TRACEVERIFIER: EVIDENCE-LINKED CLAIM GRAPHS FOR GUI AGENT TRAJECTORY VERIFICATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Reliable verification is essential for evaluating GUI agents, curating training trajectories, and producing learning signals. Yet observable-trajectory verification remains difficult: plausible final states can conceal instruction-violating execution, while long trajectories can obscure decisive evidence. A single task label further hides the evidence behind a decision and whether downstream requirements were blocked by earlier errors. We formulate Structured Trajectory Verification as a task that maps a task instruction and an observed GUI trajectory to a structured verification report rather than a single outcome label. To address this task, we present **TraceVerifier**, which resolves instruction-derived claims using localized evidence, semantics-matched operators, and assessment dependencies, producing an evidence-grounded report that distinguishes direct failures from dependency-blocked requirements. We also construct TraceVerifierBench, comprising 384 mobile GUI trajectories from 85 applications, with outcome labels and two fine-grained diagnostic pilots. Across three task-outcome evaluations, TraceVerifier records the highest accuracy and pass precision among the evaluated verifier methods, exceeding the strongest method baseline on each benchmark by 2.85–12.71 accuracy points. Beyond final-label performance, it achieves 82.27% evidence-localization F_1, while its automatically compiled dependencies reach 90.18% blocked-claim F_1 and 81.59% immediate-blocker F_1. Finally, TraceVerifier's structured outputs serve as fine-grained supervision: Process-Trace Distillation yields a best observed Qwen3.5-9B accuracy of 85.66%, within 0.82 points of the untrained Qwen3.5-27B reference.

1 INTRODUCTION

Graphical user interfaces (GUIs) are the interaction layer through which users operate websites, desktop software, and mobile applications. GUI agents operate this layer from natural-language instructions by observing interface states and issuing grounded actions. Recent systems span web interaction, trajectory synthesis, and mobile-app control (Deng et al., 2023; Xu et al., 2025; Rawles et al., 2025).

As GUI agents become more capable, determining whether their trajectories actually satisfy user instructions becomes a central infrastructure problem. A GUI-agent verifier maps an instruction and the resulting sequence of observations and actions to a task decision. Verifier outputs affect benchmark scores, trajectory curation, and policy learning, so their reliability and granularity determine what systems can measure, diagnose, and learn (Xu et al., 2025; Lù et al., 2025; Wan et al., 2026).

GUI-agent verification broadly follows three approaches. Programmatic verifiers provide reproducible decisions given privileged environment state; human evaluators interpret open-ended goals but are costly to scale; and model-based verifiers assess observable screenshots, actions, and responses. We study post-hoc model-based verification of recorded trajectories.

Yet a verifier can return the correct task outcome for the wrong reason. Reliable trajectory verification therefore requires jointly reasoning about three tightly coupled issues. First, a verifier must determine whether the required interaction was actually executed, not merely whether the final state appears correct. Different interaction paths can produce visually similar or identical final states: an agent may reach the expected page after acting on the wrong interface element, using the wrong entry point, or skipping a required operation. As illustrated in figure 1, a final-state judge can therefore

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.

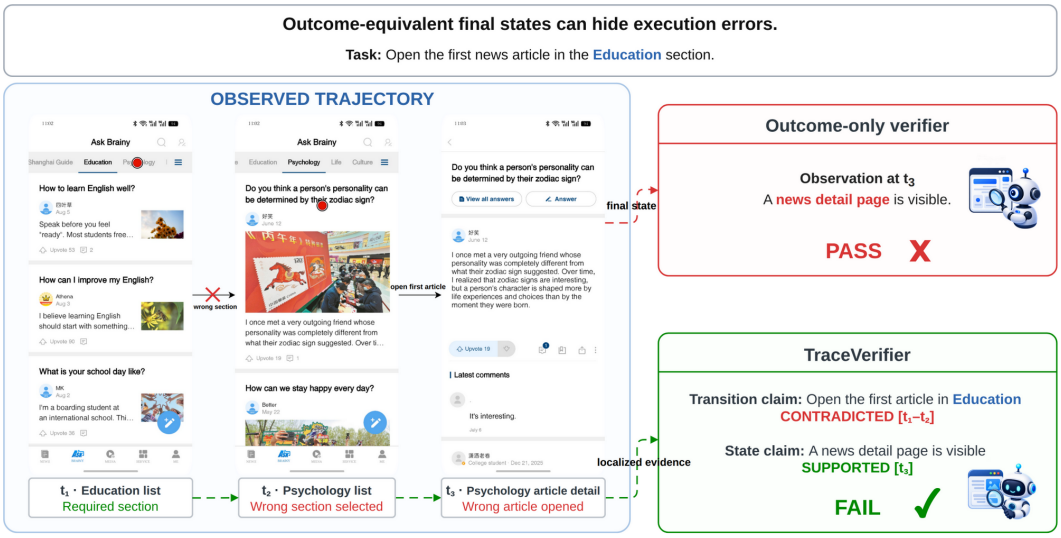


Figure 1: **Motivation: Outcome-equivalent final states can hide execution errors.** The task requires opening the first article in the Education section, but the agent enters Psychology and opens its first article. An outcome-only verifier sees a news detail page and returns PASS, whereas TraceVerifier identifies the contradicted transition claim and returns FAIL.

accept an outcome-equivalent execution error. Second, the verifier must locate the decisive evidence for each requirement. Reading an entire trajectory at once does not reliably surface that evidence: long traces contain irrelevant or competing observations, while critical feedback may appear only briefly (Wan et al., 2026). Third, the verifier must determine whether downstream requirements remain assessable after an upstream failure. If navigation fails, for example, later requirements may become unassessable; treating each of them as an independent failure obscures the root error. These challenges expose a gap between a task-level verdict and the evidence needed to audit it: a complete verification task must specify the unit of judgment, the provenance of supporting evidence, and the assessability of downstream claims.

We introduce *Structured Trajectory Verification* as a task paradigm for evaluating GUI-agent trajectories beyond a single task-level outcome. A structured verifier should expose how an instruction is decomposed into verifiable requirements, how each requirement is judged from trajectory evidence, and how uncertainty or upstream failures affect the assessment. The task therefore makes the trajectory decision auditable. Section 3.1 formalizes Structured Trajectory Verification.

We present **TraceVerifier** as a realization of this task paradigm. Structured Trajectory Verification specifies what a trustworthy verifier should expose; TraceVerifier specifies how to produce it. As shown in figure 2, TraceVerifier compiles a Verification Contract's Execution and Output Requirements into typed claims, routes each claim to localized temporal evidence and a semantics-matched operator, and resolves direct judgments under assessment dependencies. The resulting report contains per-claim statuses, cited evidence, blockers, and a task decision.

Our contributions are threefold:

- We introduce *Structured Trajectory Verification*, a task paradigm that extends GUI-agent evaluation from binary outcome prediction to auditable requirement-level assessment with evidence and failure provenance.
- We present **TraceVerifier**, an evidence-linked and dependency-aware framework that realizes this task through typed claims, claim-local evidence, specialized verification operators, dependency resolution, and task-level aggregation.
- We construct **TraceVerifierBench**, a benchmark of 384 trajectories spanning 85 real-world mobile applications. Outcome labels support task-level evaluation, while diagnostic subsets separately evaluate evidence localization and blocker attribution.

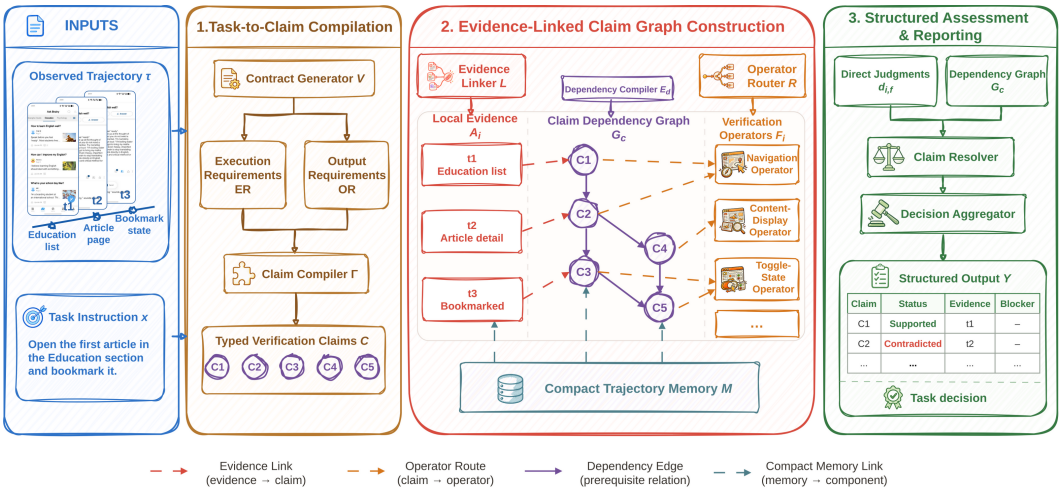


Figure 2: **TraceVerifier overview.** The framework consists of three stages: *Task-to-Claim Compilation*, *Evidence-Linked Claim Graph Construction*, and *Structured Assessment and Reporting*.

2 Related Work

Programmatic verifiers. State-based evaluators in WebArena (Zhou et al., 2024), OSWorld (Xie et al., 2024), and AndroidWorld (Rawles et al., 2025) are reproducible with privileged assertions, but task-specific instrumentation and sandbox access limit portability to unseen applications.

Human evaluators. Human evaluators in Online-Mind2Web (Xue et al., 2025) handle open-ended goals and changing interfaces, but their cost limits repeated benchmarking and supervision.

Model-based verifiers. AgentTrek (Xu et al., 2025) provides global judging or filtering, whereas WebJudge (Xue et al., 2025) and SeekJudge (Wan et al., 2026) localize evidence before producing outcome or step-level judgments. GRADE (Sui et al., 2026), Universal Verifier (Rosset et al., 2026), and OS-Themis (Li et al., 2026) further decompose requirements and manage process evidence or cascading failures.

Appendix A provides a detailed taxonomy and timeline.

3 TraceVerifier

3.1 Task Formulation

Given a natural-language instruction x and an observed GUI trajectory $\tau = \{(o_t, a_t)\}_{t=1}^{T}$, Structured Trajectory Verification determines whether the trajectory satisfies the requirements expressed by x and produces an auditable report rather than only a task-level outcome label. The report is intended to answer three questions.

Q1: What did the instruction require? The verifier identifies the required actions, intermediate states, transitions, and final properties. This decomposition distinguishes correct execution from an outcome-equivalent shortcut.

Q2: What happened for each requirement, which observations support each judgment, and is that evidence sufficient? The verifier determines what occurred, cites supporting trajectory evidence, and distinguishes decisive judgments from insufficient evidence. This makes the decision auditable and separates unresolved claims from contradictions.

Q3: What was the root failure, and which requirements did it block? The verifier distinguishes direct failures from requirements made unassessable by an earlier error. Failure provenance therefore avoids counting downstream blocked requirements as independent failures.

The verification report is

$$\mathcal{Y} = (\hat{y}, \{z_i\}, \{A_i\}, D), \tag{1}$$

where $\hat{y}$ denotes the task-level decision, z_i the judgment for requirement i, A_i the trajectory evidence associated with that judgment, and D the dependency and blocker information needed to represent failure provenance. The formulation specifies the information exposed by the report; the following sections describe how TraceVerifier constructs and resolves these components.

3.2 Task-to-Claim Compilation

Task-to-Claim Compilation converts the instruction into a Verification Contract and a set of typed claims.

Contract Generator. The Contract Generator parses the instruction into atomic requirements and organizes them into two complementary sets: *Execution Requirements* (ERs), which describe actions, intermediate states, or transitions that must occur during interaction, and *Output Requirements* (ORs), which describe properties that the final state or generated content must satisfy. It identifies these requirements from the verbs, intermediate-state constraints, transitions, and final-state or content constraints expressed in x, and derives the *Verification Contract*

$$\mathcal{V}(x) = (\mathrm{ER}, \mathrm{OR}), \qquad \mathrm{ER} = \{\mathrm{ER}_i\}_{i=1}^{m}, \quad \mathrm{OR} = \{\mathrm{OR}_j\}_{j=1}^{k}, \tag{2}$$

Each ER_i specifies a required operation, intermediate state, or transition, whereas each OR_j specifies a semantic property of the final state or generated content.

Claim Compiler. The Claim Compiler maps every requirement to a typed verification claim:

$$C = \Gamma(\mathrm{ER} \cup \mathrm{OR}) = \{c_1, \ldots, c_n\}, \qquad n = m + k, \tag{3}$$

where Γ denotes claim compilation. Each claim specifies what to verify, whether it belongs to an ER or OR, and what trajectory evidence can support its assessment. The resulting claims retain requirement-level granularity for subsequent evidence and dependency analysis.

3.3 Evidence-Linked Claim Graph Construction

The second stage constructs an evidence-linked claim graph with claim, evidence, and verification-operator nodes. Let $S_\tau = \{s_t\}_{t=1}^{T}$ denote the trajectory-step evidence nodes, $\mathcal{F}$ the operator library, and $M(\tau)$ the compact trajectory memory. For each claim c_i, *claim-centric dual routing* is defined as

$$A_i = \mathcal{L}(c_i, \tau), \qquad \mathcal{F}_i = \mathcal{R}(c_i), \qquad d_{i,f} = f(c_i, \tau_{A_i}, M(\tau)),\ f \in \mathcal{F}_i, \tag{4}$$

where the Evidence Linker $\mathcal{L}$ selects steps A_i, the Operator Router $\mathcal{R}$ selects semantics-matched operators $\mathcal{F}_i$, and $d_{i,f}$ is the direct judgment from operator f.

The two routes define claim–evidence links $E_{\mathrm{ev}} = \{(c_i, s_t) : t \in A_i\}$ and claim–operator routing links $E_{\mathrm{op}} = \{(c_i, f) : f \in \mathcal{F}_i\}$. Together with the assessment-dependency edges E_d defined below, the complete graph is

$$G = (C \cup S_\tau \cup \mathcal{F}, E_{\mathrm{ev}} \cup E_{\mathrm{op}} \cup E_d). \tag{5}$$

The graph records the evidence and operators used for each claim and the dependencies that determine its assessability.

Evidence Linker. Long GUI trajectories can dilute decisive evidence: adding irrelevant screenshots can make the decisive evidence harder to identify (Wan et al., 2026). The local view τ_{A_i} preserves a short window around informative steps, including the state before an action and its result. The global view $M(\tau)$ maintains *Compact Trajectory Memory*, a structured summary of task progress and relevant history. Together, these views retain local detail without losing the context needed to interpret it.

Compact Trajectory Memory. At step t, the system updates a raw working memory with newly extracted facts and action, then compacts it into a bounded memory. Let e_t denote the structured facts extracted from the step:

$$W_t = \mathcal{U}(W_{t-1}, o_t, a_t, e_t). \tag{6a}$$

This updates W_t using the previous memory, observation o_t, action a_t, and extracted facts e_t.

$$M_t = \Phi_\theta(W_t, M_{t-1}, x, \mathcal{V}(x)), \qquad M(\tau) = M_T. \tag{6b}$$

This compactor produces the bounded trajectory memory and sets $M(\tau) = M_T$.

$$M_t = (a_t^{\text{summary}}, K_t, B_t), \qquad |K_t| \le K_{\max}. \tag{6c}$$

This decomposes memory into the action summary a_t^{summary}, key facts K_t bounded by $K_{\max}$, and blocker records B_t. Each record stores its signal, detection step, lifecycle status, and root-cause attribution. The memory supplies global context, while claim-specific evidence grounds each obligation judgment.

Verification Operators. The Operator Router assigns each claim to semantics-matched operators, which return a direct claim judgment and its supporting evidence. Each operator checks conditions specific to the claim semantics, such as whether an action caused the required transition or whether entered content matches the target. The Evidence Linker selects observations, the Operator Router interprets them, and the Claim Resolver combines their judgments.

Dependency Compiler. The Dependency Compiler adds an edge $c_i \to c_j$ when c_i is an assessment prerequisite for c_j. The resulting claim-level subgraph $G_c = (C, E_d)$ is a directed acyclic Assessment Dependency Graph. Dependency edges encode prerequisite relations between claims, whereas evidence links connect each claim to the trajectory steps used to assess it. For example, assessing whether the agent enabled dark mode may require first establishing that it opened the display settings; the evidence link instead points to the step showing the resulting toggle state.

3.4 Structured Assessment and Reporting

TraceVerifier resolves direct claim judgments into a structured assessment of the trajectory. It instantiates the report defined in Eq. 1: $\hat{y}$ is the task decision, z_i is the resolved status of claim c_i, A_i contains the trajectory steps cited for that claim, and D records dependency and blocker provenance. In TraceVerifier, each claim receives one of four statuses:

$$z_i \in \{\text{Supported}, \text{Contradicted}, \text{Insufficient Evidence}, \text{Dependency Blocked}\}. \tag{7}$$

Supported and Contradicted require decisive local evidence; Insufficient Evidence denotes an unresolved assessable claim, while Dependency Blocked denotes an unmet prerequisite.

Claim Resolver. The Claim Resolver distinguishes direct failures from claims that cannot be assessed because a prerequisite has failed. A claim with direct evidence is assessed independently. Otherwise, a claim whose prerequisite is contradicted is marked as Dependency Blocked rather than Contradicted. For example, if the agent opens the wrong application, subsequent claims about reaching a settings page and enabling an option are attributed to that initial failure instead of being counted as additional failures. Dependency resolution changes downstream attribution while preserving an already-determined task decision.

Decision Aggregator. TraceVerifier aggregates the resolved claims into a binary task decision: all required claims being supported yields PASS; a contradicted, insufficiently evidenced, or unresolved required claim yields FAIL. Dependency-blocked claims retain their blocker provenance in the report, while the task-level evaluation maps unresolved required claims to FAIL. These resolution and aggregation rules instantiate the information requirements of Structured Trajectory Verification in TraceVerifier.

3.5 Process-Trace Distillation

A task-level label supervises only the final Pass/Fail decision and does not specify how a verifier should decompose the instruction, locate supporting evidence, or attribute downstream failures. Our structured report exposes these intermediate decisions as training targets.

For a trajectory τ, a verification process trace is an ordered sequence of intermediate records that links each claim to its resolved claim status, cited trajectory evidence, evidence sufficiency, and dependency and blocker provenance, followed by the final task decision. These fields provide distinct supervision: resolved claim statuses teach task decomposition, cited trajectory evidence teaches evidence selection, evidence sufficiency distinguishes unresolved claims from contradictions, and dependency and blocker provenance represents the effect of upstream failures. Process-Trace Distillation transfers this report-derived supervision from a Qwen3.5-27B (Qwen Team, 2025) teacher to a smaller verifier.

The teacher generates process traces for trajectories from the News subset of TraceVerifierBench whose task decisions agree with the human labels. We use these records as initial process-level targets. Because a correct task decision can still contain an erroneous intermediate target, targeted correction identifies the earliest semantic error and revises that target together with its affected downstream records. The corrected traces are then used for additional student training.

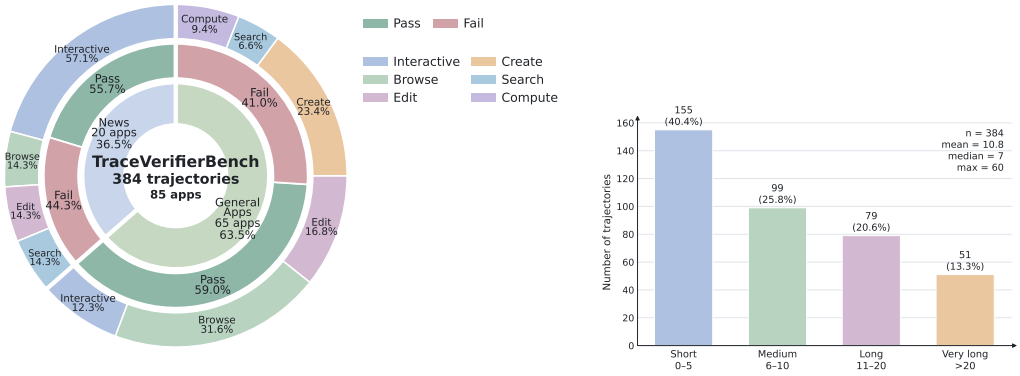
We compare three 9B settings on the disjoint General Apps-244 cohort: an untrained base model, process-trace distillation, and process-trace distillation with targeted correction. The base-to-distillation comparison tests whether report-derived intermediate targets improve over task-level inference alone. The distillation-to-correction comparison tests whether correcting erroneous intermediate targets provides additional benefit. An untrained Qwen3.5-27B verifier provides a reference comparison. Details of the training stages and correction procedure are provided in Appendix D.

4 TRACEVERIFIERBENCH

Mobile GUI benchmarks primarily measure task completion (Sui et al., 2026), while verifier-focused resources often target web trajectories, computer-use settings, or step-level rewards (Lù et al., 2025; Rosset et al., 2026; Wan et al., 2026). Few existing resources combine task-outcome labels with fine-grained annotations for evidence localization and dependency-based failure attribution for mobile GUIs. We constructed **TraceVerifierBench** independently of TraceVerifier for outcome and diagnostic evaluation; it was not used to tune the verification method.

TraceVerifierBench contains two complementary outcome subsets, General Apps and News, and each item provides a task instruction, temporally ordered GUI observations and actions, an optional final response, and a task-outcome label. Their composition and trajectory-length distributions are shown in figure 3. The full benchmark contains 384 trajectories from 85 applications, with 222 pass and 162 fail labels; General Apps contains 244 trajectories from 65 applications and News contains 140 trajectories from 20 applications.

All trajectories have pass/fail labels for task-level evaluation. We further construct *TraceVerifierBench-Dense*, a diagnostic suite within General Apps consisting of two separate pilots: evidence localization and dependency diagnosis. Detailed annotation protocols are provided in Appendix B.2.



(a) Benchmark composition.

(b) Trajectory-length distribution.

Figure 3: Overview of TraceVerifierBench: (a) composition; (b) trajectory-length distribution.

Table 1: Task-outcome verification on fixed denominators (%). Bold and underline mark the best and second-best observed values, respectively; pass is the positive class.

Framework	CUA-FARA				News				General Apps			
	Acc.	Prec.	Recall	F_1	Acc.	Prec.	Recall	F_1	Acc.	Prec.	Recall	F_1
TraceVerifier Ours	**73.58**	**35.14**	76.47	**48.15**	**75.71**	**90.74**	62.82	<u>74.24</u>	**86.48**	**93.02**	83.33	**87.91**
WebJudge COLM 2025	50.94	23.08	<u>88.24</u>	36.59	51.43	66.67	25.64	37.04	59.02	77.50	43.06	55.36
MMRubricAgent	57.55	25.86	<u>88.24</u>	40.00	49.29	55.56	44.87	49.65	47.13	56.88	43.06	49.01
AgentTrek evaluator ICLR 2025 Spotlight	25.47	17.71	**100.00**	30.09	<u>72.86</u>	67.24	**100.00**	**80.41**	<u>73.77</u>	69.42	**99.31**	<u>81.71</u>
Caption-then-Reason COLM 2024	57.55	27.42	**100.00**	<u>43.04</u>	65.71	65.00	<u>83.33</u>	73.03	69.26	67.88	<u>90.97</u>	77.74
SeekJudge	<u>62.26</u>	<u>27.45</u>	82.35	41.18	69.29	<u>71.08</u>	75.64	73.29	71.72	<u>78.63</u>	71.53	74.91

Note: CUA-FARA is the 106-trajectory `fara7b_om2w_browserbase` subset of CUAVerifierBench (17 pass, 89 fail); a constant FAIL predictor attains 83.96% accuracy, with zero pass recall and F_1.

5 Experiments

5.1 Experimental Setup

Benchmarks. We evaluate TraceVerifier on the News and General Apps subsets of TraceVerifierBench and on the 106-trajectory `fara7b_om2w_browserbase` subset of CUAVerifierBench, which we denote CUA-FARA.

Baselines and implementation details. We compare TraceVerifier with WebJudge (Xue et al., 2025), MMRubricAgent (Rosset et al., 2026), the AgentTrek evaluator (Xu et al., 2025), Caption-then-Reason (Pan et al., 2024), and SeekJudge (Wan et al., 2026). For a controlled comparison, TraceVerifier and all five baselines use the same Qwen3.5-27B backbone service. Appendix B.1 provides the complete benchmark, evaluation, and implementation details.

5.2 Task-Outcome Reliability

TraceVerifier is most effective at reducing false-pass decisions while retaining most successful trajectories. As shown in table 1, it achieves the highest accuracy and pass precision among the evaluated methods on all three benchmarks. On the primary General Apps benchmark, TraceVerifier reaches 86.48% accuracy and 93.02% pass precision, improving accuracy and F_1 over the strongest baseline, the AgentTrek evaluator, by 12.71 and 6.20 points, respectively. Its 83.33% pass recall shows that the precision gain is not obtained by rejecting most successful trajectories.

The other benchmarks reveal different precision–recall behavior. On News, TraceVerifier favors pass precision, whereas the AgentTrek evaluator achieves higher recall and F_1. This trade-off indicates that TraceVerifier applies a more conservative pass criterion and accepts fewer failed trajectories. On the browser-based CUA-FARA benchmark, accuracy mainly reflects the fail majority, so we compare pass precision and F_1: TraceVerifier attains the highest values (35.14% and 48.15%) while recovering 76.47% of successful trajectories, indicating that it remains informative beyond mobile applications. Appendix C.4 reports performance across trajectory lengths.

5.3 Component Contributions to Outcome Reliability

We first examine how individual runtime components affect the final task decision by comparing the full framework with three matched variants on the General Apps subset. **Full** routes each claim to a short claim-specific temporal window together with compact memory and assigns semantics-matched verification operators. **w/o evidence scoping** instead supplies every operator with all informative trajectory steps; **w/o specialized operators** replaces semantics-matched routing with a single generic operator for all Execution Requirements; and **w/o context compaction** passes uncompressed working memory to downstream verification. All configurations use the same task set, backbone, retry policy, and fixed-denominator scoring protocol.

Evidence scoping. Removing evidence scoping decreases accuracy from 86.48% to 78.69%, while F_1 decreases from 87.91% to 80.45%. The larger recall loss (9.02 versus 5.32 points for precision) is

Table 2: Component ablations on the General Apps subset (%).

Variant	Acc.	Prec.	Recall	F_1	ΔAcc.
Full	86.48	93.02	83.33	87.91	–
w/o evidence scoping	78.69	87.70	74.31	80.45	−7.79
w/o specialized operators	78.69	88.33	73.61	80.30	−7.79
w/o context compaction	75.00	82.68	72.92	77.49	−11.48

consistent with irrelevant trajectory content making evidence of successful execution harder to identify. These results suggest that evidence scoping helps operators focus on claim-relevant observations while retaining global context.

Specialized operators. Replacing semantics-matched operators with a generic operator decreases accuracy from 86.48% to 78.69%, while F_1 decreases from 87.91% to 80.30%. The larger recall loss (9.72 versus 4.69 points for precision) is consistent with the generic operator missing more evidence of successful execution. These results indicate that semantics-matched operators provide suitable assessment criteria across heterogeneous claim types. Appendix C.2 shows that all 12 registered operators were used and that 967 of 973 assignments for ER-derived claims (99.38%) invoked specialized operators. The ablation measures the resulting effect on task-level decisions.

Context compaction. Removing context compaction produces the largest observed degradation, reducing accuracy from 86.48% to 75.00% and F_1 from 87.91% to 77.49%. Both variants retain claim-local evidence, typed claims, and dependency relations; the ablation therefore isolates bounded compaction from uncompressed working history. Precision and recall decrease by similar amounts (10.34 and 10.41 points), so the observed degradation affects both pass-decision precision and successful-trajectory recognition. These results suggest that compact memory helps retain decision-relevant history in a bounded representation.

The three ablations above isolate evidence scoping, operator specialization, and context compaction while retaining typed claims and resolved claim statuses. Complementary comparisons in Appendix C.1 show that *TraceVerifier (Full)* obtains 3.69–4.92 points higher observed accuracy than *Independent Requirement Checks* and two task-level judge baselines. The appendix provides the variant definitions and complete outcome results.

5.4 Fine-Grained Diagnostic Reliability

The component ablations above measure reliability at the task-decision level. We next evaluate the quality of the intermediate information in the structured report through two diagnostic experiments.

Setup. We evaluate fine-grained diagnostic quality on TraceVerifierBench-Dense, the diagnostic annotation suite comprising separate evidence-localization and dependency-diagnosis pilots from the General Apps subset. Annotation and evaluation details are provided in section B.3.

Can TraceVerifier localize supporting evidence? For each progress claim, the evidence track evaluates whether TraceVerifier's cited steps match the evidence identified by a human annotator. Table 3a shows that TraceVerifier achieves 74.77% precision, 91.45% recall, and 82.27% F_1 against human relevant-step annotations. Recall remains above 89% for every claim type, showing that evidence recovery is consistent across heterogeneous claim semantics. The gap between recall and precision indicates that TraceVerifier more often includes additional context than misses evidence identified by the annotator.

Do dependencies improve blocker diagnosis? We evaluate dependency-aware diagnosis on 128 controlled scenarios while holding claims, evidence, and direct claim judgments fixed and varying only the dependency graph. We compare human-reviewed, automatically compiled, naive sequential, and empty graphs. Claim accuracy measures the correctness of the resolved claim statuses. Blocked-claim F_1 measures whether downstream claims are correctly identified as dependency blocked, while immediate-blocker F_1 measures whether each blocked claim is linked to its nearest responsible claim. The automatic dependency compiler achieves 80.3% edge precision, 87.4% recall, and 83.7% edge F_1, and produces a valid DAG for every task in this pilot. The complete scenario construction and replay protocol are provided in Appendix B.3.

Table 3: Fine-grained diagnostics on TraceVerifierBench-Dense (%).

Claim type	Claims	Prec.	Recall	F_1
Transition	74	68.61	92.16	78.66
State	84	79.17	89.86	84.18
Content	10	79.17	100.00	88.37
Overall	168	74.77	91.45	82.27

(a) Evidence localization.

Graph	Claim Acc.	Blocked F_1	Immediate blocker F_1
Human	100.00	100.00	100.00
Automatic	97.45	90.18	81.59
Naive	97.97	90.62	70.75
Empty	37.12	12.50	12.50

(b) Controlled dependency replay.

Table 4: Process-Trace Distillation on General Apps (%).

Backbone	Training	Acc.	Prec.	Recall	F_1
Qwen3.5-9B Base	None	78.69	81.94	81.94	81.94
Qwen3.5-9B Base	Process-trace distillation	80.74	81.29	87.50	84.28
Qwen3.5-9B Base	+ targeted correction	<u>85.66</u>	87.07	**88.89**	**87.97**
Qwen3.5-27B	None	**86.48**	**93.02**	83.33	<u>87.91</u>

Note: All configurations use the TraceVerifier pipeline.

Automatic dependencies achieve 97.45% claim accuracy, 90.18% blocked-claim F_1, and 81.59% immediate-blocker F_1, substantially outperforming the empty graph on all three measures. The naive graph obtains a similar blocked-claim F_1 of 90.62% but a lower immediate-blocker F_1 of 70.75%, showing that dependency quality matters primarily for identifying the nearest upstream cause rather than merely recognizing that a claim is blocked. Task decisions remain identical across graph variants because dependency resolution refines diagnostic provenance without changing the binary task-decision rule. Automatic dependencies still trail the human graph in immediate-blocker attribution, leaving nearest-blocker identification as the main diagnostic bottleneck.

On CUAStepBench, TraceVerifier achieves the highest bad-step recall (37.40%) and F_1 (40.41%) among methods that natively produce step-level outputs. Appendix C.5 reports the complete results.

5.5 Process-Supervised Verifier Training Results

We train Qwen3.5-9B on the verification process traces defined in section 3.5. Training records come from the News subset of TraceVerifierBench, and evaluation uses the disjoint 244-task General Apps subset with human-gold labels. Table 4 reports the results, and Appendix D provides the training details.

These results test whether the structured report can serve as a supervision interface for a smaller verifier. Distilling its resolved claim statuses, cited trajectory evidence, evidence sufficiency, and dependency and blocker provenance increases 9B accuracy from 78.69% to 80.74%. Adding targeted corrections to these report-derived targets raises accuracy to 85.66% and F_1 to 87.97%, a 6.97-point accuracy gain over the untrained 9B verifier. The resulting accuracy is within 0.82 points of the untrained 27B verifier, while its F_1 is slightly higher. The results show that intermediate report-derived targets improve task-level verification beyond supervision from the final Pass/Fail decision alone.

6 Conclusion

We introduced TraceVerifier, an evidence-linked claim-graph framework for auditable trajectory reports. Across three datasets, it achieves the highest accuracy and pass precision, supports evidence and blocker diagnostics, and enables process-trace distillation to supervise a smaller verifier. Future work will use it to curate GUI-agent trajectories for improved training.

AI use statement

In this work, we did not use generative AI tools for research ideation, study design, data collection, experimentation, implementation, result analysis, or the generation of scientific claims. We used generative AI tools only to improve the readability of author-written text and identify grammatical

errors. We reviewed all AI-assisted work and take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Cong Chen, Kaixiang Ji, Hao Zhong, Muzhi Zhu, Anzhou Li, Guo Gan, Ziyuan Huang, Cheng Zou, Jiajia Liu, Jingdong Chen, Hao Chen, and Chunhua Shen. GUI-Shepherd: Reliable process reward and verification for long-sequence GUI tasks. *arXiv preprint arXiv:2509.23738*, 2025.

Zhaorun Chen, Yichao Du, Zichen Wen, Yiyang Zhou, Chenhang Cui, Zhenzhen Weng, Haoqin Tu, Chaoqi Wang, Zhengwei Tong, Qinglan Huang, Canyu Chen, Qinghao Ye, Zhihong Zhu, Yuqing Zhang, Jiawei Zhou, Zhuokai Zhao, Rafael Rafailov, Chelsea Finn, and Huaxiu Yao. MJ-Bench: Is your multimodal reward model really a good judge for text-to-image generation? *arXiv preprint arXiv:2407.04842*, 2024.

Chaoqun Cui, Jing Huang, Shijing Wang, Liming Zheng, Qingchao Kong, and Zhixiong Zeng. Agentic reward modeling: Verifying GUI agent via progressive trajectory-grounded interaction. *arXiv preprint arXiv:2602.00575*, 2026.

Gaole Dai, Shiqi Jiang, Ting Cao, Yuanchun Li, Yuqing Yang, Rui Tan, Mo Li, and Lili Qiu. V-Droid: Advancing mobile GUI agent through generative verifiers. In *ACM MobiCom 2026*, 2026a.

Gaole Dai, Shiqi Jiang, Ting Cao, Yuqing Yang, Yuanchun Li, Rui Tan, Mo Li, and Lili Qiu. ProRe: A proactive reward system for GUI agents via reasoner-actor collaboration. In *International Conference on Learning Representations*, 2026b.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2Web: Towards a generalist agent for the web. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. VisualWebArena: Evaluating multimodal agents on realistic visual web tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024.

Seongyun Lee, Seungone Kim, Sue Park, Geewook Kim, and Minjoon Seo. Prometheus-Vision: Vision-language model as a judge for fine-grained evaluation. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 11286–11315, 2024. doi: 10.18653/v1/2024.findings-acl.672.

Zehao Li, Zhenyu Wu, Yibo Zhao, Bowen Yang, Jingjing Xie, Zhaoyang Liu, Zhoumianze Liu, Kaiming Jin, Jianze Liang, Zonglin Li, Feng Wu, Bowen Zhou, Zun Wang, and Zichen Ding. OS-Themis: A scalable critic framework for generalist GUI rewards. *arXiv preprint arXiv:2603.19191*, 2026.

Fanbin Lu, Zhisheng Zhong, Ziqin Wei, Shu Liu, Chi-Wing Fu, and Jiaya Jia. STEVE: A step verification pipeline for computer-use agent training. *arXiv preprint arXiv:2503.12532*, 2025.

Xing Han Lù, Amirhossein Kazemnejad, Nicholas Meade, Arkil Patel, Dongchan Shin, Alejandra Zambrano, Karolina Stańczak, Peter Shaw, Christopher J. Pal, and Siva Reddy. AgentRewardBench: Evaluating automatic evaluations of web agent trajectories. In *Conference on Language Modeling*, 2025.

Jiayi Pan, Yichi Zhang, Nicholas Tomlin, Yifei Zhou, Sergey Levine, and Alane Suhr. Autonomous evaluation and refinement of digital agents. In *First Conference on Language Modeling*, 2024.

Qwen Team. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy Lillicrap, and Oriana Riva. AndroidWorld: A dynamic benchmarking environment for autonomous agents. In *International Conference on Learning Representations*, 2025.

Corby Rosset, Pratyusha Sharma, Andrew Zhao, Miguel Gonzalez-Fernandez, and Ahmed Awadallah. The art of building verifiers for computer use agents. *arXiv preprint arXiv:2604.06240*, 2026.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Chenrui Shi, Yuwei Wu, Yang Liu, Ruining Feng, Zirui Shang, Zhi Gao, Lifeng Fan, and Che Sun. Interactive reward agent: GUI task evaluation via environment-state verification. *arXiv preprint arXiv:2607.25904*, 2026.

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

Yang Wan, Zhenhao Zhang, Jierui Wang, and Linchao Zhu. SeekJudge: A practical reward framework for reinforcement learning in computer-use agents. *arXiv preprint arXiv:2607.23263*, 2026.

Jinbiao Wei, Qianran Ma, Yilun Zhao, Xiao Zhou, Kangqi Ni, Guo Gan, and Arman Cohan. OpenComputer: Verifiable software worlds for computer-use agents. *arXiv preprint arXiv:2605.19769*, 2026.

Zhenyu Wu, Jingjing Xie, Zehao Li, Bowen Yang, Qiushi Sun, Zhaoyang Liu, Zhoumianze Liu, Yu Qiao, Xiangyu Yue, Zun Wang, and Zichen Ding. OS-Oracle: A comprehensive framework for cross-platform GUI critic models. In *2026 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 27514–27524, 2026.

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*, 2024.

Tao Xiong, Xavier Hu, Yurun Chen, Yuhang Liu, Changqiao Wu, Pengzhi Gao, Wei Liu, Jian Luan, and Shengyu Zhang. GUI-PRA: Process reward agent for GUI tasks. *arXiv preprint arXiv:2509.23263*, 2025a.

Tianyi Xiong, Xiyao Wang, Dong Guo, Qinghao Ye, Haoqi Fan, Quanquan Gu, Heng Huang, and Chunyuan Li. LLaVA-Critic: Learning to evaluate multimodal models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2025b.

Yiheng Xu, Dunjie Lu, Zhennan Shen, Junli Wang, Zekun Wang, Yuchen Mao, Caiming Xiong, and Tao Yu. AgentTrek: Agent trajectory synthesis via guiding replay with web tutorials. In *International Conference on Learning Representations*, 2025. ICLR 2025 Spotlight.

Tianci Xue, Weijian Qi, Tianneng Shi, Chan Hee Song, Boyu Gou, Dawn Song, Huan Sun, and Yu Su. An illusion of progress? Assessing the current state of web agents. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=6jZi4HSs6o.

Tianyu Yu, Haoye Zhang, Qiming Li, Qixin Xu, Yuan Yao, Da Chen, Xiaoman Lu, Ganqu Cui, Yunkai Dang, Taiwen He, Xiaocheng Feng, Jun Song, Bo Zheng, Zhiyuan Liu, Tat-Seng Chua, and Maosong Sun. RLAIF-V: Open-source AI feedback leads to super GPT-4V trustworthiness. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2025.

Danyang Zhang, Situo Zhang, Ziyue Yang, Zichen Zhu, Zihan Zhao, Ruisheng Cao, Lu Chen, and Kai Yu. ProgRM: Build better GUI agents with progress rewards. *arXiv preprint arXiv:2505.18121*, 2025.

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

A Supplementary Related Work

Following the three evaluation choices outlined in the introduction, we review programmatic, human, and model-based verification. Within model-based verification, methods differ in their evidence access, judgment granularity, and intended use. Figure 4 organizes their technical development by primary verification paradigm and first public release.

A.1 Programmatic Verifiers

Programmatic evaluators inspect URLs, databases, files, or application state. WebArena (Zhou et al., 2024), VisualWebArena (Koh et al., 2024), OSWorld (Xie et al., 2024), and AndroidWorld (Rawles et al., 2025) use functional or task-specific checks in controlled environments. OpenComputer (Wei et al., 2026) extends this approach with structured inspection endpoints and partial rewards. Such checks provide reproducible decisions when privileged state and maintained assertions are available, but extending them to closed-source or changing applications requires application-specific access and upkeep.

A.2 Human Evaluation

Human evaluators can interpret open-ended goals, visual context, and interface changes without application-specific instrumentation. Studies such as Online-Mind2Web (Xue et al., 2025) therefore use human judgments as the reference for assessing agent success and automatic evaluators. Their cost and latency, however, limit repeated evaluation for large benchmarks, trajectory filtering, and online policy optimization.

A.3 Model-Based Verifiers

General-purpose VLM judges establish rubric-conditioned visual evaluation, preference modeling, and judge reliability analysis (Lee et al., 2024; Xiong et al., 2025b; Yu et al., 2025; Chen et al., 2024); GUI verification additionally requires reasoning over actions and temporally related interface states. One line studies step-level rewards, progress signals, or action preferences for policy training and action selection (Lu et al., 2025; Dai et al., 2026a; Zhang et al., 2025; Chen et al., 2025; Wu et al., 2026). Another line gathers additional evidence through visual tools, reasoner–actor interaction, or environment probing (Dai et al., 2026b; Xiong et al., 2025a; Cui et al., 2026; Shi et al., 2026). Our work focuses on post-hoc trajectory verification; we group related methods below by their primary judgment strategy.

Global trajectory judges. Model-based judges operate on observable screenshots and actions. Agent-Eval-Refine (Pan et al., 2024) studies end-to-end and modular VLM evaluation, while Agent-Trek (Xu et al., 2025) uses evaluation for trajectory filtering. AgentRewardBench (Lù et al., 2025) shows that reliability varies with trajectory representation and failure mode. These approaches broaden evaluation beyond instrumented environments, but in long trajectories, decisive evidence can be diluted by numerous screenshots and action descriptions. This motivates decomposed verification that first identifies task-relevant evidence and then evaluates smaller judgment units.

Evidence-selective judges. Evidence selection reduces distraction from long trajectories by focusing judgment on task-relevant observations. WebJudge (Xue et al., 2025) extracts key points and informative screenshots before predicting success. SeekJudge (Wan et al., 2026) separates evidence localization from detailed extraction and produces outcome and step rewards. Their localized evidence primarily supports a global verdict or step signal.

The evolution of model-based verification also reflects a shift in evidence retrieval strategy. Early global judges inspect a trajectory largely as an undifferentiated sequence, leaving the model to discover relevant signals implicitly. Evidence-selective methods make this search more deliberate by locating informative screenshots, steps, or interaction states before judgment. A further step is requirement-driven retrieval: the verifier first defines the claims or obligations that must be assessed, then seeks evidence for each requirement and records whether the requirement is directly supported, contradicted, or made unassessable by an upstream failure. This progression moves from searching

the trajectory for anything useful to specifying what evidence is needed before searching, which is the perspective adopted by Structured Trajectory Verification.

Task-decomposed verifiers. Task decomposition instead refines what is judged. GRADE (Sui et al., 2026) describes hierarchical guidelines containing operational and output-quality requirements together with negative constraints; MMRubricAgent (Rosset et al., 2026) separates process and outcome rubrics, manages criterion-specific context, and controls downstream scoring after upstream failures; and OS-Themis (Li et al., 2026) decomposes trajectories into milestones, isolates critical evidence, and reviews the evidence chain before assigning reward.

Taken together, model-based verifiers have advanced along three complementary stages: global trajectory inspection, evidence-selective retrieval, and requirement-driven evidence retrieval with decomposition. Structured Trajectory Verification unifies these directions in a task that exposes requirement-level judgments, supporting evidence, and downstream assessability. TraceVerifier realizes this formulation through typed claims, localized evidence, and dependency-aware resolution.

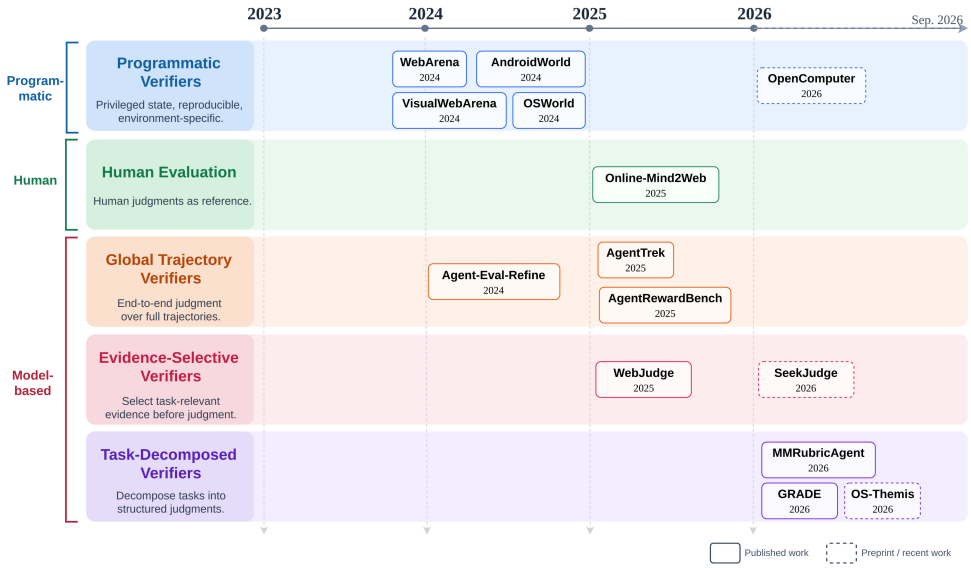


Figure 4: **GUI-agent verification by first public release (2023–2026).** The timeline follows the three evaluation families used in this section: programmatic verification, human evaluation, and model-based verification. Model-based methods are further organized by global trajectory judgment, evidence selection, and task decomposition. Solid and dashed borders distinguish published work from preprints or recent work as of September 2026.

B Benchmark and Evaluation Protocols

B.1 Experimental Setup Details

General Apps is the primary outcome and ablation benchmark, while its two disjoint Dense tracks evaluate evidence localization and blocker diagnosis. The CUA-FARA subset contains 17 pass and 89 fail trajectories and evaluates transfer from mobile to browser-based agents.

All methods use the same Qwen3.5-27B backbone service while retaining their method-specific prompts, input organization, and decision rules. WebJudge, MMRubricAgent, the AgentTrek evaluator, Caption-then-Reason, and TraceVerifier use deterministic decoding. SeekJudge retains its official temperature of 0.2 and maximum output length of 8192 tokens.

B.2 TRACEVERIFIERBENCH DATA AND ANNOTATION PROTOCOL

We generated the trajectories with a single GUI agent and designed roughly five to seven tasks for each application. Each annotation provides the task instruction and the complete recorded trajectory, including GUI screenshots, action logs, and the agent's final response.

Two annotators independently assigned a task-level label after reviewing these materials. We compared their labels and sent every disagreement to a third annotator for arbitration; the third annotator's decision is the final ground-truth label. A trajectory is labeled PASS only when all task requirements are satisfied. A trajectory is labeled FAIL when any requirement is incomplete or cannot be confirmed from the recorded trajectory.

Table 5 reports the cross-tabulation of the two independent annotators. They agreed on 350 of 384 trajectories (91.15%); the remaining 34 cases (8.85%) were resolved by a third annotator.

Table 5: Cross-tabulation of task-level labels assigned by two annotators. Counts are trajectories.

	Annotator 2		
Annotator 1	Pass	Fail	Total
Pass	201	24	225
Fail	10	149	159
Total	211	173	384

B.3 TRACEVERIFIERBENCH-DENSE ANNOTATION DETAILS

TraceVerifierBench-Dense is a diagnostic subset of General Apps designed to evaluate two distinct capabilities of TraceVerifier. It contains two separately annotated pilots, each covering 50 tasks. The evidence-localization pilot examines whether the steps cited for individual progress claims match those identified by human annotators. The dependency-diagnosis pilot examines whether the dependency graph identifies claims blocked by an upstream failure and attributes each blocked claim to the appropriate blocker. We report the two pilots separately because they use different task collections and annotation targets. Task-level Pass/Fail labels support outcome evaluation, while the dense annotations support the two diagnostic analyses.

Table 6: Composition and purpose of the TraceVerifierBench-Dense annotation tracks.

Track	Scale	Human annotation	Evaluation target
Evidence localization	50 tasks, 168 progress claims	Relevant and uncertain trajectory steps for each claim	Claim–step evidence precision, recall, and F_1
Dependency diagnosis	50 tasks, 128 controlled scenarios	Dependency edges and blocker relations	Edge quality, blocked-claim F_1, and immediate-blocker F_1

Evidence-localization protocol. This pilot evaluates whether the claim–step links produced by TraceVerifier identify the trajectory steps that support each progress claim. For each progress claim in the pilot, human annotators review the complete trajectory and mark the relevant step IDs; steps whose relevance is uncertain are recorded separately. We compare TraceVerifier's cited step IDs with the human relevant-step annotations: precision measures the proportion of cited steps that are relevant, recall measures the proportion of human-annotated relevant steps that are recovered, and F_1 combines the two. We exclude claims that only summarize the final task outcome, such as whether the task was completed or an item was successfully saved, because these claims do not correspond to a single execution event. We categorize the remaining progress claims into three types: transition claims for navigation changes, state claims for interface-state changes, and content claims for displayed or generated content.

Dependency-diagnosis protocol. This pilot isolates how the dependency graph affects blocker diagnosis. We hold the claims, evidence, and direct claim judgments fixed and vary only the graph: *Human* uses reviewed dependencies, *Automatic* uses dependencies produced by the Dependency

Compiler, *Naive* links progress claims sequentially and attaches outcome claims to the final progress claim, and *Empty* contains no dependency edges. Each controlled scenario introduces either an upstream contradiction or an evidence gap and masks its descendants. We then measure whether the resolver identifies the correct blocked claims and their blockers without rerunning the VLM.

C Additional TraceVerifier Analyses

C.1 Comparison with Simpler Verifier Architectures

This experiment asks whether simpler decision architectures can match or outperform the full TraceVerifier design. We therefore construct three simpler variants that progressively remove structure while preserving the same task and backbone: (1) *Independent Requirement Checks*, which checks the original requirements independently; (2) *Task-Level Judge (Compact Memory)*, which makes one decision from the compact trajectory memory; and (3) *Task-Level Judge (All Steps)*, which makes the same direct task-level judgment while also receiving all informative trajectory steps. Comparing these variants with TraceVerifier separates the effect of requirement-level decomposition from the effect of direct whole-task judgment and tests whether additional evidence can offset a simpler assessment structure.

Table 7 describes the full system and structural baselines along four semantic dimensions: the unit being verified, the trajectory evidence provided, the graph and evidence links available to the assessment, and the way the task decision is constructed. TraceVerifier evaluates typed claims with claim-local evidence, dependency edges, explicit claim–step links, and specialized operators, then aggregates resolved claim statuses into a task decision. Independent Requirement Checks operates on the original requirements and aggregates generic requirement checks without typed claims or dependency structure. The two task-level variants, *Task-Level Judge (Compact Memory)* and *Task-Level Judge (All Steps)*, collapse assessment to the whole task and differ only in their evidence input. The former receives compact memory, whereas the latter receives all informative steps in addition to that memory. Their paired comparison therefore tests whether broader evidence access can compensate for removing item-level structured assessment.

Table 7: Semantic comparison of the full system and structural baselines. Every variant produces a task-level decision, either by aggregating item-level statuses or by direct judgment.

Variant	Verification unit	Evidence input	Graph and links	Assessment	Output construction
TraceVerifier (Full)	Typed claims	Claim-local evidence + compact memory	Dependency graph + claim–step links	Specialized operators	Claim statuses + aggregated task decision
Independent Requirement Checks	Original requirements	Requirement-local evidence + compact memory	No dependency graph or serialized links	Generic check	Requirement statuses + aggregated task decision
Task-Level Judge (Compact Memory)	Whole task	Compact memory	None	Generic task judge	Direct task decision
Task-Level Judge (All Steps)	Whole task	All informative steps + compact memory	None	Generic task judge	Direct task decision

Table 8: Outcome results for representation-level structural baselines on the General Apps subset (%).

Variant	Acc.	Prec.	Recall	F_1	ΔAcc.
TraceVerifier (Full)	86.48	93.02	83.33	87.91	–
Independent Requirement Checks	81.56	89.60	77.78	83.27	-4.92
Task-Level Judge (Compact Memory)	82.38	89.76	79.17	84.13	-4.10
Task-Level Judge (All Steps)	82.79	88.64	81.25	84.78	-3.69

The results in table 8 favor the full structured architecture. TraceVerifier exceeds the three simpler variants by 3.69–4.92 accuracy points. Relative to *Independent Requirement Checks*, it improves accuracy by 4.92 points and F_1 by 4.64 points, showing the benefit of typed claim-level verification under this setup.

Providing all informative steps to the direct judge improves accuracy by only 0.41 points and F_1 by 0.65 points over compact memory alone. The *Task-Level Judge (All Steps)* variant still

trails TraceVerifier by 3.69 accuracy points and 3.13 F_1 points, showing that item-level structured assessment contributes beyond broader evidence access.

C.2 Operator Routing Audit

Were the specialized operators actually used? The full General Apps run exercises the complete operator library. Every one of the 1,444 serialized claims has an operator assignment, and all 12 registered operators appear in executed assessments. Of the 973 assignments for ER-derived claims, 967 (99.38%) use specialized operators and only six use the generic operator. The full system therefore uses semantics-specific checks in practice instead of collapsing these assignments into one generic check.

How were the operators used? The Operator Router dispatches each ER-derived claim according to its interaction semantics, as defined in equation (4). For example, navigation claims invoke the navigation operator, while text-entry and submission claims invoke their corresponding checks. A claim can use several operators when it combines multiple interaction semantics. The 835 ER-derived claims produce 973 assignments, with 138 claims using more than one operator. The 609 OR-derived claims follow a separate route to the output-quality operator. Table 9 shows the resulting distribution.

Table 9: Executed operator assignments in the Full General Apps run.

Operator	Assignments	Operator	Assignments
Navigation	312	App launch	235
Content display	151	Input content	101
Submission	76	Toggle state	47
Scroll reach	32	Input activation	7
Media control	5	Popup dismissal	1
Generic obligation	6	Output quality	609

C.3 Backbone Sensitivity

Table 10: TraceVerifier backbone sensitivity on the General Apps subset (%). Metrics use pass as the positive class; only the Qwen3.5 backbone changes.

Backbone	Acc.	Prec.	Recall	F_1
Qwen3.5-9B	78.69	81.94	81.94	81.94
Qwen3.5-27B	86.48	93.02	83.33	87.91
Qwen3.5-397B-A17B (GPTQ-INT4)	82.38	90.40	78.47	84.01

TraceVerifier maintains strong performance across all three Qwen3.5 backbones, with accuracy ranging from 78.69% to 86.48% and F_1 from 81.94% to 87.91%. Replacing Qwen3.5-9B with Qwen3.5-27B improves accuracy by 7.79 points (exact McNemar $p = 0.00940$). The largest setting, Qwen3.5-397B-A17B, is a GPTQ-INT4-quantized mixture-of-experts model with 397B total parameters and 17B activated parameters. Its 4.10-point accuracy gap relative to Qwen3.5-27B is not significant ($p = 0.13250$). Overall, these results indicate that TraceVerifier remains effective across a wide range of backbone scales, although the exact accuracy varies by backbone.

C.4 Performance across Trajectory Lengths

Figure 5 shows that TraceVerifier remains reliable as trajectories become longer. Its accuracy is 79.35%, 82.83%, 83.54%, and 90.20% for the 0–5, 6–10, 11–20, and > 20 step groups, respectively. Pass precision remains above 86% in every group, indicating that the framework continues to avoid false-positive task completions even when the trajectory contains many observations and actions. For trajectories longer than 20 steps, TraceVerifier achieves its highest accuracy and F_1, at 90.20% and 87.18%, together with 89.47% precision and 85.00% recall. Relative to the 0–5-step group, accuracy and F_1 improve by 10.85 and 4.02 points.

TraceVerifier also preserves a substantial advantage over the evaluated baselines on long trajectories. For the > 20 step group, its 90.20% accuracy and 87.18% F_1 exceed the strongest baseline, AgentTrek

Eval, by 17.65 and 13.11 points, respectively. Two factors may contribute to the higher accuracy on longer trajectories. First, the > 20 step group contains a larger fraction of fail cases (31/51, compared with 48/155 in the 0–5 step group); because TraceVerifier maintains strong fail recognition, these cases contribute many true-negative decisions. Second, longer trajectories expose more intermediate states and transitions, providing additional evidence for linking observations to claims and resolving their dependencies. Together, these factors help explain TraceVerifier's strong performance on long trajectories.

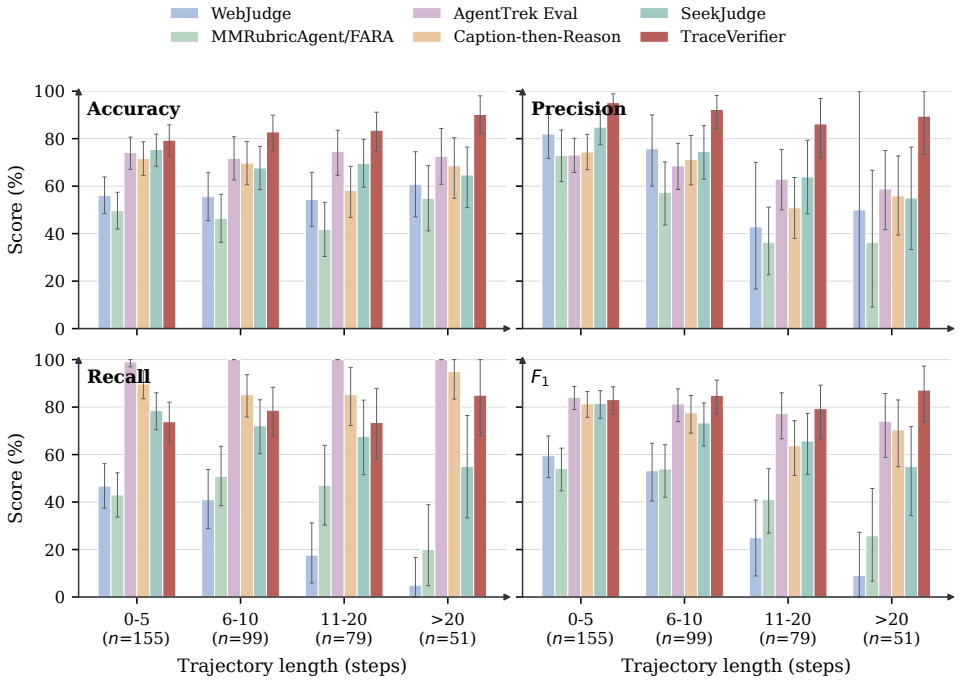


Figure 5: Task-outcome metrics by trajectory length on TraceVerifierBench. Bars compare methods within the same length group; error bars show task-level bootstrap 95% confidence intervals from 10,000 resamples.

C.5 CUAStepBench Analysis

We additionally evaluate TraceVerifier with Qwen3.5-27B on CUAStepBench to assess step-level error localization using human labels for executed steps. All metrics treat bad steps as positive.

Table 11: Step-level error-localization results on CUAStepBench (%). Metrics treat bad steps as positive. Best values are bolded and second-best values are underlined. ‡Results reported by Wan et al. (2026) under their original models and settings. †Methods without native step-level outputs; step labels are obtained using SeekJudge's extraction procedure with the same base model.

Framework	Model	Accuracy	Precision	Recall	F_1
CUAJudge‡†	GPT-5-mini	<u>90.4</u>	<u>49.7</u>	30.8	38.0
CUAJudge‡†	Qwen3VL-8B	**90.9**	**67.7**	12.7	21.3
OSThemis‡†	Qwen3VL-8B	83.6	17.0	14.3	15.6
SeekJudge‡	Qwen3VL-8B	89.4	41.4	20.4	27.3
SeekJudge‡	SeekJudge-9B	89.6	44.5	<u>33.3</u>	<u>38.1</u>
TraceVerifier	Qwen3.5-27B	89.66	43.95	**37.40**	**40.41**

Among methods that natively produce step-level outputs, TraceVerifier achieves the strongest overall balance on CUAStepBench. Relative to SeekJudge-9B, it obtains nearly identical accuracy (89.66% versus 89.6%) and precision (43.95% versus 44.5%), while improving bad-step recall by 4.10 points

and F_1 by 2.31 points. Because the step labels are highly imbalanced, the gains in recall and F_1 provide stronger evidence of dense error-localization quality than accuracy alone.

D PROCESS-TRACE DISTILLATION DETAILS

Distilling report structure and routine semantics. Starting from Qwen3.5-9B Base, the student first undergoes two GRPO stages. Stage A uses Group Relative Policy Optimization (GRPO) (Shao et al., 2024) to learn valid JSON, schema and type compliance, core fields, and decision consistency. Stage B adds routine-specific rewards for numeric values, fact and list overlap, and textual evidence coverage. Together, these stages teach the smaller verifier to reproduce the structured report and its routine-level semantics.

Improving report-derived supervision. A correct teacher decision can still contain an erroneous intermediate target. For challenging training tasks, annotators locate the earliest semantic error and correct that target together with its affected downstream targets. The corrected records are mixed with clean replay for supervised fine-tuning (SFT), followed by a short GRPO stage. This procedure improves the quality of the report-derived supervision while preserving the dependency order of the verification process. Process traces are constructed from the News subset of TraceVerifierBench, while the disjoint General Apps-244 cohort is reserved for final evaluation.

Training setup. Training uses four A800-80G GPUs and one random seed. Stage A runs for 30 steps and weights JSON validity, schema compliance, field types, core decisions, and content shape by 0.30, 0.25, 0.15, 0.20, and 0.10, respectively. Stage B continues for 50 steps and weights JSON, schema, type, decision, numeric agreement, fact/list F_1, and text coverage by 0.10, 0.10, 0.10, 0.30, 0.10, 0.20, and 0.10. Each prompt uses four rollouts with temperature 0.8 and top-p 0.95; validation uses deterministic decoding with one rollout. Maximum prompt and response lengths are 16,384 and 1,024 tokens. The corrected SFT stage uses one epoch, followed by 30 steps of GRPO with increased decision weight.