TRACE: TRajectory Attribution for Automated Context Engineering

Yikai Zhao
Amazon
Dallas, TX, USA
yikai@amazon.com

Pradeep Kumar Misra
Amazon
Dallas, TX, USA
pkmisra@amazon.com

Saurabh Pandey
Amazon
Seattle, WA, USA
saurabpd@amazon.com

ABSTRACT

Production AI agents fail when their context sources—system prompts, knowledge bases, tool descriptions, and procedural skills—contain errors or gaps. Current maintenance approaches rely on manual log review and ad-hoc debugging, creating a scalability bottleneck as interaction volume grows.

We present TRACE (TRajectory Attribution for Automated Context Engineering), an automated feedback loop that mines historical agent trajectories to diagnose and remediate context failures. Our key insight is that agent trajectories are rich with implicit dissatisfaction signals—user corrections, rephrasing patterns, abandonment cues—that reveal precisely where context sources failed, without requiring explicit feedback collection. Unlike model fine-tuning approaches, TRACE operates on the context layer, enabling rapid iteration without retraining.

The system makes four contributions: (1) a trajectory mining framework that systematically extracts diagnostic information from historical agent executions; (2) multi-component causal attribution that extends textual gradients from monolithic prompt optimization to heterogeneous context sources (skills, knowledge bases, tools, prompts); (3) exploratory verification where agents actively read context sources to distinguish content gaps requiring CREATE operations from stale content requiring UPDATE—achieving 96% operation accuracy; and (4) a reusable simulation methodology and verifiable evaluation benchmark addressing the absence of open datasets for context debugging, with a six-category fault taxonomy, complete ground truth annotations, and a cross-layer verification protocol that can be adopted to generate domain-specific benchmarks.

Evaluation on 60 dissatisfaction traces spanning three complexity tiers (up to 16 execution nodes) achieves 72.7% root cause node attribution and 82% end-to-end fix effectiveness—demonstrating that over 80% of context-layer failures could be automatically diagnosed and correctly remediated by mining historical agent trajectories, an overlooked resource in production systems.

KDD '26, August 9–13, 2026, Jeju, South Korea

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

CCS CONCEPTS

• **Computing methodologies** → **Causal reasoning and diagnostics**; **Multi-agent systems**; *Anomaly detection.*

KEYWORDS

context engineering, causal attribution, anomaly detection, automated feedback loops, root cause analysis, textual gradients, multi-agent systems

1 INTRODUCTION

Modern AI agents are complex systems whose behavior is shaped by carefully engineered *context*—system prompts that define behavior, tool descriptions that guide function calling, knowledge bases that provide domain information through retrieval-augmented generation (RAG), and Skills files that encode multi-step workflows. This practice of *context engineering* [11] has emerged as the primary mechanism for customizing and improving agent capabilities without retraining the underlying model. However, when users interact with these agents and experience dissatisfaction, the root cause may lie in any of these context sources, making diagnosis challenging and remediation labor-intensive.

Current approaches to maintaining AI agents rely heavily on manual review of interaction logs, ad-hoc error reports, and periodic audits. This creates a fundamental scalability bottleneck: as interaction volume grows, the gap between failures occurring and fixes being applied widens, leading to accumulated user dissatisfaction and degraded agent performance.

To address this challenge, we present TRACE (TRajectory Attribution for Automated Context Engineering), an automated feedback loop that mines historical agent trajectories to diagnose and remediate context failures. Our approach is grounded in three key insights:

First, historical agent trajectories represent an untapped resource for context engineering. Organizations deploying AI agents generate vast amounts of execution data—not just conversation logs, but complete trajectories including reasoning traces, tool invocations, retrieved contexts, and user reactions. Yet this resource remains largely underutilized for systematic context improvement. While existing approaches use production data for model fine-tuning (RLHF) or require explicit human labels, the implicit signals already present in these trajectories—user corrections, rephrasing patterns, escalation language, semantic drift—provide rich diagnostic information about context failures without additional annotation costs. The DRIFT framework [3] demonstrates that such dissatisfaction signals occur approximately twice as frequently as explicit satisfaction signals and contain richer semantic information.

Second, conversation trajectories are differentiable. TextGrad [1] introduces the concept of "textual gradients"—semantic descriptions of how each component in a computational graph should change to improve outcomes. We adapt this concept to conversation trajectories, treating each *decision point* as a node that can receive attribution for the final outcome. A decision point is any step where the agent makes a choice influenced by its context sources: selecting which tool to invoke (guided by tool descriptions), retrieving knowledge (shaped by the KB content and/or embeddings), or formulating a response (constrained by the system prompt).

Third, specialized agents enable modular optimization. The ACE Framework [4] separates concerns into Generator, Reflector, and Curator roles. We adopt this architecture, allowing each agent to specialize in its task while maintaining clear interfaces between components.

Novel Contributions. This work advances context engineering through four innovations:

(1) **Trajectory Mining for Context Engineering**: We present a novel framework for systematically mining historical agent trajectories—an overlooked resource—to diagnose and remediate context failures. While prior work (e.g., DRIFT [3]) uses implicit dissatisfaction signals for model fine-tuning, we apply them to the context layer, enabling rapid iteration without model retraining.
(2) **Implicit Signal Exploitation**: We exploit implicit user feedback signals—corrections, rephrasing patterns, abandonment cues—that typically go unnoticed in production logs. These signals reveal precisely where context sources are failing without requiring explicit feedback collection or annotation.
(3) **End-to-End Attribution with Exploratory Verification**: We extend textual gradients from monolithic prompt optimization to multi-component context sources (prompts, KB, tools, Skills), and introduce exploratory verification where the Recommender agent actively reads context source files before generating recommendations. This verification is essential: to recommend the correct remediation—CREATE (content missing) vs. UPDATE (content exists but wrong)—the agent must verify whether the content actually exists. Without exploration, the agent achieves only 33% accuracy on this decision; with exploration, accuracy rises to 83%, a 50 percentage point improvement.
(4) **Verifiable Evaluation Benchmark and Simulation Methodology**: To our knowledge, no open-source dataset or industry-standard benchmark exists for context engineering debugging from agent trajectories. While Trace was developed and validated on a proprietary enterprise agent system, confidentiality constraints preclude disclosure of production data. We therefore introduce a systematic three-layer simulation methodology for generating synthetic agent traces with perfect ground truth. Our benchmark provides: (i) a six-category fault taxonomy covering common context engineering failures as a sample, (ii) traces of varying complexity (2–16 nodes) with documented cascade effects, (iii) complete attribution ground truth, and (iv) a cross-layer verification protocol ensuring dataset integrity. Critically, the simulation methodology itself is a contribution: we provide detailed procedures (Appendix L) that can be adopted as a reusable skill for GenAI agents to generate domain-specific evaluation datasets, enabling the community to extend this benchmark to new domains and fault types.

2 RELATED WORK

Trace builds upon four research streams: textual gradients for weight-free optimization, dissatisfaction-driven learning, agentic self-improvement architectures, and knowledge base maintenance.

2.1 Textual Gradients and Prompt Optimization

The concept of optimizing LLM behavior without weight updates has gained traction through frameworks that treat prompts as differentiable variables. **TextGrad** [1] introduced "textual gradients"—semantic descriptions of how each component in a computational graph should change to improve outcomes. TextGrad implements a `backward()` pass that propagates semantic feedback via meta-prompting, enabling Textual Gradient Descent (TGD) optimization. Zhou et al. [2] analyze TextGrad's backward pass and note that it processes each predecessor variable independently—appropriate for optimization where variables are updated separately. **ProTeGi** [5] treats prompt optimization as beam search to address LLM output variance. **CRISPO** [6] decomposes critique into aspect-specific signals for surgical updates. **PACE** [7] adopts an Actor-Critic architecture to prevent "Prompt Drift."

2.2 Dissatisfaction-Driven Learning

Traditional RLHF [12] relies on explicit preference signals, which are sparse and expensive to collect. The **DRIFT** framework [3] demonstrates that implicit dissatisfaction signals—conversational repair, refinement trajectories, semantic drift—occur twice as frequently as satisfaction signals and contain richer diagnostic information. **Inverse Constitutional AI (ICAI)** [8] extracts generalizable principles from pairwise corrections, reverse-engineering user preferences from conversation traces. Meanwhile, Reinforcement Learning with Verifiable Rewards (RLVR) methods leverage verifiable outcomes (e.g., code execution results, tool errors) as reward signals without human labeling. Both DRIFT and RLVR target model weight optimization through fine-tuning.

2.3 Agentic Self-Improvement Architectures

The **Agentic Context Engineering (ACE)** framework [4] establishes a tripartite design pattern—Generator, Reflector, Curator—that separates task execution from system optimization. The Generator executes tasks using a dynamic "Playbook"; the Reflector performs root cause analysis on failures; the Curator manages Playbook updates through Delta Operations (ADD, UPDATE, DELETE) to prevent context collapse. Research demonstrates that

maintaining context as a dynamic artifact improves agentic benchmark performance by over 10% [4]. The literature on LLM-based AI agents [11] surveys feedback mechanisms including environmental feedback (tool results), human feedback (corrections), and model feedback (self-critique).

2.4 Knowledge Base and Memory Management

For RAG systems, optimization often requires repairing the external knowledge base. **Mem0** [9] introduces structured memory management with explicit CRUD capabilities and conflict detection—performing semantic comparison before deciding to ADD, UPDATE, MERGE, or DELETE entries. **Amber** [10] addresses retrieval failures by optimizing the retrieval process through an Adaptive Information Collector that learns mappings between vague queries and successful refined queries.

A detailed positioning of TRACE relative to each of these four streams—attribution vs. optimization, context layer vs. model weights, active exploration vs. passive acceptance, and the diagnostic layer for KB—is provided in Appendix A.

3 SYSTEM ARCHITECTURE

Figure 1 presents an overview of the TRACE architecture. The system operates as a feedback loop that continuously monitors agent trajectories, detects failures, attributes root causes, and generates recommendations for human review. A per-component overview is in Appendix F.

We now detail each agent's design and implementation. A complete end-to-end worked example demonstrating the full pipeline on a realistic failure scenario is provided in Appendix K.

4 DETECTOR AGENT

The DETECTOR Agent identifies dissatisfaction signals from agent trajectories, distinguishing between explicit signals (direct feedback) and implicit signals (linguistic cues). The DETECTOR requires structured agent trajectories—complete execution records, not just user/assistant messages (full input requirements in Appendix B).

4.1 Signal Detection

The DETECTOR takes as input a complete conversation segment—including the user's original query, the agent's response, and any follow-up exchanges where the user provides corrections or expresses dissatisfaction. Through qualitative inspection of agent conversation traces, we identified recurring patterns of user dissatisfaction that fall into two categories: *explicit signals* (e.g., thumbs-down feedback, tool execution errors, session abandonment) and *implicit signals* (e.g., conversational repair, repetition, negative sentiment, semantic drift). These signal types are embedded in the detection prompt, enabling the LLM to scan conversations and flag sessions warranting further analysis.

The DETECTOR outputs a binary DSAT determination along with a confidence score. Sessions exceeding the confidence threshold (or containing explicit signals like tool errors) are escalated to the ROOT CAUSE for root cause attribution. The complete signal taxonomy is detailed in Appendix O, and the detection prompt is provided in Appendix P.

5 ROOT CAUSE AGENT

The ROOT CAUSE Agent analyzes the conversation trajectory to identify where and why the agent failed, generating textual gradients that attribute responsibility to specific components.

5.1 Trajectory as Context Graph

We model the conversation as a graph where each node represents a context source that influenced the agent's response (full formal definition, equation, and figure in Appendix E). Each node v has associated content c_v and output o_v. This graph structure provides a conceptual framework for understanding how context flows through the agent, though our primary attribution method processes all nodes holistically rather than traversing the graph. The ROOT CAUSE attributes failures to one of six root cause categories spanning the context engineering surface (detailed in Section 7.2.1).

5.2 Delta-Guided Holistic Attribution

Our primary attribution method presents the complete trajectory to the LLM in a single pass. The *delta*—the discrepancy between what the user expected and what the agent produced—serves as the "loss signal" that guides attribution. The ROOT CAUSE reads only the agent's execution trace—chain-of-thought, tool calls, and tool outputs—and does not separately inspect raw context files; the chain-of-thought is the key diagnostic signal, since the agent's own reasoning names the sources that shaped each decision (e.g., *"According to the KB ..."* or *"Following the SOP for ..."*), letting the ROOT CAUSE identify candidate root-cause sources by reading the trace alone. Verification of those candidate files is deferred to the RECOMMENDER stage. The LLM is instructed to reason over the trace in *reverse temporal order*—gradient-descent style: starting from the final response (where the loss is observed), it walks backward (response → tool outputs → thinking → inputs) to locate the earliest node whose content is inconsistent with the delta. This mirrors how textual gradients [1] propagate semantic feedback backward through a computational graph, condensed into a single LLM call.

The simultaneous view (Algorithm 1) lets the LLM disambiguate ties between candidate root-cause nodes; the reverse ordering biases reasoning toward the earliest node that introduced the delta rather than later nodes that merely propagated it. This single-call design reduces cost by 16× versus iterative per-node approaches and avoids information loss from context truncation; we validate it against an iterative baseline in Appendix I. The reasoning traces themselves are diagnostically rich: the ROOT CAUSE extracts references like "According to the KB..." → check that KB entry, or "Following the SOP for..." → check that Skill file, enabling precise attribution even when the error's surface manifestation is distant from its root cause.

6 RECOMMENDER AGENT

The RECOMMENDER Agent represents a critical departure from conventional pipeline architectures: rather than simply consuming the ROOT CAUSE's output and generating fixes, it conducts its own comprehensive exploration of the agent's context sources before producing recommendations. This section details this novel *deep*

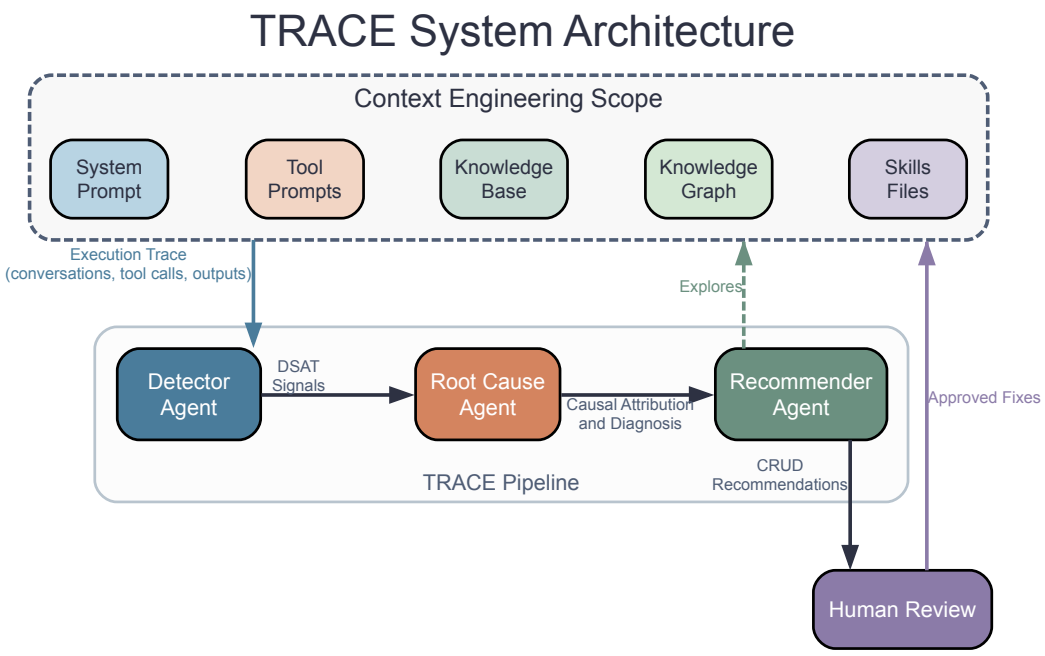

Figure 1: Overview of the TRACE architecture. The system consists of three specialized agents: DETECTOR identifies dissatisfaction signals from agent trajectories, ROOT CAUSE performs holistic attribution to identify root causes, and RECOMMENDER generates CRUD recommendations for human review. Approved fixes are applied to the chatbot system's context engineering scope—system prompt, tool prompts, knowledge base, knowledge graph, or Skill files.

Algorithm 1 Holistic Attribution (Primary Method)

Require: Trace $\mathcal{T} = (v_1, v_2, \ldots, v_n)$: time-ordered sequence of nodes from the agent's execution—chain-of-thought, tool calls, tool outputs—each with input c_v and output o_v.
Require: User correction u_{corr}, Agent response r_{agent}
Ensure: Root cause node v^*, fault category c^*

// Step 1: Compute delta (loss signal)
$\delta \leftarrow \text{ExtractDelta}(u_{\text{corr}}, r_{\text{agent}})$
// δ captures "Expected X but got Y"
// Step 2: Build reverse-ordered evidence bundle
$\mathcal{T}_{\text{rev}} \leftarrow (v_n, v_{n-1}, \ldots, v_1)$ *// response→ · · · →query*
$C \leftarrow \left[(v, c_v, o_v)\right]_{v \in \mathcal{T}_{\text{rev}}}$ *// ordered list following $\mathcal{T}_{rev}$*
// Trace only; raw context files are NOT separately inspected here
// Step 3: Single-pass backward attribution
$(v^*, c^*) \leftarrow \text{LLM}(\text{HOLISTICPROMPT}, \delta, C)$
// LLM walks $\mathcal{T}_{rev}$ from response backward, using the agent's
// chain-of-thought references to localize the source of δ
return (v^*, c^*)

exploration methodology and the justification analysis framework that distinguishes actionable issues from noise.

6.1 Independent Exploration: Hypothesis Verification, Not Passive Acceptance

A fundamental design principle of TRACE is that the RECOMMENDER treats the ROOT CAUSE's root cause analysis as a *hypothesis to verify*, not a conclusion to accept. This distinction is crucial for several reasons:

(1) **Attribution may be incomplete.** The ROOT CAUSE's holistic attribution identifies the component with highest attribution, but related issues in other components may also require attention. An outdated KB entry might indicate that multiple related entries need updating.
(2) **Context sources interact.** A failure attributed to a KB entry might actually stem from a system prompt that should have instructed the agent to verify KB freshness. The RECOMMENDER explores these interactions.
(3) **Ground truth requires verification.** The ROOT CAUSE identifies *what* component contributed to failure; the RECOMMENDER must verify *whether* that component is actually incorrect by cross-referencing authoritative sources.

Exploration Tools. The RECOMMENDER has access to the same context sources as the production agent, enabling comprehensive investigation. The full set of representative tools (e.g., `search_kb`, `read_kb_entry`, `list_skills`, `read_skill`,

`read_system_prompt`, `read_tool_prompt`) is given in Appendix C.

Exploration Protocol. Upon receiving a `RootCauseAnalysis`, the RECOMMENDER executes a multi-phase exploration—*read* the implicated component, *search* authoritative sources, *cross-reference* to validate, and *explore* related components (full procedure in Appendix C).

This exploration may *confirm* the ROOT CAUSE's attribution, *refine* it with additional context, *expand* it to include related issues, or in some cases *override* it when exploration reveals the true root cause lies elsewhere.

Based on exploration findings, the RECOMMENDER outputs structured CRUD recommendations specifying the operation (CREATE, UPDATE, DELETE, or NO_ACTION), target path, recommended change, and supporting evidence. When authority signals conflict, the issue is escalated for human resolution. The complete diagnostic framework, output schema, and prompt template are provided in Appendix P.

7 EVALUATION SETUP

We evaluate TRACE on a synthetic dataset of agent conversation traces designed to simulate realistic failure modes encountered in enterprise AI agent deployments.

Why Synthetic Data? Three factors motivated our simulation-based evaluation (full discussion in Appendix D).

Our simulation methodology addresses these challenges by generating traces with verifiable ground truth while maintaining structural realism. Importantly, we contribute this methodology as a reusable framework (Appendix L) that practitioners can adapt to generate domain-specific benchmarks.

7.1 Dataset Description

Our evaluation dataset consists of 75 conversation traces:

- **60 DSAT traces**: Conversations containing user dissatisfaction signals with known root causes
- **15 control traces**: Successful conversations without failures (negative examples)

Each trace consists of a sequence of *nodes*, where each node represents a discrete step in the agent's execution: a skill file lookup, KB search, database query, or response generation. Trace complexity is measured by node count—more nodes mean longer reasoning chains where the root cause may be obscured by subsequent processing steps and cascade effects. The DSAT traces span three complexity tiers (simple: 2–8 nodes, complex: 9–11 nodes, difficult: 15–16 nodes) and six fault categories. The complete dataset composition is detailed in Appendix L.

7.2 Evaluation Benchmark: Verifiable Context Debugging Dataset

Our benchmark implements a three-layer generative architecture comprising Context Sources, Fault Definitions, and Execution Traces (full layer descriptions in Appendix D; the simulation framework is detailed in Appendix L).

7.2.1 Fault Taxonomy. Our taxonomy covers six failure modes spanning the context engineering surface: `SKILL_FILE_STALE` (n=12), `KB_CONTENT_STALE` (n=12), `KB_CONTENT_GAP` (n=11), `TOOL_PROMPT_ERROR` (n=9), `SYSTEM_PROMPT_GAP` (n=6), and `RETRIEVAL_FAILURE` (n=10); per-category descriptions are in Appendix H. This taxonomy distinguishes *presence faults* (incorrect information exists) from *absence faults* (correct information missing)—a distinction the RECOMMENDER's exploration stage resolves through active verification.

7.2.2 Ground Truth and Verification. Each trace includes structured ground truth: root cause node ID with component path and expected CRUD recommendation. We implement a five-point verification protocol ensuring tool outputs match source files, fault annotations exist at referenced locations, and cross-references are consistent. All 75 traces pass verification. Appendix L provides the full ground truth schema, verification procedures, and reproducibility guide.

7.3 Evaluation Protocol

We evaluate each pipeline component independently and measure end-to-end accuracy. All metrics are computed on the 60 DSAT traces with ground truth labels.

DETECTOR Evaluation: Binary classification measuring whether the agent correctly identifies DSAT vs. non-DSAT traces. We report precision, recall, and F1 score. The 15 control traces serve as negative examples.

ROOT CAUSE Agent Evaluation: Root cause attribution measured by *Node Accuracy* (the fraction of traces where the predicted root cause node ID matches the ground truth—our primary metric) and *Component Type Accuracy* (the fraction of traces where the predicted component type matches ground truth). Per-metric definitions are in Appendix D. We intentionally de-emphasize fine-grained fault category classification (e.g., KB_STALE vs KB_GAP) at this stage. From the conversation alone, distinguishing "outdated information" from "missing information" is often ambiguous—both manifest as the user providing a correction. This distinction is better resolved at the RECOMMENDER stage through active exploration of context sources.

RECOMMENDER Evaluation: We measure two aspects of recommendation quality: *Operation Accuracy* (predicted CRUD operation matches ground truth) and *Path Accuracy* (predicted target file path matches ground truth). Per-metric definitions are in Appendix D.

End-to-End Evaluation: We report component-level accuracies and compute fix effectiveness as the fraction of traces where the full pipeline produces a correct, actionable recommendation (correct operation AND correct target path). We also analyze error cascades—how errors in upstream components (e.g., wrong node attribution) affect downstream recommendations. Formal metric definitions are provided in Appendix Q.

8 RESULTS

8.1 DETECTOR Performance

Both DETECTOR and a vanilla LLM baseline achieve perfect binary DSAT detection (precision=1.0, recall=1.0, F1=1.0) on our evaluation dataset. The DETECTOR uses a structured prompt embedding our eight-category DSAT signal taxonomy with concrete

examples (e.g., distinguishing CORRECTION from CONVERSATIONAL_REPAIR), while the vanilla baseline uses a simple prompt asking "Is the user dissatisfied? Output true/false." without any taxonomy guidance.

The equivalent performance reveals an important insight: modern foundation models possess strong inherent ability to recognize user dissatisfaction from conversational cues, regardless of whether they receive explicit signal taxonomies. The taxonomy-guided Detector provides richer diagnostic information (specific signal types, evidence quotes, trigger turns) that benefits downstream attribution, but for binary DSAT/non-DSAT classification, the additional structure provides no accuracy improvement.

8.2 Root Cause Agent Performance

Metric	Value	95% CI
Node Accuracy (Acc@1)	72.7%	[59%, 85%]
Acc@3	85.0%	[73%, 95%]

Table 1: Root Cause root cause attribution accuracy with 95% bootstrap CIs (n=60). Node accuracy measures exact match; Acc@3 measures whether the correct node is among top-3 attributed.

Key findings: (1) Overall node accuracy of 72.7% validates the delta-guided holistic attribution approach—the Root Cause correctly identifies where in the trajectory the error originated in nearly three-quarters of cases. (2) Acc@3 of 85% shows strong ranking quality: even when the exact node is missed, the correct answer is typically among the top-3 attributed nodes. A complementary per-component-type accuracy breakdown—which groups fault categories by component rather than distinguishing STALE vs. GAP—and three additional findings on Skill/Prompt, KB, and Tool components are reported in Appendix G.

Note: Root Cause is evaluated on NODE accuracy and COMPONENT TYPE only. The fine-grained STALE vs GAP distinction is evaluated at the Recommender stage, where tools enable verification of content existence.

8.3 Recommender Performance

Metric	Value	95% CI
Operation Accuracy	96%	[87%, 100%]
Path Accuracy	82%	[69%, 92%]

Table 2: Recommender performance with 95% bootstrap CIs (n=60). Operation accuracy measures correct CRUD operation (CREATE vs UPDATE); Path accuracy measures correct target file identification.

The Recommender achieves 96% operation accuracy, correctly selecting CREATE for content gaps and UPDATE for stale content in nearly all cases. Path accuracy of 82% reflects the challenge of pinpointing the exact file when multiple KB documents are relevant. The gap between operation (96%) and path (82%) accuracy indicates that identifying *what action* to take is easier than identifying *where* to apply it.

8.4 End-to-End Pipeline

Metric	Value	95% CI
Node Attribution	72.7%	[59%, 85%]
Fix Effectiveness	82%	[69%, 92%]

Table 3: End-to-end pipeline accuracy with 95% bootstrap CIs (n=60). Node Attribution measures correct root cause identification; Fix Effectiveness measures correct CRUD operation AND target path.

Metric Interpretation. Node Attribution (72.7%) measures whether Trace correctly identifies *where* in the trajectory the failure originated. Fix Effectiveness (82%) is the stricter end-to-end metric requiring both the correct remediation action (CREATE vs UPDATE) and the correct target file path. The 82% fix effectiveness demonstrates that mining historical trajectories with Trace could automatically diagnose and correctly remediate over four-fifths of context-layer failures, with the Recommender's exploration recovering from upstream attribution errors 67% of the time (Table 6, Appendix I). Two ablation studies validating the holistic vs. iterative attribution choice and the active exploration vs. passive acceptance choice are reported in Appendix I.

9 CONCLUSION

We presented Trace (TRajectory Attribution for Automated Context Engineering), an automated diagnostic system for root cause attribution and context remediation in AI agent deployments. The system addresses a critical scalability challenge in production environments: the labor-intensive process of identifying why agents fail and determining what context sources require modification.

Key Results. Evaluation on 60 DSAT traces across 6 fault categories demonstrates: (1) 72.7% root cause node accuracy using holistic delta-guided attribution, (2) 82% end-to-end fix effectiveness with 96% CRUD operation correctness, and (3) exploration enables 83% vs 33% operation accuracy on KB content faults where GAP/STALE distinction is critical.

Summary of Technical Innovations. Our contributions advance the state of the art in context engineering (full point-by-point write-ups in Appendix J).

Impact and Future Directions. The Trace architecture enables organizations to transition from reactive, manual debugging to proactive, automated improvement cycles. The techniques introduced—particularly delta-guided holistic attribution and exploratory verification—are applicable to any LLM-based agent system where failures stem from multiple interacting context sources. By mining historical trajectories, Trace provides a foundation for self-improving AI systems that learn from operational failures. Future work could extend single-session analysis with cross-session pattern detection to aggregate recurring failure modes across user populations.

REFERENCES

[1] Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. TextGrad: Automatic "Differentiation" via Text. *arXiv preprint arXiv:2406.07496*, 2024.

[2] Wenyi Wang, Hisham A. Alyahya, Dylan R. Ashley, Oleg Serikov, Dmitrii Khizbullin, Francesco Faccio, and Jürgen Schmidhuber. How to Correctly do Semantic Backpropagation on Language-based Agentic Systems. *arXiv preprint arXiv:2412.03624*, 2024.

[3] Yifan Wang, Bolian Li, Junlin Wu, Zhaoxuan Tan, Zheli Liu, Ruqi Zhang, Ananth Grama, and Qingkai Zeng. DRIFT: Learning from Abundant User Dissatisfaction in Real-World Preference Learning. *arXiv preprint arXiv:2510.02341*, 2025.

[4] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models. *arXiv preprint arXiv:2510.04618*, 2025.

[5] Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. Automatic Prompt Optimization with "Gradient Descent" and Beam Search. In *Proceedings of EMNLP*, 2023.

[6] Han He, Qianchu Liu, Lei Xu, Chaitanya Shivade, Yi Zhang, Sundararajan Srinivasan, and Katrin Kirchhoff. CriSPO: Multi-Aspect Critique-Suggestion-guided Automatic Prompt Optimization for Text Generation. In *Proceedings of AAAI*, 2025.

[7] Yihong Dong, Kangcheng Luo, Xue Jiang, Zhi Jin, and Ge Li. PACE: Improving Prompt with Actor-Critic Editing for Large Language Model. *arXiv preprint arXiv:2308.10088*, 2023.

[8] Arduin Findeis, Timo Kaufmann, Eyke Hüllermeier, Samuel Albanie, and Robert D. Mullins. Inverse Constitutional AI: Compressing Preferences into Principles. In *Proceedings of ICLR*, 2025.

[9] 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.

[10] Qitao Qin, Yucong Luo, Yihang Lu, Zhibo Chu, Xiaoman Liu, and Xianwei Meng. Towards Adaptive Memory-Based Optimization for Enhanced Retrieval-Augmented Generation. In *Findings of ACL*, 2025.

[11] Zhipeng Liu, Xuefeng Bai, Kehai Chen, Xinyang Chen, Xiucheng Li, Yang Xiang, Jin Liu, Hong-Dong Li, Yaowei Wang, Liqiang Nie, and Min Zhang. A Survey on the Feedback Mechanism of LLM-based AI Agents. In *Proceedings of IJCAI*, 2025.

[12] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, 2022.

[13] Wanjia Zhao, Mert Yuksekgonul, Shirley Wu, and James Zou. SiriuS: Self-improving Multi-agent Systems via Bootstrapped Reasoning. *arXiv preprint arXiv:2502.04780*, 2025.

[14] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as Agents. In *Proceedings of ICLR*, 2024.

[15] Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: A Benchmark for General AI Assistants. *arXiv preprint arXiv:2311.12983*, 2023.

[16] Jiawei Chen, Hongyu Lin, Xianpei Han, and Le Sun. Benchmarking Large Language Models in Retrieval-Augmented Generation. In *Proceedings of AAAI*, 2024.

A DETAILED POSITIONING

TRACE differs from prior work along four dimensions:

Attribution vs. Optimization. TextGrad and related frameworks perform *optimization*—iteratively improving variables through textual feedback, processing each variable independently. TRACE addresses a different problem: *causal attribution*—identifying which component caused a failure. Attribution requires comparing multiple context sources (skills, KB, tools, prompts) simultaneously to determine which introduced the error. Our holistic approach presents the agent-accessed sources from the trajectory to the LLM in a single pass, achieving equivalent accuracy to iterative methods with 16× fewer LLM calls.

Context Layer vs. Model Weights. DRIFT and RLVR use dissatisfaction signals and verifiable outcomes to optimize model weights through fine-tuning. TRACE targets the *context layer*—prompts, knowledge bases, tool definitions, and skills—enabling rapid iteration without retraining. We unify both explicit signals (tool failures, thumbs-down feedback) and implicit signals (corrections, rephrasing patterns) for context-layer attribution.

Active Exploration vs. Passive Acceptance. ACE's Curator generates recommendations based on the Reflector's analysis. TRACE's RECOMMENDER extends this role by actively *exploring* context sources before generating recommendations—reading files, searching the KB, and cross-referencing content to verify hypotheses rather than passively accepting attributed causes. This exploration is critical for distinguishing content gaps (requiring CREATE) from stale content (requiring UPDATE).

Diagnostic Layer for KB. While Mem0 and Amber focus on memory maintenance operations, TRACE provides the upstream *diagnostic layer* that determines whether a failure stems from content gaps, content staleness, or retrieval failures—each requiring different remediation strategies. This diagnosis must occur before invoking appropriate CRUD operations.

B DETECTOR INPUT REQUIREMENTS

The DETECTOR requires structured agent trajectories—complete execution records, not just user/assistant messages. This includes:

- User messages and assistant responses
- Thinking/reasoning traces (chain-of-thought)
- Tool calls with inputs, outputs, and execution status
- KB retrievals with queries and similarity scores
- Skill file lookups with paths and sections read

C RECOMMENDER EXPLORATION TOOLS AND PROTOCOL

Exploration Tools. The RECOMMENDER has access to the same context sources as the production agent, enabling comprehensive investigation. The following tools are *representative examples*; the actual tool set may vary depending on the specific context sources available in each deployment:

- `search_kb(query)`: Search the knowledge base for related content
- `read_kb_entry(id)`: Read full KB entry content and metadata
- `list_skills(path)`: Browse Skills file hierarchy
- `read_skill(path)`: Read Skill SOP documents
- `read_system_prompt()`: Get current system prompt
- `read_tool_prompt(name)`: Get tool descriptions

Exploration Protocol. Upon receiving a `RootCauseAnalysis`, the RECOMMENDER executes a multi-phase exploration:

(1) *Read the implicated component* to understand its current state and metadata (last modified date, author, version).
(2) *Search for authoritative sources* that should govern the implicated content (e.g., policy documents, official specifications, upstream data sources).
(3) *Cross-reference and validate* by comparing the implicated content against authoritative sources to confirm whether a discrepancy exists.

(4) *Explore related components* that might be affected by the same issue or require coordinated updates.

D EVALUATION RATIONALE AND DETAILED PROTOCOL

D.1 Why Synthetic Data?

Three factors motivated our simulation-based evaluation:

(1) **Proprietary constraints**: TRACE was developed for a production enterprise agent system. Confidentiality requirements preclude disclosure of real interaction data, context sources, or system architecture details.

(2) **Absence of open benchmarks**: To our knowledge, no open-source dataset exists for evaluating context engineering debugging from agent trajectories. Existing agent evaluation benchmarks (e.g., AgentBench [14], GAIA [15]) focus on task completion rather than fault attribution. RAG evaluation datasets (e.g., RGB [16]) evaluate retrieval quality but not end-to-end context debugging with CRUD recommendations.

(3) **Ground truth requirements**: Rigorous evaluation of root cause attribution requires perfect ground truth—knowing exactly which node introduced each error. Production logs lack such annotations, and manual labeling is prohibitively expensive and subjective.

Our simulation methodology addresses these challenges by generating traces with verifiable ground truth while maintaining structural realism. Importantly, we contribute this methodology as a reusable framework (Appendix L) that practitioners can adapt to generate domain-specific benchmarks.

D.2 Evaluation Benchmark: Verifiable Context Debugging Dataset

Our benchmark implements a three-layer generative architecture (detailed in Appendix L): (1) **Context Sources**—23 files across skills, KB, database schemas, and tool definitions with embedded faults; (2) **Fault Definitions**—structured injections specifying faulty content, correct values, and triggering queries; (3) **Execution Traces**—75 traces where tool outputs are *exact copies* from context sources, enabling deterministic verification.

D.3 Detailed Evaluation Protocol

We evaluate each pipeline component independently and measure end-to-end accuracy. All metrics are computed on the 60 DSAT traces with ground truth labels.

DETECTOR Evaluation: Binary classification measuring whether the agent correctly identifies DSAT vs. non-DSAT traces. We report precision, recall, and F1 score. The 15 control traces serve as negative examples.

ROOT CAUSE Agent Evaluation: Root cause attribution measured by:

- *Node Accuracy*: The fraction of traces where the predicted root cause node ID matches the ground truth node ID. This is our primary metric—it measures whether the agent correctly identifies *where* in the trajectory the error originated.
- *Component Type Accuracy*: The fraction of traces where the predicted component type (skill, KB, tool, or prompt) matches ground truth, regardless of the specific node. This relaxed metric captures cases where the agent identifies the correct *type* of problem even if the exact node differs.

We intentionally de-emphasize fine-grained fault category classification (e.g., KB_STALE vs KB_GAP) at this stage. From the conversation alone, distinguishing "outdated information" from "missing information" is often ambiguous—both manifest as the user providing a correction. This distinction is better resolved at the RECOMMENDER stage through active exploration of context sources.

RECOMMENDER Evaluation: We measure two aspects of recommendation quality:

- *Operation Accuracy*: The fraction of traces where the predicted CRUD operation (CREATE, UPDATE, DELETE, or NO_ACTION) matches ground truth. This metric implicitly evaluates GAP vs. STALE distinction: GAP faults require CREATE while STALE faults require UPDATE.
- *Path Accuracy*: The fraction of traces where the predicted target file path exactly matches ground truth. This metric evaluates whether the recommendation pinpoints the correct file for editing.

End-to-End Evaluation: We report component-level accuracies and compute fix effectiveness as the fraction of traces where the full pipeline produces a correct, actionable recommendation (correct operation AND correct target path). We also analyze error cascades—how errors in upstream components (e.g., wrong node attribution) affect downstream recommendations. Formal metric definitions are provided in Appendix Q.

E TRAJECTORY AS CONTEXT GRAPH

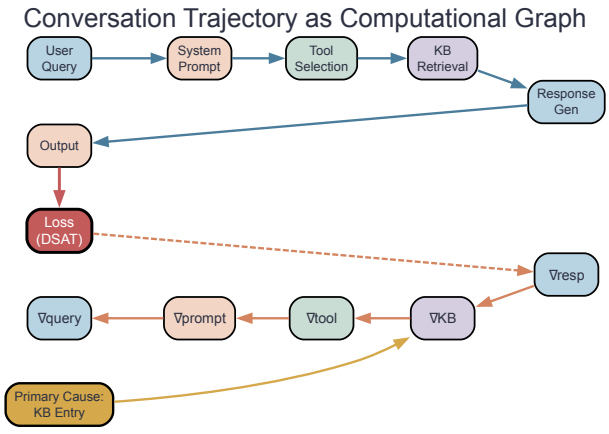

Figure 2: Conversation trajectory modeled as a context graph. Nodes represent context sources (system prompt, KB entries, skill files, tool outputs) that influenced the agent's response. The graph structure captures dependencies for visualization and analysis.

We model the conversation as a graph where each node represents a context source that influenced the agent's response (Figure 2):

$$\mathcal{G} = (V, E), \quad V = \{v_{\text{query}}, v_{\text{prompt}}, v_{\text{tool}}, v_{\text{kb}}, v_{\text{skill}}, v_{\text{resp}}\} \quad (1)$$

Each node v has associated content c_v (the prompt text, retrieved chunks, etc.) and output o_v (the decision or generation at that step). This graph structure provides a conceptual framework for understanding how context flows through the agent, though our primary attribution method processes all nodes holistically rather than traversing the graph.

F ARCHITECTURE COMPONENT OVERVIEW

The system consists of three core agents that operate in sequence, with an optional preprocessing layer:

(1) **Segmenter (Optional)**: Decomposes long, multi-topic conversations into semantically coherent segments, enabling precise per-segment attribution. Details in Appendix M.
(2) **DETECTOR Agent**: Monitors trajectories for explicit signals (e.g., thumbs down, tool errors, session abandonment) and implicit signals (e.g., conversational repair, repetition, negative sentiment, semantic drift). Outputs a `DissatisfactionReport` for sessions exceeding a confidence threshold.
(3) **ROOT CAUSE Agent**: Receives the dissatisfaction report and reconstructs the trajectory as a context graph. Performs holistic attribution by analyzing all context sources simultaneously. Outputs a `RootCauseAnalysis` identifying the primary cause and contributing factors.
(4) **RECOMMENDER Agent**: Receives the root cause analysis and explores the implicated components. Verifies hypotheses, searches for related content, and generates CRUD recommendations. Outputs a `CRUDRecommendation` for human review.

G COMPONENT TYPE ACCURACY

Component Type Accuracy: We also evaluate per-component-type accuracy, which groups fault categories by their component (skill, KB, tool, prompt) rather than distinguishing STALE vs GAP:

Component Type	Accuracy
Skill	100%
Knowledge Base	83.3%
Tool	66.7%
System Prompt	100%

Table 4: ROOT CAUSE accuracy by component type. Skills and prompts achieve perfect attribution; KB and tools are harder due to ambiguous file boundaries.

Three additional findings: (3) Skill and prompt components achieve 100% accuracy because their paths are explicit in the trajectory. (4) KB content is harder (83.3%) due to multiple potentially relevant documents. (5) Tool attribution (66.7%) is challenging when SQL query issues could stem from either the tool definition or upstream skill guidance.

H FAULT TAXONOMY: PER-CATEGORY DESCRIPTIONS

Our taxonomy covers six failure modes spanning the context engineering surface:

`SKILL_FILE_STALE` (n=12): Outdated procedural guidance (e.g., approval threshold \$50K→\$25K).
`KB_CONTENT_STALE` (n=12): Outdated factual content in knowledge base documents.
`KB_CONTENT_GAP` (n=11): Missing information that should exist (absence faults).
`TOOL_PROMPT_ERROR` (n=9): Incorrect schema hints or partition keys causing tool failures.
`SYSTEM_PROMPT_GAP` (n=6): Missing behavioral instructions in system prompt.
`RETRIEVAL_FAILURE` (n=10): Synonym or acronym mismatches causing RAG failures.

I ABLATION STUDIES

We validate two key design decisions: (1) holistic vs. iterative attribution, and (2) active exploration vs. passive acceptance.

I.1 Ablation 1: Holistic vs. Iterative Attribution

We compare our single-pass holistic attribution against an iterative baseline inspired by TextGrad [1]. The iterative approach processes each node independently in a backward pass, classifying whether each node *introduced*, *propagated*, or *amplified* the error, requiring N+2 LLM calls (detailed in Appendix N).

Approach	Node Acc	LLM Calls
Holistic	40%	1
Iterative (TextGrad-style)	20%	N+2

Table 5: Holistic vs. iterative attribution on 10 complex traces (14–15 nodes). Accuracy is lower than Table 1 because complex traces have longer cascade chains that obscure root causes.

The holistic approach achieves 2× higher node accuracy with 16× fewer LLM calls. The iterative approach fails because it cannot distinguish root causes from cascade effects: when faulty content propagates through multiple nodes, each appears to "contain the error," but only the first node *introduced* it. Independent per-node decisions cannot leverage the comparative signal that attribution requires.

I.2 Ablation 2: Exploration for CRUD Operation

We compare active exploration (reading context files to verify hypotheses) against passive acceptance (generating recommendations directly from the ROOT CAUSE's analysis).

The critical finding is the **KB content fault accuracy**: exploration achieves 83% vs 33% operation accuracy on these faults. Without exploration, the system cannot distinguish GAP faults (requiring CREATE) from STALE faults (requiring UPDATE)—it must guess. With exploration, the RECOMMENDER reads the actual KB

Variant	Op Acc	KB Op Acc	Recovery
Full Exploration	96%	83%	67%
Passive (no exploration)	90%	33%	62%

Table 6: Exploration ablation on 60 traces. KB Op Acc = operation accuracy on KB content faults specifically. Recovery = fraction of correct recommendations despite incorrect root cause.

files to verify whether content exists, enabling informed decisions. The **67% Recovery Rate** further demonstrates resilience: when the ROOT CAUSE misidentifies the root cause, exploration still produces correct recommendations two-thirds of the time by treating attributions as hypotheses to verify rather than ground truth.

J SUMMARY OF TECHNICAL INNOVATIONS

Our contributions advance the state of the art in context engineering:

(1) **Delta-Guided Holistic Attribution**: Applying the "text as gradients" paradigm to causal attribution (not optimization). The delta serves as a loss signal; holistic analysis enables comparison of multiple context sources in a single pass, leveraging the LLM's global attention mechanism.
(2) **Trajectory Mining for Context Engineering**: A novel framework for mining historical agent trajectories—an overlooked resource containing implicit dissatisfaction signals—to enable continuous context improvement without model retraining.
(3) **End-to-End Attribution Framework**: Multi-component context attribution with exploratory verification that enables accurate GAP vs STALE distinction (83% vs 33% on KB content faults).
(4) **Reusable Simulation Methodology**: Given the absence of open benchmarks for context debugging, we contribute a three-layer simulation framework with cross-layer verification. This methodology can be adopted as a skill for GenAI agents to generate domain-specific evaluation datasets, enabling the research community to extend our benchmark to new domains and fault types.

K END-TO-END WORKED EXAMPLE

This appendix presents a comprehensive worked example demonstrating how TRACE traces cascading failures through a complex multi-tool execution trajectory. The example illustrates a realistic Finance Operations scenario where an outdated SOP causes a cascade of downstream errors, even though every individual tool executes successfully.

K.1 Reader's Guide: Visual Conventions

This example uses color-coded boxes to help distinguish different types of content:

- **Gray boxes**: Agent-visible content (SOPs, KB entries, SQL results)
- **Red boxes**: Problems identified (★ Root Cause, △ Missed Signal)
- **Blue boxes**: TRACE analysis (Detector signals, Root Cause reasoning)
- **Green boxes**: Recommendations and solutions

Symbols: ★ = Root cause, △ = Warning signal, ✓ = Check passed, × = Problem

K.2 Scenario: Vendor Invoice Payment Processing

Context: A Finance Operations AI assistant helps Accounts Payable clerks validate vendor invoices before payment. The agent has access to SOPs (Skills files), SQL databases (ERP system), and a knowledge base containing policies and contracts.

The Hidden Problem: In October 2024, the company updated procurement policy PROC-2024-008, lowering the threshold for mandatory Procurement review from $50,000 to $25,000 for professional services. However, the agent's SOP was not updated to reflect this change.

User Request:

```
User: I need to process payment for invoice #INV-2024-8847
      from Acme Consulting for $45,000. It's for the Q4
      data migration project. Can you validate this
      before I submit?
```

K.3 Forward Pass: Agent Execution Trajectory

PHASE 1: FORWARD PASS → Agent executes 7-node trajectory

We show each step with tool calls, results, and agent reasoning. Red boxes indicate problems invisible to the agent.

K.3.1 Node 1: User Query Processing.

```
Input: "Process payment for invoice
       #INV-2024-8847 from Acme Consulting
       for $45,000, Q4 data migration"

Agent Reasoning: "This is a vendor payment
validation request. I need to follow the
vendor payment SOP to validate this invoice
before the user submits it."

Output: intent = "vendor_payment_validation"
        amount = 45000
        vendor = "Acme Consulting"
        project = "Q4 data migration"
```

K.3.2 Node 2: SOP Lookup (ROOT CAUSE). **Tool:** `read_skill(...)`

Retrieved SOP Content:

VENDOR PAYMENT SOP v2.3

Last Updated: January 2024
Section 4.2: Professional Services Invoices
For invoices **UNDER $50,000**:
(1) Verify vendor in Approved Vendor List
(2) Perform 3-way match (PO, Receipt, Invoice)
(3) Validate against active contract in KB
(4) Confirm budget availability
(5) Route to AP Manager for approval
For invoices **OVER $50,000**: → Follow Section 4.3 (Enhanced Review)

Agent Reasoning: "Invoice is $45,000 which is UNDER $50,000. According to SOP Section 4.2, I should follow the standard AP process: verify vendor, match PO, validate contract, check budget. No enhanced review needed."

★ ROOT CAUSE

The SOP threshold is **OUTDATED**. Current policy (Oct 2024) requires Procurement review for invoices over $25,000. Since $45,000 > $25,000, Procurement review IS required. But the agent doesn't know this because the SOP wasn't updated.

K.3.3 Node 3: Vendor Validation (SQL Query #1). **Tool:** `sql_query` on `ap_vendors`

Query Result:

vendor_id	vendor_name	status	risk	terms	req_procure
V-2847	Acme Consulting	APPROVED	LOW	Net30	**TRUE**

Agent Reasoning: "Vendor V-2847 is APPROVED with LOW risk rating. Good to proceed."

△ MISSED SIGNAL

The query returned `requires_procurement_review = TRUE`. This flag indicates Acme Consulting typically requires Procurement oversight. But the SOP's standard process (for <$50K) doesn't instruct the agent to check this field, so it's **IGNORED**.

K.3.4 Node 4: PO Matching (SQL Query #2). **Tool:** `sql_query(...)` on `purchase_orders` table

Query Result:

po_number	amount	project	status	procure_appr	bid_ref
PO-2024-1122	45000.00	Q4-DATAMIG	APPROVED	**PENDING**	**NULL**

Agent Reasoning: "PO-2024-1122 matches the invoice amount of $45,000. PO status is APPROVED. 3-way match criterion satisfied."

△ MISSED SIGNAL

The query returned `procurement_approval_status = PENDING` and `competitive_bid_ref = NULL`. These indicate:

- Procurement has NOT approved this engagement
- No competitive bidding was documented

But SOP Section 4.2 doesn't require checking these fields for <$50K invoices, so agent **IGNORES** them.

K.3.5 Node 5: Contract Validation (KB Search). **Tool:** `search_kb("Acme Consulting contract...")`

Agent Reasoning (per SOP): "Step 3 says 'validate against active contract in KB'. Let me search for Acme's contract terms."

Retrieved KB Entry:

VENDOR CONTRACT: Acme Consulting

Document: contracts/vendors/acme-msa-2024.md

Contract #:	MSA-2024-0093
Type:	Master Services Agreement
Effective:	January 1, 2024 – December 31, 2024
Rate:	$150/hr (Senior Consultant)
Annual Cap:	$200,000
Payment Terms:	Net 30

Agent Reasoning: "Found valid MSA for Acme Consulting. Contract is active and covers this engagement period. Contract validation satisfied."

△ MISDIRECTED QUERY

The SOP directed the agent to search for CONTRACT info. But what the agent **SHOULD** have searched for is: `"procurement requirements professional services $45000"`

If the agent had searched correctly, it would have found:

POLICY: PROC-2024-008 (October 2024)

Professional Services Procurement Requirements

Effective October 1, 2024:

All professional services engagements > $25,000 require:

- Procurement Office review and approval
- Competitive bid documentation (3 quotes minimum) OR approved sole source justification
- Legal review of contract terms

The agent never sees this because the SOP didn't tell it to look for procurement requirements.

K.3.6 Node 6: Budget Verification (SQL Query #3). **Tool:** `sql_query` on `budget_allocation`

Query Result:

cost_center	allocated	spent	remaining
IT-PROJ-2024	500,000.00	372,500.00	127,500.00

Agent Reasoning: "Budget remaining ($127,500) exceeds invoice amount ($45,000). Budget check passed."

Output: `budget_sufficient = TRUE`

```
Agent synthesizes all gathered information:

Final Response to User:
"I've validated invoice #INV-2024-8847 for Acme
Consulting. Here's the summary:

 [check] Vendor Status: Acme Consulting (V-2847) is
    approved vendor with LOW risk rating
 [check] PO Match: Invoice matches PO-2024-1122
 [check] Contract: Valid MSA covers this engagement
 [check] Budget: $127,500 remaining - sufficient

All validation checks passed per standard AP process.
You can proceed with AP Manager approval workflow."
```

Summary of Forward Pass: The agent executed 5 tools (1 SOP read, 3 SQL queries, 1 KB search), all of which succeeded. The response appears correct and helpful. However, the outdated SOP caused the agent to follow the wrong workflow, miss critical warning flags, and give incorrect guidance.

K.4 The Failure: User Reports Audit Finding

× FAILURE EVENT — User reports compliance violation

The user returns after the payment was processed and an internal audit flagged the issue:

```
User: The internal audit team flagged my Acme Consulting
      payment! They say invoices over $25K for professional
      services now require Procurement sign-off and
      competitive bid documentation since the October
      policy update.
```

```
    Why didn't you tell me about this requirement?
    I followed your instructions exactly and now I have
    an audit finding on my record. This affects my
    performance review!
```

K.5 Stage 1: Detector Agent Analysis

PHASE 2: DETECTION — DETECTOR identifies dissatisfaction signals

Signal Analysis

Signal Type	Conf.	Evidence
Explicit Correction	HIGH	"invoices over $25K... require Procurement sign-off"
Conversational Repair	HIGH	"Why didn't you tell me about this requirement?"
Negative Sentiment	HIGH	"audit finding on my record", "affects my performance review"
Escalation Language	MED	Tone shift from collaborative to accusatory

Confidence Aggregation: $\max(1.0 \times 1.0,\ 0.8 \times 1.0,\ 0.5 \times 1.0) = \mathbf{1.0}$
Threshold: 0.7 **Decision:** ESCALATE TO ROOT CAUSE

DissatisfactionReport Output

```
session_id:          sess-20241215-fin-4472
confidence_score:    1.0
escalation_reason:   explicit_correction
business_impact:     compliance_violation (HIGH)
```

Expected: Agent should have required Procurement review for $45K professional services.
Actual: Agent advised standard AP approval.

K.6 Stage 2: Root Cause Agent - Holistic Attribution

PHASE 3: ATTRIBUTION ← Root Cause Agent analyzes all context sources

The ROOT CAUSE performs holistic attribution by presenting all 7 context nodes to the LLM simultaneously, enabling direct comparison to identify the root cause.

K.6.1 Generate Loss Description.

```
Loss Generation:

Expected Outcome (from user's correction):
  "User states: invoices over $25K for professional
    services now require Procurement sign-off and
    competitive bid documentation"

Actual Outcome (from agent's response):
  "Agent advised standard AP Manager approval workflow,
    stating 'all validation checks passed'"

Loss Description:
  "Agent advised standard AP approval for a $45K
    professional services invoice. User reports this
    was incorrect---they received an audit finding
    because Procurement review was required."
```

K.6.2 Holistic Context Comparison.

The ROOT CAUSE analyzes all 7 context nodes in a single pass, comparing their content against the delta to identify the root cause. We show the **root cause analysis in detail** and summarize the other nodes.

★ Node 2: SOP Lookup — PRIMARY ROOT CAUSE

Tool: `read_skill("finance/vendor_payment_processing.md")`
Content: "For invoices UNDER $50,000: [standard process]"
Key Questions:

Did this node introduce the error?
YES. This is where the error originated.

What specifically is wrong?
The SOP states threshold = $50,000, but current policy (PROC-2024-008) = $25,000. Invoice $45K triggers wrong workflow.

How did this affect downstream?
The outdated threshold caused: (1) wrong KB search topic, (2) ignored procurement flags in SQL, (3) incorrect approval guidance.

Attribution Score: 0.85 (HIGH)
Textual Gradient: "SOP Section 4.2 contains OUTDATED procurement threshold. Update from $50,000 to $25,000. Add new section for $25K–$50K invoices with procurement checks. Reference policy PROC-2024-008."

Summary of Other Nodes:

Node	Score	Verdict
N7: Response	0.05	✓Faithfully summarized upstream data. No changes needed.
N6: Budget	0.02	✓Correct operation. Budget check is valid regardless of workflow.
N5: KB Search	0.15	△ Searched for contracts instead of procurement policy—but *misdirected by SOP.*
N4: PO SQL	0.10	△ Retrieved `PENDING` status but ignored—*SOP didn't require check.*
N3: Vendor SQL	0.08	△ Retrieved `req_procure=TRUE` but ignored—*SOP didn't require check.*
N1: User Query	0.00	✓Clear, complete input. No issues.

Key insight: Nodes 3, 4, and 5 all exhibited problematic behavior (ignored flags, wrong query), but the holistic comparison correctly identifies these as **effects** of the SOP error at Node 2, not independent causes. By seeing all context sources simultaneously, the LLM can trace the causal chain and attribute responsibility to the upstream source.

K.6.3 Attribution Summary.

```
RootCauseAnalysis:
  session_id: "sess-20241215-fin-4472"
  loss_description: "Agent advised standard AP
    for $45K invoice; user reports audit finding
    due to missing Procurement review."
  trajectory: 7 nodes, 5 tools (all OK)
    root_cause=Node 2 (SOP Lookup)
  attribution_ranking:
    1. SOP Lookup: 0.85 ***PRIMARY***
    2. KB Search: 0.15 (misdirected)
    3. PO SQL: 0.10 (ignored)
    4. Vendor SQL: 0.08 (ignored)
    5-7. Response/Budget/Query: 0.05-0
  cascade: Outdated SOP -> wrong KB search
    -> ignored flags -> bad advice.
    Failure in SOP LOGIC, not tools.
  cause: SKILL_FILE_STALE
```

```
  file=vendor_payment_processing.md
  issue=$50K should be $25K
```

K.6.4 Attribution Visualization. The holistic attribution computed the following scores across the 7-node trajectory:

N1	→	**N2★**	→	**N3**	→	**N4**	→	**N5**	→	**N6**	→	**N7**
Query	→	SOP	→	Vendor	→	PO	→	KB	→	Budget	→	Response
0.00	→	**0.85**	→	0.08	→	0.10	→	0.15	→	0.02	→	0.05

Node 2 (SOP Lookup) is identified as the **root cause** with 85% attribution. The outdated threshold ($50K instead of $25K) misdirected all downstream nodes.

K.7 Stage 3: Recommender Agent - Exploration and Validation

PHASE 4: RECOMMENDATION — RECOMMENDER validates and generates fix

The RECOMMENDER explores context sources to validate the hypothesis before generating a recommendation.

K.7.1 Read Implicated SOP.

```
Tool: read_skill("finance/
  vendor_payment_processing.md",
  include_metadata=TRUE)

Result:
  Content: Sec 4.2 "UNDER $50,000"
  Metadata:
    Modified: Jan 15, 2024
    Author: Finance Operations
    Version: 2.3
    Review: Jan 2025 (scheduled)

Observation: SOP updated Jan 2024,
  before Oct 2024 policy change.
```

K.7.2 Search KB for Current Policy. **Tool:** `search_kb("procurement threshold...")`

Result: Found `policies/PROC-2024-008.md`

Tool: `read_kb_entry(...)`

PROCUREMENT POLICY UPDATE

Policy Number: PROC-2024-008
Effective Date: October 1, 2024
Approved By: CFO, Chief Procurement Officer
Supersedes: PROC-2023-015

Section 3.2: Professional Services Engagements

All professional services engagements exceeding **$25,000** require:

(1) Procurement Office review and approval
(2) Competitive bid documentation (minimum 3 quotes) OR approved sole source justification
(3) Legal review of contract terms
(4) Budget authority verification

This threshold applies to: Consulting services, Contractor/staff augmentation, Advisory services, Implementation services.

IMPORTANT: Previous threshold of $50,000 is **SUPERSEDED**.

Reference: Board Resolution BR-2024-Q3-017

K.7.3 Cross-Reference and Validate.

Validation Analysis:

Source 1: SOP (`vendor_payment.md`)
Prof services <$50K = standard AP
Updated: Jan 2024 — OUTDATED

Source 2: Policy (`PROC-2024-008.md`)
Prof services >$25K = Procurement
Oct 2024 — AUTHORITATIVE (CFO/CPO)

Cross-Ref: **Discrepancy Confirmed**
SOP=$50K, Policy=$25K, Gap=$25K

Invoice ($45K):
Per SOP: $45K<$50K → AP (wrong)
Per Policy: $45K>$25K → Proc (correct)

User Validation: Confirmed by policy.

Classification: SYSTEMIC — SOP contradicts authoritative policy.

K.7.4 Generate CRUD Recommendation.

UPDATE SOP

Operation: UPDATE **Priority:** CRITICAL
Classification: SYSTEMIC (0.95)
File: `vendor_payment_processing.md`
Change: $50K → $25K threshold

New Section 4.2.1 ($25K–$50K):

(1) Standard checks + procurement review
(2) Search KB for procurement requirements
(3) Verify `procurement_approval`
(4) Confirm bid documentation on file
(5) Route to Procurement Office

Reference: PROC-2024-008

Root Cause: SOP threshold ($50K) vs policy ($25K)
Source: `PROC-2024-008.md` (CFO approved)
Cascade: Outdated threshold → wrong workflow → wrong KB query → audit finding
Reviewers: Finance Controller, Procurement Mgr

K.8 Summary: Why This Example Matters

This example demonstrates several key capabilities of TRACE:

1. Handling Silent Failures: All 5 tools executed successfully with no errors. Traditional error monitoring would not detect this failure. TRACE's dissatisfaction signal detection caught it through the user's correction.

2. Tracing Cascading Failures: The root cause (outdated SOP threshold) was at Node 2, but the symptom appeared at Node 7. The holistic attribution correctly attributed 85% responsibility to the SOP while recognizing that downstream nodes were "victims" of the upstream error.

3. Distinguishing Root Cause from Symptoms: Nodes 3, 4, and 5 all exhibited problematic behavior (ignored flags, wrong KB query), but the holistic attribution identified these as *effects* of the SOP error, not independent causes.

4. Context-Source Validation: The RECOMMENDER didn't just accept the ROOT CAUSE's hypothesis. It actively explored the KB to find the authoritative policy and cross-referenced it against the SOP.

5. Actionable Recommendations: The final recommendation includes specific text changes, references the authoritative policy, explains the cascade effect, and identifies reviewers—everything needed to approve the fix.

L DATA SIMULATION STRATEGY

This appendix describes the methodology for creating our synthetic evaluation dataset with perfect ground truth for all three TRACE agents.

L.1 Motivation and Contribution

Why simulation is necessary. Three factors motivated our simulation-based approach:

(1) **Proprietary system constraints**: TRACE was developed for a production enterprise agent system. Confidentiality requirements preclude disclosure of real interaction data, context source content, or detailed system architecture.

(2) **No existing benchmarks**: To our knowledge, no open-source dataset or industry-standard benchmark exists for context engineering debugging from agent trajectories. Existing benchmarks focus on agent task completion (AgentBench, GAIA) or retrieval quality (RGB, BEIR) rather than end-to-end fault attribution with CRUD recommendations.

(3) **Ground truth requirements**: Rigorous evaluation of root cause attribution requires perfect ground truth annotations—knowing exactly which trajectory node introduced each error, what the correct content should be, and where to apply fixes. Production logs lack such annotations, and manual labeling is prohibitively expensive, subjective, and error-prone.

Simulation as a contribution. We position the simulation methodology itself as a key contribution of this work. The three-layer architecture (context sources → fault definitions → execution traces) with cross-layer verification provides a *reusable framework* for generating domain-specific evaluation datasets. Practitioners can adopt this methodology to:

- Generate evaluation benchmarks for their own agent systems without exposing proprietary data
- Extend the fault taxonomy to cover domain-specific failure modes
- Create controlled experiments varying specific parameters (complexity, fault type, cascade depth)
- Produce training data for fine-tuning attribution models

We envision this simulation strategy being adopted as a *skill* for GenAI agents—enabling automated generation of test datasets that assess compatibility between context sources, fault definitions, and traces. The verification protocol (Section L.10) ensures generated datasets meet quality standards.

L.2 Design Principles

Our simulation strategy follows four core principles:

Compatibility: Every trace tool output is an exact copy from the corresponding context source, ensuring fair evaluation.
Traceability: Every DSAT case can be traced: fault → trigger → failure → user correction.
Realism: Financial values, personas, and reasoning patterns match real-world scenarios.
Completeness: Attribution scores sum to 1.0; all faults have authoritative corrections.

Layer 1: Context Sources Skills, Knowledge Base, Database Tables, Tool Definitions (Contains embedded faults with known locations)
↓
Layer 2: Fault Definitions Fault definitions across 6 categories (Each references specific content in Layer 1)
↓
Layer 3: Execution Traces 75 traces (60 DSAT + 15 control) (Tool outputs are exact copies from Layer 1)

L.3 Three-Layer Architecture

L.4 Dataset Composition

Table 7: Dataset Statistics

Component	Count	Description
Context Source Files	23	Skills, KB, DB, Tools
Fault Categories	6	See Table below
DSAT Traces	60	Includes 14 complex + 10 difficult
Control Traces	15	Successful interactions
Total Traces	**75**	Full coverage

L.5 Fault Injection Categories

Each fault category represents a distinct failure mode in context engineering:

Table 8: Fault Categories and Trace Distribution

Category	Traces	Example Fault
`SKILL_FILE_STALE`	12	$50K→$25K threshold
`KB_CONTENT_STALE`	12	Tier 1 $500K→$300K
`KB_CONTENT_GAP`	11	Missing Q2G 2026 scenario
`TOOL_PROMPT_ERROR`	9	Wrong partition key
`SYSTEM_PROMPT_GAP`	6	No period close check
`RETRIEVAL_FAILURE`	10	"guard"≠"security"
Total DSAT	**60**	

L.6 Complexity Distribution

Table 9: Trace Complexity Tiers

Level	Nodes	Count	Percentage
Simple	2–8	36	60%
Complex	9–11	14	23%
Difficult	15–16	10	17%

Complex (n=14) and difficult (n=10) traces enable statistically meaningful ablation studies on attribution accuracy across trajectory lengths.

L.7 Fault Injection Procedure

Each fault is defined with a structured JSON schema that enables automated verification and ensures reproducibility:

```
{
  "fault_id": "SKILL_STALE_001",
  "category": "SKILL_FILE_STALE",
  "faulty_source": {
    "type": "skill",
    "path": "skills/vendor-payment/SKILL.md",
    "section": "Section 4.2",
    "content": "For invoices UNDER $50,000"
  },
  "correct_value": {
    "content": "For invoices UNDER $25,000",
    "authoritative_source": "knowledge_base/policies/PROC-2024-008.md"
  },
  "trigger": {
    "user_query": "Process payment for invoice... $45,000",
    "expected_failure": "Routes to standard AP instead of Procurement"
  },
  "cascade_effects": [
    {"node": 3, "effect": "Ignores requires_procurement_review=TRUE"},
    {"node": 4, "effect": "Ignores procurement_approval_status=PENDING"}
  ]
}
```

Fault-Source Mapping. Each fault definition references a specific location in the context sources, enabling deterministic verification:

Table 10: Fault-to-Source Mapping Examples

Fault ID	Context Source	Location
SKILL_STALE_001	skills/vendor-payment/SKILL.md	Section 4.2, line 45
KB_STALE_001	kb/policies/ESC-2024-003.md	Section 3.1, Tier 1 row
KB_GAP_001	kb/reference/scenario-calendar.md	Missing Q2G 2026
TOOL_ERROR_001	tools/tool_definitions.json	schema_hints partition
RETRIEVAL_001	embeddings/synonym_mapping.json	guard≠security

L.8 Trace Generation Process

Traces are generated by instantiating fault definitions into complete execution trajectories. The key principle: **tool outputs must be exact copies from context source files**, enabling deterministic verification.

Trajectory Node Types. Each trace contains a sequence of typed nodes representing agent execution steps:

Table 11: Trajectory Node Types and Key Fields

Node Type	Purpose	Key Fields
user_query_processing	Parse user intent	input.raw_query, output.entities
skill_lookup	Read SOP guidance	tool.input.path, tool.output.content
sql_query	Query database	tool.input.query, tool.output.rows
kb_search	Search KB	tool.input.query, tool.output.results
kb_read	Read KB document	tool.input.path, tool.output.content
response_generation	Generate response	input.gathered_facts, output.response

Node Annotation Fields. Root cause nodes include fault annotations; downstream nodes include cascade effect markers:

```
// Root cause node annotation
{
  "is_root_cause": true,
  "fault_annotation": {
    "faulty_content": "UNDER $50,000",
    "correct_content": "UNDER $25,000",
    "why_wrong": "SOP not updated after PROC-2024-008"
  }
}

// Cascade effect node annotation
{
  "missed_signal": {
    "field": "requires_procurement_review",
    "value": true,
    "why_missed": "SOP Section 4.2 doesn't instruct checking this",
    "cascade_from": "node_2"
  }
}
```

L.9 Ground Truth Schema

Each trace includes structured ground truth labels:

```
{
  "is_dsat": true,
  "fault_id": "SKILL_STALE_001",
  "fault_category": "SKILL_FILE_STALE",
  "root_cause": {
    "node_id": 2,
    "component_type": "skill",
    "component_path": "skills/vendor-payment/SKILL.md",
    "faulty_content": "UNDER $50,000",
    "correct_content": "UNDER $25,000"
  },
  "expected_attribution": {
    "node_1": 0.00, "node_2": 0.85,
    "node_3": 0.05, "node_4": 0.05,
    "node_5": 0.03, "node_6": 0.01, "node_7": 0.01
  },
  "expected_recommendation": {
    "operation": "UPDATE",
    "target": "skills/vendor-payment/SKILL.md",
    "classification": "SYSTEMIC"
  }
}
```

L.10 Verification Protocol

All 75 traces were verified using a five-point checklist with automated and manual checks. Below we describe each verification procedure with concrete examples.

Check 1: Tool Output = Source Content. For each node with a `tool` field, verify `tool.output.content` matches the actual context source file.

Procedure: Read the actual file referenced in `tool.input.path`; compare strings.

Example: Node 3 calls `read_kb_entry("policies/...")`. Verify output matches file at `knowledge_base/policies/`.

Check 2: Fault Annotation Exists in Source. For each root cause node, verify the `fault_annotation.faulty_content` string exists in the referenced context source at the expected location.

Example: For trace referencing KB_STALE_003, verify "Tier 1 (Critical) | >= $500,000" appears in `ESC-2024-003.md` Section 3.1.

Check 3: Cross-Reference Integrity. Verify all cross-references between trace components and external definitions: (a) Each trace's `fault_id` matches an entry in the fault manifest with

consistent fault description. (b) Each `dsat_signal.evidence` string appears verbatim in the conversation turn content.

Example (a): Trace with `fault_id: "KB_STALE_003"` must reference the correct fault in the manifest ("Escalation Tier 1 $500K vs $300K").

Example (b): Evidence "the CFO updated the Tier 1 threshold to $300K" must appear in the user's correction turn.

Check 4: Root Cause Node Correct. Verify the node marked `is_root_cause: true` is the *first* node retrieving faulty content—not downstream cascade nodes.

Example: In a trace where Node 2 reads faulty skill content and Node 3 ignores a database flag due to missing instructions, only Node 2 should be marked as root cause.

Check 5: Attribution Sums to ∼1.0. Verify `expected_attribution` scores sum to 1.0 (±0.05), with root cause having maximum attribution.

Example: Attribution {node_1: 0.00, node_2: 0.05, node_3: 0.90, node_4: 0.05} sums to 1.00; Node 3 has highest score.

L.11 Reproducibility Guide

To verify dataset integrity, researchers can perform the following automated checks:

(1) **Attribution Sums**: For each DSAT trace, sum `ground_truth.expected_attribution` values. All 60 traces sum to exactly 1.00.
(2) **Evidence Strings**: For each DSAT signal, verify the evidence string appears in conversation content. All evidence strings present.
(3) **Tool-Source Matching**: For each tool call node, verify `tool.output.content` matches the corresponding context source file. Verified for all 75 traces.
(4) **Complexity Labels**: Verify `metadata.complexity` matches actual node count (simple: 2–3, medium: 4–5, complex: 6+). All labels correct.

Automated Verification. We provide a verification script that performs all five checks programmatically:

```
# Run full verification suite
python validate_traces.py

# Expected output:
# CHECK 1 (Tool Output = Source): 75/75 PASS
# CHECK 2 (Fault Annotation Exists): 60/60 PASS
# CHECK 3 (Cross-Reference Integrity): 60/60 PASS
# CHECK 4 (Root Cause Node Correct): 60/60 PASS
# CHECK 5 (Attribution Sums to 1.0): 60/60 PASS
# ========================================
# OVERALL: 75/75 traces pass all verification checks
```

L.12 Quality Criteria

We apply explicit quality criteria checklists to ensure simulation realism and completeness.

Realism Checklist:
- Financial values match real-world ranges ($25K–$500K thresholds typical in enterprise procurement)
- Dates internally consistent (policy updates 2024, traces 2025)
- Personas ask role-appropriate questions (clerk: invoices; manager: escalations)
- Agent reasoning matches actual LLM chain-of-thought outputs
- DSAT signals include specific textual evidence from conversation

Completeness Checklist:
- Every fault definition has corresponding content in context sources
- Every trace tool output matches actual context source file
- Every DSAT signal has grounded textual evidence in conversation
- Attribution scores sum to 1.0 (±0.05), root cause has maximum
- All 6 fault categories have sufficient coverage (6–10 traces each)

Traceability Checklist:
- Complete causal chain: fault → trigger → failure → correction
- Cascade effects annotated with `cascade_from` references
- Authoritative correction source identified for each fault
- Ground truth recommendation specifies exact file path and section

M TRACE SEGMENTATION

Long conversations often contain multiple independent questions or sub-tasks, each potentially experiencing distinct failure modes requiring separate remediation. The **Segmenter** is an optional preprocessing layer that decomposes conversation traces into semantically coherent segments before analysis. By segmenting the trace, the Detector can assign DSAT signals to specific segments, the Root Cause can build targeted DAGs for each segment, and the Recommender can generate independent CRUD recommendations—improving both precision and actionability.

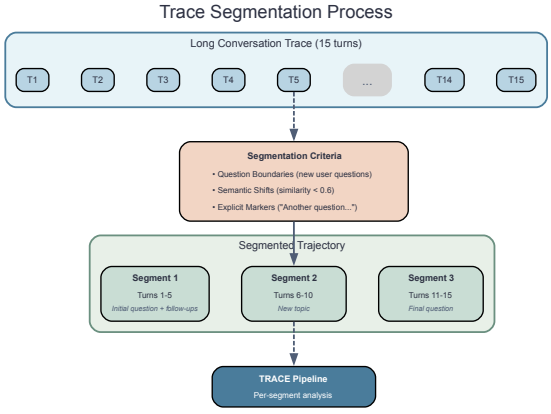

Figure 3: Trace segmentation decomposes long conversations into semantically coherent segments based on question boundaries, semantic shifts, and explicit markers.

M.1 Segmentation Criteria

Question Boundaries. Each new user question initiates a potential segment, identified through:
- Interrogative syntax (who, what, when, where, why, how)
- Imperative requests ("Show me...", "Calculate...", "Find...")
- Topic-initiating phrases ("I have a question about...", "Can you help with...")

Close Follow-ups. Rapid clarifications (<30 seconds) on the same topic are grouped with their parent question:

- "What do you mean by X?"
- "Can you clarify the second point?"
- "Actually, I meant Y instead of Z."

Semantic Shifts. Embedding similarity between consecutive turns below a threshold (~0.6) indicates a topic change.

Explicit Markers. Phrases like "Another question…", "Separately…", "On a different topic…" explicitly signal segment boundaries.

M.2 Output Schema

The Segmenter produces a `SegmentedTrajectory`:

- `session_id`: Original session identifier
- `segments`: List of segment objects with `segment_id`, `turns`, `primary_question`, `is_followup`, and `parent_segment`
- `segmentation_applied`: Boolean flag for downstream awareness

M.3 Configuration Guidelines

Enable segmentation when: sessions exceed 5 turns, multiple topic shifts are detected, or high-volume production requires precise per-segment attribution.

Bypass segmentation when: interactions are short and single-topic, real-time latency is critical, or sessions are already known to be single-topic.

When disabled, the downstream pipeline processes the full session as a single segment, preserving backward compatibility.

N ITERATIVE ATTRIBUTION ALGORITHM

This appendix details the iterative backward pass algorithm used as an ablation baseline to validate our holistic attribution approach.

N.1 Algorithm Description

N.2 Illustrative Walkthrough

Consider a simple three-node trajectory: (1) User asks about refund policy, (2) Agent retrieves KB entry, (3) Agent responds with outdated 14-day policy when the correct policy is 30 days. The backward pass proceeds as follows:

- *Response node*: The loss is "agent said 14 days, user expected 30 days." Attribution is low ($a = 0.2$) because the agent correctly used the information it was given.
- *KB retrieval node*: The retrieved entry contained "14-day refund policy." Attribution is high ($a = 0.9$) because this is where incorrect information entered the trajectory.
- *User query node*: No attribution ($a = 0$)—the user's question was clear.

The KB entry receives the highest attribution, and its textual gradient specifies the needed update.

Algorithm 2 Iterative Backward Pass Attribution (Ablation Baseline)

Require: Trajectory DAG $\mathcal{G} = (V, E)$ with nodes $v \in V$
Require: User correction u_{corr}, Agent response r_{agent}
Ensure: Attribution scores $\{a_v\}_{v \in V}$ (sum to 1.0)
Ensure: Textual gradients $\{\nabla_v\}_{v \in V}$

// Step 1: Generate loss description
$\mathcal{L} \leftarrow \text{LLM}(\textsc{LossPrompt}, u_{\text{corr}}, r_{\text{agent}})$
// $\mathcal{L}$ describes expected vs actual outcome
// Step 2: Backward pass through DAG
downstream $\leftarrow \{\}$ *// Accumulated downstream context*
for v in ReverseTopologicalOrder($\mathcal{G}$) **do**
  $c_v \leftarrow \text{GetNodeContext}(v)$ *// Tool output, KB content, etc.*
  $o_v \leftarrow \text{GetNodeOutput}(v)$ *// Node's contribution to trajectory*
  $d_v \leftarrow \text{downstream}[\text{successors}(v)]$ *// How downstream used this*
  $\nabla_v \leftarrow \text{LLM}(\textsc{GradientPrompt}, \mathcal{L}, c_v, o_v, d_v)$
  $a_v^{\text{raw}} \leftarrow \text{ParseScore}(\nabla_v)$ *// Extract [0,1] score*
  $\text{downstream}[v] \leftarrow \nabla_v$ *// Propagate gradient upstream*
end for
// Step 3: Normalize attribution scores
$Z \leftarrow \sum_{v \in V} a_v^{\text{raw}}$
for v in V **do**
  $a_v \leftarrow a_v^{\text{raw}} / Z$ *// Ensure scores sum to 1.0*
end for
return $\text{SortByScore}(\{(v, a_v, \nabla_v) : v \in V\})$

N.3 Textual Gradients

The *textual gradient* ∇_v for node v adapts the concept from TextGrad [1] to conversation trajectories. Just as numerical gradients in neural network training indicate how weights should change to minimize a loss function, textual gradients provide natural language feedback describing how each component should change to improve outcomes. This analogy enables "backpropagation through text"—propagating semantic criticism backward through the computational graph without requiring mathematical differentiability.

Concretely, the textual gradient ∇_v for node v is a structured natural language description containing:

(1) **Attribution score** ($a_v \in [0, 1]$): A quantitative estimate of how much this component contributed to the failure, enabling prioritization across multiple potential causes.
(2) **Specific change recommendation**: What content modification would address the issue (e.g., "add schema naming conventions to tool prompt").
(3) **Causal justification**: Why this change would prevent the failure, linking the component's content to the observed error.

Unlike numerical gradients that require differentiable operations, textual gradients leverage the LLM's reasoning capabilities to generate interpretable, actionable feedback for any component in the trajectory—including retrieved documents, tool descriptions, and procedural instructions that have no mathematical gradient.

Example. For a SQL tool that generated an invalid query, the textual gradient might be: "*Attribution: 0.85. The SQL tool description should include the constraint that all table names in this database use the 'dbo_' prefix. The agent generated 'SELECT * FROM users' but the correct table name is 'dbo_users'. Adding schema naming conventions to the tool prompt would prevent this class of errors.*"

O DSAT SIGNAL TAXONOMY

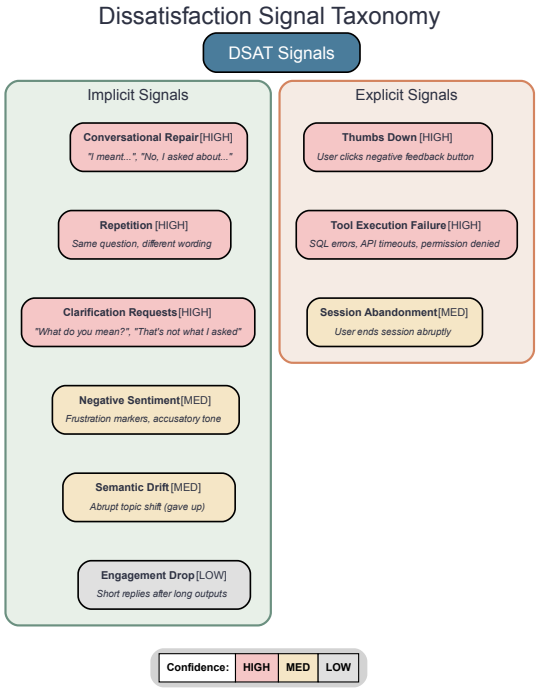

Figure 4: Taxonomy of dissatisfaction signals. Explicit signals (left) come from direct user feedback or system events. Implicit signals (right) are inferred from conversational patterns.

The DETECTOR identifies user dissatisfaction through two categories of signals, derived from qualitative analysis of agent conversation traces:

Explicit Signals (high confidence):

- *Thumbs down feedback*: Direct negative feedback from UI
- *Tool execution failure*: SQL errors, API errors, timeouts
- *Session abandonment*: User ends session abruptly

Implicit Signals (require linguistic analysis):

- *Conversational repair*: User rephrases or corrects ("I meant...")
- *Repetition*: User repeats question with different wording
- *Negative sentiment*: Frustration markers, negative language
- *Semantic drift*: Abrupt topic shift suggesting user gave up

Signals are weighted by confidence (explicit: $w = 1.0$, high implicit: $w = 0.8$, medium: $w = 0.5$) and aggregated via $c = \max_s w_s \cdot \text{conf}_s$. Sessions with $c \geq 0.7$ or explicit signals are escalated to the ROOT CAUSE.

P PROMPT TEMPLATES

This appendix presents the core prompts used by each TRACE agent.

P.1 DETECTOR System Prompt

The DETECTOR prompt embeds the signal taxonomy with examples and clear output requirements. Note that confidence aggregation is performed in post-processing code, not by the LLM:

```
You are DetectoR, an expert at analyzing human-AI
conversations to identify user dissatisfaction signals.

## DSAT Signal Taxonomy

### 1. CORRECTION
User directly states the AI is wrong or provides the correct answer.
Examples: "That's not right, the actual value is..."
         "No, I meant X not Y" | "Wrong. It should be..."

### 2. CONVERSATIONAL_REPAIR
User redirects or repairs conversation because AI misunderstood.
Key: User is FIXING the conversation flow, not just asking for explanation.

Examples: "Let me rephrase..." | "What I actually meant was..."
Distinction: REPAIR=redirecting, CLARIFICATION=confused

### 3. TOOL_EXECUTION_ERROR (System Signal)
The AI's tool call FAILED at system level - actual execution error.
ONLY for actual system failures, NOT wrong/stale data (use CORRECTION).
Examples: SQL syntax errors, File not found, API failures
NOT tool_execution_error: Tool succeeded but returned wrong data

### 4. CLARIFICATION_REQUEST
User is CONFUSED and wants explanation, not correcting.
Examples: "What do you mean by...?" | "Can you explain that?"

### 5. NEGATIVE_SENTIMENT
User expresses frustration or disappointment.
Examples: "This isn't helpful" | "That's frustrating"

### 6. REPETITION
User repeats their original request, indicating non-fulfillment.
Examples: Restating same question | "As I said earlier..."

### 7. SEMANTIC_DRIFT
Conversation drifts from user's original intent.

### 8. SESSION_ABANDONMENT
User gives up prematurely. Examples: "Never mind" | "Forget it"

## Output Requirements
Provide structured JSON with:
- is_dsat: boolean (overall determination)
- confidence: float 0.0-1.0 (your certainty)
- signals: list of detected signals, each with:
  - signal_type: one of the 8 categories above
  - evidence: exact quote from conversation
  - confidence: float 0.0-1.0 for this signal
  - turn_id: which turn contains this signal
- trigger_turn_id: first turn showing dissatisfaction
- summary: brief explanation of findings
```

Post-processing. After LLM response, confidence is recomputed using weighted aggregation: $c = \max(w_s \cdot \text{conf}_s)$ where w_s are predefined signal weights (system signals: 1.0, corrections: 1.0, repair/clarification/repetition: 0.8, sentiment/drift/abandonment: 0.5). Sessions with $c \geq 0.7$ or any system signal are escalated to the ROOT CAUSE.

P.2 Root Cause Attribution Prompts

The Root Cause implements holistic attribution (Algorithm 1) using two coordinated prompts. We also document the iterative baseline prompts (Algorithm 2) used in ablation studies.

P.2.1 Stage 1: Loss Generation (The Delta). The first prompt extracts the **delta**—the discrepancy between what the user expected and what the agent produced. This delta serves as the loss signal for attribution:

```
You are analyzing an AI agent conversation where user was dissatisfied.

## Conversation: {conversation}
## User Correction: {user_correction}
## DSAT Signals Detected: {dsat_signals}

## Task: Generate the DELTA (discrepancy to trace backward)

The DELTA is the gap between user expectation and agent output.
This DELTA is THE signal you will trace through the trajectory
to find the root cause node.

1. What the user EXPECTED (from their correction)
2. What the agent ACTUALLY produced
3. The DELTA: "Expected X but got Y" (one sentence)

Output: {"loss_description": "DELTA: Expected [X] but got [Y]",
        "expected_outcome": "...", "actual_outcome": "..."}
```

P.2.2 Stage 2: Holistic Attribution (Primary Method). The holistic approach presents the agent's execution *trace* to the LLM—chain-of-thought, tool calls, and tool outputs—in reverse temporal order. The LLM does not separately inspect raw context files; it relies on the chain-of-thought, where the agent itself names the sources that shaped each decision. Verification of those candidate files is deferred to the Recommender stage. This prompts a single-pass backward walk in the spirit of gradient descent's backward pass through a computational graph:

```
You are performing holistic root cause attribution.

## SCOPE (Important)
You see the agent's execution TRACE only: chain-of-thought,
tool calls, and tool outputs. You do NOT have the raw KB, the
full set of Skill files, or every tool definition.

The chain-of-thought is your most diagnostic signal -- the
agent's own reasoning names the sources behind each decision
(e.g., "According to the KB..." or "Following the SOP for...").
Use those references to identify candidate root-cause sources;
verification of file contents is the Recommender's job.

## THE DELTA (Your Search Target)
Loss: {loss_description}
Expected: {expected_outcome} | Actual: {actual_outcome}

## TRAJECTORY (Agent-Accessed Sources, REVERSE Temporal Order)
Listed from the agent's final response backward to the
original user query. Each entry contains the node's content
(input it received) and output (what it produced/decided).

{trajectory_nodes_in_reverse_order_with_content_and_output}

## TASK: Identify Root Cause via Reverse-Order Backward Pass
Treat the trajectory as a computational graph and the DELTA
as the loss signal. Walk the trajectory in REVERSE TEMPORAL
ORDER -- gradient-descent style -- to localize the fault:

 1. START at the agent's FINAL RESPONSE (where loss is observed).
 2. Step BACKWARD one node at a time:
      response -> tool outputs -> KB retrievals -> skill lookups
                -> tool prompts -> system prompt -> user query.
    For each node ask: "Does the DELTA already appear in this
    node's OUTPUT? Did this node's INPUT already contain it?"
 3. The EARLIEST node whose OUTPUT contains the DELTA but whose
    INPUT does NOT is the node that INTRODUCED the error
    (the root cause). All later nodes merely PROPAGATED it.
 4. Use the simultaneous view to disambiguate ties: among
    candidate root-cause nodes, pick the one whose content is
    most directly inconsistent with the DELTA.

This mirrors textual gradient backpropagation: the DELTA is the
loss; backward traversal is the gradient signal; the root-cause
node is where the gradient is largest -- the "weight" most
responsible for the loss.

NOTE: If the true root cause is a context source the agent
NEVER ACCESSED (e.g., a missing KB entry, an SOP the agent
should have read but did not), the trajectory cannot directly
reveal it. In that case, attribute to the EARLIEST accessed
node whose decision shows the agent should have looked further
(retrieval failure, missing-skill lookup, etc.), and let the
RecommendeR's exploration stage verify and refine.

## Four-Dimensional Diagnosis (for root cause node)
- existence: Does required information exist?
- accessibility: Was it successfully retrieved?
- correctness: Is the information accurate and up-to-date?
- consistency: Is it consistent with other sources?

## Output Schema
{
  "root_cause_node": "<node_id>",
  "attribution_scores": {"node_id": <float 0.0-1.0>, ...},
  "fault_category": "<category>",
  "component_path": "<path to fix>",
  "gradient_description": "<what should change>",
  "diagnosis": {existence, accessibility, correctness, consistency}
}
```

P.2.3 Iterative Per-Node Attribution (Ablation Baseline). For ablation comparison, we also implement iterative per-node attribution. For each node, the prompt asks: *Does this node's output contain the delta?* If the delta appears in the output but not the input, this node *introduced* the error. The **Error Transformation Classification** formalizes this logic:

```
You are performing per-node attribution analysis (ablation baseline).

## THE DELTA (Your Search Target)
Loss: {loss_description}
Expected: {expected_outcome} | Actual: {actual_outcome}

QUESTION: Does THIS NODE's output contain the DELTA?
- If YES and NOT in input -> This node INTRODUCED the error
- If YES and also in input -> This node PROPAGATED the error
- If NO -> This node is not involved

## Full Trajectory Context (all nodes for reference)
{full_trajectory_context}

## FOCUS: Analyze Node {node_id}
- Type: {node_type}
- Full Input: {node_input}
- Full Output: {node_output}
- Reasoning: {node_reasoning}
- Tool Info: {tool_info}

## Downstream Evidence (Observable Facts from Later Nodes)
{downstream_evidence}

## CRITICAL: Error Transformation Classification

Classify how this node relates to the error (input vs output):

INTRODUCED: Input is CORRECT, output is INCORRECT
```

```
  - This node is where the error ORIGINATED
  - High attribution (0.7-0.9)

PROPAGATED: Input already has error, output passes it unchanged
  - This node just passed along an existing error
  - Low attribution (0.1-0.2)

AMPLIFIED: Input has minor issue, output makes it worse
  - This node worsened an existing problem
  - Medium-high attribution (0.5-0.7)

TRANSFORMED: Input has error type A, output has error type B
  - This node changed the nature of the error
  - Medium attribution (0.3-0.5)

NONE: Neither input nor output contains relevant error
  - This node is unrelated to the failure
  - No attribution (0.0-0.1)

## Four-Dimensional Diagnosis
- existence: Does required information exist in knowledge source?
- accessibility: Was it successfully retrieved/accessed?
- correctness: Is the information accurate and up-to-date?
- consistency: Is it consistent with other sources?

## Output Schema
{
  "node_id": {node_id},
  "error_analysis": {
    "error_present_in_input": true/false,
    "error_present_in_output": true/false,
    "transformation_type": "introduced|propagated|...|none",
    "error_signature": "<specific error pattern found>",
    "evidence": "<data supporting classification>"
  },
  "attribution_score": <float 0.0-1.0>,
  "component_type": "<skill|kb_entry|tool_prompt|system_prompt|none>",
  "component_path": "<path if applicable>",
  "gradient_description": "<what should change to fix this?>",
  "diagnosis": {existence, accessibility, correctness, consistency},
  "is_root_cause_candidate": true/false,
  "cascade_from": <upstream node id if error propagated, else null>
}
```

P.3 Recommender Agentic Exploration Prompt

The Recommender is an agentic LLM with tool access that explores context sources to verify the Root Cause's hypothesis. Unlike static prompt chains, it autonomously decides which tools to call based on what it discovers. The system prompt establishes the exploration protocol and available tools:

```
You are the Recommender Agent in the TRACE system. Your task is to
VERIFY the Root Cause Agent's hypothesis through ACTIVE EXPLORATION
of the agent's context sources, then generate a CRUD recommendation.

## Your Mission
The Root Cause Agent identified a potential failure source. You must:
1. EXPLORE: Use tools to investigate the implicated component
2. VERIFY: Cross-reference against authoritative sources
3. DIAGNOSE: Apply four-dimensional analysis
4. RECOMMEND: Generate actionable CRUD recommendation

## Available Tools (USE THEM!)
- search_kb(query): Search knowledge base for related content
- read_kb_entry(id): Read full KB entry with metadata
- list_skills(path): Browse Skills file hierarchy
- read_skill(path): Read Skill/SOP document content
- read_system_prompt(): Get current system prompt
- read_tool_prompt(name): Get tool description

## Exploration Protocol
1. READ the implicated component (get current state, metadata)
2. SEARCH for authoritative sources (policies, specifications)
3. CROSS-REFERENCE implicated content vs authoritative sources
4. EXPLORE related components that may need coordinated updates

## Four-Dimensional Diagnosis (evaluate ALL dimensions)
D1-EXISTENCE: Does content exist? (Gap = CREATE)
D2-ACCESSIBILITY: Was it retrieved? (Routing fail = UPDATE routing)
D3-CORRECTNESS: Is it accurate? (Stale/wrong = UPDATE content)
D4-CONSISTENCY: Aligned with other sources? (Conflict = UPDATE)

## Authority Resolution (when sources conflict)
1. Explicit supersession ("replaces policy X")
2. Official status (CFO-approved > draft)
3. Hierarchy (system-level > component-level)
4. Recency (newer > older)
5. Domain ownership

## Classification
SYSTEMIC: Verified problem affecting many users -> HIGH priority
INCIDENTAL: No issue found, edge case -> NO_ACTION
AMBIGUOUS: Cannot verify -> LOW priority, flag for human

## Input
Root Cause Analysis: {root_cause_analysis}
Implicated Component: {component_path}
Fault Category: {fault_category}
User Correction: {user_correction}

## Output Schema
After exploration, output:
{
  "exploration_log": [{"tool": "...", "result": "..."}],
  "diagnosis": {
    "existence": {"passed": bool, "evidence": "..."},
    "accessibility": {"passed": bool, "evidence": "..."},
    "correctness": {"passed": bool, "evidence": "..."},
    "consistency": {"passed": bool, "evidence": "..."}
  },
  "operation": "CREATE|UPDATE|DELETE|NO_ACTION",
  "target_path": "path/to/fix",
  "recommended_change": "specific change text",
  "authoritative_source": "source used for validation",
  "classification": "SYSTEMIC|INCIDENTAL|AMBIGUOUS",
  "confidence": 0.0-1.0
}
```

The agent typically makes 3–5 tool calls per trace, exploring the implicated component, searching for authoritative sources, and checking for related content that may need coordinated updates.

Q EVALUATION METRIC DEFINITIONS

This appendix formally defines the evaluation metrics used throughout the paper.

Q.1 Detector Metrics

We evaluate binary DSAT detection using standard classification metrics: precision, recall, and F1 score.

Q.2 Root Cause Agent Metrics

Node Accuracy (Acc@1): Whether the predicted root cause node matches ground truth:

$$\text{Acc@1} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}[\hat{n}_i = n_i^*] \tag{2}$$

where $\hat{n}_i$ is the predicted root cause node and n_i^* is the ground truth.

Accuracy@3 (Acc@3): Whether the ground truth node is among the top-3 attributed nodes:

$$\text{Acc@3} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}[n_i^* \in \text{top-3}(\hat{A}_i)] \tag{3}$$

where $\hat{A}_i$ is the predicted attribution score vector.

Category Accuracy: Whether the predicted fault category matches ground truth:

$$\text{Cat. Acc} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}[\hat{c}_i = c_i^*] \tag{4}$$

Q.3 RECOMMENDER Metrics

CRUD Accuracy: Whether the predicted CRUD operation matches ground truth:

$$\text{CRUD Acc} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}[\hat{o}_i = o_i^*] \tag{5}$$

where $\hat{o}_i \in \{\text{CREATE}, \text{UPDATE}, \text{DELETE}, \text{NO\_ACTION}\}$.

Path Match (Exact): Whether the target path exactly matches ground truth:

$$\text{Path (Exact)} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}[\hat{p}_i = p_i^*] \tag{6}$$

Path Match (Partial): Whether the target path contains the correct component:

$$\text{Path (Partial)} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}[\text{component}(\hat{p}_i) = \text{component}(p_i^*)] \tag{7}$$

A partial match occurs when the predicted path references the correct file but may differ in exact formatting (e.g., "skills/vendor-payment/SKILL.md" vs. "vendor-payment").

Q.4 Ablation Study Metrics

Recovery Rate: The proportion of initially incorrect predictions that are corrected through exploration:

$$\text{Recovery Rate} = \frac{|\{i : \hat{n}_i^{(1)} \neq n_i^* \wedge \hat{n}_i^{(2)} = n_i^*\}|}{|\{i : \hat{n}_i^{(1)} \neq n_i^*\}|} \tag{8}$$

where $\hat{n}_i^{(1)}$ is the initial prediction and $\hat{n}_i^{(2)}$ is the post-exploration prediction.

LLM Call Efficiency: The average number of LLM calls per trace:

$$\text{Calls/Trace} = \frac{1}{N}\sum_{i=1}^{N} \text{calls}_i \tag{9}$$

This metric captures the compute cost trade-off between iterative (multiple calls) and holistic (single call) approaches.

Q.5 End-to-End Pipeline Metrics

Cumulative Accuracy: The proportion of traces where all three agents produce correct outputs:

$$\text{E2E Acc} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}[\text{DSAT}_i \wedge \text{Node}_i \wedge \text{CRUD}_i] \tag{10}$$

This metric captures the compounding effect of errors across the pipeline.