Debugging the Debuggers: Failure-Anchored Structured Recovery for Software Engineering Agents

Chenyu Zhao
Nankai University
Tianjin, China

Shenglin Zhang*
Nankai University
Tianjin, China

Yihang Lin
Nankai University
Tianjin, China

Wenwei Gu
Nankai University
Tianjin, China

Zhimin Chen
Yongqian Sun
Nankai University
Tianjin, China

Dan Pei
Tsinghua University
Beijing, China

Chetan Bansal
Saravan Rajmohan
Microsoft
Redmond, USA

Minghua Ma
Microsoft
Redmond, USA

Abstract

Software engineering agents are increasingly deployed in evaluable engineering environments, yet post-failure recovery remains costly, manual, and largely ad hoc. Existing systems expose traces for inspection or generate follow-up feedback, but they do not turn heterogeneous runtime evidence into grounded, bounded recovery guidance for a subsequent attempt. We present *PROBE*, a failure-anchored framework for structured recovery in software engineering agents. *PROBE* organizes failed-run telemetry into structured evidence, structured diagnosis, and bounded recovery guidance, realized by a Telemetry Layer, a Diagnosis Layer, and a Guidance Gate. The Telemetry Layer preserves fine-grained runtime signals, the Diagnosis Layer fuses cross-signal evidence into a grounded diagnosis, and the Guidance Gate constructs diagnosis-derived recovery guidance only when it is evidence-grounded, actionable, and within the scope of agent-side behavior.

We evaluate *PROBE* across three software engineering settings spanning repository-level software repair, enterprise workflow recovery, and AIOps service mitigation. On 257 initially unresolved cases, *PROBE* achieves 65.37% Top-1 diagnosis accuracy and a 21.79% recovery rate, outperforming the strongest non-*PROBE* baseline by 43.58 and 12.45 percentage points, respectively. The results reveal a substantial diagnosis–recovery gap: accurate diagnosis is necessary but not sufficient unless it can be translated into bounded guidance that the subsequent attempt can execute and verify. Beyond controlled evaluation, a Microsoft IcM prototype shows that *PROBE* can attach as a non-intrusive side channel to existing service-diagnosis agent workflows without changing the agent policy, toolset, or execution budget. These results suggest that telemetry-grounded, failure-anchored recovery is a practical path to improving post-failure recoverability of software engineering agents under realistic engineering constraints.

*Corresponding author.

Conference'17, Washington, DC, USA

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

CCS Concepts

• **Software and its engineering** → **Software maintenance tools**; • **Computing methodologies** → *Planning and scheduling*; Knowledge representation and reasoning; • **Computer systems organization** → Maintainability and maintenance.

Keywords

LLM agents, AgentOps, failure recovery, telemetry, structured diagnosis, bounded guidance

ACM Reference Format:
Chenyu Zhao, Shenglin Zhang, Yihang Lin, Wenwei Gu, Zhimin Chen, Yongqian Sun, Dan Pei, Chetan Bansal, Saravan Rajmohan, and Minghua Ma. 2026. Debugging the Debuggers: Failure-Anchored Structured Recovery for Software Engineering Agents. In . ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Software engineering agents powered by large language models are increasingly deployed in complex real-world settings, including repository-level code repair [2, 37, 41], enterprise workflow automation [23], and cloud service operations [1, 9, 10, 17]. As these agents tackle broader and more complex tasks, failed executions have become a pervasive practical concern [26, 37]. In practice, such failures are often handled by inspecting a few error messages, appending prompt instructions, and rerunning the agent [5, 38]. However, this ad hoc approach is unreliable: naive retry has been shown to yield diminishing returns without targeted feedback [26]. More importantly, such recovery is weakly tied to failed-run provenance, obscuring which evidence should guide the subsequent attempt and which behaviors should be avoided.

In practice, failed executions produce rich runtime evidence. However, most existing benchmarks focus primarily on whether the task objective is satisfied and do not preserve the fine-grained diagnostic signals needed to support recovery [18, 23]. Recent analyses further show that many agent failures stem from recurring behavioral patterns rather than random errors [6, 25, 37]. Process-oriented studies show that execution errors can be categorized at the step level and that recurring coding-agent misbehaviors, such as specification drift, hallucinated reasoning, and malformed tool calls, can often be addressed through targeted guidance [11, 25]. These findings indicate that failed executions already contain actionable evidence for subsequent attempts, yet existing mechanisms do not

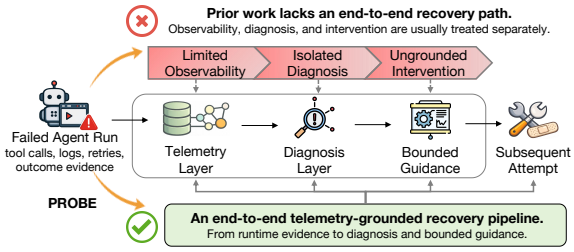

Figure 1: Prior work treats observability, diagnosis, and intervention separately. *PROBE* connects runtime evidence, diagnosis, and bounded guidance into one recovery loop.

systematically turn such evidence into diagnosis and recovery guidance.

Recent work and emerging practice have begun to make agent executions more observable, measurable, and diagnosable [14]. However, as Figure 1 illustrates, existing efforts remain fragmented: observability systems expose telemetry and traces for inspection [12, 19]; diagnosis-oriented methods localize critical failure steps or generate explanations [3, 8, 33]; and iterative refinement methods provide feedback for subsequent attempts [22, 32], but such feedback is typically weakly grounded in the runtime telemetry most relevant to recovery. Taken together, prior work improves inspection, explanation, or retry feedback, but does not connect runtime telemetry, failure interpretation, and follow-up intervention into a single recovery process.

We make this missing recovery process concrete as a progression from telemetry to evidence, diagnosis, and guidance. Runtime telemetry can be organized into *structured evidence*: localized and fused failure-relevant signals preserved with provenance. Such evidence can support a *structured diagnosis*: a schema-constrained account of the failure anchor, primary cause, behavioral mistake, supporting evidence, and confidence. The diagnosis can then be translated into *bounded recovery guidance*: actionable and verifiable instructions for the subsequent attempt. This progression leads to three practical challenges.

Challenge 1. Preserving recovery-critical evidence. Effective recovery depends on more than a final error message or a coarse execution summary. It requires retaining evidence such as exception signatures, repeated tool failures, execution order, agent and environment state, and evaluator feedback. Since this evidence is scattered across execution steps, it is easily lost when telemetry is compressed into generic summaries or undifferentiated logs before being organized into structured evidence.

Challenge 2. Building structured diagnosis. Runtime signals differ in source, format, granularity, and reliability. Metrics capture progress and resource use, logs record localized failures, traces preserve temporal structure, while status, environment, and outcome signals provide contextual information at the execution and task levels. A useful framework should integrate these signals without collapsing their distinct roles, and produce a structured diagnosis that remains traceable to evidence and usable for recovery guidance.

Challenge 3. Producing grounded and bounded guidance. A diagnosis explains the failure, but not how the subsequent attempt should act. Guidance must be evidence-grounded to avoid unsupported recovery instructions, and bounded to constrain the agent to verifiable, in-scope actions. A practical system must therefore translate diagnosis into guidance with an explicit target, operation, verification signal, and boundary condition.

To address these challenges, we present *PROBE*, an end-to-end framework for failure-anchored structured recovery in software engineering agents. *PROBE* uses failed execution telemetry as the anchor for recovery and realizes the progression from structured evidence to structured diagnosis and bounded recovery guidance through a *Telemetry Layer*, a *Diagnosis Layer*, and a *Guidance Gate*. The *Telemetry Layer* instruments the agent run at span level and organizes failed-run telemetry into typed signal families, including metrics, logs, traces, agent intent, tool–environment state, and optional external outcome signals. The *Diagnosis Layer* localizes failure-relevant signals and fuses them into structured evidence, then derives a structured diagnosis through anchor-first diagnosis generation. The *Guidance Gate* acts as a grounding, actionability, and scope filter, converting the diagnosis into bounded recovery guidance only when it is telemetry-supported, actionable, and within the scope of agent-side behavior.

To make the role of the *Guidance Gate* concrete, consider a service-remediation case from AIOpsLab (Section 4.3.1). A structured diagnosis can identify a misconfigured `targetPort` and premature submission, but it does not determine how the subsequent attempt should proceed. *PROBE* therefore translates the diagnosis into bounded recovery guidance by specifying the target to revisit, the operation to perform, the verification signal to check, and the boundary condition that prevents premature completion.

Our contributions are as follows.

- We formulate *failure-anchored structured recovery* for software engineering agents as a transformation from failed-run telemetry to structured evidence, structured diagnosis, and bounded recovery guidance for a subsequent attempt.
- We implement *PROBE* as a portable, framework-agnostic Python package that provides side-channel recovery support through span-level telemetry, evidence-aware diagnosis, and generation of grounded and bounded guidance, without modifying the agent policy, toolset, executor, or evaluator.
- We evaluate *PROBE* across three software engineering settings: SWE-bench [18], EnterpriseOps-Gym [23], and AIOpsLab [9]. On 257 initially unresolved cases, *PROBE* achieves 65.37% Top-1 diagnosis accuracy and a 21.79% recovery rate, outperforming the strongest baseline by 43.58 and 12.45 percentage points while revealing a diagnosis–recovery gap.
- We validate the industrial integration boundary through a non-intrusive Microsoft IcM service-diagnosis prototype, showing that *PROBE* can attach to existing service-diagnosis agent workflows without modifying the agent policy, toolset, or execution budget.

2 Background and Motivation

2.1 Agent Stacks and Execution Layers

Modern software engineering agents are typically realized as layered, tool-using systems rather than single model calls. Across code repair, enterprise workflow automation, and AIOps service

Table 1: Human-annotated failure-cause taxonomy for the 257 initially unresolved cases.

Reviewed failure cause	Keywords	Explanation	Count
Insufficient validation	missing verification, outcome unchecked, incomplete confirmation	The agent proceeded without adequately verifying whether the intended outcome was achieved.	83
Tool/subprocess failure handling	tool anomaly, timeout, execution failure, invalid output	The agent failed to correctly recognize or respond to tool-level execution anomalies.	47
State/workflow error	wrong target, state mismatch, missing field, workflow violation	The agent operated on the wrong target or left the workflow in an incorrect final state.	42
Patch/submission workflow	premature submission, incomplete artifact, workflow breakdown	The run broke down near artifact production or submission, leaving the final deliverable incomplete or invalid.	35
Retry/no-progress loop	repeated retries, no progress, no adaptation, stalled loop	The agent repeated ineffective actions without making meaningful progress or adapting strategy.	26
Runtime/environment handling	environment precondition, missing dependency, infrastructure blocker	Execution was blocked by an unresolved environment or infrastructure precondition.	24

mitigation, runs involve multi-step interactions with tools and execution environments [2, 3, 37, 41]. These agents expose recurring execution boundaries, including model calls, orchestration decisions, tool interactions, workflow state updates, and evaluator feedback [15, 34, 36, 47]. Representative stacks such as LiteLLM [4], LangChain-style orchestration [7], MCP [24], and tool/API-oriented agents [21, 27, 28, 30] instantiate these boundaries in different ways.

This common boundary structure motivates *PROBE*'s framework-agnostic design. Rather than encoding recovery logic for a specific agent framework, *PROBE* records recovery-relevant signals through a shared telemetry schema that can be instantiated across heterogeneous stacks [34, 36, 39, 47]. In our implementation, the same schema supports LiteLLM-based SWE-agent runs [41], LangChain-based ReAct with MCP-mediated tool access in EnterpriseOps-Gym [23], and AIOpsLab [9], keeping the recovery pipeline unchanged across settings.

2.2 Related Work

Prior work covers several ingredients of recovery for software engineering agents, but usually treats them as separate problems rather than as stages of a single recovery process. Observability systems such as AgentOps [12] and LangSmith [19] collect telemetry and expose execution traces for inspection, but they primarily treat telemetry as an inspection artifact rather than converting it into structured diagnosis or bounded recovery guidance. Diagnosis-oriented work studies failure localization, explanation, and debugging support [3, 6, 8, 10, 17, 40]; for example, AgentRx [3] localizes critical failure steps with a cross-domain taxonomy, and LLMRCA [33] shows that metrics, logs, and traces can support multilevel diagnosis for deployed LLM applications. Cloud-incident systems further show the value of using historical incidents, runtime diagnostic information, and domain-specific queries for diagnosis and incident investigation [1, 10, 17, 45]. Traceability-oriented work such as CodeTracer [20] reconstructs code-agent logs into hierarchical trace trees, localizes failure-responsible stages, and uses localized diagnostic signals for reflective replay. Refinement methods such as Reflexion [32] and Self-Refine [22] generate feedback for a later attempt, but the feedback is often only weakly grounded in runtime telemetry and is typically not bounded by a structured diagnosis. Overall, existing methods either expose traces for inspection, localize failure-relevant steps for explanation or replay, or provide generic feedback for retry. What remains missing is a recovery process that treats failed-run telemetry as the anchor, converts multi-signal evidence into a structured diagnosis, and admits only grounded, actionable, and in-scope guidance into the subsequent attempt.

2.3 Motivation

We focus on software engineering agents whose executions admit task-level evaluation, where each run can be judged as success or failure by an environment-specific evaluator, verifier, or benchmark harness. To understand what makes failed executions recoverable, we analyze the 257 initially unresolved first-attempt reports used in our recovery evaluation: 102 from SWE-bench [18], 106 from EnterpriseOps-Gym [23], and 49 from AIOpsLab [9].

For each report, two professional software engineers independently inspect the saved execution trajectory, runtime records, and outcome evidence, without access to *PROBE*'s structured diagnosis or gated guidance, and assign a reviewed failure-cause category. To avoid circular evaluation, the annotators do not use *PROBE*'s structured diagnosis or gated guidance when assigning the reviewed failure cause. Disagreements are resolved through discussion and category consolidation. This analysis clarifies what recovery-relevant evidence is already present in failed executions before any recovery feedback is generated, and why final outcome labels alone are insufficient for iterative recovery.

2.3.1 Finding 1. Initially unresolved runs are dominated by process-level failures. The initially unresolved reports are not dominated by a single surface error. Instead, they exhibit recurring process-level failure modes, as summarized in Table 1. Across all 257 reports, the largest categories are insufficient validation, tool/subprocess failure handling, and state/workflow error, which together account for 172 cases (66.93%).

This distribution suggests that failed executions are better understood as process failures than as isolated final verdicts. Many failures arise from how the agent interprets tool feedback, verifies intermediate results, reacts to runtime conditions, and decides

whether a run is ready to terminate or submit. These results motivate recovery support that operates on execution evidence rather than relying only on a final failure signal.

2.3.2 Finding 2. Effective recovery requires heterogeneous runtime signals. The failure categories summarized in Table 1 cannot be reliably identified from final evaluator outcomes alone. A final unresolved verdict indicates that the task objective was not satisfied, but it rarely explains whether the failure comes from missing validation, broken workflow execution, repeated tool errors, incorrect state transitions, or runtime/environment handling. These distinctions require inspecting how the execution unfolded over time.

Different failure categories depend on different combinations of runtime evidence. For example, insufficient validation requires linking intermediate actions, evaluator outputs, and the agent's decision to stop or submit. Tool/subprocess failure handling depends on tool-call sequences, return codes, and whether the agent adapts after failure. Runtime/environment-related failures require distinguishing agent-recoverable configuration issues from external infrastructure blockers. No single raw signal is sufficient across these categories.

These observations motivate the design of *PROBE*. To support iterative recovery for software engineering agents, the system should preserve fine-grained runtime records, organize heterogeneous signals into structured evidence, diagnose the actionable failure cause, and pass only bounded, evidence-grounded guidance to the subsequent attempt.

3 PROBE

3.1 Overview

PROBE models post-failure recovery for software engineering agents as a structured transformation from failed-run telemetry to bounded recovery guidance for the subsequent attempt. As shown in Figure 2, the *Telemetry Layer* records failed-run telemetry into typed signal families, the *Diagnosis Layer* localizes and fuses these signals into structured evidence before deriving a structured diagnosis, and the *Guidance Gate* validates and constructs bounded recovery guidance. Formally, let a failed run produce a telemetry bundle

$$\mathcal{T} = \{T_{\text{metrics}}, T_{\text{logs}}, T_{\text{traces}}, T_{\text{intent}}, T_{\text{env}}, T_{\text{outcome}}\}, \tag{1}$$

where each component corresponds to one signal family, and T_{outcome} is included when external evaluator signals are available.

PROBE processes the telemetry bundle through the structured pipeline

$$\mathcal{T} \to \mathcal{E} \to \mathcal{D} \to \mathcal{G}, \tag{2}$$

where $\mathcal{T}$ denotes the typed telemetry bundle, $\mathcal{E}$ denotes structured evidence formed by localizing and fusing failure-relevant telemetry signals, $\mathcal{D}$ denotes the structured diagnosis object, and $\mathcal{G}$ denotes bounded recovery guidance. This benchmark-agnostic schema abstracts model-access, orchestration, and tool-interface events into a shared telemetry representation, enabling *PROBE* to instrument existing agents without changing their policy, toolset, executor, or evaluator.

3.2 Telemetry Layer

To preserve recovery-critical evidence, the *Telemetry Layer* records runtime events as span-level telemetry and organizes them into typed signal families. The resulting telemetry bundle $\mathcal{T}$ preserves signal provenance for downstream evidence construction and diagnosis.

3.2.1 Telemetry Representation. The *Telemetry Layer* represents a software engineering agent run using four notions: spans, traces, intent, and tool–environment state. A *span* σ_t is the smallest recorded execution unit at step t, such as a model response, tool call, verifier result, metric snapshot, or runtime exception. Each span records its event type, timestamp, payload, status, and available tool, model, or error metadata.

A *trace* is the temporally ordered sequence of spans from one run. It preserves execution structure such as repeated cycles and delayed verification. To capture agent-side behavior, *PROBE* augments each step with intent i_t, which describes what the agent is trying to do, such as gathering evidence, editing artifacts, running verification, or preparing submission. When explicit phase labels are unavailable, *PROBE* infers intent from model messages, tool choices, command types, verifier calls, and submission events. To capture environment-side context, *PROBE* records tool–environment state s_t, including tool-return status, workspace or workflow state, and evaluator-side signals. Together, i_t and s_t distinguish what the agent intended to do from what the environment actually exposed or changed.

3.2.2 Signal Families. Given this representation, *PROBE* partitions collected telemetry into signal families with distinct diagnostic roles.

Metrics. *PROBE* derives T_{metrics} from runtime counters, span-level metadata, and scalar features computed from intent i_t and tool–environment state s_t. We organize these metrics into four recovery-oriented groups. *Cost and capacity* capture token velocity, context saturation, tool-call density, and retry dominance. *Recovery progress* measures observable state changes in s_t, or uses intent transitions in i_t as a fallback when explicit workflow progress is unavailable. *Progress–cost coupling* characterizes whether observed progress is commensurate with the resources consumed in the same window. *Behavioral stability* captures intent volatility, intent run-length ratio, and tool-switch volatility. Together, these groups help identify runs that make little progress, consume resources without corresponding state change, or fall into repetitive or unstable action patterns.

Logs. T_{logs} is extracted from tool returns, command outputs, parser failures, runtime exceptions, and system messages. *PROBE* deduplicates repeated messages, truncates long outputs, and canonicalizes common errors into reusable signatures. This preserves local failure anchors such as repeated errors, invalid tool arguments, subprocess failures, and environment-side breakdowns.

Traces and intent. T_{traces} and T_{intent} capture the temporal structure and functional role of each step in the run. They expose repeated failures, missing progress, unstable action transitions, and role shifts across the run.

Tool–environment state. T_{env} records environment-side context, including tool availability, tool-return status, workspace or

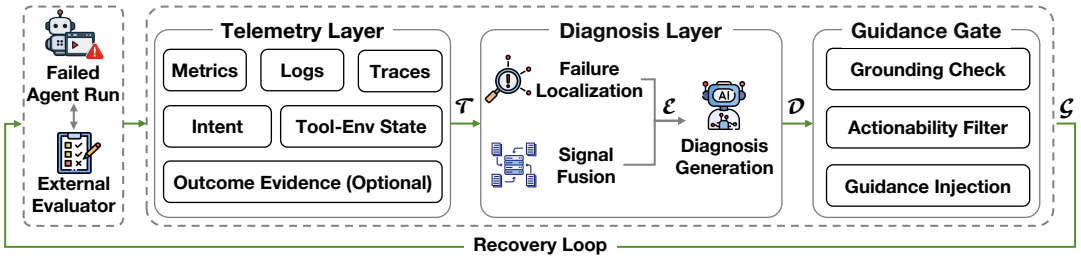

Figure 2: Overview of the *PROBE* framework. *PROBE* organizes failed-run telemetry into a typed telemetry bundle ($\mathcal{T}$), constructs structured evidence ($\mathcal{E}$), derives structured diagnosis ($\mathcal{D}$), and produces bounded recovery guidance ($\mathcal{G}$), forming a recovery loop for subsequent attempts.

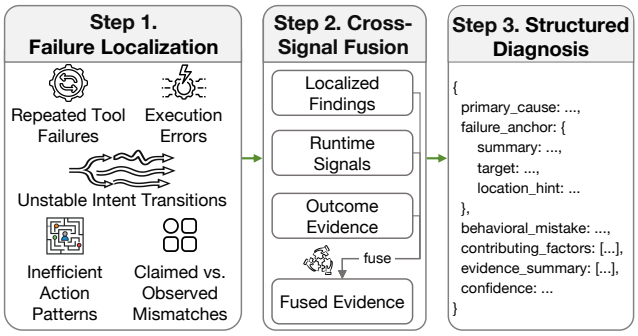

Figure 3: Detailed workflow of the *Diagnosis Layer*.

workflow state, and evaluator-side observations. It helps distinguish agent-side behavioral failures from environment-side constraints.

Optional external outcome evidence. T_{outcome} is included when evaluator-side feedback is available, such as benchmark verdicts, test results, incident-resolution checks, or task-specific scoring scripts. It grounds telemetry against the task objective and helps detect mismatches between agent claims and evaluator observations.

3.3 Diagnosis Layer

To support bounded recovery guidance, the *Diagnosis Layer* converts $\mathcal{T}$ into structured evidence $\mathcal{E}$ and then into structured diagnosis $\mathcal{D}$ through failure localization, cross-signal fusion, and structured diagnosis generation, as shown in Figure 3.

3.3.1 Failure Localization. Given telemetry bundle $\mathcal{T}$, failure localization applies signal-specific detectors to identify failure-relevant steps, events, and execution windows as typed localized findings.

For metrics, *PROBE* uses robust window-based anomaly scoring: it computes robust z-scores with the median absolute deviation (MAD) [29], derives tail thresholds from empirical quantiles, and emits findings when metrics cross these thresholds. When no metric dominates, *PROBE* applies Isolation Forest [16] to window-level feature vectors and retains the highest-scoring window as aggregate metric evidence.

For logs and tool–environment state, *PROBE* localizes execution errors and repeated failures from tool returns, verification records, subprocess returns, guardrail signals, and error-bearing log segments. Related evidence is grouped by stable anchors such as tool name, argument fingerprint, error signature, return code, and nearby control context. When an auxiliary model is available, *PROBE* derives grounded semantic error signatures; otherwise, it retains raw signatures and infrastructure clues such as timeout, connection, or out-of-memory signals.

For traces and intent, *PROBE* localizes unstable intent transitions and run-level action patterns from the ordered execution history. It scores local transitions with a smoothed n-gram model and emits findings for transitions in the high-surprise tail. For longer-range patterns, it groups repeated failed actions using stable keys and summarizes the run timeline together with recent verification records, outcome evidence, and metric findings.

When external outcome evidence is available, evaluator verdicts and failing checks are converted into typed findings so that internal execution evidence can be compared with the task objective.

3.3.2 Cross-Signal Fusion. Cross-signal fusion aligns localized findings that refer to the same failure-relevant event, behavior pattern, or outcome condition. It converts localized findings into fused evidence records, where each record preserves a shared anchor, supporting signal sources, temporal scope, severity, and agreement or conflict across signals.

To construct fused evidence, *PROBE* first normalizes each localized finding into a standard evidence unit containing an anchor, source, time scope, severity, and evidence reference. It then groups compatible units by anchor, time, or semantic content and aggregates each group into a fused evidence record that preserves both support and disagreement. For example, repeated failures from logs, traces, and outcome evidence can strengthen the same unresolved-failure anchor, while an internal success claim contradicted by evaluator feedback is preserved as a conflict.

The resulting fused evidence records form structured evidence $\mathcal{E}$, preserving anchors, support, conflicts, and signal provenance for downstream structured diagnosis.

3.3.3 Structured Diagnosis. Given fused evidence $\mathcal{E}$, *PROBE* constructs a structured diagnosis through an anchor-first diagnosis protocol. Following decomposition-based prompting methods [35, 42, 46], the default implementation instantiates this protocol with an LLM-based diagnosis model prompted to emit a schema-constrained diagnosis object. When the model is unavailable, *PROBE* falls back to a deterministic summarizer built from localized findings. The

goal is not to generate a free-form explanation, but to produce a run-specific diagnosis that is grounded in evidence and structured enough to support bounded recovery guidance.

First, the protocol is anchor-first. Each diagnosis must start from a concrete, telemetry-supported failure anchor before making causal claims. The anchor identifies where the failure becomes diagnostically actionable, such as a failed check, contradicted outcome, repeated execution error, inconsistent workflow state, or supported artifact/log location. Unlike the reviewed failure-cause category, which describes the type of failure, the failure anchor grounds the diagnosis in the specific run.

Second, the protocol records guidance-relevant fields by distinguishing the *primary cause* from the *behavioral mistake*. The primary cause explains why the task failed, while the behavioral mistake explains what the agent did or failed to do after the failure anchor appeared. This distinction matters because the same failure anchor may require different recovery guidance, such as completing a missing action, avoiding repeated invalid tool use, delaying submission until verification succeeds, or revising an incorrect hypothesis. Evidence that contextualizes but does not directly explain the main failure is recorded as *contributing factors*.

Third, the protocol is schema-constrained and confidence-aware. The structured diagnosis is emitted as a fixed object containing the *primary cause*, *failure anchor*, *behavioral mistake*, *contributing factors*, *evidence summary*, and *confidence*, matching the schema shown in Figure 3. The confidence field is a bounded estimate of evidential support for the diagnosis object, not a calibrated probability of correctness [43, 44]. *PROBE* clips model-emitted confidence to $[0, 1]$, assigns conservative fallback confidence when the model is unavailable, limits contributing factors and evidence items, and requires failure-anchor fields to be supported by fused evidence. Confidence alone does not authorize recovery guidance; the *Guidance Gate* still applies grounding, actionability, and intervention-scope checks before the diagnosis can shape the subsequent attempt.

Together, these constraints make $\mathcal{D}$ an evidence-grounded diagnosis object that can be checked before being converted into bounded recovery guidance.

3.4 Guidance Gate

To turn structured diagnosis into guidance, the *Guidance Gate* acts as a deterministic validation and injection layer between $\mathcal{D}$ and the subsequent attempt. Given a structured diagnosis $\mathcal{D}$, the gate determines whether it can contribute *diagnosis-derived recovery guidance* and constructs bounded recovery guidance $\mathcal{G}$ only when the diagnosis is evidence-grounded, actionable, and within the agent's intervention scope.

Grounding Check. To prevent unsupported assumptions from being promoted into recovery instructions, the grounding check verifies whether the diagnosis is anchored in telemetry-supported evidence. A diagnosis is considered grounded only if its primary cause or failure anchor can be traced back to structured evidence, such as a failed evaluator check, repeated tool error, contradicted outcome, inconsistent workflow state, or concrete artifact/log location. If an artifact path, entity name, service name, location hint, or failure location is not supported by fused evidence, the gate does not use it as a guidance target. When the diagnosis lacks both a supported primary cause and a supported failure anchor, the diagnosis-derived guidance component is marked as non-injectable.

Actionability Filter. To ensure that grounded diagnosis can guide concrete agent behavior, the actionability filter checks whether it can be operationalized as bounded recovery guidance. Inspired by prior work on iterative feedback and self-refinement [22, 32], operationalizable guidance must specify four elements: a target, an operation, a verification signal, and a boundary condition. The target identifies what the agent should revisit; the operation specifies what it should do; the verification signal specifies what evidence should be checked afterward; and the boundary condition specifies when the agent should stop, avoid repetition, or return to evidence collection. The filter also checks whether the proposed intervention falls within the scope of agent-side recovery. Runtime and environment-related failures are not uniformly excluded, since some correspond to recoverable configuration, dependency, or validation issues. However, diagnoses dominated by platform outage, persistent connection failure, out-of-memory error, Docker failure, container crash, or other infrastructure-level conditions are not converted into direct corrective guidance. For such cases, the gate bounds the feedback around verification, evidence collection, or avoiding premature completion instead of issuing speculative corrective instructions.

Guidance Injection. After the *Grounding Check* and *Actionability Filter*, the gate constructs bounded recovery guidance $\mathcal{G}$ from the validated fields. $\mathcal{G}$ records whether diagnosis-derived guidance is injectable and, when injectable, specifies the target, operation, verification signal, and boundary condition for the subsequent attempt. The final *recovery hint block* is produced by a formatter rather than copied directly from $\mathcal{G}$. Injectable guidance is used as the main hint. Non-injectable guidance is not converted into direct corrective instructions; instead, the formatter may add conservative telemetry-supported hints, such as re-checking evidence, rerunning verification, or avoiding premature submission.

Overall, the *Guidance Gate* preserves the distinction between diagnosis and guidance by allowing only evidence-grounded, actionable, and bounded guidance to shape the subsequent attempt.

4 Experiments

Our evaluation asks whether failed-run telemetry can be converted into useful recovery feedback for software engineering agents. We study this question along three research questions:

- **RQ1: Recovery effectiveness.** Can *PROBE* improve diagnosis quality and recovery on initially unresolved runs compared with outcome-only feedback and observability-summary baselines?
- **RQ2: Actionable and gated recovery guidance.** Can *PROBE* transform failed-run evidence into actionable and bounded feedback for the subsequent attempt?
- **RQ3: Practical overhead.** How much telemetry does *PROBE* collect, and what token and latency overhead does it introduce when generating recovery feedback for the subsequent attempt?

4.1 Experimental Setup

To evaluate post-failure recovery across heterogeneous software engineering settings, we use three benchmarks with explicit task-level

Table 2: Recovery performance and diagnosis quality on initially unresolved cases. *Rate* is the fraction of *Initial Unres.* cases recovered after one repair attempt. *Top-1* is the fraction of cases whose diagnosed failure cause matches the independently reviewed failure cause.

		Outcome Only	AgentOps Summary		LangSmith Summary		*PROBE* Full Pipeline	
Setting	Initial Unres.	Rate	Rate	Top-1	Rate	Top-1	Rate	Top-1
SWE-bench	102	1.96%	12.74%	12.75%	14.71%	29.41%	**28.43%**	**83.33%**
EnterpriseOps-Gym	106	0.00%	5.66%	3.77%	2.83%	3.77%	**11.32%**	**57.55%**
AIOpsLab	49	8.16%	10.20%	42.86%	16.33%	**44.90%**	**30.61%**	**44.90%**
Total	257	2.33%	9.34%	14.79%	9.34%	21.79%	**21.79%**	**65.37%**

evaluators. Together, they cover repository-level repair, enterprise workflow execution, and AIOps service mitigation.

4.1.1 Benchmarks and Recovery Protocol. For repository-level software repair, we use SWE-bench [18] with mini-SWE-agent, where the agent must inspect a repository, edit source code, and produce a patch that satisfies the benchmark evaluator. For enterprise workflow recovery, we use EnterpriseOps-Gym [23] with its default task executor. For AIOps service mitigation, we use Microsoft AIOpsLab [9], an open-source framework for evaluating autonomous AIOps agents in interactive microservice cloud environments.

Our recovery protocol consists of one initial attempt followed by at most one repair attempt. Note that this one-repair-attempt design is an evaluation protocol, not a framework limitation. Each task is first executed without recovery feedback and evaluated by the native evaluator. Only tasks that remain unresolved after the initial attempt enter the repair phase. The repair attempt keeps the subject agent, task, tool environment, evaluator, and execution budget fixed, and differs only in the additional recovery information supplied to the agent, thereby isolating the marginal effect of recovery feedback from gains due to repeated retries.

Unless otherwise specified, the agents use Qwen3.5-Plus for task execution, while *PROBE* and summary-based baselines use Claude Sonnet 4.5 for diagnosis or feedback generation. Although absolute recovery rates may depend on diagnosis-model capability, using the same feedback model across *PROBE* and summary-based baselines allows us to compare the structured recovery pipeline against generic trace summarization while controlling for the feedback model.

4.1.2 Baselines. We compare *PROBE* with three feedback baselines under the same recovery protocol. All baselines use the same subject agent, task, evaluator, execution budget, and repair-attempt interface as *PROBE*. They differ only in the feedback provided before the repair attempt.

Outcome Only provides only the evaluator feedback from the failed initial attempt. This baseline tests whether the native failure signal, without execution telemetry or diagnosis, is sufficient to guide recovery. *AgentOps Summary* [12] and *LangSmith Summary* [19] serve as the observability summary baselines. For each baseline, we collect the execution trace exposed by the corresponding observability system, summarize it with Claude Sonnet 4.5, and combine the summary with evaluator feedback. These baselines test whether generic summaries of observability traces are sufficient for recovery support.

PROBE Full Pipeline is the full-method condition, where the formatted recovery hint block is generated from the telemetry report, structured diagnosis, and *Guidance Gate*.

4.1.3 Evaluation Metrics. In Table 2, *Initial Unres.* denotes the number of tasks that remain unresolved after the initial attempt, and *Rate* denotes the fraction of those tasks resolved after the repair attempt. We also report *Top-1*, where a case is counted as correct when the main failure cause reflected in the method's structured diagnosis matches the independently reviewed failure cause. Top-1 evaluates diagnosis quality rather than end-to-end task success.

To quantify runtime overhead, we use three metrics. *Telemetry volume* is measured by the number of recorded spans and tool calls. *Prompt overhead* is measured by the number of additional prompt tokens introduced by the recovery information supplied to the subsequent attempt. *Feedback-generation latency* is measured from the end of the failed initial attempt to the completion of telemetry report construction, structured diagnosis, *Guidance Gate* processing, and recovery-hint formatting.

4.2 RQ1: End-to-End Recovery Effectiveness

Table 2 reports diagnosis quality and recovery performance under the same one-repair-attempt protocol. Across 257 initially unresolved tasks, *PROBE* achieves 65.37% Top-1 diagnosis accuracy and recovers 56 cases, corresponding to a 21.79% recovery rate. Compared with the strongest non-*PROBE* baselines, *PROBE* improves Top-1 diagnosis accuracy by 43.58 percentage points and recovery rate by 12.45 percentage points. The larger improvement in diagnosis accuracy than in recovery rate suggests a diagnosis–recovery gap: accurate failure-cause identification is necessary but not sufficient for successful recovery under fixed environment and budget constraints.

The advantage of *PROBE* is consistent across settings. On SWE-bench, *PROBE* achieves 83.33% Top-1 accuracy and 28.43% recovery, exceeding the best non-*PROBE* baseline by 53.92 and 13.72 percentage points, respectively. On EnterpriseOps-Gym, *PROBE* reaches 57.55% Top-1 accuracy and 11.32% recovery, improving over the strongest non-*PROBE* baseline by 53.78 and 5.66 percentage points. On AIOpsLab, *PROBE* matches the strongest non-*PROBE* baseline on Top-1 accuracy at 44.90%, but achieves a higher recovery rate of 30.61% versus 16.33%.

Outcome Only recovers only 6 of the 257 initially unresolved tasks, showing that final evaluator feedback alone is a weak basis for recovery. The observability-summary baselines recover 24

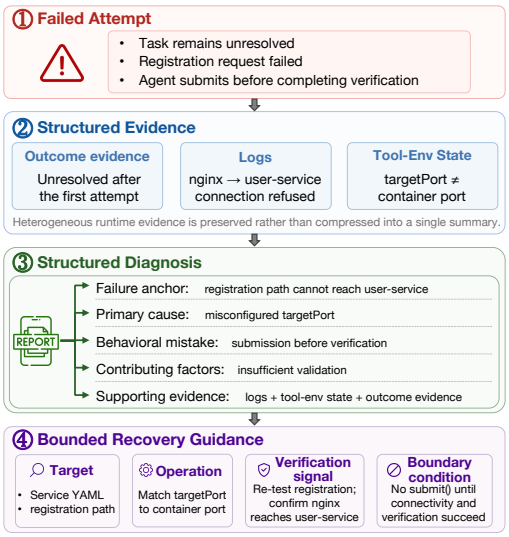

Figure 4: AIOpsLab case illustrating how *PROBE* preserves failed-run evidence, derives a structured diagnosis, and produces bounded recovery guidance for the subsequent attempt.

tasks overall, but their Top-1 accuracy remains much lower than *PROBE*, at 14.79% and 21.79%. This suggests that generic trace summaries provide useful runtime context, but often fail to preserve the signal-specific anchors needed to identify the most recovery-relevant failure cause and produce bounded recovery guidance. The AIOpsLab result further clarifies the distinction between diagnosis and recovery. Although *PROBE* and *LangSmith Summary* achieve the same Top-1 score, *PROBE* recovers substantially more cases. In interactive service environments, recovery requires operationally specific guidance, such as which configuration to inspect, which check to rerun, which verification signal to wait for, and which premature action to avoid.

Overall, RQ1 indicates that *PROBE* improves both diagnostic quality and downstream recovery rate, while also revealing a gap between diagnosis and recovery.

4.3 RQ2: Actionable and Gated Recovery Guidance

To examine the diagnosis–recovery gap, RQ2 studies whether *PROBE* can convert failed-run evidence into executable and bounded recovery guidance through the *Guidance Gate*. We first present a representative case study, and then analyze category-wise diagnosis alignment and recovery outcomes.

4.3.1 Case Study: From Diagnosis to Bounded Guidance. Figure 4 shows a representative AIOpsLab case. The initial attempt remains unresolved because the registration path cannot reach `user-service`, and the agent submits before completing verification. *PROBE* preserves three recovery-critical evidence sources: outcome evidence showing that the task remains unresolved, logs showing an `nginx` connection failure to `user-service`, and tool–environment state showing a mismatch between the Service `targetPort` and the actual container port.

The *Diagnosis Layer* converts this evidence into a structured diagnosis: the failure anchor is that the registration path cannot reach `user-service`, the primary cause is a misconfigured `targetPort`, and the behavioral mistake is premature submission before verification. This diagnosis explains the failed run, but it does not by itself specify how the subsequent attempt should proceed. The *Guidance Gate* turns the diagnosis into bounded recovery guidance by applying the Grounding Check and Actionability Filter, including intervention-scope validation. In this case, the guidance can be expressed with four actionability fields:

- *Target:* the Service YAML and the registration path involving `user-service`.
- *Operation:* compare `targetPort` with the actual container port and correct the mismatch if present.
- *Verification signal:* re-test the registration request and confirm that `nginx` can reach `user-service`.
- *Boundary condition:* do not call `submit()` until connectivity and registration verification both succeed.

This case illustrates the value of the *Guidance Gate*: diagnosis explains the failure, whereas bounded guidance turns the diagnosis into an executable instruction for the subsequent attempt.

Table 3: Category-wise diagnosis alignment and recovery under the *PROBE* full pipeline. *Aligned* is the fraction of diagnoses matching the independently reviewed failure cause; *Recovered* is the fraction of all initially unresolved cases in the category that are solved in the subsequent attempt.

Reviewed failure cause	Cases	Aligned	Recovered
Insufficient validation	83	81 (97.59%)	21 (25.30%)
Tool/subprocess failure handling	47	20 (42.55%)	3 (6.38%)
State/workflow error	42	35 (83.33%)	6 (14.29%)
Patch/submission workflow	35	12 (34.29%)	15 (42.86%)
Retry/no-progress loop	26	6 (23.08%)	5 (19.23%)
Runtime/environment handling	24	14 (58.33%)	6 (25.00%)
Total	257	168 (65.37%)	56 (21.79%)

4.3.2 Category-wise Results. Table 3 reports diagnosis alignment and recovery by reviewed failure category under the *PROBE* full pipeline. Overall, *PROBE* aligns with the independently reviewed failure cause in 168 of 257 cases, but only 56 cases are recovered, confirming that correct diagnosis is important but not sufficient for executable recovery.

The category-level results further clarify this diagnosis–recovery gap. *Insufficient validation* and *state/workflow error* show strong diagnosis alignment, reaching 97.59% and 83.33%, respectively, because they often expose explicit anchors such as failing checks, premature submission, missing state transitions, or incorrect entity linkage. However, *state/workflow error* reaches only 14.29% recovery, suggesting that recognizing an incorrect workflow state is easier than repairing it under the same execution budget. Recovery is highest for *patch/submission workflow*, suggesting that localized submission-stage corrections are often executable once surfaced in the feedback. By contrast, *tool/subprocess failure handling* has the

lowest recovery rate at 6.38%. This does not mean such failures are hard to observe; rather, their evidence can be explicit, but successful recovery often requires strategy adaptation beyond a localized correction. For *runtime/environment handling*, some cases involve recoverable configuration or dependency issues, while others are dominated by runtime conditions outside the agent's direct control. Accordingly, the *Guidance Gate* bounds the feedback around verifiable agent-side actions when possible, and around verification, evidence collection, or avoiding premature completion when direct repair is not supported.

In summary, RQ2 shows that *PROBE* goes beyond generic summarization by converting failed-run evidence into structured diagnosis and evidence-grounded recovery feedback, while the remaining diagnosis–recovery gap reflects that successful repair still depends on whether the required recovery path is executable by the agent and environment.

4.4 RQ3: Practical Overhead

To evaluate the practical overhead of *PROBE*, we measure telemetry volume, prompt overhead, and feedback-generation latency. Under *PROBE* instrumentation, the median failed run contains 578 spans and 164 tool calls on SWE-bench, 46 spans and 17 tool calls on EnterpriseOps-Gym, and 135 spans and 64 tool calls on AIOpsLab. These differences reflect task structure: repository-level repair requires repeated file inspection, editing, and test execution, whereas enterprise workflow and service-operation tasks typically involve shorter environment-interaction sequences.

The final recovery hint block remains compact across all three settings. On average, *PROBE* adds 1.07K prompt tokens on SWE-bench, 0.45K on EnterpriseOps-Gym, and 0.39K on AIOpsLab. These values remain within the same order of magnitude as the observability-summary baselines, indicating that *PROBE* does not require substantially larger feedback prompts. This result is consistent with the design of *PROBE*: the telemetry report preserves rich failed-run evidence, while structured diagnosis, *Guidance Gate* processing, and hint formatting select only recovery-relevant content for the subsequent attempt.

Across the 257 analyzed cases, the median feedback-generation latency is 210.43 seconds, with setting-level medians of 190.45 seconds on SWE-bench, 300.68 seconds on EnterpriseOps-Gym, and 130.50 seconds on AIOpsLab. This latency is incurred once after the failed initial attempt and before the repair attempt starts. It covers telemetry report construction, structured diagnosis, *Guidance Gate* processing, and recovery-hint formatting, and does not modify the subject agent's execution budget.

Taken together, these results indicate that *PROBE* scales as a side-channel recovery layer, with moderate telemetry volume, compact prompt overhead, and bounded feedback-generation latency.

5 Deployment-Oriented Prototype for IcM Service Diagnosis

In Microsoft Incident Management (IcM), service incidents are investigated by diagnosis agents that retrieve heterogeneous operational evidence, including monitor alerts, service metadata, dependency graphs, and historical incident records, and then propose mitigation actions through a structured workflow [1, 10, 13, 17, 31, 45]. When such a diagnosis-agent execution fails to resolve an incident, on-call engineers often inspect traces, review retrieved evidence, adjust prompts, and rerun the agent manually. This process is time-consuming and does not systematically reuse the runtime evidence already produced by the failed attempt.

To validate *PROBE*'s industrial integration boundary, we build a deployment-oriented proof-of-concept integration. It defines the input–output contract and examines whether *PROBE* can be attached to an existing diagnosis-agent workflow without changing the agent policy, toolset, evaluator, prompt budget, or execution logic. In this section, we focus on the engineering interface, integration boundary, and lessons for industrial adoption.

5.1 Prototype Scope and Input–Output Contract

The prototype targets the diagnosis-agent execution loop rather than the full incident-management lifecycle. It does not replace incident triage, mitigation execution, escalation, or human approval. Instead, *PROBE* observes a diagnosis-agent run as a side channel and produces recovery artifacts when the run fails or remains unresolved.

Table 4 summarizes the input–output contract of the prototype. The key design point is non-intrusive integration: the diagnosis agent keeps the same model, tools, evaluator, prompt budget, and execution logic, while *PROBE* records boundary events and produces artifacts after the run.

The four interfaces capture common execution boundaries in IcM diagnosis-agent workflows: run initialization, boundary-event recording, report finalization, and handoff. More generally, they apply to tool-using or task-evaluable agents whose executions expose comparable boundary events, such as agent calls, tool interactions, validation results, and final outcomes.

5.2 Integration Design

To keep the integration lightweight, the prototype exposes a small set of interface-level adapters rather than requiring framework-internal hooks. A service team only needs to wrap the existing diagnosis-agent loop with a telemetry session and register boundary callbacks for model calls, tool calls, validation results, and final outcomes. In practice, this requires only run-level session creation, boundary-event recording, and report finalization, rather than changes to the agent's prompt, tool schema, or orchestration logic.

The integration follows a fail-open design. If *PROBE* is unavailable, the diagnosis agent continues with its original workflow. If the execution raises an exception, the telemetry session is finalized in the cleanup path when possible, so the failed run can still produce an execution record. This property is important for IcM-style workflows, where recovery support must not introduce a new operational failure mode.

The prototype supports two handoff modes. In a human-facing workflow, the telemetry report is attached to the incident record as a diagnosis artifact, helping on-call engineers review the evidence, failure anchor, and candidate verification or corrective action. In an iterative-agent workflow, the formatted recovery hint block is supplied to the subsequent attempt as compact feedback. Both modes consume the same report artifact.

Table 4: Input–output contract and prototype interfaces for integrating *PROBE* into an IcM diagnosis-agent workflow.

Stage	Prototype interface	Input to *PROBE*	Output from *PROBE*	Agent impact
Run start	Session initialization	Incident context, task, tools, telemetry links, model ID	Run metadata and tool snapshot	None
During run	Boundary-event recording	Model/tool calls, retrieved evidence, claims, checks, exceptions	Telemetry spans, correlation IDs, evidence refs., outcome observations	Side-channel only
Run end	Session finalization and report generation	Final status, evaluator outcome, run metadata	Telemetry report with evidence, diagnosis, and Guidance Gate output	None
Handoff	Report export and hint formatting	Telemetry report and incident context	Incident artifact or recovery hint block	No policy/tool/evaluator/budget change

The Guidance Gate is deterministic by design. To make the guidance decision transparent, reproducible, and auditable, it applies the *Grounding Check* and *Actionability Filter*, including intervention-scope validation, before allowing structured diagnosis to contribute diagnosis-derived recovery guidance.

In our controlled evaluation, the median feedback-generation latency is 210.43 seconds across 257 failed runs. For human-facing workflows, the report can be generated off the critical path; for automated reruns, this latency is treated as feedback-generation time before the subsequent attempt, not as part of the agent's execution budget. This separation supports incremental industrial adoption by allowing teams to use *PROBE* first as an auditable diagnosis artifact and later as automated recovery feedback.

5.3 Lessons Learned

Lesson 1: Recovery support needs boundary evidence, not only final outcomes. The prototype showed that useful recovery artifacts require events from multiple boundaries: model calls, tool calls, tool returns, validation attempts, exceptions, and evaluator outcomes. Final success or failure labels are insufficient because they do not explain where the agent lost progress or which action should be revisited.

Lesson 2: Recovery artifacts should serve both human review and agent retry. The same telemetry report should serve two consumers. For on-call engineers, it should be readable as an incident artifact that identifies the failure anchor, primary cause, and supporting evidence. For iterative agents, it should be formatted into a compact recovery hint block with verifiable actions and stopping conditions.

Lesson 3: Adoption depends on preserving the existing execution boundary. The prototype suggests that recovery support should be integrated at workflow boundaries rather than inside the agent's trusted execution path. Accordingly, *PROBE* should remain a wrapper or boundary observer rather than an inline dependency. This preserves the agent's policy, tools, evaluator, prompt budget, and execution logic in IcM-style workflows, while avoiding a new blocking component in the incident-diagnosis path.

Lesson 4: Live deployment should evaluate operational usefulness, not only diagnosis accuracy. The proof-of-concept validates the integration boundary and artifact flow, but not production impact on live incidents. A full IcM deployment should measure whether the report reduces engineer inspection effort, whether generated guidance is accepted by on-call engineers, and whether the side-channel integration remains robust under production load. These metrics would assess the operational usefulness of the produced artifacts, beyond validating artifact generation and handoff.

6 Discussion and Limitations

Our evaluation has two main limitations. First, the one-repair-attempt protocol measures single-step recoverability rather than the practical ceiling of iterative recovery. Multi-round protocols would clarify whether *PROBE* continues to provide marginal gains across debugging loops or whether recovery saturates after the first guided attempt. Second, the dependence of recovery on diagnosis-model capability remains to be quantified. Same-model and model-swap ablations, such as using the task-execution model for diagnosis generation, would further separate the contribution of the structured recovery pipeline from that of the diagnosis model. Future evaluations will incorporate these extensions to better characterize the robustness and generality of *PROBE* under iterative recovery settings and across diagnosis models with different capabilities.

7 Conclusion

We presented *PROBE*, a failure-anchored framework that converts heterogeneous runtime telemetry into structured diagnosis and bounded recovery guidance for software engineering agents through three coupled stages: a Telemetry Layer, a Diagnosis Layer, and a Guidance Gate. Evaluation on 257 initially unresolved cases across three settings shows 65.37% Top-1 diagnosis accuracy and a 21.79% recovery rate, exceeding the strongest non-*PROBE* baseline by 43.58 and 12.45 percentage points, respectively. The results confirm that accurate diagnosis is necessary but insufficient without grounded, bounded, and executable guidance. A lightweight prototype integration for Microsoft IcM service diagnosis further demonstrates that *PROBE* can be attached to existing diagnosis-agent workflows without modifying the agent policy, toolset, or execution budget, and that the engineering principles behind its design, including fail-open instrumentation, deterministic guidance gating, and boundary-level adaptation, are applicable to broader industrial software engineering agent workflows.

8 Data Availability

We have made our source code and datasets publicly available through an anonymous repository at https://anonymous.4open.science/r/6114a1c7/README.md.

References

[1] Toufique Ahmed, Supriyo Ghosh, Chetan Bansal, Thomas Zimmermann, Xuchao Zhang, and Saravan Rajmohan. 2023. Recommending Root-Cause and Mitigation Steps for Cloud Incidents using Large Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1737–1749. doi:10.1109/ICSE48619.2023.00149

[2] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C., Arun Iyer, Suresh Parthasarathy, Sriram K. Rajamani, Balasubramanyan Ashok, and Shashank Shet. 2024. CodePlan: Repository-Level Coding using LLMs and Planning. *Proc. ACM Softw. Eng.* 1, FSE (2024), 675–698. doi:10.1145/3643757

[3] Shraddha Barke, Arnav Goyal, Alind Khare, Avaljot Singh, Suman Nath, and Chetan Bansal. 2026. AgentRx: Diagnosing AI Agent Failures from Execution Trajectories. arXiv:2602.02475 [cs.AI] https://arxiv.org/abs/2602.02475

[4] BerriAI. 2024. LiteLLM. https://docs.litellm.ai/. Unified interface for calling multiple large language model providers. Accessed: 2026-04-09.

[5] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. arXiv:2403.17134 [cs.SE] https://arxiv.org/abs/2403.17134

[6] Islem Bouzenia and Michael Pradel. 2025. Understanding Software Engineering Agents: A Study of Thought-Action-Result Trajectories. In *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, Republic of, November 16-20, 2025*. IEEE, 2846–2857. doi:10.1109/ASE63991.2025.00234

[7] Harrison Chase. 2022. LangChain. https://github.com/langchain-ai/langchain. Open-source framework for building LLM applications and agents. Accessed: 2026-04-09.

[8] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug. arXiv:2304.05128 [cs.CL] https://arxiv.org/abs/2304.05128

[9] Yinfang Chen, Manish Shetty, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Jonathan Mace, Chetan Bansal, Rujia Wang, and Saravan Rajmohan. 2025. AIOpsLab: A Holistic Framework to Evaluate AI Agents for Enabling Autonomous Clouds. In *Proceedings of the Eighth Conference on Machine Learning and Systems, MLSys 2025, Santa Clara, CA, USA, May 12-15, 2025*, Matei Zaharia, Gauri Joshi, and Yingyan (Celine) Lin (Eds.). OpenReview.net/mlsys.org. https://openreview.net/forum?id=3EXBLwGxtq

[10] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, Jun Zeng, Supriyo Ghosh, Xuchao Zhang, Chaoyun Zhang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Tianyin Xu. 2024. Automatic Root Cause Analysis via Large Language Models for Cloud Incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, 674–688. doi:10.1145/3627703.3629553

[11] Zhi Chen, Wei Ma, and Lingxiao Jiang. 2026. Beyond Final Code: A Process-Oriented Error Analysis of Software Development Agents in Real-World GitHub Scenarios. arXiv:2503.12374 [cs.SE] https://arxiv.org/abs/2503.12374

[12] Liming Dong, Qinghua Lu, and Liming Zhu. 2024. AgentOps: Enabling Observability of LLM Agents. arXiv:2411.05285 [cs.AI] https://arxiv.org/abs/2411.05285

[13] Drishti Goel, Fiza Husain, Aditya Singh, Supriyo Ghosh, Anjaly Parayil, Chetan Bansal, Xuchao Zhang, and Saravan Rajmohan. 2024. X-Lifecycle Learning for Cloud Incident Management using LLMs. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, Marcelo d'Amorim (Ed.). ACM, 417–428. doi:10.1145/3663529.3663861

[14] Antonio Gulli, Lavi Nigam, Julia Wiesinger, Vladimir Vuskovic, Irina Sigler, Ivan Nardini, Nicolas Stroppa, Sokratis Kartakis, Narek Saribekyan, and Alan Bount. 2025. *Agents Companion*. Technical Report. Google. https://www.kaggle.com/whitepaper-agent-companion Google Whitepaper. Available at: https://cdn.jsdelivr.net/gh/abncharts/abncharts.public.1/abnasia.org/1743681605381_www.abnasia.org.pdf.

[15] Zikang Guo, Benfeng Xu, Chiwei Zhu, Wentao Hong, Xiaorui Wang, and Zhendong Mao. 2026. MCP-AgentBench: Evaluating Real-World Language Agent Performance with MCP-Mediated Tools. In *Fortieth AAAI Conference on Artificial Intelligence, Thirty-Eighth Conference on Innovative Applications of Artificial Intelligence, Sixteenth Symposium on Educational Advances in Artificial Intelligence, AAAI 2026, Singapore, January 20-27, 2026*, Sven Koenig, Chad Jenkins, and Matthew E. Taylor (Eds.). AAAI Press, 30888–30896. doi:10.1609/AAAI.V40I37.40347

[16] Songqiao Han, Xiyang Hu, Hailiang Huang, Mingqi Jiang, and Yue Zhao. 2022. ADBench: Anomaly Detection Benchmark. arXiv:2206.09426 [cs.LG] https://arxiv.org/abs/2206.09426

[17] Yuxuan Jiang, Chaoyun Zhang, Shilin He, Zhihao Yang, Minghua Ma, Si Qin, Yu Kang, Yingnong Dang, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. 2024. Xpert: Empowering Incident Management with Query Recommendations via Large Language Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 92:1–92:13. doi:10.1145/3597503.3639081

[18] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66

[19] LangChain. 2026. LangSmith: A Platform for Observability and Evaluation of LLM-based Systems. https://docs.langchain.com/langsmith/observability. Accessed: 2026-03.

[20] Han Li, Yifan Yao, Letian Zhu, Rili Feng, Hongyi Ye, Jiaming Wang, Yancheng He, Pengyu Zou, Lehan Zhang, Xinping Lei, Haoyang Huang, Ken Deng, Ming Sun, Zhaoxiang Zhang, He Ye, and Jiaheng Liu. 2026. CodeTracer: Towards Traceable Agent States. arXiv:2604.11641 [cs.SE] https://arxiv.org/abs/2604.11641

[21] Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. API-Bank: A Comprehensive Benchmark for Tool-Augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, 3102–3116. doi:10.18653/V1/2023.EMNLP-MAIN.187

[22] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. arXiv:2303.17651 [cs.CL] https://arxiv.org/abs/2303.17651

[23] Shiva Krishna Reddy Malay, Shravan Nayak, Jishnu Sethumadhavan Nair, Sagar Davasam, Aman Tiwari, Sathwik Tejaswi Madhusudhan, Sridhar Krishna Nemala, Srinivas Sunkara, and Sai Rajeswar. 2026. EnterpriseOps-Gym: Environments and Evaluations for Stateful Agentic Planning and Tool Use in Enterprise Settings. arXiv:2603.13594 [cs.AI] https://arxiv.org/abs/2603.13594

[24] Model Context Protocol Contributors. 2025. Model Context Protocol Specification. https://modelcontextprotocol.io/specification/2025-06-18. Accessed: 2026-04-09.

[25] Rahul Nanda, Chandra Maddila, Smriti Jha, Euna Mehnaz Khan, Matteo Paltenghi, and Satish Chandra. 2026. Wink: Recovering from Misbehaviors in Coding Agents. arXiv:2602.17037 [cs.SE] https://arxiv.org/abs/2602.17037

[26] Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2024. Is Self-Repair a Silver Bullet for Code Generation? arXiv:2306.09896 [cs.CL] https://arxiv.org/abs/2306.09896

[27] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2024. Gorilla: Large Language Model Connected with Massive APIs. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/e4c61f578ff07830f5c37378dd3ecb0d-Abstract-Conference.html

[28] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=dHng2O0Jjr

[29] Lukas Ruff, Jacob R. Kauffmann, Robert A. Vandermeulen, Gregoire Montavon, Wojciech Samek, Marius Kloft, Thomas G. Dietterich, and Klaus-Robert Muller. 2021. A Unifying Review of Deep and Shallow Anomaly Detection. *Proc. IEEE* 109, 5 (May 2021), 756–795. doi:10.1109/jproc.2021.3052449

[30] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html

[31] Manish Shetty, Chetan Bansal, Sumit Kumar, Nikitha Rao, Nachiappan Nagappan, and Thomas Zimmermann. 2021. Neural Knowledge Extraction From Cloud Service Incidents. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 218–227. doi:10.1109/ICSE-SEIP52600.2021.00031

[32] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. doi:10.48550/ARXIV.2303.11366

[33] Gou Tan, Zilong He, Min Li, Haiyu Huang, Yilun Wang, Pengfei Chen, Giuliano Casale, and Chuanfu Zhang. 2026. LLMRCA: Multilevel Root Cause Analysis for LLM Applications Using Multimodal Observability Data. *ACM Trans. Softw. Eng. Methodol.* (April 2026). doi:10.1145/3806200 Just Accepted.

[34] Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. 2024. AppWorld: A Controllable World of Apps and People for Benchmarking Interactive Coding Agents. arXiv:2407.18901 [cs.SE] https://arxiv.org/abs/2407.

18901

[35] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[36] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation Framework. *CoRR* abs/2308.08155 (2023). arXiv:2308.08155 doi:10.48550/ARXIV.2308.08155

[37] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying LLM-Based Software Engineering Agents. *Proc. ACM Softw. Eng.* 2, FSE (2025), 801–824. doi:10.1145/3715754

[38] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)*. ACM, 819–831. doi:10.1145/3650212.3680323

[39] Tianbao Xie, Fan Zhou, Zhoujun Cheng, Peng Shi, Luoxuan Weng, Yitao Liu, Toh Jing Hua, Junning Zhao, Qian Liu, Che Liu, Leo Z. Liu, Yiheng Xu, Hongjin Su, Dongchan Shin, Caiming Xiong, and Tao Yu. 2023. OpenAgents: An Open Platform for Language Agents in the Wild. *CoRR* abs/2310.10634 (2023). arXiv:2310.10634 doi:10.48550/ARXIV.2310.10634

[40] Zhe Xie, Shenglin Zhang, Yitong Geng, Yao Zhang, Minghua Ma, Xiaohui Nie, Zhenhe Yao, Longlong Xu, Yongqian Sun, Wentao Li, and Dan Pei. 2024. Microservice Root Cause Analysis With Limited Observability Through Intervention Recognition in the Latent Space. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2024, Barcelona, Spain, August 25-29, 2024*, Ricardo Baeza-Yates and Francesco Bonchi (Eds.). ACM, 6049–6060. doi:10.1145/3637528.3671530

[41] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. arXiv:2405.15793 [cs.SE] https://arxiv.org/abs/2405.15793

[42] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=WE_vluYUL-X

[43] Dylan Zhang, Xuchao Zhang, Chetan Bansal, Pedro Henrique B. Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. 2023. PACE-LM: Prompting and Augmentation for Calibrated Confidence Estimation with GPT-4 in Cloud Incident Root Cause Analysis. *CoRR* abs/2309.05833 (2023). arXiv:2309.05833 doi:10.48550/ARXIV.2309.05833

[44] Dylan Zhang, Xuchao Zhang, Chetan Bansal, Pedro Henrique B. Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. 2024. LM-PACE: Confidence Estimation by Large Language Models for Effective Root Causing of Cloud Incidents. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, Marcelo d'Amorim (Ed.). ACM, 388–398. doi:10.1145/3663529.3663858

[45] Xuchao Zhang, Supriyo Ghosh, Chetan Bansal, Rujia Wang, Minghua Ma, Yu Kang, and Saravan Rajmohan. 2024. Automated Root Causing of Cloud Incidents using In-Context Learning with GPT-4. arXiv:2401.13810 [cs.CL] https://arxiv.org/abs/2401.13810

[46] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V. Le, and Ed H. Chi. 2023. Least-to-Most Prompting Enables Complex Reasoning in Large Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=WZH7099tgfM

[47] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. WebArena: A Realistic Web Environment for Building Autonomous Agents. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=oKn9c6ytLx