Published by :
https://www.ijert.org/
An International Peer-Reviewed Journal

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 10 , October - 2026

AURA: Evidence-Gated Verification of Task Completion in Multi-Step AI Agents

Shanmukh Datta
Department of Instrumentation and Control Engineering
Dr. B. R. Ambedkar National Institute of Technology Jalandhar, Punjab, India

Abstract

AI agents complete tasks through workflows involving multiple tool calls and changes to stored data. Verifying completion requires checking whether these actions achieved the requested outcome while respecting the task's constraints. Successful tool execution alone cannot establish this. We present AURA (Agent Uncertainty and Residue Arbiter), a Python and SQLite framework that verifies postconditions, permitted mutations, process requirements, and observation coverage. Independent observations are bound to execution epochs and evaluated under explicit evidence rules to produce PASS, FAIL, or UNCERTAIN verdicts. An optional Counterexample-guided Evaluation with Risk Amortization (CERA) planner selects verification probes using finite candidate worlds, prior weights, and inspection costs; the verifier independently determines the verdict. Evaluation covers 13 scripted local scenarios, 36 controlled SQL cases, and two separate five-episode Qwen2.5 agent cohorts. Across these 59 records, AURA produced 13 PASS, 39 FAIL, and seven UNCERTAIN verdicts. All 59 agreed with supplied harness expectations or specification-sharing reference judgments; this concordance is not independently established verification accuracy. A fresh validation run reproduced these totals through 46 saved-evidence replays and 13 regenerated local scenarios, with replay comparisons limited to selected certificate fields. A separate planning study shows that CERA's modeled inspection-cost advantage depends on prior assumptions and does not hold across all evaluated regimes. The findings support the feasibility of evidence-gated task-completion verification within declared observation boundaries, without establishing general agent reliability, independent specification correctness, or detection of unobserved transient effects.

Keywords— AI agents; runtime verification; task contracts; persistent state; evidence sufficiency; reproducibility

I. INTRODUCTION

Large language model agents combine natural-language reasoning with actions such as querying databases, modifying files, and interacting with application interfaces. ReAct integrates reasoning with interaction, while Toolformer explores learned tool use [1], [2]. These capabilities extend the consequences of an agent's output beyond a textual response, as subsequent users and services may rely on the resulting state. Multi-step workflows may therefore involve coordinated changes, restrictions on unrelated records, and evidence that required steps were completed. Evaluating such workflows requires examining both the resulting environment and the agent's response.

Execution success and task success are not necessarily the same. A database operation may finish without errors but update the wrong row, while a sequence of successful operations may still omit a required report. Conversely, a task may be completed correctly through an execution path different from the developer's preferred sequence. Verified completion introduces an additional evidence requirement: the declared outcomes and constraints must be supported by observations the verifier is authorized to use. An unavailable readback must not be treated as a successful check. These distinctions motivate a completion interface that identifies both established violations and unresolved obligations.

Existing agent benchmarks provide an established foundation for outcome-based evaluation. SWE-bench uses software tests, WebArena evaluates functional task completion, and AppWorld checks application state, including collateral changes [3]–[5]. AURA does not claim to introduce state-based evaluation or to be the first framework to address unintended mutations. Instead, it focuses on combining explicit postconditions, mutation boundaries, process evidence, and observation sufficiency within a compact, inspectable runtime verifier. The central challenge is to communicate a completion judgment within declared boundaries when evidence is incomplete or inconsistent, while preserving the records needed for later examination.

AURA, the Agent Uncertainty and Residue Arbiter, brings together five elements: contract-driven verification using independent evidence; three-valued decisions supported by auditable certificates; coverage-aware checks of persistent mutations; optional cost-aware inspection through Counterexample-guided Evaluation with Risk Amortization (CERA); and controlled experiments accompanied by an explicit reproducibility audit. CERA selects evidence probes, while the verifier independently determines the completion verdict and issues the certificate. The contribution lies in integrating these mechanisms and evaluating them within a defined scope, rather than introducing a new foundation for state-based verification. This paper reports repository commit 6d4b714 [6] and distinguishes agreement with reference judgments, selected-field replay, and byte-level artifact preservation.

II. RELATED WORK

AgentBench evaluates agents across diverse interactive environments, while OSWorld extends evaluation to real computer environments [7], [8]. SWE-bench, WebArena, and AppWorld provide task-specific executable evaluations [3]–[5]. These benchmarks demonstrate that task completion can be assessed through changes in the environment rather than relying solely on an agent's verbal reports. The tau-bench benchmark examines tool-agent-user interactions under domain-specific policies [9]. ToolSandbox introduces stateful conversational evaluation with intermediate and final milestones [10], highlighting the importance of tool dependencies and environment state. However, their evaluation settings and scales differ from AURA's con-

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 10 , October - 2026

trolled SQL pilot. These works serve as relevant design precedents rather than experimental baselines evaluated in this study, and no comparative performance claims are made against them.

Tool interaction, reflection, and verification serve complementary purposes. ReAct provides action-observation trajectories, while Reflexion uses feedback to improve subsequent attempts [1], [11]. These traces can help diagnose execution and support process constraints when their capture is trusted. However, a successful tool response alone does not establish that all required final-state properties have been satisfied. AURA relies on independently admitted observations and selected process facts rather than evaluating an agent's internal reasoning or using a general learned judge. AgentSynth also addresses outcome verification and reproducible agent evaluation [12]. This paper focuses specifically on the preserved task pack and AURA adapter, rather than the full range of current AgentSynth capabilities.

The conceptual foundation of AURA draws on established software verification principles. Hoare's axiomatic approach relates program execution to preconditions and postconditions [13]. Runtime verification evaluates observed executions against specifications, while three-valued monitoring explicitly accounts for inconclusive evidence [14], [15]. Separation logic provides a formal basis for reasoning about mutable state and framing [16]. AURA draws on these concepts but does not implement a Hoare-logic prover, a linear temporal logic (LTL) monitor, or a separation-logic proof system. Instead, it uses mutation allowlists and atomic predicates as practical runtime mechanisms whose adequacy depends on the declared scope and evidence-capture assumptions.

AURA's contribution is therefore deliberately limited in scope. It combines contract evaluation, independent readback, mutation accounting, and auditable uncertainty in a compact Python implementation. Its finite inspection planner examines how probe selection affects modeled inspection costs, while the conservative verifier manages evidence admission and generates completion verdicts. The evaluation examines whether the implementation behaves as specified across the preserved scenarios. However, the small agent cohorts and reference judgments that share task specifications limit the conclusions that can be drawn. In particular, agreement with reference judgments does not establish that the task contract captures every requirement intended by the user.

III. PROBLEM FORMULATION

Consider an agent that executes a sequence of actions $a_1, \ldots, a_T$, transforming an initial persistent state s_0 into a final state s_T while producing a captured trace t. We represent the task contract as $C = (S, O, A, R)$, where S identifies the observable surfaces, O defines the required predicates, A specifies permitted mutations, and R describes the evidence and coverage requirements. This notation summarizes the existing implementation rather than defining a separate executable contract language. Observable surfaces may include SQLite tables, filesystem paths, traces, and modeled external effects.

Let S_m denote the subset of S designated as mutation surfaces under the closed-world frame policy. Comparing the initial and final snapshots identifies persistent changes within this declared scope. The frame condition is satisfied only when every detected change is authorized:

$$\begin{aligned} D &= \mathrm{diff}(s_0, s_T)|_{S_m}; \\ \text{frame_ok} &\Leftrightarrow \forall d \in D,\ A(d). \end{aligned} \tag{1}$$

Here, D represents the set of differences within S_m, and $A(d)$ indicates whether a change d is permitted by the allowlist. Satisfying this condition does not rule out temporary changes that were reversed before the final snapshot. A transient-safety obligation therefore requires a suitable complete journal, as matching initial and final states alone is insufficient.

Each obligation is evaluated as supported, violated, or unknown. Missing facts, insufficient coverage, unavailable readback, and contradictory trusted facts can affect its evidential status. Let X indicate any trusted-fact conflict, F the presence of at least one violated obligation, and U unresolved obligations or missing global frame coverage. For valid verification inputs, the implemented decision follows:

$$V = \begin{cases} \text{UNCERTAIN}, & X; \\ \text{FAIL}, & \neg X \wedge F; \\ \text{UNCERTAIN}, & \neg X \wedge \neg F \wedge U; \\ \text{PASS}, & \text{otherwise}. \end{cases} \tag{2}$$

The order of these decisions is important. A trusted-fact conflict results in UNCERTAIN, even when another predicate appears to be violated. In the absence of a conflict, a confirmed violation takes precedence over unresolved obligations. PASS is issued only when all obligations and required coverage conditions are supported. Malformed inputs are rejected before this procedure and do not produce an ordinary FAIL certificate. Thus, verified completion indicates that the declared requirements are satisfied under the admitted evidence, defined scope, and explicit trust assumptions. It does not establish universal correctness of the agent.

IV. PROPOSED AURA FRAMEWORK

A. System Architecture

The execution environment and verifier serve distinct roles. The agent performs task actions within an instrumented SQLite/filesystem environment, while the capture code records snapshots before and after execution, associated with a named epoch. Registered probes then collect structured observations. Rather than relying on the agent's completion statement, the verifier evaluates the task contract, probe catalogue, observations, and snapshot pair. Fig. 1 illustrates these relationships. The evidence ledger and certificate preserve identifiers for later inspection, while the external validation package separately computes reference judgments and replay comparisons.

Here, "residue" refers to persistent state differences remaining at the final snapshot, assessed against the declared mutation permissions. The task contract is defined before evaluation. The implementation does not provide a validated, general-purpose natural-language-to-contract compiler. Developers must explicitly specify the required facts, permissible changes, and observation boundaries. This makes the contract reviewable but also introduces a specification burden: stronger verification cannot recover requirements omitted from the contract. The framework's independence applies to its observation and evaluation paths, not to the human assumptions embedded in the task specification.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 10 , October - 2026

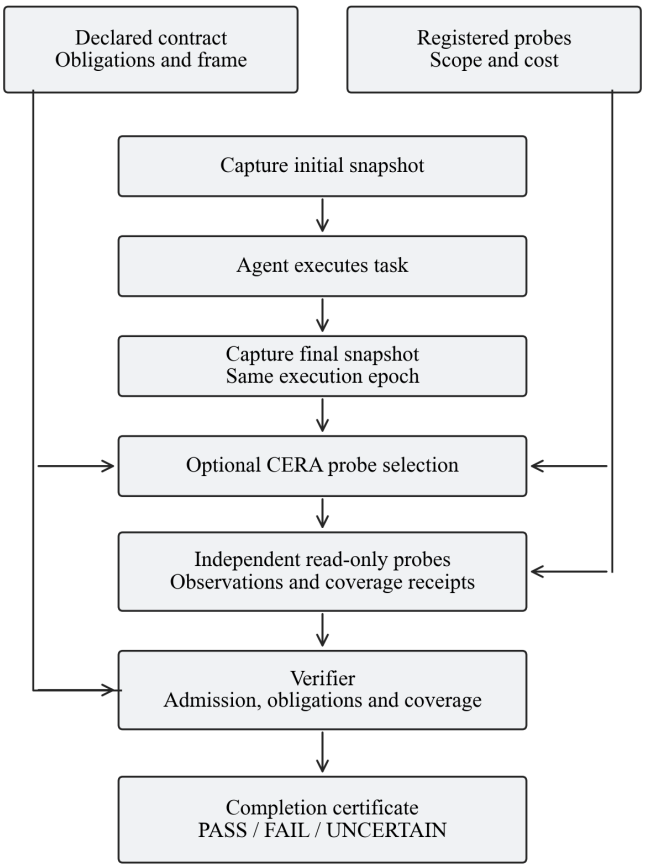

Fig. 1. AURA evidence flow. Optional CERA selects inspections; registered observations feed the verifier, which alone issues the certificate.

B. Contract and Mutation Checks

The contract schema distinguishes four types of obligations: outcome, frame, process, and external. Atomic predicates support typed equality, inequality, exact counts, Boolean tests, and comparisons between facts. Type-sensitive evaluation prevents Boolean values from being treated as integers simply because the host language permits such comparisons. Each obligation also specifies the complete coverage it requires. Under the closed-world policy, contracts must include zero-unaccounted-mutation checks for every declared mutation surface. The engine rejects contracts that lack this required accounting structure.

Snapshot differencing identifies schema changes, row insertions and deletions, modified fields, and filesystem changes. Primary keys provide row identity where available, while tables without primary keys are compared as multisets rather than assigned artificial update identities. Adapter-specific allowlists determine whether each observed change is permitted. Consequently, checking only the intended cell is insufficient for PASS if another declared surface lacks coverage. Process checks, such as requiring validation before an update, rely on the recorded trace and its trust assumptions rather than the agent's narrative explanation.

As an implementation example, the reference task updates student 101's marks to 85 and creates the report reports/student_101.md. The student_state probe provides student.101.marks and student.101.matching_count, which must equal 85 and one, respectively. Report observations confirm that the file exists and its contents agree with the database, while trace.validation_before_update must be true. Snapshot differencing must also confirm zero unauthorized mutations, supported by complete coverage receipts for the students table and reports tree. Updating student 102, modifying an unrelated name, or creating a debug file violates these restrictions. If authoritative student readback is unavailable and no consistent violation is established, the missing required facts result in UNCERTAIN rather than PASS. This example illustrates the implemented reference contract and does not represent an additional experiment.

C. Evidence Admission and Targeted Inspection

Observations are validated against registered probe identities, run identifiers, epochs, payload digests, and supported fact keys. Only observations designated as independent_verifier contribute authoritative facts to the core verifier. Complete coverage receipts must match registered methods and surfaces. Failed readbacks are recorded as unavailable-fact information rather than assigned invented values. Conflicting authoritative observations prevent a consistent interpretation and are retained as explicit uncertainty. These checks protect the evidence interface within the capture model. However, the hashes provide integrity checks, not externally authenticated signatures.

The optional CERA planner selects registered verification probes based on finite candidate worlds, inspection costs, and prior weights. For each probe, it sums the products of prior weights for successful and failing world pairs that the probe can distinguish. This sum is then divided by the squared remaining probability mass and the probe cost. When both outcomes remain possible, the score includes a small coverage term, calculated as 0.00001 times the number of newly covered surfaces divided by cost; otherwise, the full coverage term is used. This approach focuses on distinguishing counterexamples rather than maximizing general entropy. The implementation also supports fixed, greedy, exhaustive, and exact small-instance policies. CERA does not execute the agent's task or determine the authoritative certificate verdict. Its finite models and assigned costs do not guarantee better performance than greedy inspection. Required coverage must still be satisfied, even when all remaining modeled worlds meet the outcome predicates.

In a practical integration, the completion signal can be withheld until a verification certificate is available. A FAIL verdict indicates an established violation that may require repair, while UNCERTAIN indicates missing or conflicting evidence that may require further inspection. The current implementation reports these distinctions, but automated remediation and escalation policies remain outside the scope of the evaluated work.

D. Verdicts and Certificates

Fig. 2 and Algorithm 1 illustrate the verifier's decision process. Certificates record the contract digest, epoch, verdict, reasons, obligation results, observation references, inspected probes, verified frame surfaces, unresolved obligations, costs, and trust assumptions. Together, these fields distinguish established violations from unobserved requirements and clarify the scope within which a PASS verdict applies.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 10 , October - 2026

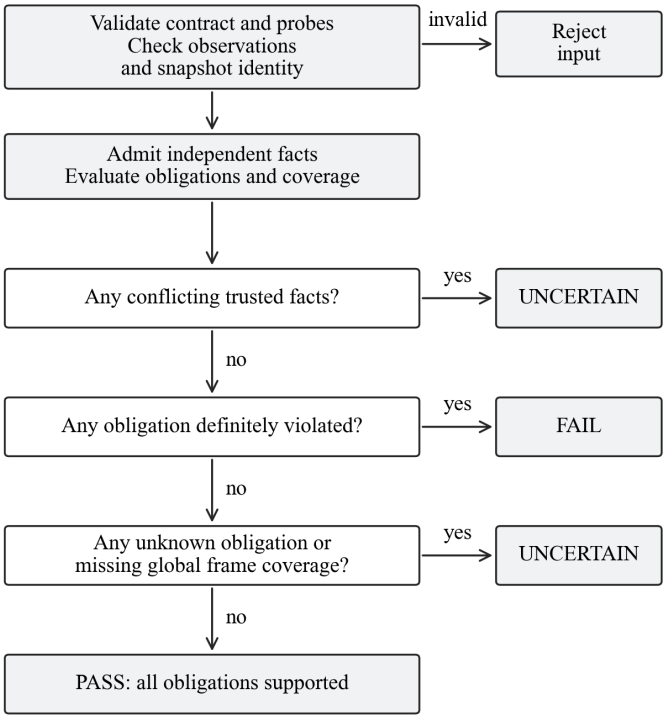

Fig. 2. Core verifier precedence for valid inputs. A trusted-fact conflict overrides apparent violations; malformed inputs are rejected before verdict generation.

ALGORITHM 1. CONSERVATIVE COMPLETION VERIFICATION

1 Validate inputs; reject malformed evidence.
2 Admit registered, same-epoch independent facts.
3 Record conflicts, unavailable facts and coverage.
4 Evaluate obligations and global frame coverage.
5 Apply conflict → violation → unknown → PASS order.
6 Emit certificate with reasons and evidence references.

E. Implementation and Replay Boundary

The primary modules are aura_evidence, aura_verifier, aura_planning, and aura_validation [6]. Python coordinates JSON-schema validation, SQLite snapshots, probes, and certificate generation. Although the evaluation includes preserved Qwen2.5 executions [17], the verifier itself does not depend on a language model. Replaying saved evidence recalculates the verdict and selected certificate fields without repeating model generation. The current replay helper compares verdicts, obligations, and reasons. However, its surface comparison uses frame_coverage.declared_surfaces rather than the emitted verified_frame_surfaces field. Consequently, the aggregate replay flag does not establish full certificate equality, which requires more complete comparisons.

V. EXPERIMENTAL METHODOLOGY

A. Questions and Cohorts

The evaluation examines three questions: whether the verifier distinguishes specified task completion from injected violations, how unavailable evidence influences verdicts, and whether preserved evidence supports consistent reevaluation. Table I summarizes the four cohorts in the 59-record validation registry. The local scenarios test actual SQLite and filesystem mutations under scripted control. The 36 controlled SQL cases cover six tasks, each with six variants: oracle completion, no operation, a falsely successful no operation, partial completion, an unauthorized additional change, and denied readback. These cases represent controlled executions rather than language-model rollouts.

The two genuine-agent cohorts each contain five Qwen2.5 episodes. The Qwen2.5-0.5B-Instruct run used a local central processing unit (CPU) with a four-turn budget, while the Qwen2.5-3B-Instruct pilot used a Kaggle T4 graphics processing unit (GPU) with a six-turn budget. The task sets also differed, so these runs provide separate feasibility observations rather than a controlled comparison of model sizes. Four earlier Stage 8.1 Qwen runs, three native AppWorld evaluations without AURA certificates, and official AgentSynth calibration runs are excluded from this count. Planner-world evaluations belong to a separate study and are not counted as additional agent episodes. The preserved Qwen task identifiers cover clearance discounts, deduplication, ticket-priority changes, refunds and restocking, and cancellation or payment operations; the two cohorts are not task matched [6].

B. Comparison Conditions and Measures

The scoring pipeline distinguishes official command-line interface (CLI) results, reproduced checker scores, reference judgments, and AURA certificates. Official CLI calibration results are not assigned to genuine-agent episodes where calibration was not performed. The reproduced checker score is the arithmetic mean, across episodes, of the fraction of included SQL checks satisfied in each episode. Because partial scores contribute to this mean, it is not the proportion of fully completed tasks, a certificate PASS rate, or an estimate of verification accuracy. For the local cohort, the corresponding field is derived from expected PASS labels rather than measured SQL outcomes. It is therefore excluded as an SQL measurement from the paper's results table.

Reference concordance is the number of matching AURA and supplied reference verdicts divided by the cohort size. The certificate PASS rate is the number of PASS verdicts divided by the same cohort size. These two denominators count episodes, whereas the reproduced checker-score mean averages within-episode check fractions. False-PASS and false-FAIL counts indicate only disagreements with the supplied reference labels. For the local and controlled SQL cohorts, these labels are based on harness expectations. The Qwen reference scorer runs separately but uses shared specifications and allowlists. Therefore, the 59-record registry does not represent 59 independently designed correctness judgments. The refusal scorer also relies on task-specific pattern heuristics rather than a general semantic oracle.

C. Finite-World Planning Study

The separate CERA study uses 48 authored synthetic task templates: 12 cases each for records, orders, access, and files. Each template defines four or five Boolean facts, a closed-world mu-

TABLE I. Evaluation cohorts

Cohort	n	Execution and reference
Local	13	Scripted SQLite/files; harness labels
SQL controls	36	Six tasks × six variants; harness labels
Qwen 0.5B	5	CPU, 4 turns; shared-specification scorer
Qwen 3B	5	T4 GPU, 6 turns; shared-specification scorer

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 10 , October - 2026

tation surface, a finite set of candidate worlds, and individual, joint, and bulk probes with assigned positive costs. All policies receive the same contract, candidate worlds, probe catalogue, and prior. A simulation oracle reveals the tested world's probe outcome only after a policy selects that probe. Fixed inspection follows catalogue order; greedy inspection uses a fact-and-coverage-gain-per-cost score; CERA prioritizes the separation of successful and failing worlds weighted by prior probabilities; and exhaustive inspection reads all probes. The exact small-instance policy first maximizes the probability of a decisive verdict, then minimizes expected additional probe cost. For each template, a policy's cost is the prior-weighted sum of its selected probe costs across candidate worlds. Table III reports the mean of these expected costs over 48 templates per prior. These are synthetic planning comparisons, not additional agent episodes or measured end-to-end latency [6].

D. Reproduction Procedure

The reported manuscript audit examined an isolated checkout at the pinned commit, reviewed the source code and episode records, recomputed cohort counts from external_episode_metrics.csv, and ran the validation pipeline with outputs stored outside the checkout. It also executed the documented test command and checked the manifests. No model inference was repeated. The validator reproduced the aggregate verdict totals, but the reported 59 matches combine 46 re-evaluations of saved episode evidence with 13 regenerated local scenarios (46 + 13 = 59). For the local scenarios, matching compares expected and actual verdicts and includes preset auxiliary comparison flags; those flags do not constitute independently recomputed certificate-field matches. These procedures are reported separately to avoid overstating replay coverage.

The documented test command produced 198 passes and eight failures, all associated with the missing _deps/agentsynth/packs/core_v2.yaml dependency. After supplying the provenance-pinned pack through an in-memory adapter-path override, an expanded test selection, including the Stage 8.3 reference tests, passed 220 tests. This result is distinct from the historical receipt of 224 passes and one skip; neither test count measures research accuracy. Current byte-level checks are also distinguished from historical receipts. Cohort checker scores reported here are derived from the preserved episode-level CSV, not substituted from separately generated cohort summaries. The accompanying evidence audit documents file paths, commands, dependency versions, test outcomes, and interpretation limits, allowing readers to reproduce the reported tables without treating successful software tests as independent validation of the research claims.

VI. RESULTS AND DISCUSSION

A. Verification Outcomes

Table II presents the preserved certificate distribution, with counts recomputed from the episode CSV. All 59 verdicts agree with their supplied reference labels, comprising 13 PASS, 39 FAIL, and seven UNCERTAIN outcomes. The table's "Ref. match column measures this label concordance and must not be read as an independently established accuracy rate. The controlled SQL cohort accounts for six PASS, 24 FAIL, and six UNCERTAIN verdicts, representing six oracle cases, four groups of six failing cases, and six denied-readback cases. The local cohort contributes three PASS, nine FAIL, and one UNCERTAIN verdict. Within these constructed scenarios, the implementation separates established violations from missing evidence as specified by its contracts; the reference agreement is not an independent accuracy estimate.

One local PASS warrants particular attention: transient_reverted_unobserved modifies an unrelated value and restores it before the final snapshot. Because the local contract does not require transient safety, the PASS verdict is consistent with its declared scope. However, this does not establish that no unauthorized action occurred during execution. This case highlights why an endpoint-based completion certificate must explicitly state its observation boundary. A similar limitation applies to external effects: without additional evidence, missing confirmation cannot establish either successful delivery or definite failure.

B. Checker Scores and Planner Costs

The 36 controlled SQL episodes had a mean reproduced checker score of 0.7500, but only six received PASS certificates (6/36). All six unauthorized_extra controls scored 1.0 on their SQL-only checks yet received FAIL certificates because their additional changes violated the mutation allowlists. The Qwen2.5-0.5B cohort had mean score 0.5833 and one PASS certificate (1/5); the Qwen2.5-3B cohort had mean score 0.7500 and three PASS certificates (3/5). These means average partial-credit checker fractions rather than counting fully completed workflows. Neither the differences between scores and PASS counts nor the 59/59 reference agreement establish verification-accuracy improvements. The small, non-matched agent cohorts and different execution budgets also prevent conclusions about model-size effects.

Table III presents a separate, preserved comparison of inspection planning across 48 templates for each prior regime. The uniform prior assigns equal probabilities to all candidate worlds, while clean_50 and clean_90 assign total success probabilities of 0.50 and 0.90, respectively, with probabilities distributed uniformly within each outcome class. Under the uniform prior, CERA achieves a mean modeled inspection cost of 3.193321, compared with 4.315745 for greedy inspection and 3.170950 for the exact policy. However, under the clean_90 prior, greedy inspection has a lower cost than CERA. These results show that relative performance depends on prior assumptions and the inspection objective, rather than establishing CERA as uniformly superior. The cost units are model inputs and do not represent measured end-to-end database or network latency.

CERA may also require more probe-selection overhead than greedy inspection, so modeled savings in probe costs do not necessarily translate into lower wall-clock latency. The planner results distinguish inspection cost from task success. Since all policies operate on the same declared finite models, a less costly

TABLE II. Certificate outcomes and supplied-reference agreement (UNC.=UNCERTAIN)

Cohort	PASS	FAIL	UNC.	Ref. match
Local	3	9	1	13/13
SQL controls	6	24	6	36/36
Qwen 0.5B	1	4	0	5/5
Qwen 3B	3	2	0	5/5
Total	13	39	7	59/59

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 10 , October - 2026

TABLE III. Mean modeled inspection cost (model units; lower is better; 48 templates per prior)

Policy	Uniform	clean_50	clean_90
Fixed	5.484673	9.870882	13.794310
Greedy	4.315745	6.474958	8.406608
CERA	3.193321	6.323659	9.120482
Exact	3.170950	5.894010	7.760971
Exhaustive	38.169375	38.169375	38.169375

inspection sequence does not indicate a more capable agent. Comparing policies across different priors is important because prioritizing early counterexamples may become less advantageous when successful worlds account for most of the probability mass.

C. Reproduction Findings

The fresh validation run reproduced the aggregate 59 reference-label matches, subject to the replay limitations discussed in Section V. Byte-level integrity checks yielded 220/246 matching Stage 8.3 entries and 128/135 matching historical entries. The remaining counts were 26/246 and 7/135 mismatches, respectively. A diagnostic LF-to-CRLF conversion reproduced expected hashes for all 26 Stage 8.3 mismatches and three of the seven historical mismatches, leaving four historical mismatches unresolved. This identifies a cross-platform source of byte differences without establishing that the original files are byte-identical. The repository remained unchanged during these checks. The recovered task pack resolved the missing test dependency for the audit run without modifying the checkout. These findings highlight the importance of preserving executable audit records alongside numerical summaries.

VII. LIMITATIONS

All seven UNCERTAIN verdicts in this registry arise from constructed unavailable-evidence conditions (six controlled denied-readback cases and one local scenario); neither Qwen cohort produced an UNCERTAIN certificate. The task contracts and reference judgments share underlying assumptions. Reference concordance can reveal implementation disagreements but cannot identify requirements omitted from both. Harness expectations also account for much of the reference set, while the refusal heuristic relies on keyword evidence and contradiction patterns. Its reliability on unrestricted language has not been established. Furthermore, the ten genuine-agent episodes are too few and differ too much in their configurations to support general claims about agent reliability or model scaling.

Verification is limited to admitted evidence and declared observation surfaces. Final-state snapshots cannot fully capture transient actions, external effects, concurrent changes, or attacks involving a compromised capture mechanism. Schema validation and hashes do not establish that evidence originates from a trustworthy external authority. The replay procedure does not repeat stochastic model generation and does not currently compare every certificate field correctly. In addition, some noninterference flags in the validation loaders are assigned by construction rather than established through fresh before-and-after database hash comparisons.

Finally, the fresh checkout contains a missing benchmark dependency and byte-integrity differences compared with preserved receipts. These remain reproducibility limitations, even where line-ending differences explain the observed mismatches. Planner results also depend on finite candidate models, prior probabilities, and assigned inspection costs. Broader deployment or efficiency claims would require larger tasks with independently defined specifications, stronger evidence capture, complete certificate replay comparisons, and experiments conducted under matched conditions.

VIII. CONCLUSION

AURA provides an evidence-gated completion mechanism for multi-step AI agents by combining explicit obligations, permitted persistent mutations, independent observations, and three-valued verdicts. The framework distinguishes between successful execution, satisfaction of declared task conditions, and completion supported by sufficient evidence. Its verification certificates make these judgments inspectable by connecting each verdict to the evaluated obligations, observation references, covered surfaces, and unresolved requirements.

Across the preserved 59-record evaluation, AURA recorded 13 PASS, 39 FAIL, and seven UNCERTAIN verdicts, all agreeing with the supplied harness expectations or specification-sharing reference judgments. A fresh validation run reproduced these totals. However, source inspection identified limitations in the interpretation of replay, noninterference, and reference independence, while fresh checks revealed dependency and artifact-preservation issues. These qualifications are an important part of the findings: claims of verified completion should remain open to examination of the evidence pipeline on which they depend.

Future work should focus on independently authored task specifications, larger and more closely matched agent cohorts, more robust natural-language checks, and trusted observation of transient actions and external effects. Correcting certificate-field comparisons and preserving byte-stable releases would also improve reproducibility. The controlled experiments demonstrate technical feasibility under the evaluated conditions. The optional CERA planner explores potential probe-cost savings that depend on modeled prior probabilities and assigned costs. AURA therefore remains an engineering prototype for distinguishing evidence-supported success, established failure, and unresolved completion within explicitly declared boundaries.

CODE AND DATA AVAILABILITY

The preserved code, episode records, planning summaries, and reproduction scripts are available from the AURA repository at the cited research commit [6], subject to the reproducibility limitations above.

DECLARATION OF AI ASSISTANCE

ChatGPT was used to assist with language refinement, manuscript organization and drafting, literature organization, repository inspection, figure preparation, and formatting. The author remains responsible for verifying all scientific content, experimental results, interpretations, and references.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 10 , October - 2026

REFERENCES

[1] S. Yao et al., "ReAct: Synergizing Reasoning and Acting in Language Models," 2023. https://arxiv.org/abs/2210.03629

[2] T. Schick et al., "Toolformer: Language Models Can Teach Themselves to Use Tools," 2023. https://arxiv.org/abs/2302.04761

[3] C. E. Jimenez et al., "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?," 2023. https://arxiv.org/abs/2310.06770

[4] S. Zhou et al., "WebArena: A Realistic Web Environment for Building Autonomous Agents," 2023. https://arxiv.org/abs/2307.13854

[5] H. Trivedi et al., "AppWorld: A Controllable World of Apps and People for Benchmarking Interactive Coding Agents," 2024. https://arxiv.org/abs/2407.18901

[6] AURA project, "AURA source and evidence release," commit 6d4b71409eba9808d171516383a93eb753576783, 2026. https://github.com/shanmukhdatta/AURA/tree/6d4b71409eba9808d171516383a93eb753576783

[7] X. Liu et al., "AgentBench: Evaluating LLMs as Agents," 2023. https://arxiv.org/abs/2308.03688

[8] T. Xie et al., "OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments," 2024. https://arxiv.org/abs/2404.07972

[9] S. Yao, N. Shinn, P. Razavi, and K. Narasimhan, "tau-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains," 2024. https://arxiv.org/abs/2406.12045

[10] J. Lu et al., "ToolSandbox: A Stateful, Conversational, Interactive Evaluation Benchmark for LLM Tool Use Capabilities," in *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 1160–1183, 2025, doi:10.18653/v1/2025.findings-naacl.65. https://aclanthology.org/2025.findings-naacl.65/

[11] N. Shinn et al., "Reflexion: Language Agents with Verbal Reinforcement Learning," 2023. https://arxiv.org/abs/2303.11366

[12] AgentSynth contributors, "AgentSynth, software repository," accessed Oct. 26, 2026. https://github.com/agentsynth/agentsynth

[13] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969, doi:10.1145/363235.363259. https://doi.org/10.1145/363235.363259

[14] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009, doi:10.1016/j.jlap.2008.08.004. https://doi.org/10.1016/j.jlap.2008.08.004

[15] A. Bauer, M. Leucker, and C. Schallhart, "Runtime verification for LTL and TLTL," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 4, art. 14, 2011. https://doi.org/10.1145/2000799.2000800

[16] J. C. Reynolds, "Separation logic: a logic for shared mutable data structures," in *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2002. https://doi.org/10.1109/LICS.2002.1029817

[17] Qwen Team, "Qwen2.5 Technical Report," 2024; revised 2025. https://arxiv.org/abs/2412.15115

(This work is licensed under a Creative Commons Attribution 4.0 International License.)