can still look complete
GPT-5	inefficient execution strategy $27 \rightarrow 10$; redundant action loop $22 \rightarrow 10$; unnecessary post-tool continuation $15 \rightarrow 8$; infeasibility misjudgment $9 \rightarrow 2$	code routing; Office persistence and restart; feasibility probe; code receipt; coordinate re-query	semantic mismatch between a self-verified artifact and evaluator state; residual coordinate errors

Table 9 retains the model-specific quantities and residual bottlenecks. A complementary cross-backbone view in Table 10 instead groups the same evidence by runtime consequence, exposing shared structure without treating similar symptoms as identical causes.

Table 10: Cross-backbone organization of recurring, harness-addressable behavioral patterns. The affected-backbone column lists models in which each pattern was verified. A pattern groups model-specific manifestations with the same runtime consequence; it does not imply an identical underlying cause.

Behavioral pattern	Affected backbones	Shared intervention principle	Model-specific realizations
Execution and control			
Inefficient execution strategy	Qwen3-VL-8B and 32B; Gemini 3.1 Pro; GPT-5	route bulk or repetitive operations through a shorter compatible execution path	document maps and application references for Qwen; code routing and Office finalization for Gemini and GPT-5
Redundant action loop	Qwen3-VL-8B and 32B; GPT-5	revise or withhold an action when repeated execution produces no useful state change	loop rejection and fresh restart; route retirement; coordinate re-query and alternative code routing
Completion judgment			
Premature completion	Qwen3-VL-8B and 32B; OpenCUA-72B; Gemini 3.1 Pro	require the task-relevant edit to be committed or persisted before accepting completion	save nudge; save gate and completion adjudication; pre-completion save; Office-file normalization
Infeasibility misjudgment	all six backbones	use sufficiently supported infeasibility evidence to select the correct terminal outcome	budget verdict and capability checks for general models; constrained feasibility probes for frontier models; text-grounded terminal remapping for OpenCUA
Tool and action interfaces			
Tool–application state mismatch	Qwen3-VL-32B; Gemini 3.1 Pro; GPT-5	return structured execution evidence and synchronize the modified artifact with the visible application state	code receipts; Office normalization and reload; controlled restart after external edits
Action-interface mismatch	OpenCUA-32B and 72B; GPT-5	normalize or preserve a valid model emission at the runtime dispatch boundary	drag duration and API mapping; preservation of a valid terminal action; coordinate re-query

The labels describe observable behavior rather than assigning a unique causal module. In particular, *inefficient execution strategy* excludes tasks that legitimately require GUI interaction or merely exceed a short step budget; it applies when the evidence shows an avoidably low-throughput route despite an available compatible alternative. This differs from a *redundant action loop*, where the same or equivalent action recurs without useful state progress.

Two recurring patterns illustrate why a shared label does not imply a universal patch. Infeasibility misjudgment appears in all six backbones, but the selected mechanisms range from budget and capability checks, to constrained feasibility probes, to text-grounded terminal remapping. Likewise, premature completion recurs across four backbones, yet is addressed through a save nudge, completion adjudication, pre-completion persistence, or Office-file normalization according to the available runtime interface.

The intervention granularity also separates along model families. Qwen3-VL-8B relies on action-level enforcement when feedback alone does not change its next action, whereas Qwen3-VL-32B combines enforcement with application-aware routing and state evidence. The frontier models benefit mainly from routing, persistence, and completion-policy changes. OpenCUA remains close to its native coordinate-action interface, concentrating its adaptations on action normalization and terminal semantics.

Across families, the remaining errors increasingly involve actions that execute successfully but target the wrong object, route, or evaluator-relevant state. Such trajectories provide less observable evidence for a runtime guard than an unchanged screen, an invalid call, or an unsaved file, explaining why later candidate edits are more likely to plateau or trade gains across tasks.

D.2 Per-backbone harness evolution and final changes

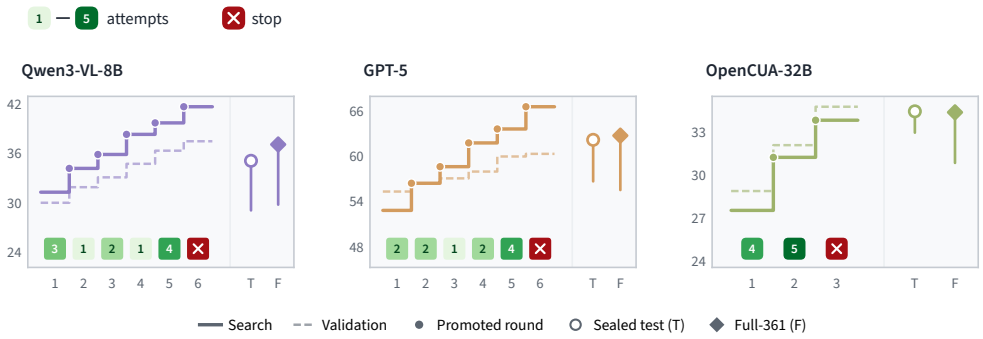

Figure 8: Harness evolution for Qwen3-VL-8B, GPT-5, and OpenCUA-32B, complementing Figure 3. Solid and dashed curves show Search and Validation scores, respectively; filled circles mark promoted harnesses, and the rightmost markers show Test (T) and Full (F) scores after optimization stops. The numbered tiles report candidate-edit attempts in each optimization round, and the red cross marks the terminal round.

Figures 3 and 8 expose both accepted updates and the search effort behind them. Qwen3-VL-32B promotes seven updates in eight rounds with 25 candidate-edit attempts; Qwen3-VL-8B promotes five in six rounds with 16 attempts. Gemini 3.1 Pro and GPT-5 promote six of seven and five of six rounds, using 18 and 16 attempts, respectively. The specialized OpenCUA backbones stop earlier: OpenCUA-72B promotes three updates in four rounds with 17 attempts, whereas OpenCUA-32B promotes two in three rounds with 14 attempts. Thus the optimization path depends on the backbone and initial harness: fewer rounds do not necessarily imply fewer edit attempts or an easier adaptation problem.

E WindowsAgentArena Transfer

E.1 WindowsAgentArena benchmark context

The paired transfer gains are summarized in the main paper. Table 11 reports the corresponding runs, and Table 12 places them in the broader six-domain WAA context. Published rows retain their reported runtimes, step budgets, and aggregation protocol. The OSWorld optimization stage precedes WAA access; transfer supplies only the platform-specific observation, action, and logging adapters.

Table 11: Same-backbone transfer to WindowsAgentArena at 50 steps. The GUI-HARVEST harnesses are selected on OSWorld and frozen before WAA evaluation; no WAA trajectory enters optimization. The published Qwen operating point is from Yang et al. (2026a).

Backbone	Harness	WAA ↑
Qwen3-VL-32B-Instruct	Qwen	31.68
	Agent S3	38.21
	GUI-HARVEST	**44.68**
GPT-5	Agent S3	50.88
	GUI-HARVEST	**64.75**

Table 12: WindowsAgentArena results. Published rows retain their reported model, runtime, and step budget; GUI-HARVEST rows transfer the frozen OSWorld-derived harness with platform adapters only. WAA contains six task domains (Bonatti et al., 2025). Published comparison values are from Yang et al. (2026a) and Han et al. (2026). For OS-Symphony rows, Avg. is the published score over all 154 tasks, including its separately reported 13-task infeasible subset; the six displayed columns are the WAA application domains.

Method / backbone	Steps	Office	Web	System	Code	Media	Utility	Avg.
Max 50 Steps								
Qwen / Qwen3-VL-32B-Instruct (Yang et al., 2026a)	50	19.05	49.66	54.17	21.05	42.19	25.00	31.68
OS-Symphony / Qwen3-VL-32B-Instruct (Yang et al., 2026a)	50	26.19	46.33	75.00	47.37	27.90	41.67	45.32
VLAA-GUI / Gemini 3 Flash (Han et al., 2026)	50	32.60	73.30	**87.50**	66.70	52.40	75.00	60.40
OS-Symphony / GPT-5-Mini (Yang et al., 2026a)	50	42.86	73.00	79.17	68.42	48.66	66.67	62.15
OS-Symphony / GPT-5 (Yang et al., 2026a)	50	**54.76**	73.00	75.00	42.11	**70.09**	75.00	63.45
GUI-HARVEST/ Qwen3-VL-32B-Instruct	50	20.93	39.66	70.83	**70.83**	47.18	33.33	44.68
GUI-HARVEST/ GPT-5	50	44.19	69.66	**87.50**	62.50	65.81	**83.33**	**64.75**
Max 100 Steps								
VLAA-GUI / Gemini 3 Flash (Han et al., 2026)	100	35.00	73.30	87.50	66.70	52.40	83.30	61.00

F Cost and Efficiency

F.1 Inference cost of the frozen harness

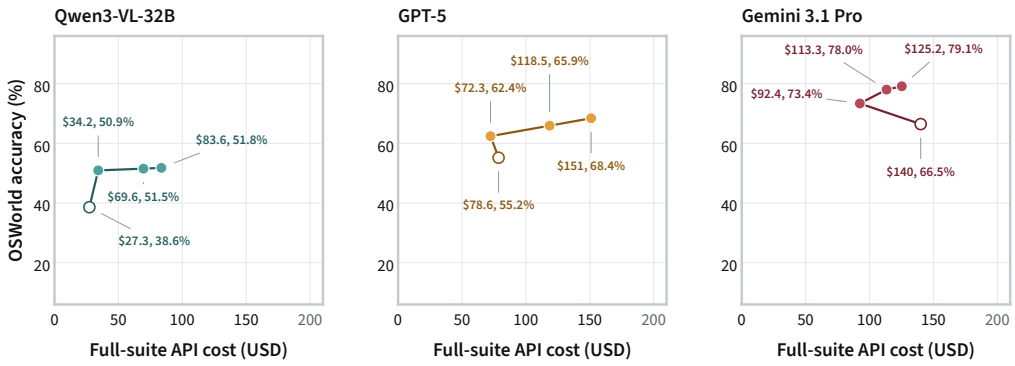

Figure 9: Full-suite API cost and OSWorld-Verified score for three target backbones. Open markers denote the 15-step Agent S3 baseline; filled markers denote the frozen GUI-HARVEST harness evaluated at 15, 50, and 100 steps. Costs cover target-model API calls for the 361-task evaluation.

Figure 9 shows that harness adaptation changes both task success and the number and length of model calls. For Qwen3-VL-32B-Instruct, the 15-step optimized harness raises accuracy from 38.6% to 50.9% while cost increases from \$27.3 to \$34.2. For GPT-5 and Gemini 3.1 Pro, the optimized 15-step harness is both more accurate and less expensive: GPT-5 moves from 55.2% at \$78.6 to 62.4% at \$72.3, and Gemini moves from 66.5% at \$140.0 to 73.4% at \$92.4. These are end-to-end measurements of the frozen harnesses.

The published comparison in Figure 4 uses cost–accuracy records from Wei et al. (2026) and Gonzalez-Pumariega et al. (2026b). The former directly reports the following accuracy and cost/task pairs: 43.3%/\$0.022 for EvoCUA-8B, 30.8%/\$0.018 for Qwen3-VL-8B, 55.4%/\$0.224 for EvoCUA-8B plus Claude Sonnet 4.5, 54.3%/\$0.423 for Qwen3-VL-8B plus Claude Sonnet 4.5, and 58.1%/\$0.881 for Claude Sonnet 4.5. Its switched-task counts and percentages imply 359 evaluated tasks (e.g., $168/46.8\%$); the plotted totals multiply the reported per-task costs by 359. The Agent S3 point combines the 62.6% 100-step accuracy and \$0.72 average cost/task reported by Gonzalez-Pumariega et al. (2026b); \$260 is $361 \times \$0.72$, rounded. Thus, the plotted public totals are direct arithmetic reconstructions from source-reported records rather than independent token-cost estimates.

For our runs, token usage is converted to API cost using the official provider prices in effect during evaluation: OpenAI and Google prices for the proprietary models, and Alibaba Cloud Model Studio prices for Qwen3-VL. Optimization calls and local compute are excluded. Public points retain the source paper's pricing assumptions. Systems without sufficient usage or cost information are omitted rather than assigned estimated values.

G More Implementation Details

G.1 Model usage and inference settings

The target backbones are Qwen3-VL-8B/32B-Instruct, OpenCUA-32B/72B, Gemini 3.1 Pro, and GPT-5. GPT-5 uses the dated `gpt-5-2025-08-07` endpoint. For Qwen3-VL and the proprietary backbones, Agent S3 uses its default UI-TARS-1.5-7B grounding model in the native 1920×1080 coordinate space; this grounder is part of the initial harness rather than an additional optimized model. OpenCUA uses its official coordinate-action runtime. All four optimizer roles use Claude Sonnet 5 with identical role prompts across target backbones. Target-model weights, API snapshots, decoding parameters, observation format, and maximum output length remain fixed within each experiment.

G.2 Environment and benchmark details

OSWorld-Verified is executed in clean virtual-machine snapshots with the benchmark's programmatic evaluators (Xie et al., 2024). Following common practice, the main protocol excludes the eight Google Drive tasks and uses the remaining 361 tasks (Han et al., 2026). The execution configuration fixes the OSWorld commit, VM and application images, screen resolution, action limit, timeout, reset and retry policy, task-group manifest, and evaluator versions. Search, validation, and test groups are fixed before optimization. The final full-361 run follows the frozen harness and stopping decision.

WindowsAgentArena evaluates frozen-harness transfer for Qwen3-VL-32B-Instruct and GPT-5 at a 50-step budget (Bonatti et al., 2025). OSWorld optimization completes before WAA access. Following the transfer setups of Agent S3 and OS-Symphony (Gonzalez-Pumariega et al., 2026b; Yang et al., 2026a), the implementation maps Windows observations, actions, and logging interfaces and updates platform-specific prompt and command conventions, including Linux-to-Windows application and shell wording. These are necessary benchmark-interface adaptations: the learned runtime mechanisms remain fixed, and no WAA trajectory enters optimization or harness selection.

G.3 Initial harnesses and action contracts

Agent S3. Agent S3 is the initial harness for Qwen3-VL and the proprietary backbones. It is an open, competitive OSWorld runtime and a widely used reference baseline, with an explicit worker, reflection, grounding, and optional code-execution structure (Gonzalez-Pumariega et al., 2026b; Qin et al., 2025). Its worker emits exactly one `agent.`$\star$ primitive per step. Pointing primitives carry a natural-language description of the target; the default UI-TARS-1.5-7B grounder maps that description to coordinates and Agent S3 compiles the result into executable PyAutoGUI code. The grounder is part of the initial harness. We retain all default runtime settings except the target backbone. Behavior best-of-N is disabled ($N = 1$), so each task attempt contributes one trajectory.

Official OpenCUA runtime. OpenCUA is trained with an action space that directly represents keyboard and coordinate-bearing mouse operations as PyAutoGUI actions (Wang et al., 2025). OpenCUA-32B/72B begin from the official OpenCUA agent. Agent S3 expects a different `agent.`$\star$ syntax followed by a separate grounding stage, whereas OpenCUA post-training directly learns coordinate-bearing PyAutoGUI actions. For these rows, H_0 denotes the official OpenCUA runtime and GUI-HARVEST edits its corresponding executable surface.

G.4 Optimization and ablation protocol

Search uses a 15-step environment budget. Unless identified as an externally published comparison, every reported GUI-HARVEST score first averages $K = 3$ independent runs within each task and then averages across tasks. Search, Validation, Test, and Full scores all follow this protocol, consistent

with the repeated-run protocol used in recent computer-use reliability analysis (Gonzalez-Pumariega et al., 2026a). Appendix C examines the trade-off between repeat count, diagnostic coverage, and rollout cost. After optimization, the same frozen harness is evaluated at 15, 50, and 100 steps without further edits. The task split, evaluator, repeat count, round cap, and promotion rule are shared across backbone families.

The main protocol allows at most $B = 10$ rounds per backbone. Each round searches from one fixed harness state, and up to $P = 5$ L1-valid candidates may enter GUI evaluation. Promotion starts the next round with a fresh five-attempt budget; if all five candidates are rejected, optimization stops at the current harness. L0/L1 repairs made before a valid candidate enters GUI evaluation consume no attempt. All stopping parameters are fixed before the sealed test is opened. A capability manifest permits edits to prompts, context and memory construction, action interfaces, control flow, verification, recovery, and termination, while protecting the model client, benchmark, evaluator, tasks, and resource limits.

Component ablations use Qwen3-VL-32B-Instruct and remove one element at a time: visual evidence, the Cross-task Clusterer, repeated optimizer evidence ($K = 1$), or the behavioral soft gate. Variants share the initial harness, editable scope, task split, and optimization-round cap. All selected harnesses, including the $K = 1$ optimizer variant, use the same $K = 3$ final evaluation protocol (Table 3).

G.5 Per-backbone optimization records

Figure 8 summarizes promoted rounds and candidate edit attempts for all six target backbones. Tables 9 and 10 connect the retained behavior changes to the runtime mechanisms in each final harness.

G.6 Role interfaces, prompts, and schemas

Evidence analysts and intervention validators are short-lived task-local workers that run in parallel. The Cross-task Clusterer receives verified task-local findings grouped by outcome category and induces recurring behavior modes supported by independent tasks. The harness engineer is one bounded tool session per candidate and can search, edit, and test only paths allowed by the capability manifest. Deterministic programs perform log parsing, citation checks, score aggregation, L0/L1 checks, promotion, rollback, and ledger writes.

GAFT converts each raw execution into a step-aligned evidence bundle. For every step it retains the model text, planned and executed action, tool outputs, and before/after screen references, and computes only reproducible visual facts: the whole-screen changed-pixel fraction Δ_{screen} and, for pointing actions, Δ_{local} in a fixed window around the landing point. The initial analyst context includes event-selected keyframes—the initial and final screens, large screen transitions, and the start of sustained no-change stretches—plus landing-point crops when they are informative. The analyst may then request additional full-resolution frames or crops by run and step. Retrieval expands the evidence available for semantic diagnosis without asking the preprocessing code to infer what the screen means. Figure 10 illustrates the landing-point evidence exposed for a pointing action.

A verified finding identifies the failed run and decisive step, summarizes the observed action–state mismatch, and links it to run/step evidence; when available, it also records a successful-run contrast. Screens are supplied with the task bundle and can be retrieved by run and step during analysis. A behavior mode contains an outcome category, observable mechanism, membership test, behavioral target, member findings, and independent-task support. Findings that do not form a recurring mode remain available as residue; the engineer may consider one only when it can justify a shared source lever and a cross-task-safe intervention. The update manifest connects selected targets and source locations to the patch, target tasks, predicted changes, and protected interfaces. Validator outputs contain one task-level verdict and run/step citations.

Appendix I presents an end-to-end record exported from the implementation.

G.7 Split and leakage controls

Tasks are assigned to disjoint Search, Validation, and Test sets of 80, 80, and 201 tasks through a domain-stratified random split. Each domain contributes in proportion to its size; tasks within a

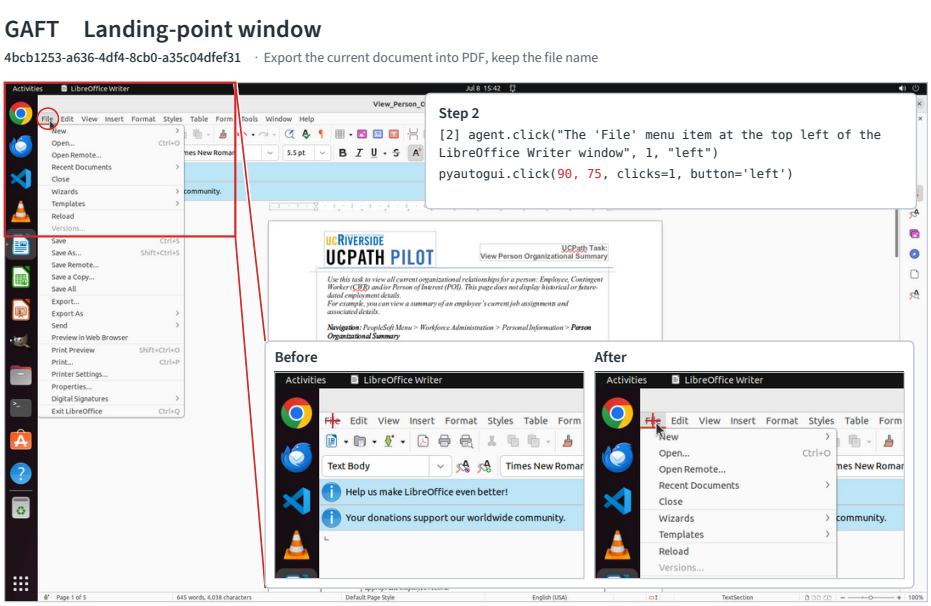

Figure 10: GAFT evidence for a pointing action. The toolkit aligns the executed click with before/after screenshots and reports pixel change over the full screen and a local window around the landing point. These measurements are mechanical evidence supplied to the Evidence Analyst, not semantic judgments.

domain are sampled uniformly with a public seed. The assignment is fixed before rollout and reused across backbones. The optimizer receives Search task cards and rollout artifacts. Validation execution is performed by the outer runner, which exports the aggregate mean needed by the score gate. Test task identifiers and outcomes remain inaccessible until the selected harness commit is frozen. The fixed task manifest records the group identifier for every task.

G.8 Comparison methods

Paired harness comparisons. The H_0–H^* comparisons fix the target backbone and evaluation protocol to measure the effect of harness adaptation. Full scores additionally provide operating points for comparison with published systems; Test scores measure generalization to tasks withheld during optimization.

Harness optimizers. For Self-Harness and Meta-Harness, we freeze Qwen3-VL-32B-Instruct and start from the same Agent S3 commit and writable source tree used by GUI-HARVEST. Both adaptations use Claude Sonnet 5 and a maximum of ten optimization rounds. Self-Harness receives serialized Agent S3 plans, actions, tool outputs, and scores through its trajectory interface; Search and Validation serve as its held-in and held-out sets. We retain its failure-key clustering, single-hook proposals, and score-based acceptance rule. Meta-Harness receives a filesystem containing candidate source, Search scores, trajectories, instructions, results, and screenshots. It proposes two harnesses per round and selects the final harness by Search score, with Validation and sealed Test withheld from selection. These adapters change data formats and GUI execution plumbing, not the optimizers' diagnosis or selection policies.

The LFF paper and repository do not provide a complete experimental configuration, an executable optimization procedure, or the optimizer/framework source required to rerun the search. The repository provides the resulting OpenCUA patch artifacts. We therefore apply those released patches to the matching OpenCUA-32B and OpenCUA-72B runtimes and evaluate them on our 15-step Test

and Full protocol. Table 2 therefore reports within-backbone comparisons and does not aggregate across the Qwen and OpenCUA blocks.

Published GUI systems. Comparisons include Agent S3, OS-Symphony, VLAA-GUI, OpenCUA, LFF, and related systems. Each published row retains its reported model, runtime, step budget, and aggregation protocol (Appendix B). LFF's 100-step OpenCUA-32B/72B results are retained as published operating points in Table 6 (Sun et al., 2026); they come from independent executions and are not paired trials against our initial harnesses.

G.9 Runtime validity and failure handling

Each rollout starts from the benchmark's clean snapshot. Runs with environment startup failure, missing screenshots, evaluator failure, or unrecoverable runner exceptions are marked invalid and rerun under a predeclared retry policy. The policy is identical for the current and candidate harness. Partial evaluator scores are retained as raw values. A bundle enters failure diagnosis when at least one valid run has score zero. Zero-score runs are finding targets, while the remaining runs provide comparison evidence.

G.10 Evaluation metrics and cost scope

The primary metric is the mean raw evaluator score, averaging the K runs within each task and then across tasks (Eq. 1). Scores are expressed as percentages, and H_0–H^* gains are absolute percentage-point differences. For the repeated-execution analysis in Appendix C, a task is *always positive* if all K runs have positive scores, *never positive* if none does, and a *flip* if both zero- and positive-score outcomes occur. This decomposition captures outcome variation that is hidden by a single rollout.

Full-suite inference API costs cover target-model calls during the 361-task evaluation of a frozen harness. Optimization-process costs are not included in these measurements. Appendix F specifies the scope and pricing caveats for published cost comparisons.

H Formal Definitions for GUI-HARVEST

This appendix summarizes the optimization loop and specifies the records and promotion rule used in Section 3. Utility is defined in Equation 1, and the score-based acceptance condition is given in Equation 2.

Algorithm 1 GUI-HARVEST optimization

Require: Frozen model M, seed harness H_0, splits $\mathcal{S}, \mathcal{V}, \mathcal{T}$, repeats K, round cap B, attempts per round P

$H \leftarrow H_0$; $\mathcal{L} \leftarrow \emptyset$; evaluate H on $\mathcal{S}, \mathcal{V}$ with K rollouts per task
for $b = 1, \ldots, B$ **do**
  Build or reuse search bundles from rollouts of the current H
  Evidence Analyst: diagnose failed runs with verified evidence, or reuse existing findings
  Cross-task Clusterer: induce recurring modes; reuse verified modes if H is unchanged
  $promoted \leftarrow$ false
  for $p = 1, \ldots, P$ **do**
    Harness Engineer: propose $\widetilde{H}$, target tasks, and predicted changes; repair until L0/L1-valid
    Validator: evaluate $\widetilde{H}$ on $\mathcal{S}, \mathcal{V}$ with K rollouts per task; apply utility and behavior gates
    Append the patch, predictions, scores, and verdicts to $\mathcal{L}$
    if $\widetilde{H}$ is promoted **then**
      $H \leftarrow \widetilde{H}$; retain its search rollouts
      $promoted \leftarrow$ true; **break**
    else
      Roll back to H
    end if
  end for
  if $promoted =$ false **then**
    break
  end if
end for
Freeze $H^* \leftarrow H$; perform the sealed-test evaluation on $\mathcal{T}$
return H^*

H.1 Execution evidence and findings

For task x_i, run r under the current harness H_t at round t is recorded as

$$\rho_{i,r}^{(t)} = \left(\tau_{i,r}^{(t)}, s_{i,r}^{(t)}, z_{i,r}^{(t)}\right), \tag{3}$$

where $\tau_{i,r}^{(t)}$ is the multimodal trajectory, containing model text, planned and executed actions, screenshots, and tool outputs; $s_{i,r}^{(t)}$ is the run's benchmark score; and $z_{i,r}^{(t)}$ contains runtime and termination metadata. Repeated runs are grouped into a task-round bundle:

$$\mathcal{B}_i^{(t)} = \left(x_i, \mathcal{C}(H_t), \{\rho_{i,r}^{(t)}\}_{r=1}^{K}\right). \tag{4}$$

The task record x_i contains the instruction, domain, related applications, feasibility annotation, setup steps, and any hint published with the task. The Harness Card $\mathcal{C}(H_t)$ summarizes the current runtime and its observable outputs. The bundle separates this shared information from the evidence specific to each run.

The Evidence Analyst produces findings of the form

$$f = (r,\, t^\star,\, \sigma,\, m,\, \bar{a},\, E), \tag{5}$$

where r identifies the failed run, $t^\star$ is its decisive step, σ records the decisive action, observed outcome, and subsequent response, m is the observable mechanism, $\bar{a}$ is an evidence-supported alternative behavior when available, and E contains run/step evidence and any successful-run contrast. Findings remain linked to their task bundle and harness version. The evidence checks in Section 3.2 validate these records before clustering.

H.2 Behavior modes and update plans

The Cross-task Clusterer first maps each finding signature σ to a deterministic outcome category g. It then represents each recurring behavior mode within one category as

$$c = (g,\, d_c,\, \phi_c,\, b_c,\, F_c), \tag{6}$$

where d_c describes its observable mechanism, ϕ_c is a membership test, b_c is the shared behavioral target, and F_c contains member findings from at least two independent tasks. Category g narrows the clustering search space without fixing the mode vocabulary or the eventual intervention. Member findings retain links to their task-local evidence.

Before candidate evaluation, the Harness Engineer records an update plan

$$u = (c, E, a, \delta, \mathcal{P}, A, q). \tag{7}$$

Here c and E identify the target mode and evidence, a is the source surface being changed, δ is the intervention, $\mathcal{P}$ contains observable predictions, and $A \subseteq \mathcal{S}$ contains the target tasks used to check them. The invariants q specify interfaces or behaviors that must remain intact. The plan links the proposed code change to evidence that can be checked in subsequent executions.

H.3 Candidate scoring and promotion

For each split $D \in \{\mathcal{S}, \mathcal{V}\}$, the candidate's score change is

$$\Delta_D = \widehat{J}_{D,K}(\widetilde{H}_t) - \widehat{J}_{D,K}(H_t). \tag{8}$$

The utility hard gate in Equation 2 requires both changes to be nonnegative and at least one to be strictly positive. Validation contributes only aggregate scores; detailed execution evidence is available only for search tasks.

The behavioral soft gate checks every prediction in $\mathcal{P}$ against the post-edit executions of its target tasks in A. Each prediction requires at least one determinate task verdict (supported or contradicted) and more supporting than contradicting verdicts. L0 and L1 denote the code checks defined in Section 3.5. Subject to the fixed execution-cost constraint, the candidate is promoted exactly when

$$\mathrm{Promote}(\widetilde{H}_t) = \mathrm{L0} \wedge \mathrm{L1} \wedge \mathrm{HardGate}_{\mathcal{S},\mathcal{V}} \wedge \mathrm{SoftGate}_{\mathcal{S}}\,. \tag{9}$$

Promotion sets $H_{t+1} = \widetilde{H}_t$; rejection keeps the current harness unchanged. The update ledger retains the patch, predictions, score changes, task verdicts, and decision for subsequent rounds.

I A Complete Evidence-to-Edit Chain

Figure 11 reproduces one end-to-end record from the optimizer. The task asks the agent to rename a Chrome profile to Thomas. In all three pre-edit executions, the field visibly contains the requested name; two runs nevertheless call `done()` while the field remains in edit mode and fail, whereas the successful run first clicks outside the field. The analyst therefore identifies an uncommitted edit rather than equating visible text with a completed state change. The clusterer links this finding to independent Impress and Calc tasks in which the requested edit is likewise visible but unsaved when termination is requested, yielding the cross-task mode Edit-Not-Committed-at-Done. The engineer implements an action gate that tracks edit actions, recognizes explicit saves, and withholds the first premature `done()` while returning the objection to the planner. It also records the prediction that a withheld termination should be followed by a commit action before a later `done()` is accepted. The validator then checks three post-edit executions of the Chrome task. Each first termination is withheld; the agent leaves the field, and termination is accepted only after the committed name is observable again. This chain shows how repeated executions separate a stable behavioral defect from a successful contrast, while visual state distinguishes typing the requested value from completing the edit.

Evidence Analyst

ID: 2ae9ba84-3a0d-4d4c-8338-3a1478dc5fe3

Instruction:

Lately I have changed my English name to Thomas. I want to update my username. Could you help me change the username in chrome profiles to Thomas?

(a) After typing Thomas



Run A, step 5
Types Thomas. Field still in edit mode.

Run A, step 6
done(). The run fails.

Run B, step 5
Types Thomas. Field still in edit mode.

Run B, step 6
done(). The run fails.

Run C, step 4
Clicks outside the field. Edit ends.

Run C, step 5
done(). The run passes.

(b) Finding

type not_committed

All three runs type Thomas into the profile-name field, and the field shows Thomas.

Run A and Run B call done() while the field is still in edit mode (blue underline, text cursor). Both fail.

Run C clicks outside the field first, so the edit ends before done(). It passes.

The name reads the same in all three. What differs is whether the edit was committed before done().

Cross-task Clusterer

Outcome category PREMATURE_DONE

3 findings · 3 tasks

(a) Findings

Chrome flip

2ae9ba84

type | not_committed | done()

Type Thomas into the profile name.

done() while the field is still editing.

Impress always fail

9cf05d24

click | not_committed | done()

Pick green for the slide background.

done() with the change unsaved.

Calc always fail

51b11269

click | not_committed | done()

Sort the records by amount, ascending.

done() with the change unsaved.

(b) Mode

EDIT_NOT_COMMITTED_AT_DONE

Mechanism	The edit is on screen, but it has not reached the file or the profile when done() is called.
Membership	An edit action is followed by done() with no save and no exit from the field in between.
Target	Commit the edit before done(): save the file, or leave the field.
Members	2ae9ba84 Chrome	9cf05d24 Impress	51b11269 Calc

Harness Engineer

Mode EDIT_NOT_COMMITTED_AT_DONE change_kind action_gate

(a) Edit

gui_agents/s3/agents/grounding.py

type() → _mark_mutation()

```
+    self._mark_mutation("type")
     ...
+    self._uncommitted_edits += 1
```

_account_for_keys()

```
     modifiers = lowered & {"ctrl", "control", "command", "cmd"}
+    if modifiers and "s" in lowered:
+        self._uncommitted_edits = 0
+    elif lowered & _MUTATING_KEYS:
+        self._mark_mutation("keys")
```

done()

```
+    spent = self._save_gate_step is not None
+    if self._uncommitted_edits > 0 and not spent:
+        self._save_gate_step = self._step_seq
+        return self.withhold("DONE WITHHELD. "
+            f"{self._uncommitted_edits} action(s) that alter a document have been dispatched ...")
+    if getattr(self, "document_modified_fact", None) is True and not spent:
+        self._save_gate_step = self._step_seq
+        return self.withhold("DONE WITHHELD. The open LibreOffice document reports unsaved ...")
     return """DONE"""
```

gui_agents/s3/agents/worker.py

_adjudicate_done()

```
     terminal = self.grounding_agent.done()
+    if terminal.strip() != "DONE":
+        note = getattr(self.grounding_agent, "last_withheld_note", None)
+        return terminal, {"verdict": "WITHHELD", "objection": note}, note
```

Explanation:

Every type, drag, cell write, or editing key adds one to an uncommitted-edit count. Ctrl+S resets it.

done() with edits outstanding, or with LibreOffice showing unsaved changes, is withheld once.

The reason goes back to the planner in the same step, so the next action commits the change.

The second done() is always taken, so a task that writes nothing is not trapped.

(b) Predicted change

Test	When done() comes with an edit not yet committed, it is withheld, and the run commits the edit (save, or leave the field) before a later done() is accepted.
Holds when	An edit action was dispatched since the last save, and the run calls done().
Targets	2ae9ba84 9cf05d24 51b11269

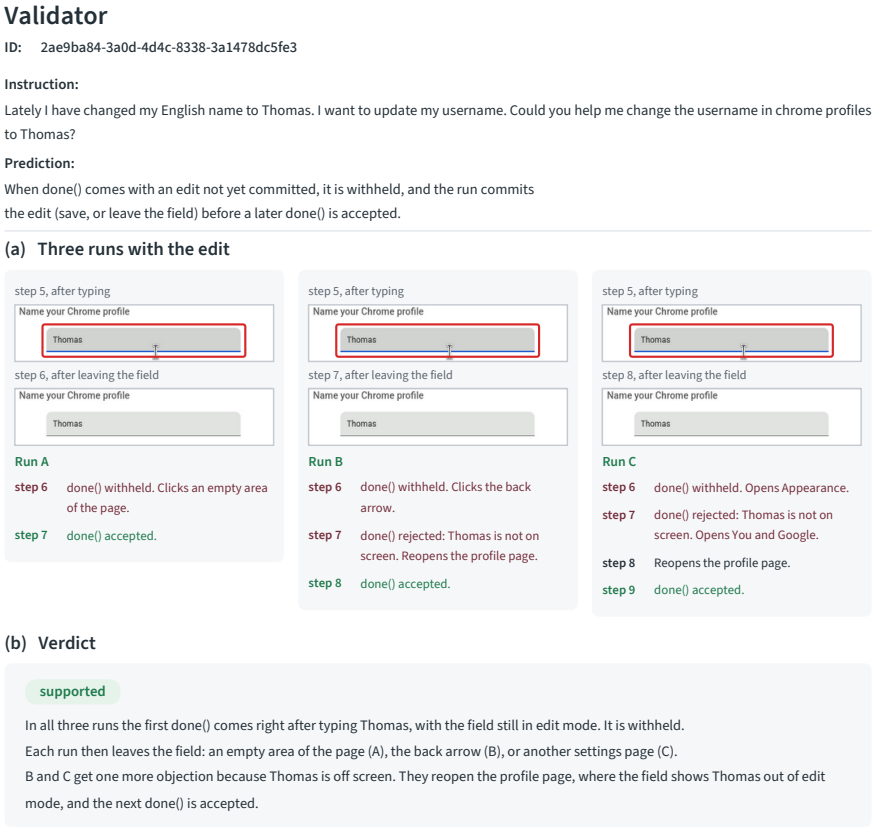

Figure 11: Real end-to-end evidence chain from repeated multimodal executions to an executable harness update. Two failed Chrome runs terminate with the renamed profile field still in edit mode, while a successful contrast commits the edit first. The finding is aggregated with analogous Impress and Calc failures, translated into a termination gate, and supported by three post-edit executions. Task, run, and step links are retained throughout diagnosis, clustering, editing, and behavioral validation; promotion additionally requires the Search/Validation utility gate.

J Representative Role Prompt Templates

The following templates are abridged from the prompts used in our experiments. They preserve the model-visible inputs, decision rules, and structured output contracts while eliding long vocabularies and repeated formatting checks.

Evidence Analyst — Task-round diagnosis

```
You are given every run of ONE task in the current round --
{n_runs} runs, each with its benchmark score.
At least one scored 0.

YOUR QUESTION
For each run that scored 0: at which step did it stop getting
nearer to the end state required by the task, and what exactly did it
do there? Answer in observations, not in causes.

WHAT YOU ARE GIVEN
(1) HARNESS CARD: loop, primitives, semantics, recorded fields.
(2) TASK CARD: instruction, domain, related apps, feasibility,
    setup steps, and any task-published hint.
(3) RUNS: actions, planner text, screenshots, and run scores.

PROCEDURE
```

```
1. If some run scored, use it as the REFERENCE and find the
   first outcome-relevant divergence.
2. Find the earliest DECISIVE STEP that introduced an error
   not fully repaired later.
3. Record did, machine, signature, mechanism, counterfactual.
4. Ground the finding in a verbatim decisive-step quote.

OUTPUT (one JSON object)
{
  "screens_wanted": [{"run":"C","step":7,"kind":"before"}],
  "findings": [{
    "run":"C", "decisive_step":N,
    "contrast":{"reference_run":"A","step":N,
                "reference_did":"..."} | null,
    "signature":{"outcome":"...","noticed":"..."},
    "did":"...", "machine":"...", "mechanism":"...",
    "counterfactual":"..." | null, "quotes":["VERBATIM"]
  }]
}
```

Cross-task Clusterer — Recurring behavior modes

```
Outcome category {outcome_category}: {outcome_category_def}.

Below are all {n} findings in this outcome category from
{n_tasks} tasks. Each contains an Analyst mechanism, what the
record showed, an optional successful contrast, and an optional
evidence-backed counterfactual.

Default to ONE behavioral mode. Split only when modes require
materially different behavioral_target values AND each has a
membership predicate that the other mode's members would fail.

A mode is publishable only if its members span at least TWO
distinct task ids. Put unsupported findings in residue. Every
finding id must appear exactly once. Do not name a harness
module, patch surface, prompt edit, or task-specific answer.

FINDINGS
{findings}

OUTPUT (one JSON object)
{
  "modes": [{
    "label":"UPPER_SNAKE",
    "mechanism":"one cross-task observable mechanism",
    "membership_test":"predicate over one Analyst finding",
    "behavioral_target":"smallest supported alternative" | null,
    "members":["finding_id", "..."]
  }],
  "residue":["finding_id", "..."]
}
```

Harness Engineer — Evidence-linked source update

```
You are the GUI-HARVEST Harness Engineer. Follow
ENGINEER_BRIEF.md exactly. Inspect only the baseline, candidate,
and frozen round evidence. Do not run GUI evaluation or decide
promotion.

YOUR JOB
1. Select one candidate, or a small compatible set explained by
   one shared mechanism.
2. Before editing, write proposal.pre.json and freeze targets,
   source attribution, predictions, invariants, and risks.
3. Edit only capability-approved candidate files; update source
   and HARNESS_CARD.md in the same edit.
4. Run local L0/L1 tests and write ENGINEER_REPORT.md.

proposal.pre.json  [schema: gui-harvest-intervention-v1]
{
  "proposal_id":"...",
  "targets":[{"candidate_id":"...", "lane":"...",
              "outcome_category":"..."}],
```

```
  "attribution":{"surface":"...", "mechanism":"...",
    "source_locations":[{"path":"...", "symbol":"...",
                         "anchor":"..."}]},
  "intervention":{"change_kind":"...", "summary":"...",
                  "genericity":"..."},
  "predictions":[{"prediction_id":"...",
    "target_candidate_ids":["..."], "statement":"...",
    "observable_test":"...", "target_task_ids":["..."],
    "required_for_promotion":true}],
  "invariants":["..."], "expected_regression_risks":["..."],
  "prior_attempts_addressed":[...]
}
```

Behavioral Validator — Prediction–execution check

```
You are the behavioral Validator in a GUI-agent harness experiment.
Judge only whether the frozen behavioral prediction is supported by
the post-edit trajectories. Do not judge task success, utility, or
promotion. Evaluate all supplied runs.

VERDICT RULES
- supported: relevant trigger(s) occurred and the claimed behavior held.
- contradicted: a relevant trigger occurred and a clear response
  violated the prediction.
- inconclusive: no relevant trigger occurred, evidence is missing,
  or the observable test cannot be decided from supplied rows.
- Never infer behavior from evaluator score; no score is supplied.
- Cite exact run labels and RESPONSE ROW step numbers.

OUTPUT (exactly one JSON object)
{
  "prediction_id":"...", "task_id":"...",
  "verdict":"supported|contradicted|inconclusive",
  "summary":"brief evidence-based explanation",
  "run_assessments":[
    {"run":"A", "verdict":"supported|contradicted|inconclusive",
     "reason":"..."}
  ],
  "citations":[
    {"run":"A", "step":3, "claim":"what this row proves"}
  ]
}
```