JOURNAL OF INFORMATION SCIENCE AND ENGINEERING XX, XXXX-XXXX (2024)
DOI: XX.XXX.XXXX.XX

Tuco: Evidence Discipline for LLM-Based Exploratory Web Testing*

CHORNG-SHIUH KOONG[1], YUAN-CHUN LIN[1,+] AND YAO-TING CHEN[2]
[1]*Department of Computer Science*
National Taichung University of Education
Taichung, 403, Taiwan
E-mail: csko@mail.ntcu.edu.tw; dpes8693@gmail.com
[2]*Department of Electrical and Mechanical Technology*
National Changhua University of Education
Changhua, 500, Taiwan
E-mail: yaoting.chen.academic@gmail.com

Language-model browser agents now operate real web applications, but a report that keeps the symptom and loses the trajectory cannot separate a product defect from the agent's own mis-step. We evaluate an evidence-disciplined exploratory testing workflow derived from the Tuco framework: recorded exploration, state checkpoints, supported expectations, verification replay, and an evidence bundle attested by an auditor that runs no language model. Tuco's original requirement-to-scenario pipeline was not executed. The controlled comparison covered two open-source systems carrying fifteen server-side seeded defects each, in four batches of eight runs. Both arms shared the worker model, browser client, recorder, schedule and budget. The treatment comprised the exploration and verification instructions together with a form-state checkpoint tool. A grading session blind to arm labels confirmed 99.2% of the disciplined arm's 128 graded candidates against 80.0% of the baseline arm's 120, or 88.1% once eleven environment-class candidates are removed. Detection favoured the disciplined arm in three batches and reversed in the fourth, and no seeded defect was confirmed by the disciplined arm alone.

Keywords: large language models; exploratory web testing; browser agents; test oracles; evidence contract; fault injection; seeded defects; LLM graders

1. INTRODUCTION

Exploratory testing of a web application asks a tester to operate the product, decide what each screen ought to have shown, and report what did not match. Language-model agents now do the operating: they drive real browsers over self-hosted open-source applications [1] and over production-grade projects [2], and they return fluent prose about what they did. Two gaps remain at the ends of that chain.

The first is the expectation gap. Oracles derived from code tend to encode the behaviour a program exhibits rather than the behaviour a specification demands [3, 4], and over half of the oracle studies in a recent review issue verdicts with no grounding in a specification artefact [5]. An agent that reports a surprise without stating what it expected, and on what basis, leaves a reader nothing to check the surprise against.

[+]Corresponding author
*Communicated by the editor. This article extends a paper presented at the 22nd Taiwan Conference on Software Engineering (TCSE 2026).

The second is the evidence gap. A report keeps the symptom and loses the trajectory: the reader receives a paragraph asserting that an export failed, without the operations and captures that would show it, and an agent optimizing for completion also walks past anomalies it did not need [6]. A claim no evidence can confirm costs a reviewer as much time as the defect it reports.

Tuco was proposed as an end-to-end answer to both gaps. It compiles supplied requirement documents into acceptance rules and a web operation manual, turns both into Gherkin scenarios whose *Then* clause carries assertion authority, and executes those scenarios in a browser under an evidence discipline. This article evaluates something narrower. What we ran is the evidence-disciplined exploratory testing workflow derived from that framework: an agent operates the product directly, records every operation that might become a finding, snapshots form state around a save, states an expected outcome together with the basis that supports it, replays its own suspected defects, and freezes an evidence bundle whose integrity a version-pinned auditor with no language model checks. The original four-stage pipeline was not executed here.

This article contributes:

C1 an evidence-discipline workflow for browser-based exploratory testing, which prescribes recorded operations, state checkpoints and supported expectations, and replays every candidate finding before it is frozen;

C2 an evidence bundle that binds operation identifiers, recordings, native tool logs and replay records to one another, with a non-LLM integrity audit over that linkage. The audit attests linkage and integrity; it does not decide whether a reported behaviour is a product defect;

C3 a descriptive four-batch comparison over two subject systems, 30 seeded defects and 32 runs. Candidate precision against the seeded set is 99.2% for the disciplined arm over 128 graded candidates against 80.0% for the baseline arm over 120, or 88.1% once eleven environment-class candidates are removed. Detection favours the disciplined arm in three batches and reverses in the fourth, no seeded defect was confirmed by the disciplined arm alone, and the cost comparison is bounded by runs whose cost record is incomplete.

This article extends the Tuco prototype introduced at TCSE, focusing on its principles of reviewable evidence and justified judgments. Building on the conference paper's preliminary demonstration of a requirement-to-acceptance-testing pipeline [7], we study a recorded-exploration workflow derived from those principles. The extension adds a controlled evaluation over two applications, 30 seeded defects and four experimental batches, examining candidate precision, defect detection and execution cost. The new experiments evaluate this derived workflow.

2. RELATED WORK

2.1 LLM Agents for Web and GUI Testing

WebArena introduced a self-hostable environment built from open-source applications for browser agents driven by natural-language instructions [1]. Later work asks

whether such an agent can test: PinATA quantifies automation errors and hallucinated step validations on manual test cases [8], NaviQAte reframes exploration as question answering [9], Temac reaches functionality classic crawlers [10] miss [11], WebTestPilot symbolizes GUI elements so that assertions are generated in a constrained DSL over those symbols rather than by free-form model reasoning [2], and requirement-oriented variants verify requirements on a running mobile application [12] or on a GUI prototype [13]. Every model evaluated under WebTestBench's harness scores below 30% end-to-end F1; most sit near 30% precision against recall under 25%, though the strongest trades precision for recall, on a protocol and denominators that differ from ours [14]. GUITester names goal-oriented masking, an agent suppressing anomalies that did not block completion, and Execution-Bias Attribution, a product defect blamed on the agent's mis-click [6]. These systems vary the agent; we hold the worker, the browser client and the recorder fixed and vary the workflow package the agent follows, so a difference is not attributable to a stronger tester.

2.2 Oracles and Expectation Grounding

The oracle problem predates language models [15]; fine-tuned code models now generate oracles, TOGLL producing 3.8 times more correct assertion oracles than TOGA, the prior state-of-the-art neural method [16]. What they encode is the worry. They capture actual rather than expected behaviour [3]; prompting and supplied context move accuracy more than the choice of model [4]; removing Javadoc @throws clauses changes exception-versus-assertion prediction accuracy by at most 0.54 percentage points on clause-bearing samples, and further ablations reveal reliance on shortcut cues [17]; yet assertions derived from documentation alone still beat a neural generator that reads the code [18], and assertions can be derived from business requirements without the source code [19]. The same weakness recurs in suites with full line and branch coverage but a mutation score of a few percent [20], in agent trajectories where print statements outnumber assertions [21], and on a post-cutoff dataset where LLM and human oracles kill mutants at nearly the same rate [22]. A review of 83 oracle studies finds over half issue verdicts with no specification grounding [5], and behaviour-driven development links natural-language requirements to executable checks [23], though a review of requirements-to-test-case generation still lists traceability and hallucination control as open problems [24]. The workflow studied here works one step upstream of a fixed authority: it requires the agent to name which of three bases supports each expected outcome and to file an unsupported expectation as a gap rather than a defect. One of those bases, a recorded round trip, is a weak form of the partial oracles that state a property relating two executions [25].

2.3 Evidence, Seeded-Defect Benchmarks and LLM Graders

Defect injection predates the language-model era. Bures et al. built a testbed for it, argued that mutation testing might reach its limit for faults that arise from a misread specification, and offered injection as a complement to mutation rather than a replacement [26]; that is the argument for seeding semantic server-side defects, and their caveat applies to ours, since seeded defects cannot be shown to represent those real development produces. Validated collections of reproducible server-side bugs supply the taxonomy such seeds imitate [27], and agent evaluations inject one defect per requirement across

production-grade applications [2], label a large test-item pool against applications whose defects arise from AI generation rather than injection, 1,750 items of which 448 fail [14], or pair interactive tasks with a defect-type catalogue [6]. Fault-injection studies verify that a seeded fault is activatable, so that an injection with no observable effect does not dilute the measured effect [28]. That check concerns the seeding rather than the search: a seeded defect an agent never found still counts in the denominator of fifteen per system. The familiar proxies can mislead here: coverage loses predictive power, and mutation analysis does not apply at all, once the code already contains the faults to be exposed [29], and TestExplora hides every defect-related signal from the agent and repeats its baseline configurations three times [30]. Because our confirmed-defect counts come from a fresh language-model session reading frozen bundles, the grader is part of the apparatus, and reliability work bounds what such counts carry: agreement rates are uncorrected for chance and a grader can be consistent while carrying a systematic bias [31], correlation alone does not validate a grader [32], and under formatting and paraphrase perturbations no evaluated grader is uniformly reliable [33]. We therefore treat the grader as a measuring instrument: Sec. 4 states its protocol and Sec. 7 what that protocol leaves unresolved.

3. THE EVIDENCE-DISCIPLINED EXPLORATORY TESTING WORKFLOW

3.1 Background: The Tuco Pipeline

Tuco was introduced as a prototype pipeline of four stages [7]. A rule extractor normalizes the supplied requirement documents into acceptance rules, each naming a condition and an outcome a browser can observe, and marks rules no browser action can decide so they do not silently become assertions. An explorer agent then operates the deployed product and writes a web operation manual of the controls that exist, which lets later stages name controls without inventing selectors. A scenario generator compiles rules and manual into Gherkin scenarios and records, for each scenario, the rule it came from and the manual entries its *When* steps use. Execution splits into an orchestrator that owns workflow state and executor agents that own the browser: an executor performs the *When* steps and compares what it observed against the *Then* clause, which and not the agent's impression of the page decides what counts as correct behaviour. A failing clause yields a candidate defect report, a passing one a browser test specification.

None of those four stages was executed in the study reported here. Both arms received the systems' published end-user documentation unchanged and operated the applications directly, so neither arm was supplied with an acceptance rule, an operation manual or a Gherkin scenario. What this article evaluates is the exploratory workflow of Fig. 1, which carries over Tuco's evidence discipline and nothing above it.

3.2 Recorded Exploration

Any operation that might become a defect report runs inside a recording window: the agent opens the window, acts, then submits an observation record that closes it. One operation identifier covers everything the window produced, the video, the screenshots and a bounded excerpt of the native browser-tool log. The instruction is to open the window be-

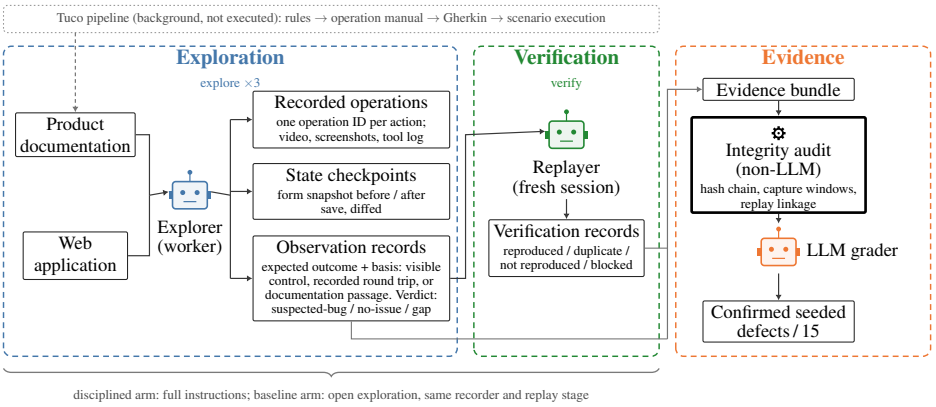

Fig. 1. The evidence-disciplined exploratory testing workflow as it was run. Recorded exploration emits one operation record per recording window; state checkpoints diff form state around a save; supported expectations name the basis for each expected outcome; verification replay re-executes each suspected defect in a fresh browser session; the evidence bundle is then frozen, checked by a non-LLM integrity audit, and handed to the grader

fore acting, submit before moving on, click real controls rather than script shortcuts, and re-read a saved object rather than trust a success message. One recorded operation from the wiki's export module shows the shape: the agent opened a window, exported a page as Markdown, re-read the downloaded bytes, and submitted a record naming the export controls it had used and what it expected them to produce. The recorder enforces unique identifiers, no new window while the previous observation is uncommitted, no capture the budget cannot finish, and schema-valid observations.

3.3 State Checkpoints

Before editing an existing object the agent snapshots the form's visible input, textarea and select controls, excluding password fields; after saving and reopening it takes a second snapshot and diffs the two by field key against the field it meant to change, flagging unedited fields that changed, fields dropped, and values incompatible with their control. The tool issues no verdict. It turns a claim about a save into a before-and-after pair.

3.4 Supported Expectations

Every record states an expected outcome and the basis that supports it, and the instruction fixes the admissible bases as one of three: a visible control, a recorded round trip, or a specific passage of the supplied documentation. An expectation none of the three supports is filed as a gap, which stays in the run's records and never becomes a candidate defect. The agent itself judges whether a basis holds, so this is a reporting discipline rather than a mechanical gate; the instructions return repeatedly to documented semantics, but documentation is not the only admissible basis and surprise alone does not license a report.

3.5 Verification Replay

Verification replay is a stage both arms run: before anything is frozen, the agent re-executes each of its own suspected defects in a fresh browser session and records whether the behaviour reproduced, whether it repeats an earlier finding of its own, or whether it could not be re-run. Only what the agent reproduced there becomes a candidate. The disciplined arm's replay text is longer than the baseline's and differs in two substantive ways: it makes the state-checkpoint tool available during replay, and it imposes stricter conditions for marking a record reproduced or a duplicate of another.

3.6 Evidence Bundle and Integrity Audit

At freeze a version-pinned auditor with no language model applies an evidence contract to each candidate. Under the standard procedure it admits a candidate only if the raw recording exists, the operation is replayable, the record falls inside a native capture window that pairs with the recording, and the record is complete and submitted, with the run's audit files hash-chained to each other and to the program that wrote them. State checkpoints lie outside the gate; the auditor never reads them.

What the gate demonstrably does is attest rather than filter. Under the standard procedure it binds a graded candidate to an intact, hash-chained, replayable window; it does not check whether an expectation is faithful to a requirement, and it does not decide whether a reported behaviour is a product defect, and its effect on the comparison was not isolated. Some runs reached grading under documented exceptions instead of the standard procedure (Sec. 7).

3.7 Grading

A frozen bundle holds one run's records, captured evidence, verification conclusions and the documentation the run was given, and is written for a triager deciding whether a candidate deserves a developer's time, a developer reproducing it, and an auditor checking the hash chain. Bundles are dominated by video; the eight bundles of one batch occupy 0.5 to 0.8 GB. A fresh grading session evaluates the shuffled run bundles of a batch using the corresponding system answer keys. It labels a candidate a hit when it names exactly one seeded defect and cites an evidence file in the bundle that shows the failure, a duplicate when it repeats a defect already credited in that bundle, and a miss when neither holds; maybe and ineligible are available for a candidate it cannot place. A run's score is the number of distinct seeded defects confirmed out of the fifteen seeded in that system. Candidate defect reports are inputs for human review, not claims of final correctness.

4. EXPERIMENTAL DESIGN

4.1 Research Questions

RQ1 (detection). Does evidence discipline change the seeded defects a grader confirms per run and the distinct defects an arm covers, and is the direction the same in every batch?

RQ2 (candidate precision). Among candidates that pass the evidence contract and reach grading, does evidence discipline change the share the grader confirms as pointing to a seeded defect?

RQ3 (cost). Under the same wall-clock budget, what does the disciplined arm cost in money and elapsed time relative to the baseline arm?

4.2 The Two Arms

The two arms are the disciplined arm, TUCO, and the baseline arm, BARE. Both use the same worker, grok-4.6, and the same browser command-line client with its embedded Chromium build. Both log through the same mechanical recorder, which stamps one operation identifier per action and writes the transcript the host audits. Both follow the same phase schedule, three exploration phases then verification replay, and both are told that a finding counts only if it is written as a record and submitted inside the recording window. BARE is therefore a direct-agent baseline, not an unconstrained model.

The disciplined arm adds the evidence-discipline requirements of Sec. 3: recorded operations, state checkpoints and supported expectations. Counted as delivered to the agent, the exploration prompt body is about 1,400 words for TUCO against about 440 for BARE, whose text is the shared recording instructions plus 115 words of open-ended guidance and which chooses its own order, inputs and tactics. Verification replay has the same form in both arms, but its instruction text is also part of the treatment: about 1,360 delivered words against about 530, and it makes the state-checkpoint tool available and tightens the rules for marking a candidate reproduced or duplicate. Both arms shared the worker model, browser client, recorder, schedule and budget. The treatment comprised the exploration and verification instructions together with a form-state checkpoint tool. The state-checkpoint tool produced 55 form diffs over the disciplined arm's sixteen runs; two reported a change in a field the operation had not targeted, four reported a field added after the save, and no diff was itself scored.

4.3 Subject Systems and Seeded Defects

The two subject systems are BookStack, a documentation wiki, and PrestaShop, an e-commerce storefront with a back office. Both are open source, publish end-user documentation that we supply unchanged as the product documentation, and are the subject systems of WebTestPilot's benchmark [2]; only the subjects are shared, since the versions and the seeded defects differ. Excluding tests and vendored libraries, BookStack 26.05.4 has about 70 thousand lines of first-party code and PrestaShop 9.1.5 about 524 thousand, spanning a small-team product and a production platform.

Each system carries fifteen server-side seeded defects, injected in application code behind the HTTP interface, in the tradition of controlled defect injection and of validated collections of reproducible server-side bugs [26, 27]. Both arms observe a defect only through the interface a user sees. The authors designed the defects, using a large language model as a brainstorming aid: the affected modules come from each system's official documentation, and the fifteen per system fall in three intended difficulty tiers of five. Each seeded defect has been verified retrospectively to be observable through the user interface by comparing a clean deployment with the seeded deployment; the recordings are part of the supplementary package. Why eight of the thirty were confirmed by no

Table 1. Seeded defects by subject system, with the number a grader confirmed in at least one run of that arm over all four batches, and the number no run of either arm confirmed

Subject system	Seeded	TUCO ever	BARE ever	Neither
BookStack (documentation wiki)	15	11	12	3
PrestaShop (storefront, back office)	15	9	10	5
Total	30	20	22	8

run of either arm was not determined case by case, and a defect can go unconfirmed at exploration, at the agent's own judgment, at replay or at grading. The answer key went only to the grader, and defect identifiers are withheld until publication so the benchmark can be reused. Table 1 summarizes the thirty defects by system.

4.4 Batches

One batch is a complete matrix of two subject systems, two arms and two replicates, eight runs; four batches gave thirty-two runs in all. Batches 1 and 2 ran in the laboratory harness, Batches 3 and 4 in the packaged benchmark, which pins the subject system images and tool versions for third-party re-execution. A batch is one matrix, not a sample of matrices, so we report the four side by side and never pool arm totals. The pooled candidate ratio in Sec. 5.2 is a different object, a count over all 32 runs, reported beside its per-batch rows.

4.5 Procedure

Fig. 2 shows one batch's procedure. Each run is registered before it starts, pinning the plan, the budget and every input the agent may read, so no instruction or document changes during a batch. The eight runs then execute in two groups of four on one host. Each group holds two disciplined and two baseline runs, one of each arm per subject system, and every run gets its own system instance, restored from the frozen snapshot before it starts, so no two concurrent runs share a deployment. Each run receives three exploration phases of at most 1,080 seconds each and a verification replay of at most 1,500 seconds, with a conditional second verification phase and a hard cap of 7,200 seconds of wall clock. At freeze the auditor applies the evidence contract (Sec. 3.6). The frozen output becomes one evidence bundle per run, relabelled with shuffled identifiers before grading, so no label reveals its arm.

4.6 Measures

The arm total out of 60 sums an arm's four runs in a batch, so a defect confirmed in both replicates counts twice. Union coverage counts distinct seeded defects an arm confirmed anywhere in a batch, fifteen per system and thirty per batch, removing that double counting.

Candidate precision against the seeded set is the share of graded candidates the grader confirmed as pointing to a seeded defect, counting hit and duplicate labels; a duplicate is a second correct report of a defect already credited. We also report the hit-only

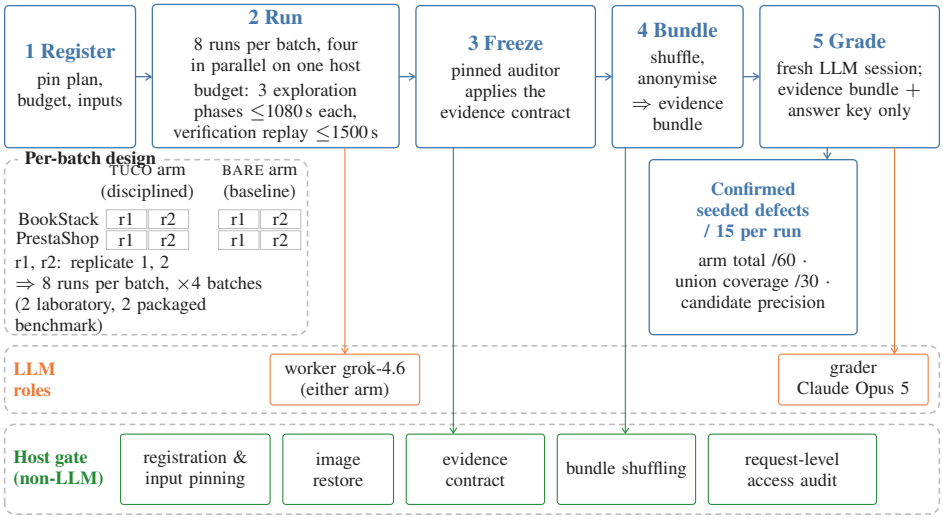

Fig. 2. Procedure for one batch: register, run, freeze under the evidence contract, bundle under shuffled identifiers, and grade the batch's bundles in one fresh session

variant, and separately the self-reported duplicates, records the agent marked as duplicates during replay, which never enter the candidate pool; because the disciplined arm's replay instructions define a duplicate more strictly, this count is not directly comparable across arms. Candidate precision is a ratio over what an arm submitted, so it has to be read together with the number of graded candidates and with union coverage, never on its own. Cost is the amount in US dollars the worker client reports for one run, with that run's wall clock.

Each cell holds two runs, so we report direction and magnitude and run no inferential statistics.

4.7 Grading Protocol

A fresh Claude Opus 5 grading session evaluated the shuffled run bundles of a batch using the corresponding system answer keys. The grader saw the bundles and the answer keys and nothing else: no method label, cost record or instruction text. It labels each candidate admitted to grading hit, maybe, miss, duplicate or ineligible, admitting a hit only if it names one seeded defect and cites an evidence file inside the bundle that shows the failure. Because grader reliability varies with task and prompt [31, 32, 33], we duplicate the instrument. The reported results use one grading round for Batch 1 and two independent grading rounds for each of Batches 2 to 4, whose per-bundle scores and confirmed-defect sets were identical in each.

4.8 Environment

Table 2 lists the models, versions and host configuration.

Table 2. Experimental setup

Item	Value
Worker model	grok-4.6, via a forked command-line client, version 1.0.12
Grader model	Claude Opus 5
Model access	All four batches ran in September 2026; neither hosted model is pinnable by snapshot, date or sampling settings
Browser tooling	Playwright command-line agent 0.1.18 with embedded Chromium build 1237, the browser both arms drove. Identical builds across arms are documented for Batch 3; elsewhere they follow from the arms sharing one runner image
Subject systems	BookStack 26.05.4; PrestaShop 9.1.5
Container runtime	Docker 29.8.0
Host	Intel Core i7-13700K, 24 threads; 62 GiB memory; Ubuntu 24.04.4 LTS, kernel 7.0.0

5. RESULTS

5.1 RQ1: Detection

Table 3 gives every cell with the arm totals and union coverage. Three of the four batches favour TUCO: the arm total is 10 to 11 points higher out of 60 and union coverage 4 to 5 defects higher out of 30. The fourth reverses on both measures.

Pairing each disciplined run with the baseline run of the same batch, system and replicate gives 16 paired cells; 12 favour TUCO and none ties. The four exceptions are BookStack and PrestaShop replicate 1 in Batch 2, where BARE leads by one point each, and both BookStack replicates in Batch 4, where it leads by two. Batch 4's difference in arm totals therefore sits entirely in BookStack, and the PrestaShop pairs of that batch still favour TUCO. Union coverage is the measure that reverses in both subject systems that batch, 11 distinct defects against 8 in BookStack and 9 against 8 in PrestaShop. The per-run scores overlap, BARE from 1 to 9 confirmed defects out of 15 and TUCO from 5 to 10, and BARE's wider spread is what produces its broader Batch 4 union coverage.

5.2 RQ2: Precision and the Candidate Funnel

Table 4 follows the candidates from record to graded verdict. The arms diverge at every stage, in the same direction in all four batches for records, self-reported duplicates and precision, and in three of the four for graded candidates: in Batch 4 BARE delivered 34 candidates against TUCO's 33. A much smaller share of the baseline's candidates survives grading. The grader confirmed 80.0% of BARE's graded candidates as pointing to a seeded defect, against 99.2% of TUCO's; under the hit-only variant the figures are 78.3% and 96.9%, so the gap does not depend on how duplicates are counted. No candidate in any run was labelled maybe or ineligible. BARE's precision rises over the four batches,

Table 3. Confirmed seeded defects per run (out of 15), by batch, arm, subject system and replicate, with per-batch arm totals (out of 60) and union coverage (out of 30)

		BookStack		PrestaShop			
Batch	Arm	r1	r2	r1	r2	Total	Union
1	TUCO	7	8	8	8	31	18
1	BARE	6	6	5	4	21	13
2	TUCO	6	10	5	8	29	18
2	BARE	7	5	6	1	19	13
3	TUCO	10	9	8	8	35	19
3	BARE	6	8	5	5	24	15
4	TUCO	7	7	7	8	29	16
4	BARE	9	9	6	6	30	20

Table 4. Candidate funnel by batch and arm. Precision is (hit + duplicate) / graded candidates; the corrected column removes the environment-class misses from the denominator. No candidate was labelled maybe or ineligible

			Self-				Miss		Prec. (%)	
Batch	Arm	Rec.	dup.	Graded	Hit	Dup.	Env.	Other	Raw	Corr.
1	TUCO	95	0	31	31	0	0	0	100.0	100.0
1	BARE	146	10	28	21	1	2	4	78.6	84.6
2	TUCO	103	0	29	29	0	0	0	100.0	100.0
2	BARE	122	9	25	19	0	3	3	76.0	86.4
3	TUCO	120	0	35	35	0	0	0	100.0	100.0
3	BARE	163	15	33	24	0	4	5	72.7	82.8
4	TUCO	133	3	33	29	3	0	1	97.0	97.0
4	BARE	163	12	34	30	1	2	1	91.2	96.9
All	TUCO	451	3	128	124	3	0	1	99.2	99.2
All	BARE	594	46	120	94	2	11	13	80.0	88.1

peaking in Batch 4, its best batch on both measures.

The host gate is not what narrows the funnel. The step from records to graded candidates comes from the agent's own verification replay, which admits only what it reproduced, and from its self-reported duplicates; the contract's own effect on the candidate set was not isolated. Those self-reported duplicates, 46 for BARE against 3 for TUCO, are the agent's own judgment under arm-specific replay instructions, not an independent measurement of redundancy, and the two counts are not directly comparable. What the contract fixes is what the grader may see: under the standard procedure a candidate reaches grading only with its recording window, operation identifiers and transcript attached. In every batch some runs reached grading through documented exceptions rather than under the standard procedure (Sec. 7).

Eleven of BARE's 24 misses report an environment defect rather than a seeded one. On the PrestaShop storefront, category covers, thumbnails and listing images return HTTP 404 and render at zero width, a property of the container image's URL rewriting; BARE

reported this nine times across the four batches and twice more reported an asset host the browser could not resolve. TUCO reported neither, although it ran the same broken configuration. Removing all eleven from the denominator raises BARE's pooled precision from 80.0% to 88.1%, 96 of 109, or to 86.5% if only the nine image reports are removed, and its Batch 4 figure to 31 of 32 against TUCO's 32 of 33, which closes the gap in that batch. Of the remaining 13 baseline misses, one made a claim the bundle's own evidence undercut, five read expected behaviour as a failure, and seven lie outside the seeded set for other reasons, mostly pre-existing demonstration data. The one disciplined miss also read expected behaviour as a failure. A miss means the candidate did not match the answer key for that system; a report about the deployment or about non-seeded demonstration data is a miss by this measure without being a fabricated report.

5.3 Anatomy of the Batch 4 Reversal

In Batch 4 BARE alone confirmed four BookStack defects. TUCO had confirmed three of them in earlier batches: whole-book export returning HTTP 404 in every format but one, in Batch 3; import of a previously exported archive reporting success while flattening the chapter structure, in Batch 2; and download of an existing attachment returning the right filename with a zero-byte body, in Batches 1 to 3. The fourth, every save in the editor demoting the current heading one level, TUCO confirmed in no batch.

The attachment download is the case worth stating precisely. In Batch 4 a TUCO run observed a zero-byte attachment download but classified it as no issue because the original file size and contents were not visible through the interface. This case illustrates uncertainty about the expected outcome; it does not isolate a causal effect of any reporting rule. The interface shows neither the stored size nor the stored contents of an attachment, so an empty body is consistent both with a serving failure and with stored content that is empty, and the run's own record says so.

The two arms also allocated their exploration differently. Across the two BookStack runs of that batch TUCO spent 22 records on the search module against 12 for BARE, while the four defects lie in export, import, attachments and the editor. The prescribed workflow is slower per operation, so unequal allocation is a possible explanation for the three remaining losses; the data do not establish that time per operation caused them, nor that judgment played no part.

Over all four batches, 20 of the 30 seeded defects were confirmed at least once by both arms and 8 by neither. Of the 22 defects confirmed by at least one arm, 2 were confirmed only by BARE, both of them in Batch 4, and none only by TUCO. Five of the eight defects that no run confirmed are in PrestaShop and three in BookStack. As a set description, evidence discipline opened up no part of the defect set that the baseline could not reach.

5.4 RQ3: Cost

Table 5 reports what one run costs: one subject system explored and replayed once, about 70 to 80 minutes of wall clock and between about five and twenty-three US dollars depending on the cell. TUCO is the more expensive arm per run in every batch on the figures the client recorded. Mean wall clock per run differs between the arms by under five minutes in Batches 2 to 4, and in Batch 1 the baseline ran about ten minutes longer,

Table 5. Cost per run in US dollars as reported by the worker client, with the number of runs in the cell whose cost record is complete, cost per confirmed seeded defect, mean wall clock per run in minutes, and the batch total as a secondary figure

		USD per run		Cost	USD per	Min.	Batch
Batch	Arm	Mean	Min–max	rec.	defect	per run	USD
1	TUCO	10.32[a]	7.41–12.47	2/4	1.33[a]	72	41.30[a]
1	BARE	9.48[a]	7.35–10.86	2/4	1.81[a]	82	37.90[a]
2	TUCO	12.87[a]	10.13–15.95	2/4	1.77[a]	75	51.47[a]
2	BARE	9.06[a]	4.85–17.73	3/4	1.91[a]	74	36.24[a]
3	TUCO	≥14.04[b]	8.97–≥19.73	3/4	≥1.60[b]	75	≥56.16[b]
3	BARE	8.73	7.23–10.24	4/4	1.45	70	34.91
4	TUCO	19.68	16.88–22.69	4/4	2.71	70	78.70
4	BARE	13.67	10.76–17.18	4/4	1.82	68	54.69

[a]Recorded spend only: some phases of this cell terminated before the client wrote a final cost line. [b]Lower bound: one run of this cell lost its final cost line.

so the extra money buys checkpoints and re-reads inside the same time budget rather than more time. Per confirmed defect, recorded spend is lower for the disciplined arm in the first two batches, 1.33 against 1.81 dollars and 1.77 against 1.91, and higher in the last two, at least 1.60 against 1.45 and 2.71 against 1.82.

Eight of the 32 runs ended a phase before the client wrote a final cost line, all in Batches 1 to 3, and both arms are affected. Every figure in Table 5 for those cells is therefore recorded spend and a lower bound on actual spend, not a total. The true cost ordering is unknown for Batches 1 and 2; in Batch 3 the disciplined arm's lower bound already exceeds the baseline arm's complete total, so the direction is decided there. The lower bounds must not be read as exact amounts. The figures are also the amounts the worker client reports for itself and are not comparable across batches, because the endpoint's throughput changed between batches and, under a fixed wall-clock budget, a faster endpoint completes more work and so spends more for an unchanged method. We do not report token counts, because the per-batch token totals could not be traced to a primary accounting record.

6. DISCUSSION

Higher precision, inconsistent detection gain. The clearest effect is on what the agent was willing to submit. BARE wrote about a third more records, delivered fewer graded candidates, and a fifth of those pointed at nothing seeded, while almost every disciplined candidate the grader read matched a seeded defect. The detection gain is not consistent: three batches favour TUCO and the fourth reverses, and across all four batches the disciplined arm opened no part of the defect set that the baseline could not reach. That last statement is a description of two sets, not a demonstration that discipline leaves the per-run confirmation rate unchanged, which it did raise in three of the four batches. Candidate precision is measured against the seeded set, so it cannot credit a report that

is right about the product but outside the answer key, and it has to be read beside the graded-candidate counts and union coverage of Sec. 5.

Uncertainty about the expected outcome. The Batch 4 attachment case is not a rule suppressing a finding the agent had established. The agent had established that the downloaded body was empty; what it could not establish from the interface was what the stored file contained, so the expected outcome itself was in doubt. The workflow admits a recorded round trip as a basis, which is the route that would settle such a case: uploading a file of known size and then downloading it makes the expectation checkable without appeal to a document. Making that route routine, rather than leaving it to the agent to think of, is a concrete improvement this study points to.

What this could mean for practice. A possible use of the workflow is a pre-screen in front of a human review queue, in which each candidate arrives with the recording and transcript needed to check it. We did not measure reviewer time, so no saving in review effort is claimed, and whether the dollar premium pays for itself remains open. Recording alone does not produce the cleaner stream, since browser tooling already captures traces and screenshots on request; what the audit adds is attestation of linkage and integrity. The audit's own effect was not isolated, so its contribution to the precision gap cannot be estimated separately.

What the comparison does and does not establish. BARE is a direct-agent baseline sharing the worker model, browser client, recorder, schedule, verification replay and budget. The comparison evaluates the workflow package as a whole; the contributions of its individual components were not isolated. It says nothing about Tuco's original pipeline, which was not executed, not varied and not evaluated here. The conference study provided preliminary evidence for that pipeline; the present evaluation focuses on the derived exploratory workflow.

7. THREATS TO VALIDITY

Two runs per cell. Each cell holds two replicates, and a batch is one matrix, not a sample of matrices. We report the four batches side by side, give direction and magnitude, and run no inferential statistics, so no difference here carries a confidence statement.

Instruction volume and specificity. The disciplined arm's instruction text is several times longer and more specific than the baseline's, so volume and specificity travel with the discipline they implement and this design cannot separate them; a length-matched control carrying no evidence requirements would be needed. The same overlap runs through the criterion: the contract asks for a replayable window and the disciplined text instructs the agent to produce one, which may affect the comparison, and the contribution of that overlap to the precision gap, and to the detection gap that depends on the same hit rule, was not isolated. The comparison still informs, because the documented exceptions retained candidates rather than removing them, and both arms were told that only submitted records count.

One worker model and one grader model. Every result comes from a single worker and a single grader on hosted endpoints that neither we nor the packaged benchmark can pin. Only the model identifier was held fixed, and the worker endpoint's throughput changed between batches, so the direction reported here is a property of this

configuration.

Grading. Candidate judgments were produced by an LLM grader and have not been independently validated by human reviewers. Agreement between grader sessions is not validity: a grader can be consistent and still carry a systematic bias [31, 32], and reliability degrades most under formatting perturbations, to which graders are less robust than to semantic paraphrase [33]. We admitted a hit only when it named one seeded defect and cited bundle evidence, and duplicated the grader in Batches 2 to 4.

Partial blinding. Bundle labels were shuffled and the grader saw no method label, cost record or instruction text, but blinding was partial rather than complete. A disciplined bundle carries state-checkpoint files a baseline bundle lacks, and three baseline bundles in Batch 2 carried a grader-visible annotation recording that their freeze followed the substitute procedure, so a grader could notice that the bundles fall into groups and could attempt to infer which group was treated. Nothing in the design bounds that effect. The annotation is arm-correlated: the three annotated runs had 12 of 17 candidates confirmed, a lower share than the 7 of 8 of the one baseline run in that batch frozen normally, and with different runs behind the two ratios neither figure isolates an effect of the annotation.

Subjects, seeds and one environment defect. Two subject systems and thirty seeded defects bound what generalizes, and seeded defects cannot be shown to represent the faults real development produces [26]. Both systems and the worker's client are public open-source software, so their code and documentation may be in the worker model's training data, a known confound in oracle evaluation [22]; the design did not test whether prior knowledge of these systems interacts with the treatment. The PrestaShop storefront also returned HTTP 404 for category and product images, a property of the container image's URL rewriting rather than a seeded defect, and two further baseline reports concerned an asset host the container could not resolve, which is why eleven baseline misses are counted separately in Sec. 5.2.

Protocol handling. Some runs required documented derivative selection, substitute freezing or adjudication of recording-window and operating-scope exceptions before grading. The supplementary materials retain the original evidence, affected runs and dispositions.

8. CONCLUSION AND FUTURE WORK

Under this configuration, the disciplined workflow package yielded a higher proportion of grader-confirmed candidates matching the seeded defects: 99.2% of 128 graded candidates against 80.0% of 120, or 88.1% once eleven environment-class candidates are removed, and each candidate arrived with the recording and transcript needed to check it. Detection favoured TUCO in three batches and reversed in the fourth; no defect was exclusive to TUCO across all batches. These results concern the recorded-exploration workflow, not the original end-to-end requirement-to-test pipeline.

Future work begins with that pipeline: a further and independent evaluation of the full requirement-to-test pipeline and of requirement traceability, neither of which this study exercised. A length-matched control arm carrying no evidence requirements would separate the discipline from its instruction volume. An ablation of supported expectations would test its contribution to candidate precision and defect detection. Human re-scoring

of a sample of graded bundles would bound how much of the precision gap belongs to the grader; measuring reviewer time per candidate would settle whether the cost premium pays for itself; and subject systems beyond a wiki and a storefront would test whether the direction observed here holds.

AVAILABILITY

A benchmark package exists and has been assembled locally. It contains the pinned subject system images, the four instruction texts of the two arms, the grader prompt, the per-run data behind every table above, and the paired clean and seeded recordings from the interface verification of the thirty seeded defects. Nothing is publicly posted at the time of writing; the package and the seeded-defect identifiers will be released upon publication.

ACKNOWLEDGMENT

We thank the reviewers of the TCSE 2026 conference version, whose comments shaped this extension, and the work received no external funding. Large language models were used in this work as the worker under test, as the grader, and as writing and analysis assistants under the authors' direction; the authors take responsibility for the content of this article.

REFERENCES

1. S. Zhou, F. F. Xu, H. Zhu, X. Zhou, R. Lo, A. Sridhar, X. Cheng, T. Ou, Y. Bisk, D. Fried, U. Alon, and G. Neubig, "WebArena: A realistic web environment for building autonomous agents," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024. [Online]. Available: https://openreview.net/forum?id=oKn9c6ytLx
2. X. Teoh, Y. Lin, D.-M. Nguyen, R. Ren, W. Zhang, and J. S. Dong, "WebTestPilot: Agentic end-to-end web testing against natural language specification by inferring oracles with symbolized GUI elements," *Proceedings of the ACM on Software Engineering*, Vol. 3, no. FSE, 2026, pp. 1933–1956.
3. M. Konstantinou, R. Degiovanni, and M. Papadakis, "Do LLMs generate test oracles that capture the actual or the expected program behaviour?" arXiv:2410.21136, 2024.
4. A. Bodicoat, G. Jahangirova, and V. Terragni, "Understanding LLM-driven test oracle generation," in *Proceedings of the 2nd IEEE/ACM International Conference on AI-Powered Software (AIware)*, 2025, pp. 29–39.
5. A. H. Mughal and M. Bilal, "LLM-based test oracles: Source-of-authority taxonomy—a systematic literature review," arXiv:2607.05031, 2026.
6. Y. Gao, J. Wu, X. Chen, Y. Yang, Z. Cui, T. Ma, J. Zhang, and J. Sang, "GUITester: Enabling GUI agents for exploratory defect discovery," arXiv:2601.04500, 2026.
7. C.-S. Koong, Y.-C. Lin, and Y.-T. Chen, "Tuco: An LLM-driven prototype framework for web acceptance testing and E2E evidence generation," in *Proceedings of the 22nd Taiwan Conference on Software Engineering (TCSE 2026)*, Taiwan, 2026.

8. A. Chevrot, A. Vernotte, J.-R. Falleri, X. Blanc, B. Legeard, and A. Cretin, "Are autonomous web agents good testers?" *Proceedings of the ACM on Software Engineering*, Vol. 2, no. ISSTA, 2025, pp. 206–228.
9. M. Shahbandeh, P. Alian, N. Nashid, and A. Mesbah, "NaviQAte: Functionality-guided web application navigation," arXiv:2409.10741, 2024.
10. A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-based web applications through dynamic analysis of user interface state changes," *ACM Transactions on the Web*, Vol. 6, no. 1, 2012, p. 3.
11. C. Liu, Z. Gu, G. Wu, Y. Zhang, J. Wei, and T. Xie, "Temac: Multi-agent collaboration for automated web GUI testing," arXiv:2506.00520, 2025.
12. Y. Hu, X. Wang, Y. Wang, Y. Zhang, S. Guo, C. Chen, X. Wang, and Y. Zhou, "AUITestAgent: Automatic requirements oriented GUI function testing," arXiv:2407.09018, 2024.
13. K. Kolthoff, F. Kretzer, S. P. Ponzetto, A. Maedche, and C. Bartelt, "GUISpector: An MLLM agent framework for automated verification of natural language requirements in GUI prototypes," arXiv:2510.04791, 2025.
14. F. Kong, J. Zhang, Y. Yue, C. Sun, Y. Tian, S. Feng, X. Yang, D. Wang, Y. Tian, J. Du, W. Zeng, H. Li, and K. Gai, "WebTestBench: Evaluating computer-use agents towards end-to-end automated web testing," arXiv:2603.25226, 2026.
15. E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, Vol. 41, no. 5, 2015, pp. 507–525.
16. S. B. Hossain and M. B. Dwyer, "TOGLL: Correct and strong test oracle generation with LLMs," in *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2025, pp. 1475–1487.
17. S. B. Hossain, M. B. Dwyer, and T. Tasnim, "Documentation vs. code patterns: What drives LLM-based exception oracle generation?" in *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Munich, Germany, 2026.
18. S. M. Khandaker, F. Kifetew, D. Prandi, and A. Susi, "AugmenTest: Enhancing tests with LLM-driven oracles," in *Proceedings of the IEEE Conference on Software Testing, Verification and Validation (ICST)*, Napoli, Italy, 2025, pp. 279–289.
19. T. Ma and N. U. Eisty, "From business requirements to test assertions: Evaluating LLM-generated oracles on real bugs," arXiv:2607.10277, 2026.
20. G. Wang, Q. Xu, L. Briand, and K. Liu, "Mutation-guided unit test generation with a large language model," arXiv:2506.02954, 2025.
21. Z. Chen, Z. Sun, Y. Shi, C. Peng, X. Gu, D. Lo, and L. Jiang, "Rethinking the value of agent-generated tests for LLM-based software engineering agents," arXiv:2602.07900, 2026.
22. D. Molinelli, L. Di Grazia, A. Martin-Lopez, M. D. Ernst, and M. Pezzè, "Do LLMs generate useful test oracles? an empirical study with an unbiased dataset," in *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2025, pp. 278–290.
23. L. P. Binamungu and S. Maro, "Behaviour driven development: A systematic mapping study," *Journal of Systems and Software*, Vol. 203, 2023, p. 111749.

24. O. Folorunsho and H. Reza, "AI-driven test case generation from natural language requirements: A survey of techniques and research gaps," arXiv:2606.06563, 2026.
25. S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on Software Engineering*, Vol. 42, no. 9, 2016, pp. 805–824.
26. M. Bures, P. Herout, and B. S. Ahmed, "Open-source defect injection benchmark testbed for the evaluation of testing," in *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2020, pp. 442–447.
27. P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, Á. Beszédes, R. Ferenc, and A. Mesbah, "BugsJS: A benchmark and taxonomy of JavaScript bugs," *Software Testing, Verification and Reliability*, Vol. 31, no. 4, 2021, p. e1751.
28. G. Tan, Z. Sun, J. Shi, T. Zhang, Z. He, Q. Wu, S. Liang, W. Sun, J. He, P. Chen, C. Zhang, L. K. Shar, and D. Lo, "AgentChaos: Chaos engineering for agent systems via programmatic fault injection," in *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Munich, Germany, 2026.
29. J. Zhao, S. Zhou, and E. Cohen, "Do coverage and mutation scores of LLM-generated test suites correlate with their effectiveness? (replicability study)," *Proceedings of the ACM on Software Engineering*, Vol. 3, no. ISSTA, 2026, p. ISSTA002.
30. S. Liu, J. Luo, X. Zhang, A. Liu, H. Liu, J. Wu, Z. Huang, Y. Huang, Y. Kang, and S. Li, "TestExplora: Benchmarking LLMs for proactive bug discovery via repository-level test generation," arXiv:2602.10471, 2026.
31. J. D. Norman, M. U. Rivera, and D. A. Hughes, "Reliability without validity: A systematic, large-scale evaluation of LLM-as-a-judge models across agreement, consistency, and bias," arXiv:2606.19544, 2026.
32. S. Han, G. Titericz Junior, T. Balough, and W. Zhou, "Judge's verdict: A comprehensive analysis of LLM judge capability through human agreement," arXiv:2510.09738, 2025.
33. S. Dev, A. Sloan, J. Kavner, N. Kong, and M. Sandler, "Judge reliability harness: Stress testing the reliability of LLM judges," arXiv:2603.05399v1, 2026. [Online]. Available: https://arxiv.org/abs/2603.05399v1