TOWARDS EFFECTIVE DATA STRUCTURES FOR AI AGENTS IN COMPLEX REAL WORLD JOBS

Anonymous authors
Paper under double-blind review

ABSTRACT

AI agents generate rich execution trajectories, but standard frameworks typically store experience as prose-heavy message logs, leaving downstream agents to reconstruct implicit interaction semantics. We introduce TRACE (**T**rajectory **R**epresentation for **A**gent **C**ontext and **E**xperience), a write-time protocol that makes interaction structure explicit without introducing additional task requirements or evaluator outcomes. TRACE labels each message with its type and coordination state, links it to the message it answers, tags evaluator feedback by dimension, and derives each evaluation's outcome; the linked messages form a directed graph that preserves response lineage across attempts. On 62 real-world web-development jobs, an agent using TRACE passes **53.2% of jobs versus 16.1%** with plain text on the first attempt, 71.0% versus 48.4% within two attempts, and 74.2% versus 69.4% within the maximum of three attempts. On the 41 jobs completed by both conditions, agents using TRACE need 37.4% fewer attempts on average.

1 INTRODUCTION

AI agents on long-running tasks call tools, receive feedback, revise, and pass or fail, and later work can learn from each step. Minsky argued that memory helps most when it restores the context in which an experience formed, not only its result Minsky (1980). Recent work shows that agents improve when they draw on what they did in earlier tasks Shinn et al. (2023); Zhao et al. (2024).

Most frameworks log these steps as messages, tool calls, and evaluations, and leave implicit how they fit together. Details such as what each message did and which message it answered stay hidden in prose. Prose also hides any available guidance on how reviewers will judge the work, such as the dimensions they score and what each covers (for example, usability covers whether a website works on mobile). Each time an agent returns to a part of the work, it may have to reinterpret these details, which costs computation and can lead it to read them differently than before.

TRACE (**T**rajectory **R**epresentation for **A**gent **C**ontext and **E**xperience) is a structured representation of agent interaction history. It reorganizes what the agent already has: the task, guidance, and feedback stay the same.

A DeepSeek-v4-flash-0731 agent works 62 real-world web-development jobs twice, once with its interaction history in TRACE and once as text. Both runs receive the same task descriptions, guidelines, history, and feedback, and neither sees the evaluator's acceptance criteria. With TRACE, the agent passes 53.2% of jobs on its first attempt, versus 16.1% with text (exact McNemar $p = 1.55 \times 10^{-6}$). Text largely catches up within three attempts (74.2% versus 69.4%, exact McNemar $p = 0.4531$), but on the 41 jobs both complete, TRACE needs 37.4% fewer attempts (Wilcoxon $p = 1.0 \times 10^{-5}$).

Our contributions are:

1. TRACE (Section 3), which labels each message with its type and coordination state and links it to the message it answers.
2. State assignment without a generative model: the sender, the attachment, and a small classifier set the state. Section 5.1 compares this with LLMs on accuracy, stability across repeated runs, and latency.

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

3. A comparison of agents using TRACE and text on 62 real-world jobs with identical information. Section 5.2 measures pass rates at each attempt and the attempts needed to pass, and Section 5.3 ablates TRACE's run-time LLM step and its history.

2 RELATED WORK

Agent communication protocols. KQML Finin et al. (1994) and FIPA-ACL Foundation for Intelligent Physical Agents (2002) made communicative intent explicit through performatives and shared ontologies. MCP Anthropic (2024), A2A Google (2025), and AutoGen Wu et al. (2023) standardize tool invocation, inter-agent messaging, and multi-agent conversation for LLM systems. These protocols specify how messages are exchanged; TRACE represents how an interaction's coordination state, response relations, and dimension-tagged feedback evolve over time.

Agent memory and experience. Reflexion Shinn et al. (2023), ExpeL Zhao et al. (2024), Voyager Wang et al. (2023), and Agent Workflow Memory Wang et al. (2024) reuse past trajectories as reflections, insights, skills, or workflows, within broader memory architectures such as CoALA Sumers et al. (2024) and Generative Agents Park et al. (2023). MemGPT Packer et al. (2023), Mem0 Chhikara et al. (2025), A-MEM Xu et al. (2025), and Zep Rasmussen et al. (2025) add explicit structure through memory tiers, linked notes, and knowledge graphs. These systems derive structure from content, typically through LLM extraction. TRACE instead records protocol-defined structure: coordination state is assigned deterministically, and response lineage is committed with each turn, much as PROV-DM W3C (2013) records how entities were produced.

Revision, feedback, and evaluation. Software and web agents revise artifacts in response to feedback Jimenez et al. (2024); Zhou et al. (2024), usually supplied as natural-language traces or critiques Kim et al. (2023); Yao et al. (2023). TRACE tags feedback with its dimension and links it to the attempt it answers. Its question detector follows dialogue-act classification Stolcke et al. (2000). Benchmarks such as AgentBench Liu et al. (2024) store or score traces in task-specific ways; TRACE offers a shared representation in which state transitions and evaluation dimensions are inspectable directly.

3 TRACE: TRAJECTORY REPRESENTATION FOR AGENT CONTEXT AND EXPERIENCE

As agents work on tasks and evaluators review their work, TRACE labels each message with its type and coordination state, links it to the message it answers, and keeps its original text (Table 1). An agent that reads the same history as text must work out what each message did, which message it answered, and which dimension each comment concerns; TRACE states these facts directly and leaves the original text in place, so anyone can check each label.

Four questions shaped TRACE. First, which states matter while an agent works? A later reader needs to know who sent a message, whether it delivers work, and whether it waits on an answer. The sender and purpose determine the message type, and the other two facts split followups into the four states in Table 2. Second, how should TRACE represent those states? SAWSDL let annotations point to concepts in any ontology (Farrell & Lausen, 2007), so every user had to supply an ontology first. TRACE instead fixes a small, closed vocabulary: four message types, seven states, a parent link, and, on feedback, one of the category's dimensions and the claims the feedback concerns. TRACE writes each record as plain JSON; a JSON-LD context maps the records to an ontology for readers who want one, but nothing depends on it. Third, how can TRACE compute these labels cheaply and the same way every time? It reads type and parent from the sender and thread, reads the deliverable signal from the attachment, and detects the question with a small classifier, so it assigns every state without a generative model. During a job run, feedback pairing is the only TRACE construction step that calls an LLM, and removing it changes little (Section 5.3). Fourth, how can an agent use its own history? Each attempt links to the feedback it answers, and each comment names its dimension and claims, so a later attempt can read what it tried and how reviewers judged each part without rebuilding that picture from prose. The experiments test this last point.

Reviewers judge work against two kinds of standard. Guidelines state what reviewers expect from any job in a category, such as whether a site works on mobile; they distill reviewer feedback on

Table 1: How TRACE produces each field.

Source	Fields
Metadata	Identifier, parent, thread, payload, message type; guidelines on the request, each with its dimension and mandatory flag; per-criterion evaluation result (passed, failed, or skipped)
Derived (rule)	State of request, feedback, and evaluation messages; outcome
Inferred (model)	State of agent messages; pairing of feedback with the agent's response, by dimension

Table 2: Message types and their states.

Message type	State	Meaning
`RequestMessage`	`Submitted`	Initial task
`FollowupMessage`	`InReview`	Deliverable attached, no question
`FollowupMessage`	`Provisional`	Deliverable attached, with a question
`FollowupMessage`	`NeedsClarification`	Question, no deliverable
`FollowupMessage`	`NeedsAttention`	Neither; deferred to a human
`FeedbackMessage`	`ChangeRequested`	Evaluator requests revisions
`EvaluationMessage`	`Evaluated`	Final verdict: `Approved` or `Rejected`

other jobs, and the agent sees them. Each guideline names the dimension reviewers score it under and says whether it is mandatory; an LLM groups the guidelines by dimension once per category, and a human checks the grouping. Acceptance criteria state what one job must satisfy; only the evaluator sees them, and they decide the outcome: `PASS` if every mandatory criterion passes, `FAIL` if any fails, and `PARTIAL` if the evaluator skips some and the rest pass.

At request time, both conditions fetch the same job description from the same job bundle and receive the same nine guideline texts. The text condition lists the guidelines as bullets under the heading "Evaluation quality bar." TRACE instead sends a `RequestMessage` that lists the job's eight evaluation dimensions and links to one node per guideline; each node holds the guideline's text, its dimension, and its mandatory flag. The TRACE prompt also labels each dimension in plain words and explains the node types. The two prompts frame the guidance differently: the text prompt calls the list a quality bar, and the TRACE prompt calls the guideline nodes the graph's rubric criteria. Appendix C reproduces both prompts.

The sender and the message's purpose set its type. The initial task arrives as a `RequestMessage`, and every agent message goes out as a `FollowupMessage`. The evaluator sends a `FeedbackMessage` when it requests revisions and an `EvaluationMessage` when it issues the final result. Table 2 lists the states each type can take. Two facts decide a followup's state: whether it attaches a deliverable, and whether it asks an unresolved question. A followup with neither falls to `NeedsAttention`, which flags the message for a human; the classifier never predicts this state, and the rule assigns it only when both signals are absent. A logistic-regression classifier on frozen all-MiniLM-L6-v2 embeddings detects the question; it scores comparably to LLM baselines and returns the same label on every run (Section 5.1). TRACE reads the attachment from the platform instead of inferring it from text, because the platform already records it and Section 5.1 shows that inference from text loses accuracy and stability.

Both conditions ask the agent to list the claims it makes about each deliverable. After each failed attempt, an LLM reads each failing criterion with the evaluator's reasoning, assigns the criterion one dimension, and links it to the claims it concerns. The agent sees the reasoning, the dimension, and the claims, but never the criterion. TRACE calls a generative model at run time only for this pairing (Appendix E), and the first ablation (Section 4.3) removes it while keeping the states, links, and history.

Each link joins a message to the one it answers, so the messages form a graph that grows with each attempt and holds what the agent has tried and how it was judged.

TRACE also acts during revision: it adds each attempt as a labeled message linked to the feedback on it, with each comment tagged by dimension and tied to the claims it concerns, so the next attempt



Figure 1: One job as each condition presents it at the start of the third attempt: the request's guidelines (top row) and two rounds of evaluator feedback (below). Highlighting marks the completeness guideline and the two auto-bet comments, which TRACE tags with the same dimension. The text panel renders the TRACE run's comments in the text format; both panels trim the comments and shorten IDs and dimension names. Appendix G gives the untrimmed guidelines and messages.

starts from what the agent tried and how each part was judged. The first attempt therefore reflects only the request-time representation; later attempts add the revision history (Section 5.2).

Figure 1 shows one web-development job, a browser dice game modeled on a commercial betting site. After the first attempt, the evaluator flags that auto-betting clamps the stake to a minimum of 0.01, and the second attempt claims a fix (claim a2-c10). The evaluator then finds a new auto-bet defect. TRACE tags each comment with its dimension, the same dimension that types the guideline, links the second comment to claim a2-c10, and links the second attempt to the feedback it answers. The text condition receives the same comments as bullets under each turn, which leaves the agent to work out which comment concerns which part of its work and how the second relates to the first.

4 EXPERIMENTAL SETUP

The experiments ask three questions: how well the classifier detects unresolved questions, whether agents perform better with their history in TRACE than in text when both receive the same information, and which components of TRACE matter.

4.1 QUESTION DETECTION

The labeled data holds 243 agent responses to messages from people building websites with web-development agents. Each response carries a human-assigned state, reviewed before training: `InReview` (100), `Provisional` (113), or `NeedsClarification` (30). `Provisional` and `NeedsClarification` responses ask an unresolved question; `InReview` responses do not. No response in the labeled data falls in `NeedsAttention`, so the evaluation covers the other three states.

The classifier is evaluated with five-fold cross-validation. All responses from one conversation stay in the same fold, so related responses never appear in both training and test data, and the reported metrics pool held-out predictions from all five folds. LLM baselines label the same 243 responses, and all methods are compared on accuracy, precision, recall, and F1.

A second test removes the attachment signal: each method must recover all three states from text alone. Because `Provisional` and `NeedsClarification` both ask a question, telling them apart means inferring from the text whether a deliverable is attached. This test reports per-class and macro F1.

Each method also labels a fixed sample of 100 responses ten times; the flip rate counts responses whose label changes in at least one run. Finally, the classifier's latency is timed per response on a CPU across all 243 responses.

4.2 AGENT PERFORMANCE

A DeepSeek-v4-flash-0731 agent, running in OpenCode (Appendix A), works each of 62 real Upwork web-development jobs (Appendix B) twice: once reading its history in TRACE, once reading it as text. The two runs share the task description, guideline texts, agent model, attempt budget, and evaluator. Each run then accumulates its own submissions and evaluator feedback.

The guidelines tell the agent what reviewers expect from jobs in the same category. To build them before this experiment, agents completed other Upwork jobs in that category, human reviewers judged the results over three rounds of revision, and the recurring feedback was summarized, grouped by dimension with an LLM, and checked by humans. No guideline repeats a job's own acceptance criteria.

In the text condition, the agent reads the guidelines as a bulleted list and the feedback as prose under each turn. In the TRACE condition, it reads the same information as structure: the request lists the guidelines as typed items, each tagged with its dimension and whether it is mandatory; each message carries its type and state and links to the message it answers; and after each attempt, each evaluator comment carries its dimension and the claims it concerns. Figure 1 shows two rounds of feedback in both formats.

GPT-5.6-sol judges both conditions as an agent with tools for inspecting the submitted work (Appendix D). It marks each acceptance criterion as passed, failed, or skipped. It runs through the Responses API, which carries a job's earlier attempts forward, so the evaluator sees what the agent fixed since its last submission. On a separate set of 281 cases, it agreed with human reviewers on every skip and on 65.1% of accept-or-reject decisions.

On each attempt, the agent submits, the evaluator judges and comments, and the agent revises, until the job passes or the agent has tried three times. The results count how many jobs pass, how many pass within one, two, or three attempts, and how many attempts each passing job takes. Paired tests compare the two conditions job by job.

4.3 ABLATIONS

Two ablations each remove one part of TRACE. TRACE without pairing drops the LLM step that ties evaluator comments to the agent's response, and keeps the states, links, and history. TRACE without history shows the agent only its latest attempt and the feedback on it. Both ablations act only during revision, so both run on the 13 jobs TRACE passed after revising.

A third ablation asks whether the first-attempt gain needs TRACE's typed guidelines. TRACE with plain guidelines reruns 27 of the jobs TRACE passed on its first attempt and gives the agent a single attempt on each job. It keeps TRACE's request node and prompt, lists the nine guideline texts as plain text instead of typed nodes, drops the plain-word dimension labels, and calls the list the job's acceptance criteria.

5 RESULTS

5.1 QUESTION DETECTION

These experiments test the classifier on three questions: how accurately it detects unresolved questions, how stable its labels are across runs, and whether TRACE needs the attachment signal.

The classifier reaches $F_1 = 0.891$, within the range of the LLM baselines (0.837–0.943; Table 3). A small local model detects unresolved questions with accuracy comparable to the LLM baselines.

Table 3: Unresolved-question detection on 243 human-labeled responses, with conversation-grouped five-fold cross-validation. Metrics pool held-out predictions across folds.

Approach	Accuracy	Precision	Recall	F1
spaCy	0.807	1.000	0.671	0.803
Classifier (MiniLM + LogReg)	0.877	0.931	0.853	0.891
Opus	0.889	0.975	0.832	0.898
Sonnet	0.926	0.970	0.902	0.935
Haiku	0.934	0.957	0.930	0.943
DeepSeek-v4-flash	0.897	0.961	0.860	0.908
Nemotron-30B-A3B	0.831	0.972	0.734	0.837

Each method labels the same 100 responses ten times. The classifier returns the same label every time; the LLM baselines change 4–29% of their labels (Table 4). The classifier labels a response in 9.27 ms at the median on a commodity CPU (mean 14.07 ms; p95 30.14 ms), so state assignment stays reproducible and cheap.

Table 4: Label stability across 10 repeated runs on a fixed sample of 100 responses. Flip rate is the share of responses whose label changes in at least one run.

Approach	Accuracy (mean $\pm$ std)	Flip rate
Classifier (MiniLM + LogReg)	0.877 (fixed)	0%
Opus	0.887 ± 0.008	4%
Sonnet	0.919 ± 0.009	4%
Haiku	0.941 ± 0.009	4%
DeepSeek-v4-flash	0.892 ± 0.012	16%
Nemotron-30B-A3B	0.835 ± 0.015	29%

An ablation shows why the attachment matters: without the attachment signal, each method must infer all three states from text (Table 5).When predicting all three states from text alone, the classifier achieves a macro-F1 of 0.611 and an F1 of 0.250 for `NeedsClarification`. Sonnet does best (macro-F1 0.860), so text sometimes reveals an attachment, but LLM labels grow less stable: flip rates range from 1% to 48%, and Sonnet's rises from 4% to 9%. The attachment supplies a deliverable signal that text gives only unreliably.

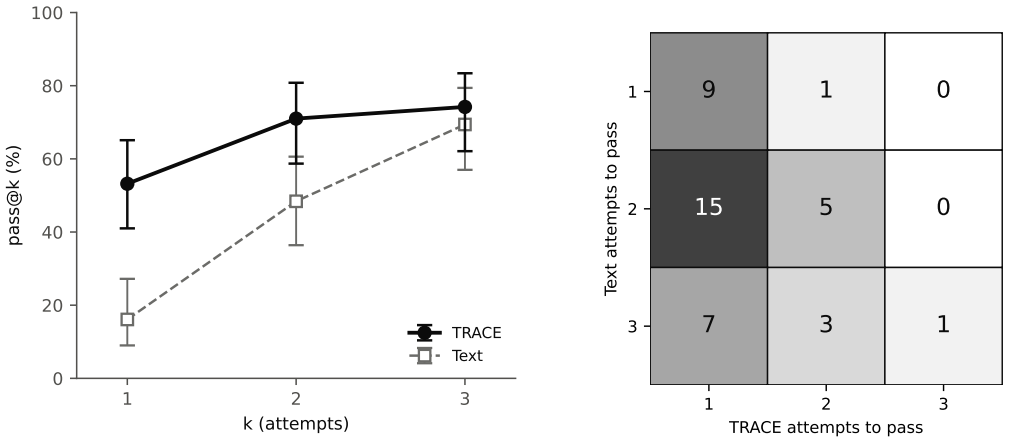
5.2 AGENT PERFORMANCE

These experiments measure how often the agent passes and how many attempts it needs, with its history in TRACE or in text.

With TRACE, the agent passes more jobs within every attempt budget, and the gap shrinks as the budget grows (Figure 2a; Table 6). On the first attempt, it passes 33 of 62 jobs, against 10 with text, a paired difference of 37.1 points (95% CI [23.1, 49.1]; exact McNemar $p = 1.55 \times 10^{-6}$); 24 jobs

Table 5: Three-way state recovery from text alone, without the attachment signal ($n = 243$). Classifier metrics pool held-out predictions from conversation-grouped cross-validation; LLM baselines label the same responses. Per-class columns report F1.

Approach	Macro-F1	Accuracy	InReview	Provisional	NeedsClarification
Classifier	0.611	0.720	0.841	0.740	0.250
Opus	0.467	0.531	0.899	0.127	0.376
Sonnet	0.860	0.872	0.896	0.880	0.806
Haiku	0.627	0.663	0.870	0.554	0.459
DeepSeek-v4-flash	0.642	0.675	0.860	0.533	0.533
Nemotron-30B-A3B	0.580	0.630	0.821	0.424	0.495



(a) (b)

Figure 2: (a) Cumulative pass rate within k attempts on 62 jobs, with Wilson 95% CIs for each condition; the intervals ignore the pairing, and Table 6 gives paired differences. (b) Attempts to pass on the 41 jobs both conditions pass; each cell counts jobs. Cells below the diagonal mark jobs where TRACE needs fewer attempts.

pass only with TRACE and 1 only with text. Within two attempts, it passes every job text passes and 14 more ($p = 1.22 \times 10^{-4}$). By the third, text nearly catches up: 46 jobs pass with TRACE (74.2%; Wilson 95% CI [62.1, 83.4]) and 43 with text (69.4%; [57.0, 79.4]), a difference that is not significant ($p = 0.4531$). The advantage concentrates in the early attempts.

Table 6: Cumulative pass rate within k attempts across 62 jobs, with Wilson 95% CIs. The difference column gives the paired difference in percentage points with a 95% CI (Newcombe's method for paired proportions). Paired counts and exact McNemar tests compare the two conditions on the same jobs.

k	Text	TRACE	Difference (points)	Both pass	Both fail	Text only	TRACE only	McNemar p
1	16.1% [9.0, 27.2]	53.2% [41.0, 65.1]	37.1 [23.1, 49.1]	9	28	1	24	1.55×10^{-6}
2	48.4% [36.4, 60.6]	71.0% [58.7, 80.8]	22.6 [11.4, 32.7]	30	18	0	14	1.22×10^{-4}
3	69.4% [57.0, 79.4]	74.2% [62.1, 83.4]	4.8 [−4.1, 13.8]	41	14	2	5	0.4531

On the 41 jobs both conditions pass, the agent needs 1.268 attempts on average with TRACE and 2.024 with text, a paired reduction of 0.756 attempts (37.4%; paired t 95% CI [0.514, 0.998], df $= 40$). It needs fewer attempts on 25 of these jobs, as many on 15, and more on 1 (Figure 2b). A Wilcoxon signed-rank test on the 26 jobs whose attempts differ rejects equal attempts ($W = 10.0$, $p = 1.0 \times 10^{-5}$). Over each condition's passing runs, the means are 1.33 and 2.07 (35.7% fewer). When both conditions pass a job, the agent gets there sooner with TRACE.

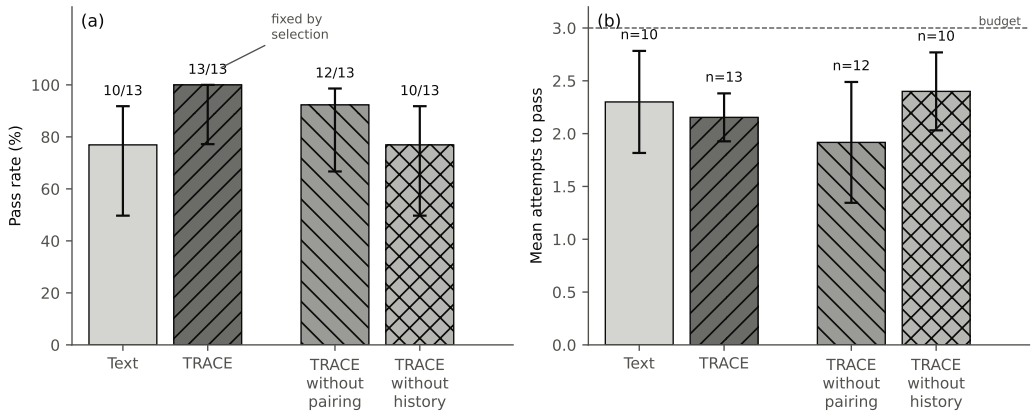


Figure 3: Ablations on the 13 multi-attempt jobs, where pairing and history can act. (a) Pass rate with Wilson 95% CIs; the jobs come from TRACE's runs, so it passes all 13. (b) Mean attempts over each condition's passing runs, with 95% t-intervals. Text and TRACE come from the main experiment; the ablations are fresh runs.

5.3 ABLATIONS

The ablations run on the 13 jobs TRACE passed after revising (Section 4.3). Because the jobs come from its own runs, TRACE passes all 13 by selection, and its pass rate does not serve as a comparison. The text and TRACE results come from the main experiment, and the ablations are fresh runs.

Removing the pairing changes little: without it, the agent passes 12 of 13 jobs and needs 1.92 attempts on its passing runs, against 2.15 with TRACE (Figure 3). Removing earlier attempts brings results down to text's level: without history, the agent passes 10 of 13, as with text, and needs 2.40 attempts, against 2.30.

Without the pairing, the agent still does better than with text on these jobs; without earlier attempts, it does no better. During revision, the results are consistent with the states and links, rather than the LLM pairing, driving the gain (pairwise tests in Appendix F).

On the first attempt, TRACE with plain guidelines passes 16 of its 27 jobs, while text passes 7 of the same jobs and TRACE, by selection, passes all 27. The evaluator returned no valid verdict on one job, so the paired comparison uses the other 26: the ablation passes 11 jobs that text fails, and text passes 1 that the ablation fails (exact McNemar $p = 0.0063$). The first-attempt gain therefore does not depend on typed guidelines alone: TRACE's request, with the guidelines left as plain text, already lifts the agent well above text on the same jobs (Figure 4).

6 DISCUSSION

TRACE changes how many attempts the agent needs, not whether it finally succeeds. Across 62 matched jobs, the final pass rates differ little (74.2% versus 69.4%, exact McNemar $p = 0.4531$), but on the 41 jobs both conditions complete, the agent needs 37.4% fewer attempts with TRACE (Wilcoxon $p = 1.0 \times 10^{-5}$). Both conditions receive the same task descriptions, category-level guidelines, interaction history, and feedback, and neither sees the evaluator's acceptance criteria; the conditions differ in how they present that information.

The pass rates within each attempt budget show where the gain arises. With TRACE, the agent passes 53.2% of jobs on the first attempt, against 16.1% with text ($p = 1.55 \times 10^{-6}$), before either condition receives any feedback on the job; the gap narrows to 22.6 points within two attempts and 4.8 points within three. Because the gap opens before feedback arrives, it points to how each condition presents the request. The text condition lists the guidelines as bullets; TRACE sends the same texts as typed nodes, tags each with the dimension reviewers score and a mandatory flag, labels the dimensions in plain words, and calls the nodes rubric criteria. TRACE thus shows the agent, before its first attempt,

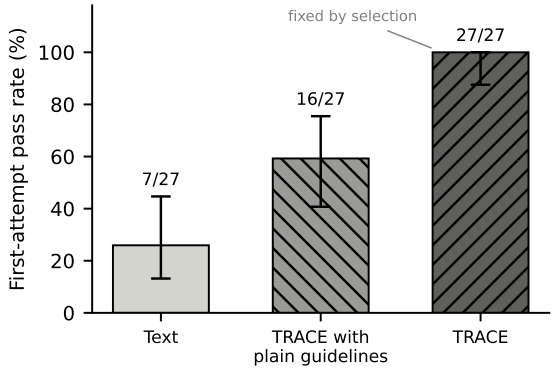


Figure 4: First-attempt pass rate on the 27 jobs of the third ablation, with Wilson 95% CIs. The jobs come from TRACE's first-attempt passes, so TRACE passes all 27.

how reviewers organize their judgment of work in the category. A third ablation separates part of this effect. Keeping TRACE's request node and prompt but listing the guidelines as plain text recovers much of the first-attempt gap between text and TRACE (16 of 26 jobs against 6; $p = 0.0063$). The guideline texts match across all three conditions, so the gain comes from how the request presents them, not from what it contains.

Because the largest gap opens before any feedback, lineage across attempts cannot explain it; lineage bears only on later revisions. After the first attempt, the gap narrows as both conditions revise. The parent chain links each revision to the feedback it answers, so the agent does not have to infer that link from where a message sits in the transcript.

The ablations bear only on revision. On the 13 jobs TRACE passes only after revising, removing the pairing changes little (12 of 13 pass; 1.92 versus 2.15 attempts), and removing earlier attempts brings results to text's level (10 of 13 pass; 2.40 versus 2.30 attempts). On this selected subset, more jobs pass without pairing than without history. This pattern suggests that retaining prior interactions may help during revision,carry TRACE's value during revision.

TRACE builds most of its structure without a generative model. It combines the observed deliverable signal with a binary question classifier that reaches $F_1 = 0.891$, comparable to the LLM baselines. The classifier returns the same label on every run, while the LLM baselines flip 4% to 29% of their labels, and it labels a message in 9.27 ms at the median on a CPU. Given the same inputs, the pipeline assigns the same coordination state, so anyone can reproduce the stored states.

Limitations. The agent runs on one model, works 62 jobs from one category, and stops after three attempts. These runs cannot show whether TRACE helps other models or other kinds of work, or whether failing jobs would pass with more attempts. The LLM evaluator agreed with human reviewers on every skip but on only 65.1% of accept-or-reject decisions, and its agreement varies widely from job to job (median 61.9%, interquartile range 45.6–77.0%; Appendix D). Both conditions use the same evaluator, but its errors may affect them differently. These errors may change both the absolute pass rates and the measured difference between conditions.The evaluator also remembers earlier attempts, so what it saw before shapes how it judges a revision; this holds in both conditions but ties each verdict to the last. TRACE decides whether a message delivers work by checking for an attachment, which works only where the platform exposes attachments; elsewhere, TRACE would have to infer delivery from text, which loses accuracy and stability (Section 5.1). The classifier learned from 243 responses on web-development work; labeling more responses, from more kinds of jobs, would show whether it transfers.

7 CONCLUSION

Agent frameworks usually store experience as text, even when a later agent must recover its structure: what a turn did, what it answered, and how reviewers judged it. TRACE records that structure

as each message arrives and links each message to the one it answers. With the same information in both conditions, an agent using TRACE passes 53.2% of real web-development jobs on its first attempt, against 16.1% with text, and needs 37.4% fewer attempts on the jobs both conditions complete; a small classifier assigns coordination state without a generative model and returns the same label on every run. For this agent and these jobs, how the history is represented changes how quickly the agent succeeds.

REPRODUCIBILITY STATEMENT

The supplementary material contains the prompts for both conditions and the question classifier with code that applies it. Appendix A describes the agent setup, Appendix B the jobs and how the guidelines were built, Appendix D the evaluator, and Appendix E the pairing step with its prompt; Appendix F gives the per-job ablation results, and Appendix G lists all nine guidelines. The jobs come from real clients and cannot be released.

LLM USAGE

Large language models assisted with drafting and revising prose, with LaTeX formatting and figure code, and with checking the text against the data, including recomputing reported statistics. The authors designed and ran every experiment, decided what the paper claims, and reviewed and edited every passage; they take full responsibility for its content. LLMs also appear in the method itself, as the baselines, the evaluator, and the pairing step described above.

REFERENCES

Anthropic. Model context protocol specification. `https://modelcontextprotocol.io/specification/2025-11-25`, 2024. Revision 2025-11-25.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025.

Joel Farrell and Holger Lausen. Semantic annotations for WSDL and XML schema. W3C Recommendation, 2007. URL `https://www.w3.org/TR/2007/REC-sawsdl-20070828/`.

Tim Finin, Rich Fritzson, Don McKay, and Robin McEntire. KQML as an agent communication language. In *Proceedings of the Third International Conference on Information and Knowledge Management*, pp. 456–463. ACM, 1994. doi: 10.1145/191246.191322.

Foundation for Intelligent Physical Agents. FIPA ACL message structure specification. Technical Report SC00061G, FIPA, 2002. URL `http://www.fipa.org/specs/fipa00061/`.

Google. Agent2agent protocol (A2A). `https://google.github.io/A2A/`, 2025. Version 0.3.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024.

Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hanglei Ding, Kaiwen Men, Kejuan Yang, et al. AgentBench: Evaluating LLMs as agents. In *International Conference on Learning Representations*, 2024.

Marvin Minsky. K-lines: A theory of memory. *Cognitive Science*, 4(2):117–133, 1980.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. *arXiv preprint arXiv:2304.03442*, 2023.

Preston Rasmussen, Pavlo Paliychuk, Travis Beauvais, Jack Ryan, and Daniel Chalef. Zep: A temporal knowledge graph architecture for agent memory. *arXiv preprint arXiv:2501.13956*, 2025.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Andreas Stolcke, Klaus Ries, Noah Coccaro, Elizabeth Shriberg, Rebecca Bates, Daniel Jurafsky, Paul Taylor, Rachel Martin, Carol Van Ess-Dykema, and Marie Meteer. Dialogue act modeling for automatic tagging and recognition of conversational speech. *Computational Linguistics*, 26 (3):339–373, 2000.

Theodore R. Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L. Griffiths. Cognitive architectures for language agents. *Transactions on Machine Learning Research*, 2024.

W3C. PROV-DM: The PROV data model. `https://www.w3.org/TR/prov-dm/`, 2013. W3C Recommendation.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, et al. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. In *Advances in Neural Information Processing Systems*, volume 38, 2025. arXiv:2502.12110.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 2024.

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

APPENDIX

A AGENT SETUP

Each agent turn runs a fresh OpenCode session (opencode-ai v1.14.30) inside a Docker container with Node 22 and a headless Playwright MCP server (v0.0.80), which the agent uses to check rendered pages. The agent model is DeepSeek-v4-flash-0731, served through Fireworks. A local proxy injects gateway credentials and strips two reasoning parameters the gateway rejects. No temperature or decoding setting is overridden, so the model runs at OpenCode's defaults. The agent may use every tool except interactive questions; `git` and `gh` commands are denied. Both conditions use the same agent model and evaluator versions, and only the representation changes.

Timeouts and retries. A turn ends after 40 minutes without events or 2 hours of wall-clock time, whichever comes first. A turn that fails within its first minutes is retried once in a fresh session. The evaluator has 10 minutes per call. The dimension tagger gets two attempts with exponential backoff (240 seconds each) and leaves feedback untagged if both fail. Deliverable uploads retry with exponential backoff (45 seconds to 5 minutes in practice, for transient gateway errors), and the orchestrator confirms each upload before evaluation starts.

Isolation between runs. The container persists across a batch, but each turn starts a fresh session, and only the job's working directory, mounted from the host, survives from one turn to the next. Before each upload, the agent removes build artifacts and scratch files; the upload tool also excludes them, along with pipeline inputs such as the request graph, and deletes its staging directory afterward. Each job's working directory is shared across runs and conditions, so it is wiped before each condition's batch, and a comparison of deliverable file sets confirms the isolation.

B JOBS AND GUIDELINES

Jobs. The jobs come from one category of client work: front-end web development, within coding and web development. They were chosen to span a range of complexity, measured by the client's budget. The pool holds 64 jobs, and 62 of them enter the experiments. Across the pool, budgets range from $5 to $1,150, with a median of $50 and a mean of $116.38 (standard deviation $173.88); jobs at $650 and $1,150 skew the distribution to the right. Each job carries 5.64 acceptance criteria on average, and only the evaluator sees them.

Guidelines. The guidelines come from earlier agent runs on other front-end jobs. Human reviewers judged those runs, and an LLM aggregated their reviews and clustered them by common theme. Labelers reviewed and refined the clusters; the feedback in each cluster was then aggregated, and guidelines were extracted from it. Labelers then compared a sample of the raw reviews against the guidelines and refined the guidelines once more. The result is nine guidelines shared by every job in the category (Appendix G), each tagged with one of eight dimensions and a mandatory flag. No guideline repeats a job's own acceptance criteria.

C PROMPTS

Both conditions start from the turn-0 prompts below, for the job in Figure 1. The prompts appear verbatim except that the job ID, the platform name, the run-format name, and namespace prefixes are replaced, the `@context` block is omitted, and seven of the nine guideline nodes are cut. The guideline nodes carry the type `up:AcceptanceCriterion`, an older name in the schema; they hold the nine category guidelines, not the job's acceptance criteria. Both prompts ask the agent for the same claims section. In later turns, the text condition receives feedback in the format shown in Figure 1, and TRACE receives the new feedback and followup nodes.

Text condition.

```
Turn 0. New job. job_id=<job_id>. Run format: text. Run `drive-fetch-job <job_id>` first to
     get your job bundle, complete the work, then run `drive-push-deliverable <job_id> 0 text`
      when done.

In NOTES.md, include a "## Claims" section listing each discrete claim about what you
     delivered as one line: `- [c1] <claim>` (c1, c2, ...). The pipeline uses these to
     attribute evaluation feedback to your work.

Evaluation quality bar:
The work is assessed across the following quality dimensions. Allocate effort across each of
     them, not only the most visible part of the task:
- **Functional behavior:** Requested interactions and workflows operate as intended; controls,
      state transitions, data handling, and navigation behave coherently.
- **Technology and delivery constraints:** The requested technologies, libraries, patterns,
     dependencies, and output formats are followed; do not substitute a different approach or
     bypass an explicit constraint.
- **Visual and structural fidelity:** The overall structure, hierarchy, spacing, styling,
     positioning, and presentation reflect the intended result, including important visual
     relationships and structural details.
- **Completeness:** The full requested experience is covered -- including less prominent
     elements, secondary states, content, assets, and supporting behavior -- not just the
     primary or most obvious portion.
- **Responsive and contextual behavior:** The result stays usable and visually coherent across
      relevant screen sizes, layouts, and usage conditions, without unintended overlap,
     clipping, or broken layout.
- **Data, integration, and business logic:** Information flows correctly through external
     systems, persistence, calculations, validation, and domain rules; described rules are
     preserved.
- **Runability and reliability:** The deliverable installs, compiles, launches, and executes
     through the requested workflow without errors, preserving existing behavior when
     modifying an existing system.
- **Implementation quality:** The solution is organized, maintainable, appropriately modular,
     and consistent with the target environment's conventions; quality supports correctness,
     not unnecessary abstraction.
- **Explicit details and constraints:** Exact values, permitted or prohibited approaches,
     required states, ordering, limits, and exceptions in the request are honored rather than
     silently replaced with assumptions.
```

TRACE condition.

```
Turn 0. New job. job_id=<job_id>. Run format: <format>. This job's request is delivered as an
     experience graph (the last section of this prompt). Run `drive-fetch-job <job_id>` first
     to get the job bundle, copy the request graph verbatim into `request.json` in your job
     directory and treat it as the source of truth for the job, complete the work, then run `
     drive-push-deliverable <job_id> 0 <format>` when done.

In NOTES.md, include a "## Claims" section listing each discrete claim about what you
     delivered as one line: `- [c1] <claim>` (c1, c2, ...). The pipeline uses these to
     attribute evaluation feedback to your work.

Evaluation dimensions (full guidance for each is in the graph's rubric criteria):
- Functional behavior (ns:FunctionalCorrectness)
- Technology and delivery constraints (ns:SpecificationAdherence)
- Visual and structural fidelity (ns:PresentationQuality)
- Completeness (ns:DeliverableCompleteness)
- Responsive and contextual behavior (ns:ResponsiveAndContextual)
- Data, integration, and business logic (ns:DataIntegrationBusinessLogic)
- Implementation quality (ns:ImplementationQuality)
- Explicit details and constraints (ns:ExplicitDetailsAndConstraints)

About the graph you are about to read: it describes a <platform> client request as connected
     message nodes, not free-form prose. An up:RequestMessage is the client's original ask (
     its `claims` list the deliverables they expect); up:FollowupMessage nodes carry
     clarifications or the work-so-far; up:EvaluationMessage nodes carry evaluator feedback
     about what passed or failed. Node `text` fields hold the actual content; node `claims`
     reference items of YOUR OWN submitted work (reuse the ids you declared under `## Claims`
```

```
    in NOTES.md). Everything in the graph is binding -- treat it the way you would treat the
    client's message thread.

{
  "@graph": [
    {
      "@type": "up:RequestMessage",
      "@id": "urn:uuid:node-245d55b654641ce0",
      "up:threadId": "urn:uuid:thread-3a8d1b44ca25c7c5",
      "up:state": "up:Submitted",
      "up:payload": {"job_id": "<job_id>", "job_description_ref": "urn:job:<job_id>"},
      "up:title": "Job <job_id>",
      "up:requestType": "job",
      "up:definesDimensions": ["ns:FunctionalCorrectness", "ns:SpecificationAdherence", "ns:
          PresentationQuality", "ns:DeliverableCompleteness", "ns:ResponsiveAndContextual", "ns
          :DataIntegrationBusinessLogic", "ns:ImplementationQuality", "ns:
          ExplicitDetailsAndConstraints"],
      "up:definesCriteria": ["urn:uuid:671b911d...", "urn:uuid:2ae39a91...", ... 9 IDs]
    },
    {
      "@type": "up:AcceptanceCriterion",
      "@id": "urn:uuid:671b911d221d4e95bee430e83a547117",
      "up:description": "Functional behavior: Requested interactions and workflows operate as
          intended; controls, state transitions, data handling, and navigation behave
          coherently.",
      "up:mandatory": true,
      "up:dimension": ["ns:FunctionalCorrectness"]
    },
    {
      "@type": "up:AcceptanceCriterion",
      "@id": "urn:uuid:b6a7f4b5c691423a9de3eb8cedc39c88",
      "up:description": "Completeness: The full requested experience is covered -- including
          less prominent elements, secondary states, content, assets, and supporting behavior
          -- not just the primary or most obvious portion.",
      "up:mandatory": true,
      "up:dimension": ["ns:DeliverableCompleteness"]
    },
    ... seven more guideline nodes of the same form
  ]
}
```

D EVALUATOR

GPT-5.6-sol evaluates each submission as a tool-using agent (Figure 5). It receives the job description, the task, the job's acceptance criteria, and the submitted files, and it returns one judgement per criterion: pass, fail, or skip, with the evidence it cites and its reasoning. The agent under test never sees these criteria.

The evaluator reads two directories: `job/` holds the client's specification, and `deliverable/` holds the submission. Every file tool is read-only, so the evaluator can inspect the work but cannot change it. Six tools gather evidence (Table 7).

Table 7: Evidence tools available to the evaluator.

Tool	What it does
File context	Lists, reads (optionally by line range), and searches files in `job/` and `deliverable/`.
Image reader	Passes an image and a targeted prompt to a vision-capable model and returns its description as text, when the evaluating model reads only text; a vision-capable evaluator reads images directly.
Code evidence	Parses Python and JavaScript source (Python AST, Node static analysis) to check structure, such as whether a file defines a required function.
HTML evidence	Checks the DOM structure of HTML files.
Artifact evidence	Checks that required files exist and forbidden files do not; it never reads file contents.
Content evidence	Runs a scoped sub-agent over files the other tools do not cover, such as reports, documents, and configuration; it can call the code-evidence tool for source context.

Gathering evidence and judging stay separate. The evaluator first checks what the submission contains (source code, HTML, images, or other files) and routes each criterion to the tools that can produce evidence for it. Each tool adds claims, check results, and gaps to a shared ledger, keyed by criterion; no tool renders a verdict. A final judging step reads the ledger and emits exactly one judgement per criterion. Every loop, including the content sub-agent, ends by calling a typed submit tool instead of writing free text, so each result arrives in a fixed schema. The judging step resolves which criterion a judgement belongs to by rule rather than from model output, and a validation loop retries any verdict set that misses or duplicates a criterion.

The evaluator runs through the Responses API, which carries a job's earlier attempts forward. On a revision, it sees what the agent changed since its last submission; the same holds in both conditions.

Agreement with human reviewers. On 281 calibration cases from 47 jobs, the evaluator agreed with human reviewers on every skip ([k] of [k]) and on 65.1% of accept-or-reject decisions (Cohen's κ = [X]). Agreement varies by job (Figure 6): the median job agrees on 61.9% of its cases, the interquartile range spans 45.6% to 77.0%, and the lowest job agrees on 12.0%. Both conditions face the same evaluator, so its errors affect them alike, but they shift the absolute pass rates.

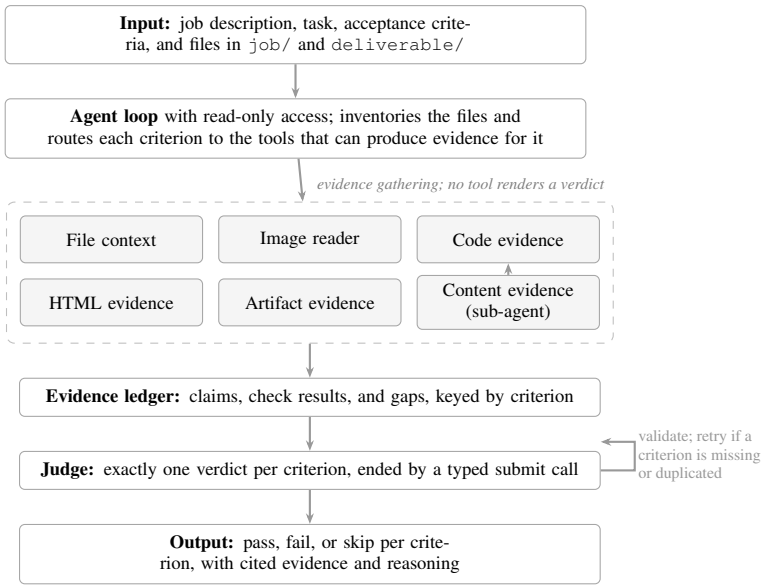


Figure 5: The evaluator's tool-calling architecture. Evidence tools read the submission and add to a ledger keyed by criterion; only the judge renders verdicts. Every loop, including the content sub-agent, ends with a typed submit call.

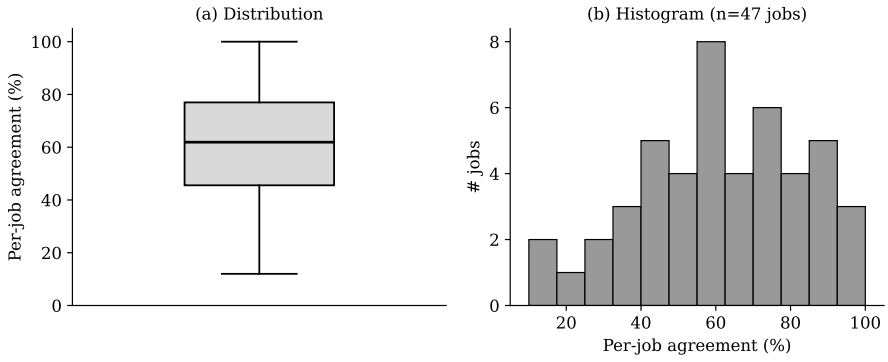


Figure 6: Per-job agreement between the evaluator and human reviewers on accept-or-reject decisions, across 47 calibration jobs. (a) Box plot; whiskers span the full range. (b) Histogram.

E PAIRING STEP

Both conditions ask the agent to add a claims section to its notes, listing each discrete claim about its deliverable on its own line with an ID (c1, c2, . . .). After each failed attempt, one call to Qwen3.8-27B pairs the evaluator's results with those claims. For each failing criterion, the call receives the criterion, the evaluator's reasoning (truncated to 500 characters), the fixed list of dimensions, and the agent's claims (each truncated to 200 characters). It returns one dimension per criterion and the IDs of the claims the criterion concerns. The agent never sees the criterion: its feedback carries the evaluator's reasoning, the dimension, and the linked claims. The text condition receives the same reasoning as bullets, without dimensions or claim links.

The prompt reads as follows; braces mark fields filled at run time, and the claims block appears only when the agent declared claims that turn.

```
Map each failing acceptance criterion to (a) the evaluation dimension it
most closely falls under and (b) the agent claim(s) it is about.

Dimensions (use exactly one of these URIs per criterion):
{sorted dimension URIs, one per line}

The agent self-declared these claims about its own work (attribute each
criterion to the claim(s) it is about, by id; empty list if none apply):
- {claim_id}: {claim text}

Criteria:
[{"criterion_id": <id>, "criterion": <criterion text>,
  "reasoning": <evaluator's reasoning>}, ...]

Respond with ONLY a JSON object: {"criterion_id": {"dimension":
"ns:Dimension", "claims": ["urn:claim:..."]}, ...}
```

The pipeline parses the reply as JSON. If that fails, it extracts the outermost brace-delimited span, because the model sometimes wraps its answer in prose or a code fence, and then tries to read the span as a Python literal. A dimension outside the fixed list becomes empty, and claim IDs the agent did not declare are dropped. The call gets two attempts; if both fail, the feedback stays untagged (Appendix A). TRACE without pairing omits this step, and the text condition never runs it.

F ABLATION STATISTICS

Table 8 lists each of the 13 ablation jobs under every condition. The text and TRACE columns come from the main experiment; the two ablations are fresh runs on the same jobs. The jobs were selected because TRACE passed them only after revising, so TRACE passes all 13 by construction.

Table 8: Attempts to pass for each ablation job; "fail" marks a job that did not pass within three attempts. *The job in Figure 1.

Job	Text	TRACE	Without pairing	Without history
J1	3	2	1	2
J2	3	2	3	fail
J3	2	2	fail	3
J4	1	2	2	2
J5	2	2	3	3
J6	fail	2	3	fail
J7	3	2	3	2
J8	2	2	1	2
J9	fail	2	1	2
J10	2	2	1	3
J11	2	2	1	2
J12	3	3	2	3
J13*	fail	3	2	fail
Passed	10/13	13/13	12/13	10/13
Mean attempts (passing runs)	2.30	2.15	1.92	2.40

Table 9 compares the conditions in pairs. For passing, an exact McNemar test compares the jobs only one condition passes. For attempts, an exact sign test compares the jobs both conditions pass and drops jobs where they need the same number. No difference reaches significance at this sample size. Every comparison with TRACE inherits the selection: a fresh run on a job chosen for needing revision may need fewer attempts, which favors the other condition. The comparisons that exclude TRACE carry no such bias.

Table 9: Pairwise comparisons on the 13 ablation jobs. "A only" and "B only" count jobs one condition passes and the other fails. Attempts compare jobs both conditions pass; "fewer", "same", and "more" count jobs where A needs fewer, the same, or more attempts than B.

		Pass			Attempts on jobs both pass		
A	B	A only	B only	p	n	Means (A, B)	Fewer/same/more, p
Without pairing	TRACE	0	1	1.000	12	1.92, 2.17	7/1/4, 0.549
Without history	TRACE	0	3	0.250	10	2.40, 2.10	0/7/3, 0.250
Without pairing	Text	3	1	0.625	9	1.89, 2.33	5/2/2, 0.453
Without history	Text	1	1	1.000	9	2.44, 2.22	2/3/4, 0.688
Without pairing	Without history	3	1	0.625	9	1.67, 2.33	6/2/1, 0.125
Text	TRACE	0	3	0.250	10	2.30, 2.10	1/6/3, 0.625

First-attempt ablation. TRACE with plain guidelines ran on 27 of the jobs TRACE passed on its first attempt. One job returned no valid verdict; text passed it. On the other 26, both conditions pass 5 jobs and fail 9, text alone passes 1, and the ablation alone passes 11 (exact McNemar $p = 0.0063$). TRACE passes all 26 by selection.

G EXAMPLE MESSAGES IN FULL

This appendix gives the guidelines and messages in Figure 1 without trimming. IDs and dimension names are shortened as in the figure.

m0, the request's guidelines. Both conditions receive the nine guideline texts below, in this order. The text condition shows them as a bulleted list after the sentences "Evaluation quality bar: The work is assessed across the following quality dimensions. Allocate effort across each of them, not only the most visible part of the task:". TRACE attaches each text to the request as a typed item with its dimension and mandatory flag; the text condition shows neither. Guidelines 1 and 7 share a dimension, so nine guidelines cover eight dimensions.

#	Guideline text (both conditions)	Dimension (TRACE only)	Mandatory
1	**Functional behavior:** Requested interactions and workflows operate as intended; controls, state transitions, data handling, and navigation behave coherently.	Functional correctness	Yes
2	**Technology and delivery constraints:** The requested technologies, libraries, patterns, dependencies, and output formats are followed; do not substitute a different approach or bypass an explicit constraint.	Specification adherence	Yes
3	**Visual and structural fidelity:** The overall structure, hierarchy, spacing, styling, positioning, and presentation reflect the intended result, including important visual relationships and structural details.	Presentation quality	Yes
4	**Completeness:** The full requested experience is covered — including less prominent elements, secondary states, content, assets, and supporting behavior — not just the primary or most obvious portion.	Completeness	Yes
5	**Responsive and contextual behavior:** The result stays usable and visually coherent across relevant screen sizes, layouts, and usage conditions, without unintended overlap, clipping, or broken layout.	Responsive and contextual	Yes
6	**Data, integration, and business logic:** Information flows correctly through external systems, persistence, calculations, validation, and domain rules; described rules are preserved.	Data, integration, and business logic	Yes
7	**Runability and reliability:** The deliverable installs, compiles, launches, and executes through the requested workflow without errors, preserving existing behavior when modifying an existing system.	Functional correctness	Yes
8	**Implementation quality:** The solution is organized, maintainable, appropriately modular, and consistent with the target environment's conventions; quality supports correctness, not unnecessary abstraction.	Implementation quality	Yes
9	**Explicit details and constraints:** Exact values, permitted or prohibited approaches, required states, ordering, limits, and exceptions in the request are honored rather than silently replaced with assumptions.	Explicit details and constraints	Yes

m2, feedback on the first attempt.

- *Dimension:* Specification adherence; *claims:* a1-c3, a1-c4, a1-c5, a1-c13, a1-c15, a1-c17.
  The delivered screenshots reproduce the broad dark 25/75 control-panel/game-board concept, but they do not match the supplied reference layout closely enough for the stated "exactly same" requirement. In deliverable/attachments/attach.png, Bet Amount has the fiat equivalent and the ½/2× controls integrated alongside the amount input, there is no separate BALANCE card or Provably fair pill in the left panel, and the header does not contain the two extra utility buttons. In deliverable/screenshots/desktop-manual-over.png, the implementation instead adds a BALANCE card, places 1/2 and 2x on a separate row, adds a Provably fair button and prominent result/status readout, and uses materially different field/card styling and spacing. These are visible structural differences, not merely minor pixel variations.

- *Dimension:* Completeness; *claims:* a1-c4, a1-c5, a1-c6, a1-c7, a1-c8, a1-c9, a1-c10, a1-c11, a1-c12, a1-c14.
  A substantial playable subset is implemented—manual rolling (app/page.js:288-313), auto betting (app/page.js:315-404), draggable target (components/DiceBoard.jsx:24-54), editable multiplier/target/chance (components/BottomStrip.jsx:26-67), history and fairness dialogs—but not all Stake dice functions/details are present or equivalent. The reference's balance/currency dropdown and fiat-equivalent displays are absent: components/BetPanel.jsx:93-100 renders a static currency chip, while the currency-change handler at app/page.js:178-187 is never exposed in the rendered controls. The submitted wallet is explicitly only a local play-money top-up UI (screenshots/desktop-wallet.png), rather than Stake wallet functionality. There is also a concrete auto-bet behavior defect: after every result, app/page.js:374 clamps the next stake to a minimum of 0.01, despite the default BTC bet being 0.000001, so Reset or percentage strategies can unexpectedly jump the stake by 10,000×. Therefore functional parity with all Stake dice functions is not demonstrated.

m3, the second attempt, claim a2-c10. Auto bet runs (count-down, stop on profit/loss, bet & cancel on win, On-Win/On-Loss stalkers) and clamps stakes to a currency-appropriate minimum (10^-decimals, e.g. 0.000001 BTC), fixing the prior 0.01 floor bug.

m4, feedback on the second attempt.

- *Dimension:* Completeness; *claims:* a2-c10.
  Many core controls are implemented: manual roll and settlement (app/page.js lines 253–322), auto-bet settings (components/BetPanel.jsx lines 150–228), editable multiplier/target/chance (components/BottomStrip.jsx lines 26–67), history (page.js lines 520–540), and fairness UI (components/FairnessDialog.jsx). However, not all Stake dice behavior is reproduced. In Stake Auto mode, Number of Bets = 0 denotes an unlimited run, but this implementation parses 0 as a finite count (app/page.js lines 343–345) and, after the first roll, computes remaining as 0 and stops (lines 390–397). The fairness rotation flow is also incomplete: rotateServerSeed retains revealedServerSeed while replacing the active seed (lines 210–215), and FairnessDialog.jsx lines 35–43 then continues displaying only the old revealed seed rather than the new active commitment. The wallet explicitly states it is only a local play-money mock with no server-side balances (components/WalletModal.jsx lines 107–110). Thus the submission demonstrates a broad functional prototype, but not all functions/behavior of the Stake.com dice client.