Delta Debugging for LLM-integrated Systems

Hao-Nan Zhu*
University of California, Davis
United States
hnzhu@ucdavis.edu

Muhammed Numair Mansur
Amazon Web Services
Germany
numairm@amazon.de

Martin Schäf
Amazon Web Services
United States
schaef@amazon.com

Zeya Chen
Amazon Web Services
United States
zeyachen@amazon.com

Tancrède Lepoint
Amazon Web Services
United States
tlepoint@amazon.com

Willem Visser
Amazon Web Services
United States
vissie@amazon.com

Abstract

Large Language Models (LLMs) are increasingly integrated into software systems as automated decision-making components. These systems rely on instruction prompts written in natural language to encode complex workflows. However, debugging these prompts when LLMs produce undesired outputs remains challenging due to their black-box nature and the impracticality of manually inspecting large, complex inputs. Unlike traditional software, LLMs provide no access to execution paths or intermediate states, making it difficult to identify which input fragments are responsible for unexpected behavior.

This paper investigates whether delta debugging can be effectively applied to identify and isolate problematic parts of LLM inputs that lead to undesired outputs. We introduce *semantic markers* as an instrumentation technique that embeds unique identifiers in LLM inputs and extracts traceability information from chain-of-thought reasoning. We systematically evaluate whether these markers accurately identify causal input fragments and enable delta debugging to isolate minimal subsets responsible for incorrect outputs.

Through experiments on two benchmarks representing development and production scenarios, we demonstrate that delta debugging with semantic markers can systematically pinpoint problematic input fragments in both development and production settings. Our investigation shows that this approach transforms prompt debugging from an ad-hoc manual process into a systematic methodology, enabling engineers to efficiently identify and address the root causes of unexpected LLM behavior in real-world applications.

ACM Reference Format:
Hao-Nan Zhu, Muhammed Numair Mansur, Martin Schäf, Zeya Chen, Tancrède Lepoint, and Willem Visser. 2026. Delta Debugging for LLM-integrated Systems. In . ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*Work conducted while at Amazon Web Services

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Across industries, Large Language Models (LLMs) are being incorporated rapidly into software systems to automate tasks that were previously infeasible to automate. LLMs can act as oracles, answering complex questions over almost arbitrary inputs. Complex workflow steps can now be encoded as *instruction prompts* in natural language that describe the task at hand and how output should be formatted. These prompts can then be appended with data in a variety of formats, such as documents or images, and sent as input to an LLM for processing.

As software engineers, we face the challenge of debugging such prompts. Specifically, when an LLM produces an undesired output, how can we identify which parts of the input to focus on?

Unlike traditional software where we can trace execution paths and inspect intermediate states, LLMs operate as black boxes with complex internal representations that are not directly accessible. Moreover, the sheer size of modern prompts, which can include lengthy documents, multiple examples, and detailed instructions, makes manual inspection and iterative refinement impractical.

LLMs are probabilistic by nature and, in most cases, inherently non-deterministic. Even with a *temperature* of zero, most of the popular models, such as Llama or Claude, still do not produce truly deterministic outputs. Debugging non-deterministic systems without specialized tools is notoriously difficult [7]. Software engineering research has produced many approaches for related challenges: debugging distributed systems [1], detecting and diagnosing Heisenbugs [8], debugging for concurrent systems [10], minimizing UI event traces [2], and debugging microservices [18]. These techniques compare inputs and executions that lead to "good" versus "bad" outcomes, and isolate patterns unique to the failures.

This approach of debugging by comparing passing and failing executions is known as delta debugging [15], which recently had its 25^{th} anniversary [16]. While delta debugging has proven effective for traditional software, applying it to LLM-integrated systems presents unique challenges due to their black-box nature and the complexity of modern prompts.

Existing LLM explainability methods [6, 17] have limitations for systematic debugging. Feature attribution and attention visualization techniques require model internals and perturbation methods only observe input-output relationships. Chain-of-thought (CoT) prompting [13] shows step-by-step reasoning, but CoT explanations can be unfaithful [12], i.e., models may generate plausible explanations that don't reflect their actual reasoning.

In this paper, we investigate whether delta debugging can be applied to identify relevant parts of an LLM input that lead to an undesired outcome without requiring access to model internals. Our investigation consists of three parts: First, we need to find a way to instrument the input to the LLM that allows us to extract the output which parts of the input were considered relevant by the LLM to produce the final output. Second, we need to show that these parts of the input are indeed relevant and that removing them from the input changes the output (and conversely, that removing unrelated parts of the input does not affect the output). And lastly, we need to show that we can use these parts of the input for delta debugging to identify a compact subset that is sufficient to debug and improve the input.

For the first task of extracting relevant parts of an LLM input from its output, we introduce the concept of *semantic markers*: we instrument the original input to the LLM by placing unique markers at regular intervals. For example, for textual input, we can place these markers to delimit every sentence. That is, for an input sentences like, "LLMs are great. Debugging is hard.", we may mark it as `<marker_1>LLMs are great.</marker_1> <marker_2>Debugging is hard.</marker_2>`. Then, we amend the original input with additional instructions to include these semantic markers `<marker_i>` in the chain-of-thought (CoT) [13] part of the output. We extract the *CoT trace* from CoT portion of the output and verify that the markers indeed exist in the LLM input. This aims at tracing the data used in CoT's and addressing its potential faithfulness problem by forcing the LLM to explicitly cite which input fragments it uses, without requiring model internals. Our evaluation shows that the LLM reliably reports these semantic markers as part of its CoT, and we can parse them as a trace.

For part two of our investigation, we need to understand if these CoT traces indeed represent parts of the input that were relevant to how the LLM made its decisions. To that end, we conduct an experiment where our input to the LLM is a factual question with a ground truth. We extract the semantic markers from the output, and gradually remove parts of the original input. We want to show that removing parts of the input that are referenced in the traces of semantic markers indeed changes the output of the LLM compared to our ground truth (i.e., not just different wording of the answer, but a different conclusion compared to our ground truth). Conversely, we need to show that removing parts of the input that are not mentioned by the semantic markers does not change the output compared to our ground truth.

With the results of task one and two, we have a mechanism that allows us to extract traces of semantic markers from an LLM output, which we can use for delta debugging. Depending on where we are in the lifecycle of developing our LLM prompt, we can envision different delta debugging applications. In the early stages, when we are not confident that our prompt is reliable, we will work with an oracle (e.g., a human expert) to determine if the LLM output yields the correct answer. If the output is deemed incorrect, we can first reduce the size of our input by stripping of everything that was not mentioned in the trace of semantic markers, and then we conduct a binary partitioning to identify the smallest subset of markers that, when removed, change the output from an incorrect answer to a correct or inconclusive answer. We explore the feasibility of semantic marker for this use case in Section 3.3.

As a second application of delta debugging of prompts, we consider the case where our prompt is already stable and ready to run in production (remember, prompt instructions and data combined are the input to the LLM). In production, we treat LLMs like probabilistic oracles, and we send the same input multiple times to ensure we don't get hit by occasional hallucinations. Before deploying to production (or when changing the model), we want to ensure a certain threshold of how many of these multiple runs need to agree on the same result, e.g., 4 out of 5.

In this scenario, we can use our semantic markers for each of the 5 runs of the LLM, and then use delta debugging to identify the parts of the input that are unique to the outlier results. We explore this use case in Section 3.3 on a system that audits contractual data, and discuss how we use the delta debugging output to improve our prompts.

Contributions. In summary, our paper makes the following contributions:

(1) **Semantic Markers for LLM Debugging and Explainability.** We introduce a novel technique for embedding and recovering unique markers from LLM outputs, enabling traceability from output elements back to specific input fragments.
(2) **Delta Debugging for LLM Inputs.** We adapt delta debugging to use semantic markers as a means of isolating minimal input fragments responsible for unexpected or incorrect outputs.
(3) **Experience Report on Industry Use Case.** We conduct experiments on an Amazon internal LLM-backed document-processing system to evaluate whether semantic markers can be reliably extracted, whether they accurately identify causal input fragments, and whether they enable systematic prompt improvement.

We frame our evaluation in terms of the following research questions:

RQ1 Can semantic markers be reliably extracted from LLM outputs, and do they accurately link to the input fragments responsible for the observed outputs?
RQ2 Can semantic markers from multiple LLM runs be used with delta debugging to isolate input fragments that explain unexpected outputs?
RQ3 Can the results of marker-based delta debugging be used to systematically improve prompt quality and stability?

2 Approach

The goal of delta debugging is to find the minimal subset of the input responsible for the unexpected behaviors. In the scenario of LLM-integrated systems, unexpected behaviors include hallucinations or incorrect answers. In this section, we introduce our approach to debug LLM-integrated systems using a running example where we want to develop a service that can answer questions about AWS documentation. As developers, when we plan such a service, we would first get the relevant documentation text and put it into a text file, or a vector database (if we want to use RAG), and write some instruction prompts that put some general guardrails around how our service can answer, and which type of answers are out of

scope. Then we would create some question-answer pairs of factual questions that we want to use for testing.

Assume now, that while testing our service, one of our tests fails: When asked "After creating lambda functions defined as container images, I need to use the lambda API to manage them. For this, do I need to create the function from the same account as the container registry in Amazon ECR?", the system incorrectly responds "Yes" due to outdated information in the documentation context that we used for our service.

How do we debug this issue? And how do we identify this outdated part of the documentation in our input? There could be different reasons why the test failed: factual errors that cause deterministic failures, and ambiguous instructions that lead to inconsistent behavior.

To enable systematic debugging of LLM inputs, we need a mechanism that provides visibility into which parts of the input context influence the model's reasoning process. Traditional debugging approaches fail here because LLMs operate as black boxes, making it impossible to trace how specific input fragments contribute to the final output.

To address this challenge, our approach first injects unique semantic markers throughout the input context, then instructing the LLM to reference these markers during its chain-of-thought reasoning. This creates an explicit trace that maps each reasoning step to the input fragments that influenced it, providing the foundation for systematic delta debugging. We first describe how we use semantic markers to generate chain-of-thought traces in Section 2.1, then we demonstrate how to use such traces to run delta debugging. We evaluate them in Section 3.

2.1 Trace Generation from Chain of Thought

Our approach generates chain-of-thought traces from the reasoning process by seeding semantic markers into the input context (in our running example, the AWS documentation and the instruction prompts) and amending the input with additional instructions that force the LLM to quote with these markers as references during its reasoning. The key insight is that by seeding unique, identifiable markers throughout the input, we can create a mapping between the LLM's reasoning steps and specific portions of the original context, enabling precise traceability and explainability.

The trace generation process begins with semantic marker injection during input preprocessing. When processing documents such as PDFs or structured text, we parse the content using standard tools (e.g., PyMuPDF) to extract textual content. Then, each semantically meaningful unit (e.g., a sentence or a paragraph) is wrapped with a unique marker that preserves the original structure while adding traceability. As illustrated in Figure 1, after seeding semantic markers into input (i.e., the AWS documentation and the instruction prompt), each sentence is wrapped with a unique identifier (e.g., `<9aad558> ... </9aad558>`). These semantic markers are generated by hashing the wrapped context elements, enabling efficient verification of whether the model accurately quotes the original text by simply re-hashing the quoted content.

Figure 1 shows how chain-of-thought traces are generated for our running example. Given the question "Do I need to create the Lambda function from the same account as the container registry in Amazon ECR?" and our input with seeded semantic markers, the model outputs its references, trace, and a final answer. The instruction prompt that is added by our approach, explicitly instructs the model to quote relevant input segments using their unique identifiers in the reasoning process, generating a chain-of-thought trace (e.g., the highlighted trace `<ea76083>, ..., <5ba7c8d>` in the model output) that reflects the reasoning path. Rather than simply asking for conclusions, this prompting strategy requires the model to justify each reasoning step by referencing specific marked segments from the input, which we refer to as *reference segments*. This creates a structured trace where each logical inference is anchored to identifiable document elements.

It is worth noting that in this example, the trace immediately provides debugging clues that were previously impossible to obtain due to the black-box nature of LLM systems. The trace reveals that segment `<40272ea>` containing the outdated "same account" requirement is referenced in the model's reasoning, directly contributing to the incorrect answer. Without semantic markers, developers would have no visibility into which specific parts of the input influenced the model's decision-making process. The resulting chain-of-thought trace can be used to map each reasoning step back to its source for post-validation, and enables systematic debugging of the decision-making process.

2.2 Delta Debugging with Semantic Markers

With the aid of the chain-of-thought traces generated through semantic markers, we propose a two-phase delta debugging methodology that systematically isolates minimal problematic input fragments. This approach handles both scenarios with and without ground truth through a unified framework that progresses from trace labelling to failure-inducing context isolation using delta debugging techniques.

2.2.1 Labeling with Ground Truth. Delta debugging relies on ground truth that indicates whether a run is correct or incorrect. Given the input and output of a model run, we assign such labels using two complementary strategies, selected based on the availability and cost of ground-truth evaluation.

Oracle-guided Labeling. Oracle-guided labeling is used when ground truth can be reliably obtained. Ground truth may come from established references, from automated verification systems, or from human judgment when the test cases are manageable. Given a question and LLM input with semantic markers, we record the model's output and extract its chain-of-thought trace, then label the output as *correct* or *incorrect*. This approach ensures high label accuracy and is particularly effective during early development phases or for critical cases where precision is the top metric.

Oracle-free Labeling. Oracle-free labeling is used when ground truth is not available, such as in large-scale testing or when manual evaluation becomes impractical. In these cases, we leverage the model's self-consistency by applying majority voting across multiple runs. We execute the same query multiple times (e.g., $N \geq 10$ runs) with identical context and question, collecting both final answers and reference segments from each run's chain-of-thought trace. The most frequent answer is treated as the likely correct outcome, while minority answers are considered potential failures. This

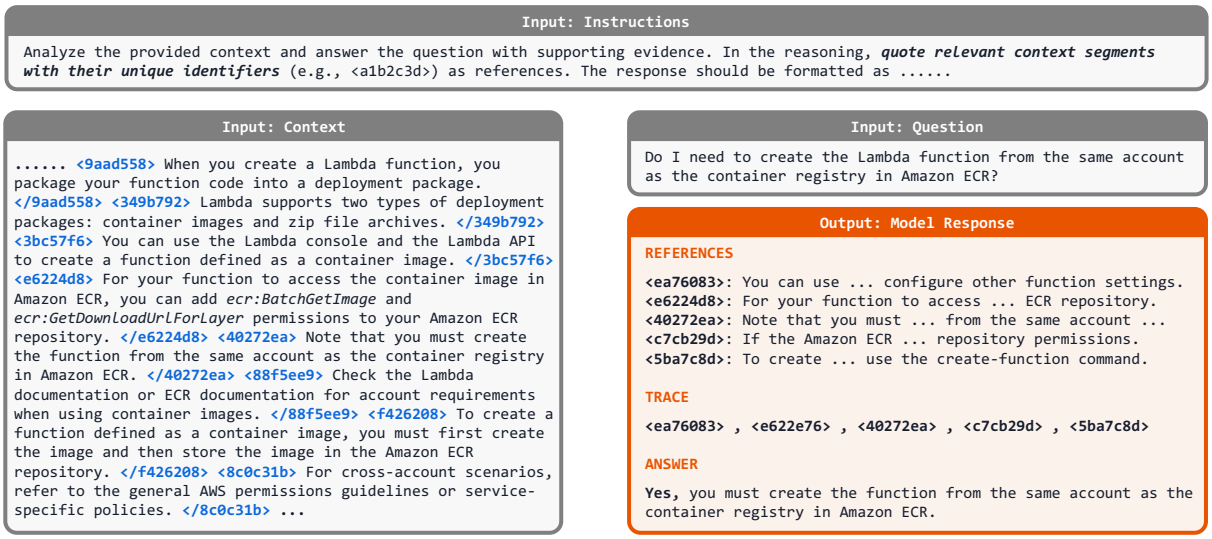

Figure 1: Semantic marker injection and chain-of-thought trace generation.

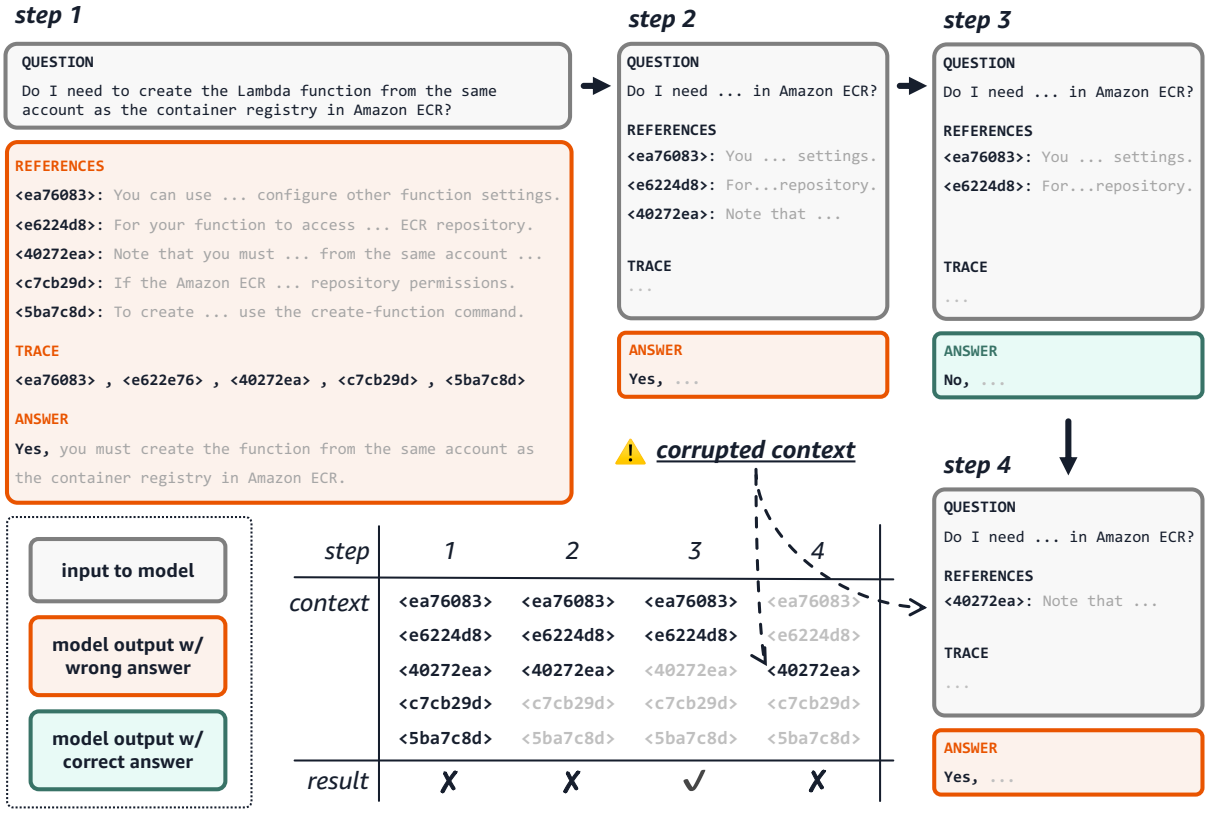

Figure 2: Delta debugging via binary partitioning.

statistical approach mirrors established self-consistency techniques in LLM reasoning, where majority voting across multiple samples improves answer reliability. When vote distributions are ambiguous, additional runs are generated to establish clearer consensus, ensuring robust pseudo-ground truth for subsequent analysis.

The combination of oracle-guided and oracle-free labeling mirrors the evolution of real-world software development. Early in system development, developers can manually inspect a small number of runs to provide high-confidence labels and fine-grained feedback. As systems scale and queries grow more complex, manual evaluation becomes infeasible, and labeling naturally shifts toward automated oracle-free strategies and statistical analysis. This progression ensures effective debugging throughout the system lifecycle while balancing accuracy with scalability.

2.2.2 Failure-Inducing Context Isolation. With labelled ground truth, we propose two delta debugging techniques to pinpoint the context segments contribute to failures. These methods represent different reduction strategies within the delta debugging framework and can be used individually or in combination depending on the specific debugging requirements.

Binary Partitioning. Binary partitioning represents the classical delta debugging approach, systematically testing subsets of context segments to identify minimal failure-inducing fragments through recursive partitioning. We adapt the `ND3MIN` algorithm, an extension of the traditional `ddmin` algorithm for handling non-determinism, to better align with the nature of LLM behaviors. For each subset test, we perform N repeated trials and require more than K failures out of N runs to classify a subset as failure-inducing. This threshold-based approach ensures statistical confidence in our outcomes, preventing false positives due to non-deterministic behaviors of LLMs.

To test subsets of context segments, we manipulate the chain of thought and guide the reasoning process by leveraging the semantic markers to constrain the model's attention to a chosen subset of references. In the AWS documentation example, the initial chain-of-thought trace contains 5 reference segments: `<ea76083>`, `<e6224d8>`, `<40272ea>`, `<c7cb29d>`, `<5ba7c8d>`. As illustrated in Figure 2, binary partitioning systematically tests subsets of these segments to isolate the failure-inducing fragment. For subset testing, we construct prompts with pre-filled partial responses containing only the subset of reference segments, allowing the model to complete the reasoning and give the final answer as a continuation. The algorithm recursively narrows down the search space, removing segments that do not contribute to the failure and further dividing those that do. This process continues until we identify a minimal subset where every segment is necessary for the failure. In our example, the process isolates segment `<40272ea>` containing the outdated "same account" requirement as the exact point from the input that is responsible for the faulty output. Removing this single segment causes the model to produce the correct answer, confirming its role as the failure-inducing segment.

Suspiciousness Scoring. This technique represents an alternative delta debugging strategy that leverages statistical analysis across multiple labeled traces to identify failure-inducing segments without explicit partitioning. Rather than systematically removing subsets as in classical delta debugging, this approach treats each context segment as a candidate for removal and uses correlation analysis to rank segments by their contribution to failures.

As illustrated in Figure 3, we aggregate traces from multiple runs, where • indicates segment presence and ∘ indicates absence in each run's chain-of-thought. In the AWS documentation example, when the context contains ambiguous instructions such as "Check the Lambda documentation or ECR documentation for account requirements", the model produces inconsistent behavior across multiple runs. Some runs may reference segments about Lambda-specific requirements while others focus on general ECR permissions, leading to different conclusions. For each segment s, we define the suspiciousness score as:

$$score(s) = \frac{|R_f^s|}{|R_f|} - \frac{|R_p^s|}{|R_p|}$$

where R_f^s and R_p^s represent the subsets of failing and passing runs that reference segment s, respectively.

This scoring mechanism captures both commission and omission errors. Positive scores indicate segments that correlate with failure (commission errors), identifying problematic content that misleads the model when present. Strongly negative scores identify segments whose absence correlates with failure (omission errors), highlighting critical information that leads to incorrect outcomes when overlooked. For instance, segment `<7b9a5ce>` appearing only in failing runs yields $score = 1$, indicating strong failure correlation, while segment `<2bf230a>` present only in passing runs yields $score = -1$, suggesting its absence contributes to failure. The resulting scores produce a prioritized ranking of potentially failure-inducing inputs, directing developers toward a systematic and efficient investigation.

2.3 Development Lifecycle Integration

The example of AWS documentation demonstrates how our approach addresses different failure types throughout the software development lifecycle. Early development phases typically exhibit clear, deterministic failures caused by factual errors like the outdated "same account" requirement in segment `<40272ea>`, making them suitable for binary partitioning-based delta debugging. As systems mature, debugging challenges shift toward subtle, non-deterministic issues caused by ambiguous instructions that create inconsistent behavior across multiple runs, requiring suspiciousness scoring for statistical pattern identification.

Both approaches leverage the same semantic marker infrastructure, creating a unified debugging framework that adapts to different failure characteristics. This complementary relationship transforms LLM debugging from ad-hoc manual inspection into a systematic methodology that scales with system complexity and development maturity.

3 Evaluation

We evaluate our approach by answering the following questions:

RQ1 To what extent do generated chain-of-thought traces represent authentic LLM reasoning? Semantic marker-based traces must reflect genuine reasoning pathways to serve as reliable foundations of subsequent debugging or analysis. Establishing the authenticity and fidelity of these traces is essential before they can be trusted as reliable representations of LLM decision-making processes for debugging purposes.

RQ2 How effective is delta debugging, based on the generated chain-of-thought traces, in isolating input segments responsible for unexpected outputs? Even if traces

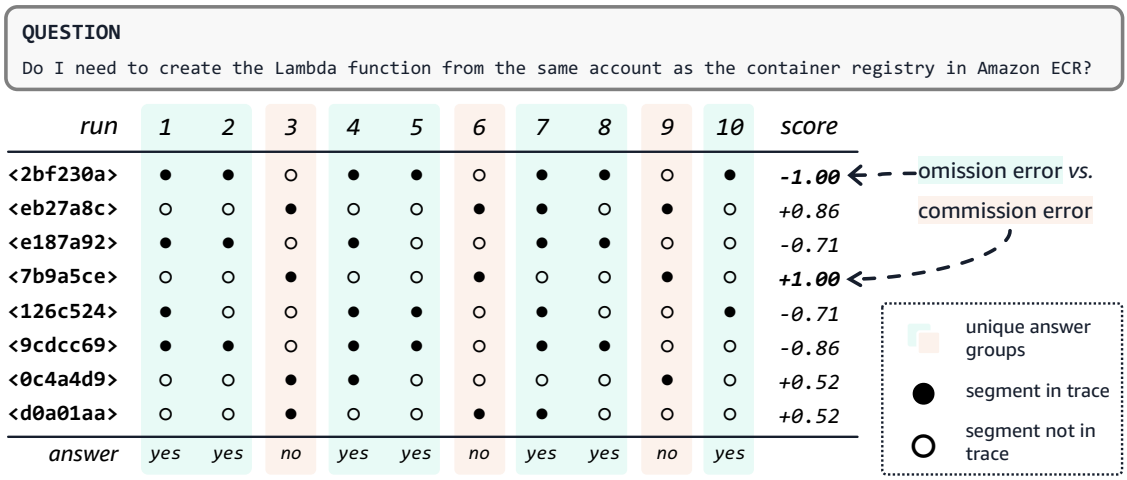

Figure 3: Delta debugging via suspiciousness scoring.

accurately represent reasoning processes, the practical utility of the approach depends on whether they can effectively guide the identification of problematic input segments. Demonstrating the effectiveness of trace-based delta debugging is crucial for validating the overall debuggability of LLM-based applications with our approach.

RQ3 **Can the outcome of trace-based delta debugging be used to systematically improve the context quality?** Beyond fault localization, the ultimate value of our debugging approach lies in its ability to provide actionable insights for system improvement. This question evaluates whether the identified problematic input segments can guide systematic enhancements to context curation, input preprocessing, or retrieval mechanisms, thereby validating the practical impact of our approach.

3.1 Experimental Setup

3.1.1 The FamilyTree Benchmark. We developed the *FamilyTree* benchmark, a controlled synthetic dataset that describes a customizable family across multiple generations. The family structure is described by only two direct relation types: *parent of* and *married to*, encoded as a formalized graph data structure where nodes represent family members and edges represent relationships.

The benchmark is accompanied by two automated engines: a context engine that generates natural language descriptions from the graph structure with precise definitions of a variety of relations, and a question engine that creates queries with controlled difficulty levels. The question engine generates questions requiring up to 5 hops of graph traversal, where each question automatically receives a difficulty score based on the number of hops (i.e., the number of facts required to be considered from the context for answering). This ranges from single-hop questions requiring direct relationship lookup (e.g., "who are the parents of Jack?") to complex multi-hop questions requiring traversal of up to 5 edges (e.g., "who are the cousins of Jack?").

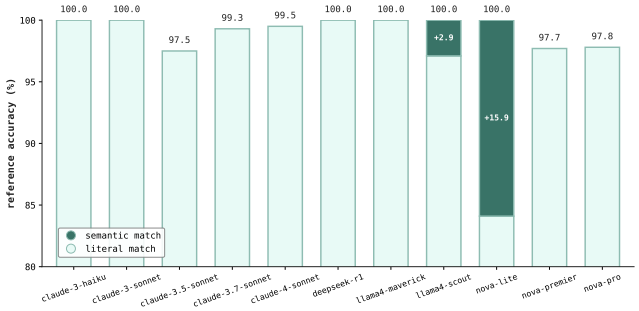

Figure 4: Reference accuracy of semantic markers.

This design fundamentally enables systematic evaluation with known ground truth, verifiable reasoning paths, and controllable complexity with the underlying graph data structure.

3.2 Fidelity of Chain-of-Thought Traces (RQ1)

To assess the reliability of our semantic marker-based approach, we evaluate two fundamental aspects: the accuracy of marker references and the authenticity of the generated chain-of-thought traces. These evaluations are essential to validate that our approach produces trustworthy debugging information.

3.2.1 Semantic Marker Reference Accuracy. We first investigate whether LLMs accurately reference the semantic markers when instructed to include them in their reasoning process. A key design feature of our semantic markers is their verifiability: each marker is generated by hashing the wrapped content, enabling efficient validation by re-hashing quoted segments and comparing them with the original marker strings.

We conducted experiments across 11 different models, with each model answering 12 questions from FamilyTree benchmark, with each question repeated 10 times. This generated a total of 1,320 traces with marker-segment pairs across all responses, where each

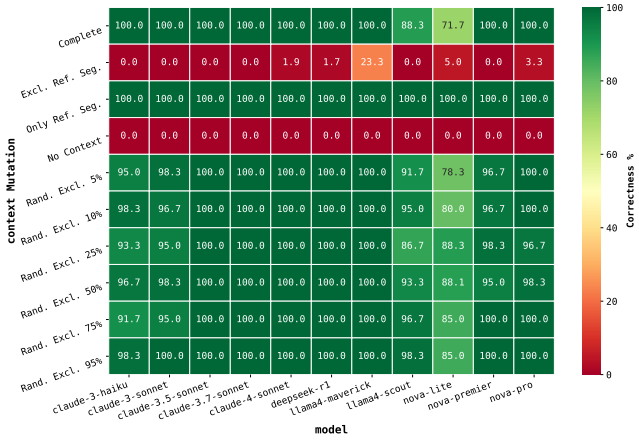

Figure 5: Mutation testing results for chain-of-thought trace authenticity validation.

pair represents a semantic marker referenced by the model and its corresponding quoted text segment. For each marker-segment pair, we performed two types of validation: *literal matching*, which leverages our hash-based marker design by re-hashing the quoted text and comparing it with the original marker to verify exact character-by-character correspondence, and *semantic matching*, which uses a judge LLM to evaluate whether minor variations preserve the original meaning. Figure 4 presents the reference accuracy results across all evaluated models.

The results demonstrate near-perfect semantic matching accuracy (100%) for almost all models, indicating that LLMs reliably reference the correct content when instructed to use semantic markers. Some discrepancies between literal and semantic matching occur primarily with definitional content, where models may omit redundant prefixes while preserving the core meaning. For example, when the original text states "child-in-law: A is a child-in-law of B if A is the spouse of B's child," the model only quotes "A is a child-in-law of B if A is the spouse of B's child," omitting the definitional prefix while maintaining semantic accuracy.

3.2.2 Chain-of-Thought Trace Authenticity. After verifying that models correctly reference individual markers, we must ensure that the complete chain-of-thought traces authentically represent the model's reasoning process. To validate this, we designed a mutation testing experiment that systematically mutates the context provided to the model and observes the impact on answer correctness.

Our hypothesis is that if the generated traces authentically reflect reasoning dependencies, then: (1) removing segments referenced in the trace should impair the model's ability to answer correctly, (2) providing only the traced segments should be sufficient for correct answers, (3) randomly removing unrelated segments should not affect performance as long as traced segments remain available, and (4) providing no context should result in poor performance, confirming that the model relies on the provided context rather than its own prior knowledge to answer the questions.

To test this hypothesis, we generated a set of multi-hop questions using our FamilyTree benchmark, where answering each question requires the model to examine and reference multiple segments from the provided context. We mutate context that directly correspond to our hypothesis: we preserved the full context as a control condition, excluded segments referenced in the trace to test hypothesis (1), provided only the traced segments to test hypothesis (2), randomly excluded varying percentages (5%, 10%, 25%, 50%, 75%, 95%) of unrelated context while preserving traced segments to test hypothesis (3), and provided no context at all to test hypothesis (4). To ensure statistical robustness, we ran each question under each context mutation 10 times for each model and averaged the results.

Figure 5 presents the mutation testing results as a heatmap, where each cell represents the answer correctness under a specific context mutation with a model. Despite some minor deviations, the results strongly support our hypothesis: models consistently fail when traced segments are removed, succeed when only traced segments are provided, maintain performance when unrelated segments are removed, and show poor performance without any context, confirming their dependence on the provided information. The only outlier is llama4-maverick, which still successfully answered 23.3% of questions when the reference segments were removed. Our manual investigation found that llama4-maverick was able to find another, albeit longer, reasoning path in the family tree to reach the correct answer. This pattern confirms that the semantic marker-based traces accurately capture the reasoning dependencies required for correct answers, validating the authenticity of our trace generation approach.

3.3 Effectiveness of Delta Debugging based on Semantic Markers (RQ2)

To evaluate the effectiveness of our semantic marker-based delta debugging, we conducted two complementary case studies: a controlled laboratory experiment and a real-world industrial system evaluation.

3.3.1 Lab-Controlled Case: FamilyTree Benchmark. We first validated our approach in a controlled setting using the FamilyTree benchmark to demonstrate systematic failure isolation under known conditions. We constructed a 42-segment context containing 16 family relation definitions and 26 direct relationships between family members, then introduced two types of failures to evaluate our debugging methodology.

In the first scenario, we planted factual errors in the relationship data while asking multi-hop questions that indirectly depended on the corrupted information. The LLM generated an initial chain-of-thought trace of 13 segments when producing incorrect answers. Binary partitioning-based delta debugging systematically reduced this to a 1-minimal set of 7 segments, where removing any single segment made the incorrect answer unreproducible. Critically, the minimal set correctly identified the planted erroneous facts despite the questions not directly querying the corrupted information, demonstrating the approach's ability to trace indirect dependencies.

In the second scenario, we introduced ambiguous family relation definitions and applied oracle-free labeling across 10 runs without ground truth. The LLM produced inconsistent answers due to the poorly defined relations. Suspiciousness scoring analysis clearly identified the ambiguous definition segments as having the

highest correlation with failures, with these segments appearing disproportionately in incorrect runs compared to correct ones.

Both scenarios validate that delta debugging reliably isolates failure-inducing fragments under controlled conditions, providing systematic evidence that the approach can distinguish between relevant and irrelevant input segments regardless of whether ground truth is available.

3.3.2 Real-World Use Case: AWS internal document processing system. We applied our approach at AWS to debug an internal service that is being developed for automating document auditing. The system is designed to cross-validate that contractual documents are operationalized correctly in JSON configurations files. Since this is an internal system, we need to omit the exact nature of these contractual documents, but we will outline the process about how they are being used. A contract, once signed by all parties, is passed to an analyst (a domain expert with the relevant legal knowledge to understand the intent of the contract). The analyst then uses a purpose-built internal website to enter the terms of the contract one by one. The terms are then stored in JSON configuration files and later used to process the customer's data. Contracts include certain identifiers, such as customer or products IDs, but also legal text that is not easily parsable by an automated system.

Our LLM-integrated auditing service integrates in this process to validate that the analyst accurately entered all contract terms into the system. To that end, our service takes three input components: (1) detailed instructions on how to conduct the audit with rules and special case handling, and (3) JSON-formatted data that was entered by the human analyst.

The instructions state that the LLM should return one of three values for each contractual term that it finds: "MATCH", if the contract and the JSON both contain the term and agree on the details; "DIFFERENCE" if both contain the term, but disagree on the meaning, or "MISSING" if only one of both contains the term. (2) contract text extracted from PDF documents.

Our evaluation demonstrates how the two delta debugging approaches complement each other across different phases of system development, addressing distinct failure patterns that emerge as the system matures.

Binary Partitioning for Clear Failures during Early Development. During initial development phases, engineers used dummy contracts to test system performance and encountered a persistent deterministic failure. When processing test contracts, the system would incorrectly flag account IDs as "DIFFERENCE" despite manual verification confirming the account identifiers were indeed identical between the contract text and JSON data. This failure occurred consistently across multiple test cases, making it a clear blocker for system development. The manifestation was straightforward: given input containing matching account information, the system would systematically produce incorrect mismatch reports, preventing progression to more complex testing scenarios.

The failing input contained 502 marked paragraphs of the input as relevant, encompassing the instruction prompt, contract text, and JSON data. The extensive length of the chain-of-thought trace made manual inspection impractical, as engineers could not efficiently identify which specific segments contributed to the incorrect behavior. When producing the incorrect "DIFFERENCE" output, the LLM generated a chain-of-thought trace referencing 21 segments consisting of instruction segments, contract text snippets, and JSON data excerpts.

We applied binary partitioning-based delta debugging to systematically isolate the failure-inducing fragments. The algorithm recursively partitioned the 21 referenced segments, testing each subset to determine whether it still produced the incorrect output. Through 28 systematic runs, the process eliminated segments that did not contribute to the failure while preserving those that did, ultimately converging on a minimal set of 3 segments: a paragraph from the instructions, one contract text snippet containing account information, and one corresponding data excerpt.

The isolated minimal set revealed the root cause: the instruction paragraph specified "12-digit account ID should strictly match" without accounting for common formatting variations such as hyphenated representations (e.g., "1111-2222-3333" vs "111122223333"). Armed with this precise diagnosis, developers corrected the instruction to accommodate formatting variations and added special case handling examples for hyphenated formats. After these targeted fixes, the account ID matching error was completely resolved, enabling the development team to progress to testing more complex contract scenarios.

Suspiciousness Scoring for Subtle Failures when System Matures. As the system matured and obvious deterministic failures were eliminated through iterative testing and fixing, engineers encountered a more subtle issue with contractual commitment terms, which are terms in the contract that specify a list of pairs of dates and monetary values. The system would sometimes produce correct commitment values and sometimes fail, creating an inconsistent and unreliable behavior pattern. Unlike the previous clear failure, this issue was non-deterministic and difficult to reproduce consistently. Manual inspection revealed that the system would correctly extract commitment values from some test contracts while producing incorrect values for seemingly similar contracts, with no obvious pattern distinguishing successful from failed cases.

Since ground truth was not immediately available for all test cases and the failure was non-deterministic, we applied oracle-free labeling with suspiciousness scoring. We executed the same problematic input 10 times, collecting both final answers and chain-of-thought traces from each run. The multiple executions produced three distinct answer groups with different commitment values. Using majority voting, we identified the most frequent answer as the likely correct outcome, treating the minority answers as potential failures.

Suspiciousness scoring analysis computed correlation scores for each segment based on its appearance frequency in failing versus passing runs. The analysis revealed high commission error correlation for two specific segments: an instruction sentence stating "Locate commitment section in the main contract or in the contract amendments" and a contract segment containing actual spend commitment information from the main contract body. These segments appeared disproportionately in runs that produced minority (incorrect) answers compared to majority (correct) answers.

This statistical analysis provided the crucial insight that the model was encountering ambiguity when spend commitments appeared in both main contracts and amendments simultaneously. The instruction failed to specify precedence rules for such cases,

leading to inconsistent model behavior depending on which information the model prioritized during reasoning. Understanding that the correct business logic requires prioritizing amendment information as it overrides main contract terms, developers added explicit precedence instructions and included special case examples demonstrating proper handling of conflicting spend commitment information. After these targeted improvements, the system achieved stable and consistent spend commitment extraction across all test scenarios.

Final Remarks. The development of our internal Document auditing system demonstrates how binary partitioning and suspiciousness scoring address different debugging needs throughout the system development lifecycle. Binary partitioning proved essential during early development for isolating precise root causes of deterministic failures with clear manifestations and available ground truth. As the system matured and obvious bugs were resolved, suspiciousness scoring became valuable for identifying subtle patterns in non-deterministic behaviors where manual evaluation was impractical, and ground truth was not immediately available. Both approaches led to concrete improvements, like corrected rule descriptions and enhanced special case handling, that systematically improved system reliability and transformed debugging from manual inspection to methodical fault isolation.

3.4 Delta Debugging-Guided Improvement (RQ3)

The ultimate goal of debugging is not merely to locate the fault but to enable systematic improvement of the target systems. With the delta debugging outcomes from Section 3.3, we have seen that our approach can identify relevant LLM input segements; now we want to inspect if we can further use these segments for automatic repair of the input. We show how a tool like Amazon Q Developer CLI, an agentic LLM command tool used by developers at AWS, can be used to process the outputs of our delta debugging from the previous section to generate patches.

Automated Factual Error Correction. For the contract auditing system described in Section 3.3, the binary partitioning process had isolated segment containing the faulty instruction about account ID formats, as well as two segments containing actual account ID data. Amazon Q Developer CLI received delta debugging outcomes from RQ2: (1) the full original input context, (2) the minimized result containing the 3 critical segments identified by binary partitioning, and (3) the incorrect "DIFFERENCE" output demonstrating the factual error. Without providing instructions or descriptions specific to the faulty segments in this case, Amazon Q Developer CLI successfully recognized that the 3 segments constitute the minimal set that causes unexpected behavior, and identified that the instruction segment contains an overly restrictive comparison rule causing semantic equivalence to be incorrectly flagged as differences.

Amazon Q Developer CLI automatically generated a patch that corrected the logic from strict formatting to semantic equivalence. The fix required only two lines of modification of the instruction bit of the LLM input, totaling 36 tokens from the original input of 29,516 tokens, representing a modification rate of 0.12% while directly addressing the root cause identified by delta debugging.

The generated patch exhibited optimal characteristics: surgical precision with minimal modifications targeting only the problematic segment identified, semantic correctness that preserved audit functionality while fixing the logic error, and causal alignment that directly addressed the fault segment. The automated approach generated a minimal corrective patch without human intervention, demonstrating that delta debugging outcomes provide actionable insights for systematic system improvement and self-healing.

Automated Instruction Enhancement. We next applied the same improvement methodology to the commitment value inconsistencies identified in mature development scenario discussed in Section 3.3. The suspiciousness scoring had identified ambiguous instruction segments causing non-deterministic behavior in spend commitment extraction. We used these results to demonstrate automated prompt instruction enhancement.

Without any fixes, during the 10 runs of the same input with the same model, the model's output could be divided into three groups with 3 different traces, where the most common trace appeared in 6 out of 10 runs. The suspiciousness scoring results had identified three highly influential segments contributing to reasoning inconsistency. The highest suspiciousness score for a segment is 0.70, indicating insufficient guidance for consistent decision-making.

For automated instruction enhancement, Amazon Q Developer CLI received (1) the original input and (2) the segments with the top 5 suspiciousness scores. During its analysis, Amazon Q Developer CLI recognized that the instructions for handling spend commitments from main contracts versus amendments were ambiguous and proposed a patch (with 37 tokens) that elaborates the instructions with "When amendments modify the original contract terms, the amended values take precedence over the original contract values."

Post-improvement validation demonstrated significant stability enhancement. Running the same 10 experiments with the improved prompt achieved perfect trace consistency: all 10 runs produced identical execution traces and reached 100% output accuracy. The number of unique traces reduced from 3 to 1, and outlier runs were completely eliminated, confirming that the targeted improvements successfully resolved the underlying reasoning inconsistencies identified.

Final Remarks. Both examples demonstrate that delta debugging outcomes enable systematic system improvements through automated analysis and targeted modifications. The factual error correction leveraged binary partitioning results to achieve targeted fixes with only 0.12% modification of the input context. The prompt instruction enhancement used suspiciousness scores to eliminate reasoning inconsistencies entirely, reducing trace variance from 3 unique patterns to 1 consistent pattern across all runs. The patches proposed by Amazon Q Developer CLI are highly aligned with the developers' patches discussed in Section 3.3, except that the Amazon Q Developer CLI patches do not contain additional examples as it was not explicitly instructed to do so.

We are not able to collect or share representative data on how developers use delta debugging internally, we have anecdotal evidence that the experience aligns with what we discussed in this paper. A promising observation that we leave for future work is that delta debugging integrated very well with our agentic developer tool Amazon Q Developer CLI. It is worth exploring if it is possible

to provide generic delta debugging tools as Model-Context Protocol (MCP) servers to assist developers in other debugging tasks.

4 Related Work

Since the introduction of Delta Debugging [14] 25 years ago [16], the approach of isolating root causes of system failures by comparing artifacts of passing and failing executions has been used in a variety of settings. Approaches vary in the type of artifacts they consider (e.g., inputs, executions traces, or logs), and in the number of executions they need (e.g., automated reasoning based fault localization that requires only a single example, vs machine learning based approaches that need large qualities of logs), but they all share the motivation of helping developers to quickly focus on the root cause of a bug.

Debugging by instrumenting inputs. A key part of our approach is to instrument the LLM input with Semantic Markers which we can then use for debugging. The idea of instrumentation and the use of capturing tools to assist debugging has been applied successfully in other contexts before:

Beschastnikh et al [1] present an approach that enhances logs of distributed systems for debugging, so they can extract *happens-before* sequences of events, which allow further fault localization and visualization.

JINSI [9] instruments Java applications to enable capture and replay of component-based systems and provides delta-debugging on top of that.

Other approach use capture-replay tools instead of specific instrumentation to collect data for delta debugging, such as [11].

Delta Debugging of different artifacts. While the original Delta Debugging approach [14] operated on input values, the approach has now been used on a variety of artifacts.

Clapp et al [2] apply delta debugging to GUI event trace.

Park et al [10] present a tool for debugging for concurrent systems by monitoring data access patterns between passing and failing executions. The Snowboard tool [4] localizes concurrency bugs in OS kernels by identifying shared memory usage between executions of generated tests.

The CHESS tool [8] detects and diagnoses Heisenbugs in multithreaded programs by taking control of the thread scheduler and to generate passing and failing interleavings for fault localization. Similarly, Zhou et al [18] introduce an approach for debugging micro-service architectures through a dedicated delta debugging controller.

LLM Explainability. The field of LLM explainability [6, 17] encompasses several approaches for understanding model behavior, each with distinct limitations for systematic debugging. Feature attribution techniques like Integrated Gradients [3] and SHAP require model internals and are computationally expensive. Attention visualizations may be unreliable [5]. Perturbation methods only observe input-output changes without understanding internal reasoning. Chain-of-thought (CoT) prompting [13] elicits step-by-step reasoning, but CoT explanations can sometimes be unfaithful to the actual reasoning process [12], i.e., models can generate plausible explanations that don't reflect actual reasoning, limiting CoT's debugging utility.

Our semantic markers approach combines CoT with input instrumentation. Unlike perturbation methods that modify inputs externally, or CoT that provides post-hoc explanations, we instrument inputs with markers that force the LLM to explicitly cite which input fragments it uses during reasoning.

5 Conclusion

Debugging LLM-integrated systems is challenging due to their black-box nature and large inputs. This paper introduces semantic markers that embed unique identifiers in LLM inputs and extract traceability from chain-of-thought reasoning, enabling delta debugging to systematically isolate problematic input fragments. We evaluate binary partitioning for deterministic failures and suspiciousness scoring for inconsistent behaviors. Results show 100% reference accuracy across models, effective isolation of minimal failure-inducing fragments (reducing 502 segments to 3 in our real-world example), and automated system improvement with 0.12% modification rates while eliminating reasoning inconsistencies. This transforms debugging for LLM-integrated systems from manual inspection into systematic methodology for reliable production systems. Beyond debugging applications, this approach also addresses growing demands for LLM explainability in regulatory contexts, providing systematic traceability to explain LLM outputs.

References

[1] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. Debugging distributed systems. *Commun. ACM*, 59(8):32–37, 2016.

[2] Lazaro Clapp, Osbert Bastani, Saswat Anand, and Alex Aiken. Minimizing GUI event traces. In *SIGSOFT FSE*, pages 422–434. ACM, 2016.

[3] Mengnan Du, Varun Manjunatha, Rajiv Jain, Ruchi Deshpande, Franck Dernoncourt, Jiuxiang Gu, Tong Sun, and Xia Hu. Towards interpreting and mitigating shortcut learning behavior of NLU models. In *NAACL-HLT*, pages 915–929. Association for Computational Linguistics, 2021.

[4] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In *SOSP*, pages 66–83. ACM, 2021.

[5] Sarthak Jain and Byron C. Wallace. Attention is not explanation. In *NAACL-HLT (1)*, pages 3543–3556. Association for Computational Linguistics, 2019.

[6] Haoyan Luo and Lucia Specia. From understanding to utilization: A survey on explainability for large language models. *CoRR*, abs/2401.12874, 2024.

[7] James Mickens. The night watch. In *USENIX Login Logout (USENIX 13)*, 2013.

[8] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, pages 267–280. USENIX Association, 2008.

[9] Alessandro Orso, Shrinivas Joshi, Martin Burger, and Andreas Zeller. Isolating relevant component interactions with jinsi. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, WODA '06, page 3–10, New York, NY, USA, 2006. Association for Computing Machinery.

[10] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. Falcon: fault localization in concurrent programs. In *ICSE (1)*, pages 245–254. ACM, 2010.

[11] Tobias Roehm, Stefan Nosovic, and Bernd Bruegge. Automated extraction of failure reproduction steps from user interaction traces. In *SANER*, pages 121–130. IEEE Computer Society, 2015.

[12] Miles Turpin, Julian Michael, Ethan Perez, and Samuel R. Bowman. Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting. In *NeurIPS*, 2023.

[13] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*, 2022.

[14] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC / SIGSOFT FSE*, volume 1687 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 1999.

[15] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.

[16] Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input: A Retrospective on Delta Debugging . *IEEE Transactions on Software Engineering*, 51(03):820–824, March 2025.

[17] Haiyan Zhao, Hanjie Chen, Fan Yang, Ninghao Liu, Huiqi Deng, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, and Mengnan Du. Explainability for large language models: A survey. *ACM Trans. Intell. Syst. Technol.*, 15(2):20:1–20:38, 2024.

[18] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Wenhai Li, Chao Ji, and Dan Ding. Delta debugging microservice systems. In *ASE*, pages 802–807. ACM, 2018.