PRAXIS: Integrating Program Analysis with Observability for Root-Cause Analysis

Shengkun Cui†, Rahul Krishna‡, Saurabh Jha‡, Ravishankar K. Iyer†
†University of Illinois Urbana-Champaign, Urbana, IL 61801, USA ‡IBM Research, Yorktown Heights, NY 10598, USA
†{scui8, rkiyer}@illinois.edu ‡{rkrsn, saurabh.jha}@ibm.com

***Abstract*—Unresolved production cloud incidents cost an average of over $2M per hour. This paper introduces PRAXIS, an orchestrator that manages and deploys an agentic workflow for diagnosing code- and configuration-caused cloud incidents. PRAXIS employs an LLM-driven structured traversal over two types of graph: (1) a service dependency graph (SDG) that captures microservice-level dependencies; and (2) a hammock-block program dependence graph (PDG) that captures code-level dependencies for each microservice. Compared to state-of-the-art ReAct baselines, PRAXIS improves RCA accuracy by up to $6.3\times$ while reducing token consumption by $5.3\times$. PRAXIS is demonstrated on a set of 30 comprehensive real-world incidents that is being compiled into an RCA benchmark.**

***Index Terms*—Cloud computing, agents, agentic approach, incident diagnosis, root-cause analysis, program analysis, reliability**

I. INTRODUCTION

Production cloud incidents cost an average of over $2M per hour [1], [2]. While operational mitigation actions such as reboot, restart, scale-out, and rollback can resolve many such incidents, approximately 24% are not resolved by operational mitigation [3]. In those cases, engineers supported by tools must perform *root-cause analysis* (RCA), a diagnostic process that identifies the origin (i.e., root causes) of a failure. When RCA is ineffective, outages can be prolonged and highly disruptive [4]–[7].

This paper introduces **PRAXIS**, an orchestrator that manages and deploys an agentic workflow for diagnosing code- and configuration-caused cloud incidents. PRAXIS employs an LLM-driven structured traversal over two types of graph: (1) a service dependency graph (SDG) that captures microservice-level dependencies; and (2) a hammock- block [8], [9] program dependence graph (PDG) that captures code-level dependencies for each microservice. Together, these graphs efficiently encode microservice- and code-level dependencies. By explicitly and jointly imposing both microservices and code dependency structures in the agentic RCA workflow, PRAXIS extends the diagnosis beyond observability symptoms, making it capable of identifying and explaining root causes that lie in code or configurations. We demonstrate the breadth of PRAXIS using 30 real-world examples, integrated into a comprehensive benchmark that is open-sourced.

Contribution. Our main contributions are as follows:
1) **PRAXIS**, an agentic approach for cloud incident RCA with structured, LLM-driven graph reasoning and traversal that utilizes microservices and program dependency graphs.
2) An application of the hammock block program dependence graph for agentic RCA, leveraging the hammock block's hierarchical nesting structure to analyze microservice code at multi-granular levels seamlessly.
3) Code-Cloud-RCA Benchmark, integrated into ITBench [10], consisting of 30 different scenarios, each incorporating a unique software, configuration, deployment, or resource-related fault observed in the real world [3]–[5], [11]–[14] and injected into a live Kubernetes cloud environment.
4) A demonstration of PRAXIS's agentic capabilities to perform cross SDG-PDG traversal to diagnose a challenging incident.

Results. We evaluated PRAXIS's RCA effectiveness in terms of root cause accuracy and token consumptions across those 30 scenarios. PRAXIS achieved a root-cause reasoning accuracy of 61.5% and a root-cause identification accuracy of 73.9%, an increase of $6.3\times$ and $3.4\times$, respectively, over state-of-the-art ReAct baselines [15], [16]. In addition, PRAXIS reduced token consumption by $5.3\times$, from 884.9k tokens to 166.5k tokens per successful diagnosis, compared to the same baselines.

An important reason for PRAXIS's effectiveness gain is its ability to reduce the context space and focus microservice- and code-level analysis on the regions most implicated by an incident, guided by the underlying dependency structure. Prior work [15]–[20] does not impose such a structure and relies entirely on the LLM's built-in autoregressive analysis over unstructured, plain-text prompts, which is shown to be less effective. PRAXIS instead explicitly constrains the LLM's reasoning along graph-defined dependencies via LLM-driven graph traversal.

II. TERMINOLOGY

Root cause analysis (RCA). Following the understanding espoused in [21], RCA is the diagnostic process that (1) identifies the microservice responsible for the incident and (2) the statement(s), function(s), or configuration(s) identifying the origin of the incident, and (3) provides concise reasoning about its propagation and manifestation to facilitate remediation.

Service dependency graph (SDG). The service dependency graph of a cloud application is a directed graph with nodes representing individual microservices and directed edges showing the dependency between them. In any given interval, the graph represents the current dynamics among the microservices.

PRAXIS implementation and benchmark scenarios used in this paper are publicly available at https://doi.org/10.5281/zenodo.19163486.

Program dependence graph (PDG). A directed graph that captures microservice program structure [22]; the nodes are hammock blocks [8], [9] and the edges are data, control, and call dependencies.

Hammock block. A hammock block restructures unstructured code into a structured block corresponding to a single-entry, single-exit (SESE) region in code [8], [9].

Observability. In microservices, observability is the ability to infer the distributed system's internal state from its generated data (e.g., logs, metrics, events, and distributed traces), enabling real-time insight through analysis.

Code context. Code context refers to code-related information.

III. Approach Overview

This section introduces PRAXIS, an orchestrator that manages and deploys an agentic workflow for cloud incident RCA. At the core of this approach is an LLM-driven structure traversal over two types of graph: (1) an SDG that captures microservice-level dependencies (providing a high-level localization of faulty microservices); and (2) a hammock-block PDG that captures code-level dependencies for each microservice (allowing for fine-grained RCA decisions that identify offending code paths and configurations responsible for the incident). PRAXIS's hammock-block PDG provides an efficient structural representation of a microservice program, enabling seamless analysis of microservice code at different granularities. By integrating tools, observability data, and dependency graphs, PRAXIS delivers demonstratively accurate RCA for live cloud incidents, demonstrated on microservice-based Kubernetes applications. In summary, the core functionalities of PRAXIS's workflow include the following.

Data gathering. Like prior agentic approaches [15]–[20], [23], PRAXIS uses cloud monitoring tools [24]–[26] to actively collect a set of automated and user-specified alerts referred to as *golden-signal alerts* (e.g., error-rates or latencies exceeding user-defined thresholds) and observability data consisting of cloud application and microservice logs, distributed traces of microservice calls, Kubernetes events, and application metrics. PRAXIS also actively retrieves microservice topology, as in [27], [28], using a cloud topology monitoring platform [29], which is used to build the SDG. In addition, PRAXIS uses microservice codebases for constructing PDG for microservices.

SDG and PDG construction. As the key differentiator from prior work [16], [20], PRAXIS's agentic workflow utilizes two modalities of graphs. The first is a dynamically evolving service dependency graph [27], [28] for the cloud application, with currently deployed microservices as nodes and inter-microservices dependencies as edges, generated via a real-time topology monitor [29]. The second is a hammock-granularity PDG for each microservice node in the SDG, with hammock blocks as nodes and control, data, and function-call dependencies as edges. This PDG is generated by static program analysis tools [30], [31] and represents the software dependency of a microservice application relevant to root-cause analysis.

Together, the SDG and PDG capture relevant microservice-level and code-level dependencies for RCA. These graphs serve as bases for RCA via LLM-driven graph traversal, first at the microservice level and subsequently at the code level, to identify responsible microservices, faulty code sites, and/or configurations that might explain the observed erroneous signals (such as active alerts, error logs, traces, metrics, and events), providing in-depth RCA down to the code level.

Structure-aware agentic RCA. After being triggered by active golden-signal alert(s), PRAXIS presents the relevant data (active alert(s) and error traces) to an LLM to identify and select microservices correlated with error signals in those data. The selected microservices become initial candidates for agentic focus. Subsequently, the LLM investigates each microservice by performing structure-aware reasoning through traversal of the corresponding hammock blocks. PRAXIS uniquely constructs the PDG using hammock blocks at different granularities spanning module, class, function, and statement levels, allowing the LLM to seamlessly move between levels as required to optimize analysis efficiency and accuracy. Leveraging this structure, the LLM iteratively traverses first the SDG and then the hammock-block PDG at the required level to narrow down the pool of potentially responsible microservices and the corresponding fault sites in code or configuration, until it reaches a conclusive decision on the root cause.

Unlike prior agentic approaches that process code as monolithic text [16], [20] for RCA, PRAXIS provides an efficient approach for strategically traversing the SDG and the corresponding hammock blocks in PDG, analyzing local code context alongside observability data. The graph traversal constrains the LLM's diagnosis to incident-relevant dependency paths and accumulates code insights that could explain the observed error signals (e.g., alerts, error traces, error logs), while filtering out irrelevant paths. Thus, instead of relying solely on the LLM's attention mechanism to implicitly infer code structure and focus on context relevant to the incident, PRAXIS's workflow explicitly imposes structure-aware, concise, and accurate diagnosis of the underlying incident. Based on the observability data and the accumulated code insights, the LLM classifies the microservice under investigation as PRIMARY FAILURE, SYMPTOM ONLY, or UNRELATED. Upon finishing its investigation of the current microservice, PRAXIS moves on to investigate the other selected microservices and the dependees according to the SDG. Once all initially selected microservices and their dependees have been investigated, PRAXIS calls the LLM to consolidate the per-microservice investigation decision into a final RCA report detailing the fault's origin, propagation, and impact.

Cross-SDG-PDG traversal: a challenging scenario. A key innovation of our approach is the ability to handle incidents for which multiple microservices are identified as potential root causes. In such cases, the LLM must traverse from the first microservice into its specific hammock block within the PDG, and then transition to a second microservice, according to the SDG and its respective hammock blocks, to form a coherent and holistic RCA decision. PRAXIS facilitates this by

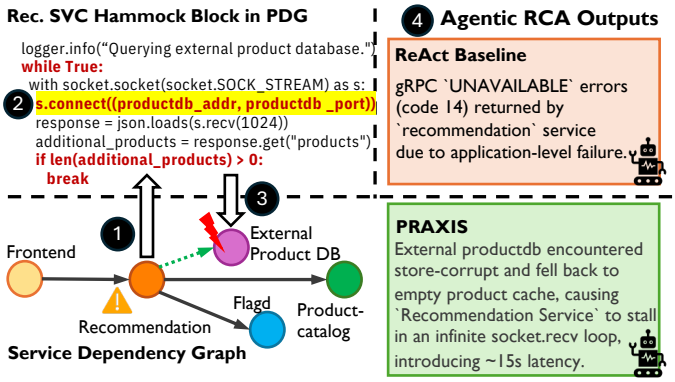

Fig. 1: **Incident:** Degraded external database returned empty responses, triggering a silent retry loop in the Recommendation service that manifested solely as a high-latency alert associated with the Recommendation service, without explicit error logs or error traces. **Cross-SDG-PDG traversal:** (1) LLM selects the Recommendation service for investigation based on the observed alert. (2) Investigation of the Recommendation service reveals a silent retry loop with missing error logs. Code traversal shows that an unresponsive External Product Database triggers this loop. (3) Subsequent investigation shows a storage failure in the External Product DB as the root cause. (4) The ReAct agent baseline fails to pinpoint the precise root cause, whereas PRAXIS successfully isolates the root cause.

attaching the PDG of each microservice to the global SDG, ensuring that the LLM can perform these cross-layer traversals when needed. We demonstrate the necessity of this unique capability in Figure 1, using a challenging incident that the baseline ReAct agent failed to solve.

IV. PRAXIS: METHODOLOGY

This section details the methodology of PRAXIS, which employs an iterative reasoning process to perform RCA on cloud incidents. PRAXIS operates in the following phases:

Phase 1 (§IV-A, Figure 2): Data gathering and dependence graph construction. PRAXIS collects the *incident context*: error traces from Jaeger [25], sustained alerts from Prometheus [24], the SDG snapshot from the cloud topology monitoring platform [29], and the PDG, constructed using Tree-sitter [30] and CLDK [31], from the microservice source code.

Phase 2 (§IV-B, Figure 3): Microservice candidate(s) selection. Using the incident context (i.e., error traces and sustained alerts), PRAXIS employs an LLM to identify initial *root-cause candidate* microservices.

Phase 3 (§IV-C, Figures 4 and 5): RCA decision-making. PRAXIS directs an LLM to iteratively traverse the microservice's PDG to construct *program context* and diagnose code regions relevant to the incident, then assign an RCA judgment (PRIMARY FAILURE, SYMPTOM ONLY, or UNRELATED) before advancing to the next microservice as suggested by the SDG.

Phase 4 (§IV-D, Figure 6): Final RCA summary. Upon completing all investigations, PRAXIS consolidates the LLM's judgments and reasoning to generate a comprehensive RCA report (Figure 6) that provides (1) root-cause identification and (2) root-cause reasoning down to the responsible statements, functions, or configurations.

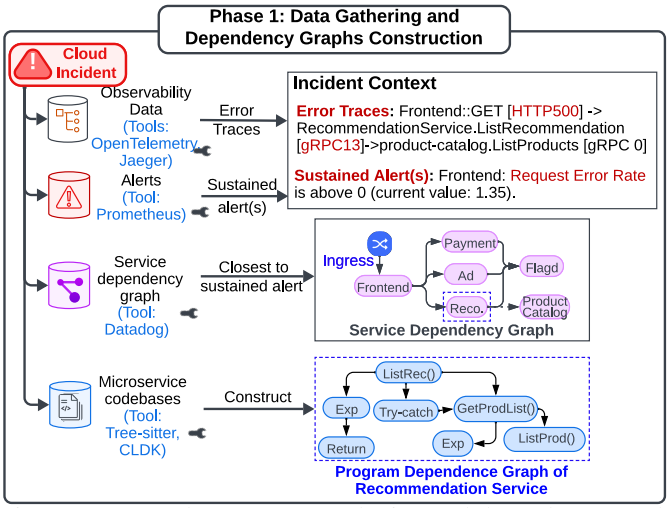

Fig. 2: PRAXIS Phase 1: Data gathering and dependency graph construction.

A. Phase 1: Data Gathering

In Phase 1 (Figure 2) of the agentic RCA workflow, PRAXIS leverages cloud monitoring tools [24]–[26] to actively monitor alerts (e.g., error rates, latency). Upon detecting a sustained alert indicating a sustained SLO violation, it collects and aggregates the relevant observability data to construct the *incident context*, as detailed below.

1) *Alerts.* Alerts signal service-level objective (SLO) violations, including high latencies and elevated error rates. PRAXIS continuously polls the Prometheus API for any active sustained alerts; these alerts serve as the initial triggers for PRAXIS's RCA workflow.

2) *Distributed error traces.* Distributed error traces, obtained via cloud-native observability tools (e.g., OpenTelemetry and Jaeger), record runtime request latencies and propagations of inter-service requests, explicitly highlighting error dependency paths across microservice boundaries for initial fault localization.

3) *Logs.* Cloud application and microservices log entries and their associated error/warning patterns[1] provide rich runtime context (e.g., information, warnings, errors, stack traces, exceptions), offering direct insight into the timing and nature of runtime executions, errors, and failures.

4) *Events.* Kubernetes events regarding deployment updates and lifecycle changes that affect the microservice or its resources (e.g., ReplicaSet scaling, container restarts) provide diagnostic context on deployment/resource configuration changes that could reveal the root cause.

5) *Metrics* include aggregated error and warning counts derived from logs and events, alongside span and trace latency measurements to quantify the severity of observed anomalies.

[1]To avoid context overflow, only the K most recent entries are collected. We set $K = 40$ for our evaluation, as empirical testing showed that increasing the value further does not significantly alter RCA accuracy.

PRAXIS aggregates sustained alerts and error traces into the incident context, which serves as the initial description of the cloud incident. Moreover, logs, metrics, and events will be used in Phase 3 for per-microservice investigation.

Figure 2 illustrates an example incident context for a cloud incident in which the Frontend service is unable to recommend products: (1) a sustained `Request Error Rate` alert is observed in the Frontend service, and (2) the corresponding error traces reveal `gRPC13` errors during calls to the `Recommendation.ListRecommendation` endpoint.

SDG & PDG PRAXIS's agentic workflow utilizes two graphs: (1) a dynamically evolving SDG for the cloud application, representing the current microservices configuration that is being deployed and executed; and (2) a hammock-block PDG for each microservice node in the SDG, representing the software dependency of a microservice program relevant to root-cause analysis. Together, these graphs capture relevant microservice-level and code-level dependencies and serve as bases for agentic RCA via LLM-driven graph traversal. PRAXIS constructs these graphs in Phase 1 after gathering data, as shown in Figure 2.

a) Service dependency graph (SDG): PRAXIS employs a *service dependency graph* (SDG) to capture the currently deployed microservices and their dependencies at the time of the incident. A *SDG* models the dependency structure of a cloud application $C = \{e_1, \ldots, e_n\}$, where each e_i is a microservice or one of its associated resources (pod or ConfigMap) [27], [28]. It is defined as a directed graph $G_C = (V, E_{\text{depend}}, \lambda_V)$ with $V = C$, where an edge (u, v) indicates that microservice u depends on v (another microservice or one of its resources), and non-microservice nodes serve as leaves with no outgoing edges. Each node is annotated by λ_V to record its kind and name, completing the definition of the SDG. The SDG G_C is constructed and maintained by a cloud topology monitor [29]. To capture the dynamic nature of the cloud environment, the graph is updated at predefined intervals. Consequently, PRAXIS retrieves the graph snapshot at the time of the alert to bootstrap the RCA process. Since the investigation can extend to non-microservice cloud components such as a pod or a configMap associated with a microservice, we hereinafter refer to a node in the SDG as an *entity*. Figure 2 presents an example SDG[2] of the cloud application [32] employed by our benchmark.

b) Program dependence graph (PDG): To perform fine-grained diagnosis, PRAXIS augments each microservice node in the SDG with a *program dependence graph* (PDG) [22], enabling reasoning over code paths and potential fault sites. For a microservice program P comprising a finite set of single-entry–single-exit (SESE) hammock blocks $B = \{b_1, \ldots, b_n\}$ [8], [9], the PDG is defined as a labeled directed graph $G_P = (V, E, \lambda_V, \lambda_E, \prec)$. Here, $V = B$, and edges $E = E_{\text{ctl}} \cup E_{\text{data}} \cup E_{\text{call}}$ capture control-, data-, and call-dependence between blocks. λ_V and λ_E annotate nodes and edges with their corresponding semantic attributes: λ_V includes node type, defined and used variables, string literals, and code snippets, and λ_E includes dependency type (control-flow, data-flow, or caller-callee) and associated variable or function names. A containment relation $\prec$ specifies when one hammock block syntactically contains another, inducing a hierarchical structure over module-, class-, function-, and statement-level regions. This hierarchy allows PRAXIS to traverse and reason about the program at the most informative granularity for RCA.

[2]For brevity, we do not show all nodes/edges in the figure.

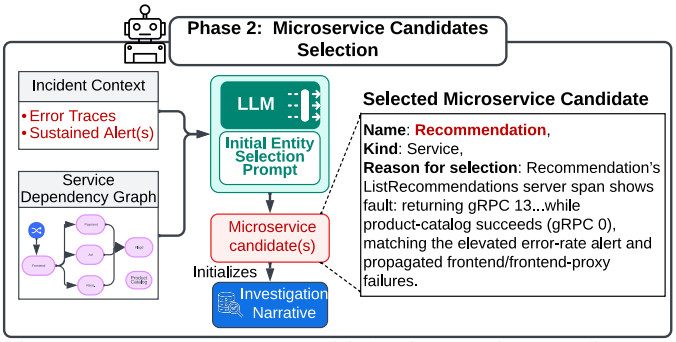

Fig. 3: PRAXIS Phase 2: Initial microservice candidate(s) selection.

Provided with the microservice codebases, PRAXIS automatically generates the PDG for microservices as part of the initialization process, using static program analysis tools: Tree-sitter [30] for parsing hammock blocks and CLDK [31] for data, call, and control relationship analysis. The PDGs generated are stored as adjacency lists in JSON files. This ensures a lightweight, language-agnostic universal format that facilitates efficient storage, management, update, and query. During agentic RCA, PRAXIS loads the PDG for each microservice under investigation at runtime.

Figure 2 shows a representative PDG of the Recommendation service from [32], comprising function- and statement-level hammock blocks.[3] `Exp` denotes expression statements (blocks of contiguous, straight-line statements without branching), whereas other nodes represent branching logic (e.g., the `try-catch` node). Function-level blocks are explicitly labeled with function names (e.g., the `GetProdList()` node).

B. Phase 2: Microservice Candidate(s) Selection

In Phase 2 (Figure 3), PRAXIS invokes an LLM to select an initial set of cloud *microservice candidates* potentially responsible for the incident to start the investigation process using the SDG and the incident context from Phase 1. PRAXIS prompts the LLM (denoted by $\mathcal{M}$) with ψ_{select}, conditioning the selection on the SDG $G_C = (V, E)$ and the incident context, which comprises active sustained alert(s) (A) and collected error traces (T). The LLM suggests a set of initial microservice candidates (up to N[4]) from nodes in G_C, and outputs an initial *investigation narrative* I_0 describing the observed alerts, traces, and the reasons why they were selected. In addition, PRAXIS initializes an investigation queue Q to actively keep track of

[3]We omit the full PDG for brevity, as the actual structure contains 44 hammock blocks.

[4]N is a user-defined variable that we set to $N = 5$ for our experiments. Further increases of this value yielded no improvements to our results.

microservices that need to be investigated and initializes Q with those selected candidates: $Q \leftarrow \mathcal{M}(\psi_{select}(A, T, \lambda_V(G_C)))$.

Figure 3 shows an example microservice candidate selection, based on the sustained alert and error traces from Phase 1. LLM selects the Recommendation service as an initial candidate, noting that `gRPC` calls from the Frontend service exhibit `gRPC 13` errors. These errors propagate upstream, manifesting as `HTTP500` errors observed in the Frontend service.

The initial investigation narrative I_0 and the investigation queue Q serve as the starting point for PDG traversal and RCA decision-making in Phase 3.

C. Phase 3: RCA Decision-making

In Phase 3 (Figure 4), given the investigation queue Q and the initial investigation narrative I_0 from Phase 2, PRAXIS iteratively dequeues and investigates an entity e from the head of Q and determines its role: a root cause, a symptom, or unrelated to the cloud incident.

For each dequeued entity $e_i \in Q$ (designated as the *focal entity*) at iteration i, PRAXIS first constructs the *observability context* c_i and retrieves the corresponding PDG G_P. Crucially, PRAXIS then employs an *LLM-driven PDG traversal* that traverses the entity's (e_i's) PDG (G_P) to construct an incident-centric program context C_P that explains the observed error symptoms from the perspective of entity code context. Given the observability context c_i and program context C_P of the focal entity, PRAXIS uses the LLM to determine its role in the cloud incident as one of PRIMARY FAILURE (the root cause), SYMPTOM ONLY (a symptom), or UNRELATED. After finishing the investigation of the current entity, PRAXIS moves on to investigate the other LLM-selected microservices and the dependees according to the SDG G_C by adding them to the investigation queue Q. PRAXIS repeats this analysis on all entities in Q until Q is empty.

Specifically, this phase consists of the following steps:

(a) Observability context construction. PRAXIS first retrieves the focal entity e_i's observability data, comprising logs, metrics, and events, the SDG (G_C), and the focal entity's PDG (G_P), built in Phase 1. PRAXIS additionally collects inter-service interactions (e.g., inbound/outbound calls) as indicated by the SDG and physical infrastructure attributes (e.g., cluster, node IDs) for each microservice-type entities. It also collects the current configuration status using `Kubectl` for ConfigMaps-type entities. PRAXIS structuralizes these collected data into a JSON representation at the time of incident investigation for the LLM to consume as *observability context* c_i, providing the most up-to-date runtime evidence to anchor the agentic program analysis and RCA reasoning.

(b) LLM-driven PDG traversal. Leveraging the observability context c_i and the graph G_P, the LLM agent operates in an iterative loop to traverse code paths defined by hammock blocks and code dependencies in the PDG. It analyzes how the underlying code logic aligns with the observed signals to pinpoint the root cause and explain the incident.

Initial hammock block (PDG node) selection. The first step to ensure that the PDG traversal begins from a semantically relevant point tied to the observed cloud incident is to identify the initial node b_0 in the PDG G_P as the starting point of the traversal. This node represents the code region most relevant to the observed symptom described in the entity's observability context c_i (e.g., error descriptions, exception stack-trace, file name, and line numbers).

Given c_i and the PDG, we employ a language model $\mathcal{M}$ conditioned on a prompt composition function ψ_{match} that encodes the runtime observability context c_i and the node attributes of the PDG λ_V (see §IV-A0b). Specifically, $\psi_{match}(c_i, \lambda_V)$ constructs a joint representation with:
(1) observability data such as error logs, exception stack frames, error events, and metric anomalies;
(2) PDG node metadata including the method names, corresponding code fragments, and file/line identifiers and string literals.

The language model $\mathcal{M}$ then evaluates the correspondence between those two modalities and selects an initial hammock block node from G_P:

$$b_0 = \mathcal{M}\left(\psi_{match}\left(c_i, \lambda_V \in V(G_P)\right)\right)$$

where $b_0 \in V(G_P)$ is a hammock block whose attributes (e.g., a string literal) correlate to the observed signals (e.g., a log entry). PRAXIS defaults to the hammock block corresponding to the entry-point (e.g., the `main` function) or request-handling methods (e.g., an API-handler function) of the microservice program when observability data are absent or insufficient due to the incident or when the matching is ambiguous (e.g., multiple matches).

PDG traversal. Once the starting hammock block b_0 (i.e., a node in PDG) has been identified, PRAXIS orchestrates the LLM's iterative agentic traversal of the PDG (G_P) to construct program context C_P for in-depth RCA. Here, the program context C_P summarizes the microservice code paths that are likely related to the incident and explains the observed error symptoms from the perspective of this code path. As such, C_P provides rich, in-depth diagnostic context beyond symptom-level observability context for determining the microservice's role in the incident.

We define this agentic process as a 4-tuple:

$$\mathcal{A} = \langle \mathcal{M}, \mathcal{T}, \mathcal{S}, G_P \rangle$$

- $\mathcal{M}$ is the LLM used for reasoning and decision-making;
- $\mathcal{T}$ is the tool-set used by $\mathcal{M}$ to traverse over the PDG;
- $\mathcal{S}$ is the agent's state space;
- G_P is the PDG for microservice e_i that acts as a transition function according to the tool action.

The PDG traversal starts from the initial hammock-block node, $b_j = b_0$, which serves as the initial anchor in the PDG. At each PDG-traversal iteration j, we define the state as:

$$s_j = \langle b_j, B_{related}, c_i, H_j \rangle \in \mathcal{S}$$

where $b_j \in V(G_P)$ is the current hammock-block node under analysis; $B_{related}$ is the set of blocks in G_P related to b_j by control, data, or call dependencies ($B_{related} = \{b | (b_j, b_k) \in E(G_P)\}$); c_i is the current microservice's observability context; and H_j is the past traversal history up to step j as PDG-traversal memory.

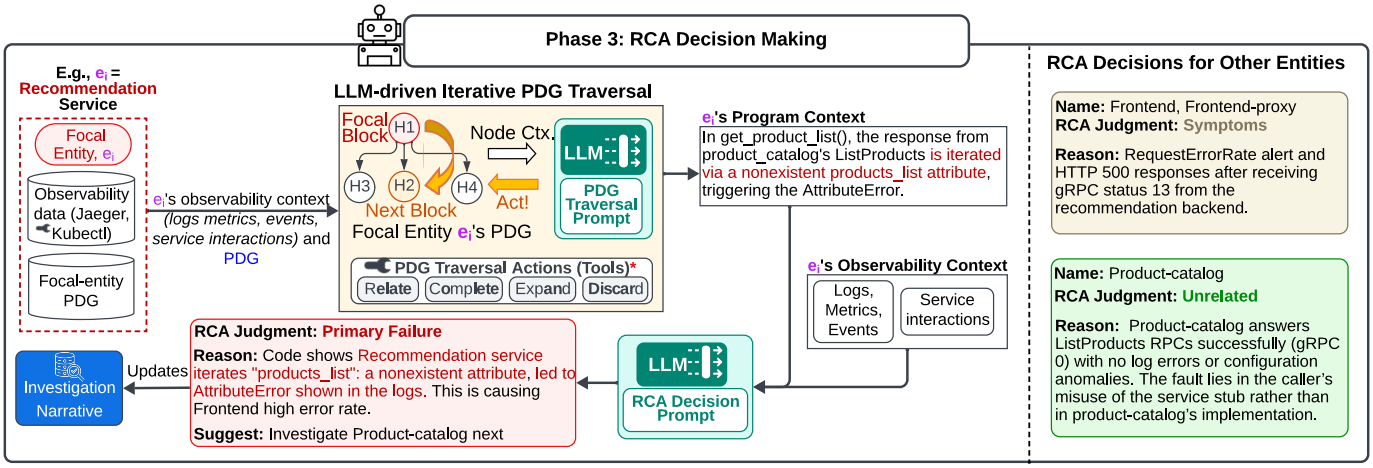

Fig. 4: PRAXIS Phase 3: RCA decision-making. This process is repeated for the next focal entity that is (1) a dependee of the current focal entity and/or (2) suggested by the LLM based on the focal entity's RCA decision.

Conditioned on the current state s_j, the LLM $\mathcal{M}$ produces a code-insight (ι_j) and the graph-traversal action (a_j) pair, i.e., $(a_j, \iota_j) = \mathcal{M}(\psi_\tau(s_j))$, where $\mathcal{M}$ is the LLM executing the reasoning and policy selection. ψ_τ is the prompt composer that encodes the current state s_j into a structured form parametrized by a template τ. In particular, b_j and $B_{related}$'s corresponding code snippets and relations are fetched by ψ_τ. $a_j \in \mathcal{T}$ is the next chosen action (mapped to our tools), and ι_j is the natural-language insight derived as the outcome of the current state/action. Notice that the LLM only consumes code context local to hammock blocks (b_j, $B_{related}$) for each traversal step, instead of large code files or codebases; this reduces the reasoning context space and allows the LLM to focus on concise, accurate context relevant to the incident, as indicated by c_i.

The agent's traversal action is implemented with a set of PDG traversal tools $\mathcal{T}$. Each tool in $\mathcal{T}$ corresponds to a concrete, admissible traversal action on the PDG G_P. Table I provides a detailed summary of the above tools. When an inadmissible action (e.g., a nonexistent block) is rejected by $\mathcal{T}$, PRAXIS prompts the LLM to regenerate a traversal action, subject to a predefined retry limit. Each action invocation updates the traversal history (H_{i+1}) and fetches the next node ($b_{j+1} \in G_P$) deterministically based on a_j via $\mathcal{T}$ on G_P: $b_{j+1} = \mathcal{T}(G_P; a_j)$, $H_{j+1} = H_j \cup \{b_j, \iota_j, a_j\}$; we can then construct s_{j+1} accordingly for the next iteration.

Figure 5 illustrates a PDG traversal step in which the LLM selects the `Relate` action to focus on Block 1 (the `get_product_list` function) for further investigation of the observed `AttributeError`. The LLM reasons that although the error is caught in the current focal block (the `try-catch` statement), it likely arises during execution of `get_product_list`.

The traversal terminates when one of the actions, COMPLETE or DISCARD, is produced, when all the nodes are exhausted, or when the user-defined budget (e.g., max number of blocks) is reached. Upon completion, the agent updates its memory state H by committing the final traversal record, yielding the complete traversal history H_T (assuming T traversal steps in total). Notice that the visited code blocks ($\{b_0, b_1, ..., b_T\}$) encompassed in H_T form a code dependency path admitted by control, data, and call dependencies. Then, the LLM uses a prompt ψ_{syn} to combine traversal history and observability context, producing the final program context $C_p(e_i)$ for microservice e_i: $C_P(e_i) = \mathcal{M}(\psi_{syn}(H_T, c_i))$. Here, C_P highlights code regions, dependencies, and explanations, which link error signals in the observability data with program evidence (code snippets and dependency flows) that might explain the incident.

TABLE I: PDG traversal tools available to the agent during graph-based program analysis.

Tool	PDG Operation	Description	Effect on Reasoning
Expand	parent(b_i)	Moves to the immediate parent or dominator node.	Broaden to examine higher-level hammock blocks or callsites.
Relate	$\Gamma(b_i)$	Visits neighboring nodes linked by call, control, or data dependencies.	Explores adjacent code paths to collect supporting or refuting evidence.
Complete	—	Terminates traversal and triggers synthesis of program context C_P.	Consolidates and finalizes the code-level hypothesis.
Discard	—	Terminates traversal without using current findings.	Prunes irrelevant or low-confidence trajectories.

The traversal tool-set $\mathcal{T} = \{\text{Expand}, \text{Relate}, \text{Complete}, \text{Discard}\}$ defines the discrete action space of the agent. Each action is implemented as a callable operation on the versioned PDG stored in the Neo4j-like graph database.

(c) Entity role judgment and queue update. The final stage in this phase is entity role judgment and an investigation queue update, in which PRAXIS provides a diagnostic judgment on the entity e_i's role in the incident and an admissible explanation with evidence, and updates the investigation queue accordingly.

Entity role judgment. After the program context for the entity e_i has been synthesized, PRAXIS determines the role of the current entity e_i in the incident: whether e_i is the root cause (primary cause of failure), a symptom, or unrelated. The role is inferred by a language model $\mathcal{M}$ conditioned on a prompt ψ_j that composes observability (c_i) and program context (C_P) into a reasoning prompt for the LLM $\mathcal{M}$, i.e., $J = \mathcal{M}(\psi_j(c_i, C_P))$, where J is one of {PRIMARY FAILURE, SYMPTOM ONLY,

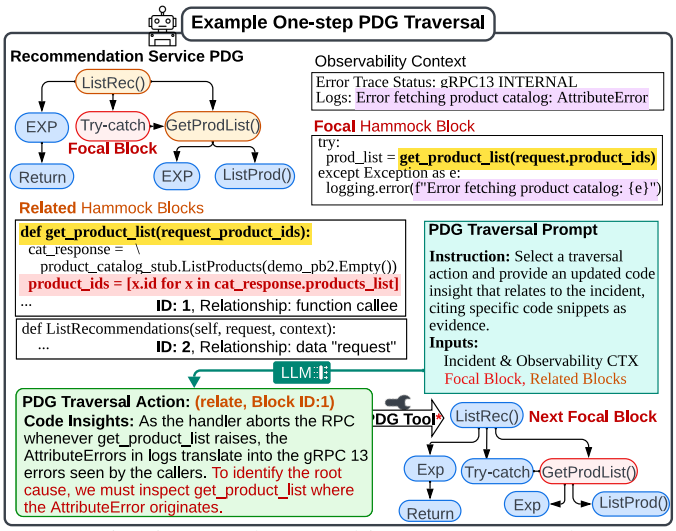

Fig. 5: Example LLM-driven PDG traversal.

or UNRELATED}. The model returns both the categorical judgment J and its natural language reasoning (I_{e_i}) with evidence grounded in both observability and program context (c_i and C_P). I_{e_i} is then appended to the global investigation narrative $I = I \cup I_{e_i}$ used in subsequent investigation.

Investigation queue update. The entity investigation queue Q is then expanded to include new entities that are (1) dependees of the current focal entity, and/or (2) suggested by the LLM based on the focal entity's RCA decision. For (1), PRAXIS enqueues all unvisited dependees of e_i according to the SDG G_C, denoted by $Q_{G_C} = G_C.\texttt{getDependees}(e_i)$, where each dependee is a cloud entity on which e_i depends. For (2), PRAXIS prompts the LLM $\mathcal{M}$ using the entity-queuing prompt ψ_Q to identify up to k additional entities[5] from G_C. This selection is based on the judgment J, the observability context c_i, the program context C_P, and the nodes in the SDG G_C, such that $Q_{\mathcal{M}_Q} = \mathcal{M}(\psi_Q(J, c_i, C_P, G_C))$. Finally, the queue is updated: $Q = Q \cup Q_{\mathcal{M}_Q} \cup Q_{G_C}$. Mechanism (2) is crucial as it allows the LLM to investigate entities in G_C that surface from the program context C_P despite having partial or missing observability data (due to insufficient coverage or the incident itself). Our evaluation confirms that this capability helps diagnose entities that do not emit observability data (e.g., a ConfigMap) or have failed silently.

Figure 4 depicts the investigation of the Recommendation service (selected in Phase 2, Figure 3), which was classified as a PRIMARY FAILURE based on its program (C_P) and observability (c_i) contexts as well as subsequent RCA judgments for the Frontend, Frontend-proxy, and Product-catalog services.

PRAXIS **repeats** this phase (Phase 3) for the next entity e_{i+1} at the head of the updated queue Q. By cross-traversing between the SDG and PDG, PRAXIS can resolve cloud incidents that require multiple hops to root-cause the incident while maintaining a coherent RCA summary via a consistent stream of investigation narrative I, a feature that is unobtainable by simply combining independent agents [18], [19], [33], [34]. Critically, leveraging code-level insights to bypass gaps in the SDG, PRAXIS can traverse across missing links (e.g, error traces, logs) in observability data, as shown by the Cross-SDG-PDG traversal example in §III.

[5]We set $k = 3$ in our implementation; however, across all evaluation runs, the LLM selected at most two entities.

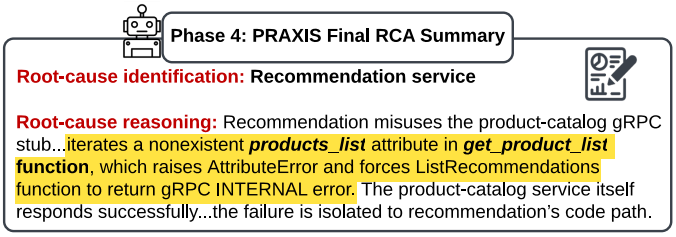

Fig. 6: PRAXIS Phase 4: Final RCA summary.

Moreover, by iteratively traversing the PDG node-by-node, PRAXIS progressively refines the RCA scope, transitioning from broader service-level observability signals to a more focused analysis of specific code regions for each focal entity. Consequently, at each step, the LLM is strictly exposed to the local context of the current graph node rather than multiple microservices or codebases/code files. This isolation of unrelated information (other node contexts) improves context efficiency, reduces token consumption, and mitigates potential context overflow, allowing PRAXIS to scale effectively to large cloud applications and codebases.

D. Phase 4: Final RCA Summary

After all entities in Q have been investigated, the LLM $\mathcal{M}$ synthesizes a comprehensive RCA report using its accumulated investigation history I from Phase 3 with a prompt $\psi_{summary}$ to produce a diagnosis report D, i.e., $D = \mathcal{M}(\psi_{summary}(I))$. D identifies the primary failure entities (root-cause entities), root-cause reasoning, and fault propagation chain, while providing a justification for each entity's diagnosis, citing evidence from the microservice code and observability data. This step consolidates the investigation and generates a structured, human-readable RCA report grounded in both observability and program-level reasoning.

Figure 6 presents an example final RCA report. PRAXIS correctly identifies the Recommendation service as the root cause, specifically attributing the failure to a schema mismatch (accessing nonexistent fields), and suggests remediation. It also accurately classifies Frontend and Frontend-proxy as SYMPTOM ONLY and Product-catalog as UNRELATED.

V. EXPERIMENT SETUP

This section first clarifies the problem and evaluation scope, and then presents the cloud incident scenarios, baselines, and metrics used for evaluation.

A. Problem and Evaluation Scope

We evaluate PRAXIS on application-level incidents, focusing on microservice codebases for which source code is available for analysis. Our approach is beneficial even under partial code availability, as reasoning over available code improves diagnostic accuracy (see §VI-A); when source code is unavailable,

PRAXIS falls back to using only observability and incident contexts for RCA. Gains are naturally limited when no code is available (see §VI-B). PRAXIS's scope is confined to incident diagnosis within the investigation/diagnosis phase of the cloud incident lifecycle [21], [35].

B. Cloud Incident Evaluation Scenarios

We developed the *Code-Cloud-RCA Benchmark* comprising 30 cloud incidents using [32], the same cloud application used in [23], supported by a standard monitoring stack [24]–[26], [29], [36]. These scenarios are distilled from root-cause types commonly reported in production studies [3], [11]–[14], [37] and real-world incidents [12], [38]. Table II enumerates the scenario templates, fault mechanisms, expected symptoms, and reference to real-world incidents; the rightmost column (#) records the number of scenarios per template. As with any benchmark, comprehensive coverage cannot be guaranteed, and by providing this benchmark, we invite the broader community to contribute to and expand its scope. Scenarios driven by the same template vary in timing (startup vs. serving), observability (stack traces vs. error logs vs. error codes), failure modes (stalls vs. rapid retries), or configurations, and are annotated with ground-truth root-cause microservices and reasoning.

C. Baselines

We benchmarked PRAXIS against ReAct-style [44] RCA agents [15]:

SRE-Agent (baseline). ReAct-style SRE-Agent [15] that uses standard SRE tools to access the Kubernetes API, traces, metrics, events, and microservices application logs.

SRE-Agent+CT (baseline+). Extends SRE-Agent with *code tools (CT)* for code retrieval, inspection, and summarization, adding code context to the baseline.

PRAXIS (ours). The proposed method, PRAXIS, employs LLM-driven reasoning and traversal over program dependence and service dependency graphs.

D. Evaluation Metrics

We evaluated RCA effectiveness with respect to the below metrics following the same definitions as [15]:

Root cause identification (RCI) Pass@1 (%): The percentage of instances for which the identified faulty microservice(s) exactly match the ground truth. In other words, RCI measures whether the agent correctly localizes the root cause to the responsible microservice(s) or associated resource(s).

Root cause reasoning (RCR) Pass@1 (%): Pass@1 accuracy of the agent's RCA reasoning, measured by whether its explanation correctly identifies the faulty statement(s), function(s), or configuration(s) relative to the ground truth, following *Pass@K* [45] with $K = 1$. In other words, RCR measures whether the agent pinpoints the underlying fault (e.g., a faulty code-site or configuration) and correctly explains (reasons about) how it led to the incident and the observed symptoms.

We also report RCA efficiency via **MTTD** (mean time to diagnosis) and **ATC** (average token consumption). See §VI-C for a detailed explanation of MTTD and ATC.

TABLE II: Incident scenario descriptions and real-world reference. Deployment/resource cases are included as negative examples to highlight PRAXIS's limitations.

Scenario description	Mechanism / How simulated	Typical symptoms (Real-world incidents)	#
Data schema mismatch/incompatibility between services	Access a nonexistent field (`products_list`) in a protobuf response from another microservice.	Unhandled exception; 5xx bursts; request failures. (Monzo 2017 outage [4]; report [3], [12], [14])	4
Improper handling of external dependency failures	Upstream DB/service hangs or returns empty results; dependent service stalls without timeout or enters rapid retries.	Latency spikes; retry storms; high error rate. (Discord 2023 outage [39]; report [3], [11], [13], [14])	6
Internal logic bug	Invert an assertion or suboptimal implementation (e.g., an exponential-time LCS ranking algorithm).	Slowness/timeouts; intermittent crashes; incorrect results. (Cloudflare 2019 outage [40]; report [11], [13], [14], [37])	2
Resource label mismatch	Hardcoded, outdated label prevents discovery of intended DB/service.	Connection refused; startup failure; unavailability. (Reddit 2023 outage [5]; report [11], [12], [14])	2
Constant misconfiguration	Wrong env/DB constant (e.g., array length) distorts control flow.	Out-of-bounds access; crashes; error bursts. (CrowdStrike 2024 outage [41]; report [3])	2
Feature-flag ConfigMaps misconfiguration	Incorrect shared Flagd rules/config; effects depend on code paths and flag usage.	Crashes; pod terminations; CPU/memory pressure. (Spotify 2025 incident [42]; report [3], [14])	9
Deployment manifest error	Invalid image tag or replicas=0.	ImagePullBackOff; service unavailable. (Quryio 2020 outage [43]; report [3])	2
Resource- & Infrastructure-related fault	Chaos Mesh: CPU stress, node-level network outage, JVM corruption.	Saturation; packet loss; node/pod failures. (Multiple cases in [12])	3

E. PRAXIS Implementation

PRAXIS is implemented in Python 3.12 using the LangGraph agentic framework [46]. PRAXIS supports PDG construction for microservices written in Python and Java, a scope we chose based on current toolchain availability and the popularity of these languages [47]. For evaluation purposes, all implemented faults reside in, or are associated with, codebases available for analysis and supported by PRAXIS. We will submit our artifacts for evaluation and open-source the implementation.

VI. EVALUATION RESULTS

We answer the following research questions (RQs):

RQ-1 (§VI-A): How accurate is PRAXIS compared to baseline agents for cloud incident RCA?

TABLE III: Evaluation result of baseline and PRAXIS.

LLM model	**RCR Pass@1%↑**	**RCI Pass@1%↑**
SRE-Agent		
gpt-oss-120b	0.00 ± 0.00	1.53 ± 1.07
deepseek-r1	0.00 ± 0.0	9.79 ± 2.49
o4-mini	3.42 ± 1.50	11.64 ± 2.65
mistral-medium-3.1	9.65 ± 2.77	21.93 ± 3.88
gpt-5-codex	0.00 ± 0.00	11.43 ± 3.80
SRE-Agent w/ Code Tools (CT)		
gpt-oss-120b	0.68 ± 0.69	1.38 ± 0.97
deepseek-r1	0.00 ± 0.0	11.41 ± 2.60
o4-mini	2.68 ± 1.32	14.77 ± 2.91
mistral-medium-3.1	10.07 ± 2.55	20.86 ± 3.45
gpt-5-codex	0.00 ± 0.00	7.96 ± 2.54
PRAXIS (Ours)		
gpt-oss-120b	48.28 ± 4.15	<u>71.03</u> ± 3.77
deepseek-r1	37.50 ± 4.74	57.69 ± 4.84
o4-mini	<u>54.16</u> ± 4.15	70.14 ± 3.81
mistral-medium-3.1	4.37 ± 1.74	23.36 ± 3.61
gpt-5-codex	**61.54** ± 4.27	**73.85** ± 3.85

Metrics: (1) **RCR Pass@1**: Pass@1 % score for root cause reasoning. (2) **RCI Pass@1**: root cause identification accuracy. Best-performing agent+LLM model is shown in **bold**, and the second-best is <u>underlined</u>. Higher is better.

RQ-2 (§VI-B): How does incident-centric program context via PDG reasoning contribute to PRAXIS's improvement?
RQ-3 (§VI-C): What are the diagnosis overheads of incorporating program context in cloud RCA?

A. Baseline Comparison (RQ1)

For RQ-1, we evaluated PRAXIS by comparing it against the current state of the art, i.e., the SRE-Agent [15], a ReAct-style RCA agent with default tools, and an extended SRE-Agent with code retrieval and summarization tools (which we call SRE-Agent+CT) to study whether code context can improve on the baseline performance.

Evaluation setup. We evaluated the agent across five different LLMs: o4-mini, gpt-5-codex, mistral-medium-3.1, deepseek-r1, and gpt-oss-120b. For statistical robustness, we repeated each ⟨LLM, scenario⟩ experiment five times with distinct random seeds, totaling 2,250 trajectories.

Overall results. As shown in Table III, PRAXIS consistently performed better than SRE-Agent and SRE-Agent+CT across all LLMs that were evaluated, both on root-cause reasoning (RCR) accuracy and root-cause identification (RCI) accuracy. PRAXIS's RCA reasoning accuracy correlates strongly with root-cause identification accuracy, reflecting how human SREs rely on accurate fault pinpointing for effective root-cause analysis. Reasoning models (gpt-5-codex, o4-mini) exhibited better RCA accuracies overall. Moreover, the strong performance of gpt-5-codex likely stemmed from its superior code-understanding capabilities (demonstrated in [48]), enabling it to better correlate the code context and program behavior with the incident. The low accuracy of mistral-medium-3.1 was due to its failure to select the correct initial candidates and its inability to follow output format instructions, which resulted in parsing errors.

Why did baselines underperform? SRE-Agent underperformed because it lacks code visibility, which is essential for diagnosing code-related scenarios. Adding code tools (SRE-Agent+CT) yielded only marginal gains, not because code is unhelpful, but because the search policy remained unguided. Even with code and SRE tools, gpt-5-codex and similar models exhibited *premature closure*: once local evidence seemed to *explain* alerts (e.g., an *HTTP 500* handler or nearby error traces), they terminated the investigation. Without explicit awareness or external imposition of the microservice or program dependencies, these baselines had neither a mechanism for traversing, nor an obligation to traverse the multi-hop dependency chain (which in our scenarios often involved 7–10 hops across different microservices and their hammock blocks) in which the true faults either resided or were indicated. Notably, despite explicit instructions to invoke code tools in the prompt, SRE-Agent+CT invoked code tools for only 46% of runs in which code analysis was required for RCA, resulting in low RCA accuracy and inconsistent execution trajectories and highlighting SRE-Agent+CT's fragility in agent tool-calling [49], [50].

In contrast, PRAXIS treats service and code-level dependencies as first-class as it constrains the LLM's reasoning onto the dependency paths in PDG and systematically expands each graph node's neighborhood with admissible graph traversal actions, enforcing iterative context gathering and reasoning reconciliation. This graph-aware reasoning prevents early stopping on symptomatic context and steers the investigation to the upstream root cause in code, achieving in-depth RCA.

B. Ablation Study (RQ2)

For this RQ, we conducted an ablation study to assess the impact of PRAXIS's core components on RCA effectiveness by evaluating the following PRAXIS variants.
PRAXIS (Obs. Ctx.): The observability context-only variant disables PDG-traversal and program context (C_P), so only the incident context and observability context (c_i) are used in the RCA decision in §IV-C. This experiment characterized the performance gains resulting from the use of program context.
PRAXIS (Raw Code): The Raw Code variant skips PDG construction (§IV-A0b) and LLM-driven PDG traversal (§IV-C) when constructing C_P and instead uses an LLM with a modified ψ_{syn} prompt (§IV-C) to consolidate and generate program context directly from aggregated raw code files. We limited the raw code to 800k characters (∼200 tokens) to avoid overflowing the LLM's context window.

Evaluation setup. We evaluated these PRAXIS variants using the best-performing LLM for PRAXIS–gpt-5-codex. For statistical robustness, we repeated each ⟨variant, scenario⟩ experiment five times with distinct random seeds, yielding a total of 300 trajectories.

Ablation results. Table IV shows that augmenting the observability context c_i with program context C_P (PDG construction + LLM-guided traversal) is the primary driver of RCA effectiveness. Relative to PRAXIS (Raw Code), PRAXIS improves overall RCR Pass@1 by 28.8 percentage points, from

TABLE IV: Ablation study results on all scenario instances.

Agent Variant	Overall RCA Accuracy	
	RCR Pass@1%↑	RCI Pass@1%↑
PRAXIS (Obs. Ctx.)	12.93 ± 2.77	41.50 ± 4.06
PRAXIS (Raw Code)	32.65 ± 3.87	59.18 ± 4.05
PRAXIS (Ours)	**61.54** ± 4.27	**73.85** ± 3.85
	Data Schema Mismatch	
PRAXIS (Obs. Ctx.)	40.0 ± 14.14	**100.0** ± 0.0
PRAXIS (Raw Code)	25.0 ± 18.97	85.0 ± 14.14
PRAXIS (Ours)	**70.0** ± 8.94	95.0 ± 8.93
	Improper Ext. Failure Handling	
PRAXIS (Obs. Ctx.)	10.0 ± 11.54	33.3 ± 1.6
PRAXIS (Raw Code)	36.67 ± 20.0	43.33 ± 19.32
PRAXIS (Ours)	**85.83** ± 12.57	**86.0** ± 12.60
	Internal Logic Bug	
PRAXIS (Obs. Ctx.)	20.0 ± 15.49	100.0 ± 0.0
PRAXIS (Raw Code)	50.0 ± 17.89	90.0 ± 12.65
PRAXIS (Ours)	**80.0** ± 17.89	**100.0** ± 0.0
	Resource Label Mismatch	
PRAXIS (Obs. Ctx.)	30.0 ± 15.4	100.0 ± 0.0
PRAXIS (Raw Code)	40.0 ± 21.91	100.0 ± 0.00
PRAXIS (Ours)	**80.0** ± 15.5	**100.0** ± 0.0
	Constant Misconfiguration	
PRAXIS (Obs. Ctx.)	0.0 ± 0.0	80.0 ± 15.0
PRAXIS (Raw Code)	20.0 ± 17.89	90.0 ± 12.65
PRAXIS (Ours)	**90.0** ± 13.0	**100.0** ± 0.0
	Feature-flag ConfigMaps Misconf.	
PRAXIS (Obs. Ctx.)	0.0 ± 0.0	0.0 ± 0.0
PRAXIS (Raw Code)	45.56 ± 19.71	54.44 ± 19.71
PRAXIS (Ours)	**62.96** ± 16.67	**65.74** ± 18.16

We used the same RCA accuracy metrics as in Table III.

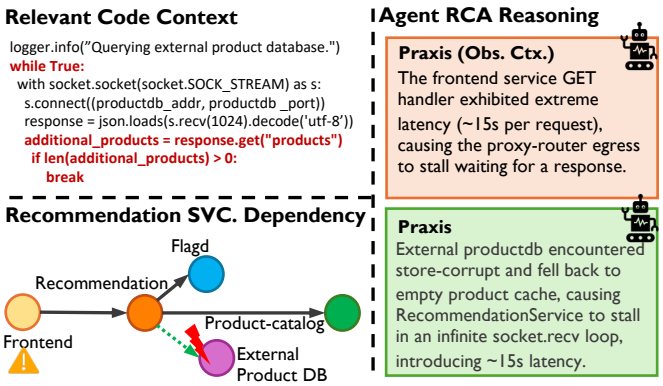

Fig. 7: RCA reasoning of PRAXIS (Obs. Ctx.) and PRAXIS. Nodes are microservices; solid arrows are dependencies; the green dotted arrow is a dependency with missing traces and had to be derived from program context. SVC. = Service.

32.7% to 61.5% (an 88.5% improvement), and RCI Pass@1 by 14.7 percentage points, from 59.2% to 73.9% (a 24.8% improvement). Against PRAXIS (Obs. Ctx.), the gains are larger, at 48.6 percentage points and 32.4 percentage points for RCR Pass@1 and RCI Pass@1, respectively.

Why do program context and LLM-driven PDG help? Similar to the SRE-Agent baseline, PRAXIS without program context (PRAXIS (Obs. Ctx.)) failed to identify root causes within, or hinted by, the internal code logic, resulting in superficial symptom-level RCA. Although the PRAXIS variant without PDG (PRAXIS (Raw Code)) retains service-level dependency awareness, the absence of PDG-guided diagnosis halved its accuracy compared to that of the full approach. Lacking explicit awareness of program-level dependencies, PRAXIS (Raw Code) cannot effectively focus its reasoning on code regions relevant to the incident. Instead, it "fixated" on ostensibly faulty regions—code that appears erroneous on the surface (e.g., manually thrown exceptions)—that are not pertinent to the observed failure. With code context exceeding 200k tokens at times, PRAXIS (Raw Code) exhibited *needle in a haystack* [51] and *context rot* [52] phenomena, leading to degraded RCA accuracy. These findings confirm that PDG-guided diagnosis is essential: it enforces the alignment of observability data with concise, accurate code context structurally presented to the LLM, grounding the RCA in relevant code paths and code regions to yield the observed performance gains.

Illustration. We illustrate the RCA reasoning specificity improvement using an "improper handling of external dependency failure" scenario. Here, a degraded external database returned empty responses, triggering a silent retry loop in the Recommendation service that manifested solely as high latency, without explicit error logs. As shown in Figure 7, PRAXIS without program context failed to detect this behavioral pattern, wrongly identifying the high latency, a symptom, as the root cause. Conversely, utilization of program context enabled PRAXIS to identify the retry logic and uncover the database dependency, correctly isolating the underlying storage issue (storage corruption → empty product list) as the true root cause.

Per-scenario trends. PRAXIS's improvements hold across most scenario types. For example, for *Constant Misconfiguration*, PRAXIS attained **90**% RCR and **100**% RCI while other agents achieved around 0%–20% RCR. A minor regression appeared in *Data Schema Mismatch* (RCI Pass@1 100.0% → 95.0%), likely because error signals in logs occasionally allow the observability-only variant (PRAXIS (Obs. Ctx.)) to correctly identify the faulty microservice without code analysis, although it is unable to arrive at the correct root cause. *Feature-flag ConfigMaps misconfiguration* scenarios remain challenging (RCI ≈ 66%), likely due to insufficient observability data for PRAXIS to exploit during the PDG-traversal and RCA decision-making.

C. Diagnosis Overheads (RQ3)

For this RQ, we evaluated the RCA overheads of SRE-Agent baselines, PRAXIS, PRAXIS (Obs. Ctx.), and PRAXIS (Raw Code) using mean time to completion (MTTC) in seconds and average token consumption (ATC) in thousands of tokens, measured across all scenarios using the best-performing LLMs.

Evaluation setup. To account for differences in RCA reasoning accuracy, we normalized both MTTC and ATC by RCR Pass@1% to assess diagnosis efficiency: *Normalized metric* = $\frac{\textit{MTTC or ATC}}{\text{RCR Pass@1\%}}$. The resulting normalized MTTC (i.e., mean time to diagnosis or MTTD) and normalized ATC (Effective ATC) reflect time and token costs per successful diagnosis. Here, MTTC is the average time to complete a diagnosis, regardless of correctness. MTTD is the normalized MTTC and represents

TABLE V: Diagnosis overheads across all scenario instances.

	MTTC↓	ATC↓	MTTD↓	Eff. ATC↓
SRE-Agent	99.69	85.39k	1,033.06	884.87k
SRE-Agent+CT	**90.18**	85.03k	**842.80**	794.67k
PRAXIS (Obx. Ctx.)	1,321.02	**75.59k**	10,475.97	599.44k
PRAXIS (Raw Code)	838.44	329.53k	2,567.96	1,009.28k
PRAXIS (Ours)	907.43	102.44k	1,474.54	**166.46k**

Metrics: (1) **MTTC** = mean time to completion in seconds; (2) **ATC** = average token consumptions in tokens; (3) **MTTD** = mean time to diagnosis; (4) **Eff. ATC** = effective ATC. Best-performing variant is shown in **bold**; second-best is underlined. Lower is better.

the expected time to reach a *correct* diagnosis. Thus, MTTD jointly captures duration and accuracy: an agent that completes diagnosis faster but is less accurate may still require more time, in expectation, to reach a correct diagnosis, resulting in an MTTD longer than that of a more accurate agent. The same interpretation applies to ATC and Effective ATC.

Table V shows that although SRE-Agent has lower raw MTTC and ATC than PRAXIS, their MTTDs, which include LLM backend response times, are comparable. However, ReAct consumed $5.1\times$ more tokens per successful diagnosis, indicating much lower token efficiency. When we compare PRAXIS with its ablated variants (PRAXIS (Raw Code)) and PRAXIS (Obs. Ctx.)), construction of program context via LLM-driven PDG traversal reduced MTTD and effective ATC by up to 85.9% and 83.5%, respectively, highlighting their significant contributions to efficient RCA. Notably, LLM-driven PDG traversal reduced ATC by $6.1\times$ compared to providing raw code as plain text in the prompt, as in [20], [53]. PRAXIS's MTTD of 1,474 seconds is substantially shorter than that of manual RCA, which can take several hours for incidents with similar root causes.

D. Discussions and Future Work

Scenarios with weak observability. PRAXIS's effectiveness is contingent on the availability and quality of observability data to anchor its PDG traversal. Sparse or ambiguous traces, logs, metrics, or events can limit PRAXIS's ability to accurately align runtime error signals with corresponding dependency paths to traverse. When a fault offers no direct or clear observable signal at the application layer to anchor RCA decision-making, PRAXIS tends to falsely attribute the incident to fault-free code paths or configurations, leading to false positives. While collecting exhaustive observability data could mitigate this, it would incur significant production overheads. We explicitly identify the fault categories most vulnerable to this limitation in our scope discussion below.

Graph representation. PRAXIS's effectiveness relies on the correctness and completeness of the dependency graphs. Dependency graphs that are incorrect or incomplete (e.g., an outdated PDG, or a missing microservice in SDG) hinder PRAXIS's ability to diagnose a root cause. Because PRAXIS diagnoses faults by traversing dependency edges anchored on observability signals, missing edges break the structural path from symptom to root cause: observability may reveal the symptomatic component, but without the corresponding edge, the agent cannot reach the true root-cause node. For isolated nodes or a complete absence of dependency structures (e.g., infrastructure/resource dependencies), traversal terminates prematurely at the upstream symptom, reducing RCA accuracy. These implications explain PRAXIS's practical scope and limitations in its present form, described in the next paragraph.

PRAXIS's scope and limitations. Our evaluation (Table IV) shows that PRAXIS is effective at diagnosing (1) code-related incidents that fall under the fault models *data schema mismatches, logical bugs, failure to handle external dependencies, and resource/address/label mismatches*, and (2) misconfiguration-related incidents caused by *misconfigured variables/constants in code or in ConfigMaps* (see §VI-B, Table IV). Collectively, these fault models represent the vast majority of code-related and misconfiguration incidents observed in production [3], [13], [14]. PRAXIS works well here as these faults emit observability signals and either are directly visible in code or can be more accurately pinpointed via code examination (e.g., a misconfiguration altering execution paths).

As discussed, PRAXIS's effectiveness depends on the availability and quality of observability signals and the completeness of its dependency graphs. Weak observability signals (e.g., missing traces or ambiguous logs) and incomplete or incorrect graphs can hinder diagnosis. This explains why PRAXIS and its variants achieve near-zero accuracy on *deployment-manifest* and *resource/infrastructure*-related incidents: those incidents lack direct application-layer observability and their dependency structures are not captured in PRAXIS's graphs. We exclude these incidents from Table IV because all ablated variants behave and perform identically, so including them would not change the ablation comparison.

For the same reason, we consider *concurrency/timing* bugs out of scope: PRAXIS in its current form lacks dynamic runtime analysis to observe microservices' internal runtime state and does not encode timing-dependency structures between microservices. These limitations are not inherent to the graph-based approach, and PRAXIS can be extended to address them, as discussed in the "Future work" section. Finally, while PRAXIS may generalize to fault types beyond those evaluated, its effectiveness on such faults remains unverified.

Access to source code. PRAXIS's gains from PDG traversal are naturally limited when microservice code is unavailable for graph construction or code access is substantially limited (e.g., extensive reliance on proprietary third-party libraries), in which case PRAXIS must rely primarily on observability signals for diagnosis. In practice, partial code access is common (and matches our evaluation): Devs/SREs can access code developed in-house but not proprietary third-party binaries. Even with partial code access, PRAXIS can exploit clues in the available code captured by the PDG (e.g., call sites and configuration-controlled branches) to localize the root cause to external dependencies, thereby achieving higher RCA effectiveness than observability-only variants, see Figure 1 and §VI-B for examples.

Static code analysis. An inherent limitation of static code analysis is its inability to accurately capture dynamic bindings (e.g., Java reflection), which can leave unlinked gaps in the PDG. Although PRAXIS cannot eliminate this limitation, its

hierarchical hammock-block traversal alleviates it by leveraging higher-level structural relations (e.g., class-level) to potentially bypass missing links at lower levels. Conversely, conditioning too strongly on static code context risks unintended bias, where PRAXIS may mistakenly attribute faults to code blocks when the incident actually stems from external factors (e.g., deployment or resources). This risk can be mitigated by incorporating runtime instrumentation to confirm whether implicated code paths were indeed executed at the time of the incident.

PDG construction overheads and continuous deployment. Regenerating all PDGs in our case took 116.42 seconds. PRAXIS builds PDGs by using a syntactic parser (e.g., Tree-sitter) to extract hammock blocks, and running CLDK-provided static analysis (e.g., WALA/RTA for Java and CodeQL/Scalpel for Python) [31]. In practice, construction time is dominated by the static-analysis backend, and lightweight backends (e.g., the RTA [54] backend used by CLDK) scale near-linearly with microservice program size [55]. Subsequent PDG construction takes $O(N + E)$, where N is the number of hammock blocks and E is the number of dependency edges extracted. Going forward, PRAXIS will be integrated with a CI/CD[6] pipeline. PRAXIS can avoid fully regenerating the PDGs because (1) PDGs can be constructed and parallelized per microservice, and (2) each PDG update can be incremental [57], [58], recomputing only nodes and transitive dependence regions affected by code changes. Moreover, caching versioned PDGs by commit SHA enables fast reuse and comparison.

Extensibility to other languages and platforms. PRAXIS is extensible beyond Python/Java and Kubernetes. SDG construction requires discovery of the runtime inter-service topology and dependencies, capabilities already supported by observability stacks (e.g., OpenTelemetry, Datadog) across platforms (e.g., OpenShift, ECS). PDG generation requires a syntactic parser for extracting hammock blocks and a static-analysis backend to derive control/data-flow edges; thus, specific language support is primarily limited by the availability of such toolchains. Because PRAXIS constructs PDG using Tree-sitter [30] and CLDK [31], which provide broad multi-language support, it can be readily extended to additional languages, including Go, Rust, TypeScript, and C/C++.

Future work. Future iterations of PRAXIS will be extended to diagnose resource-, infrastructure-, and deployment-related incidents by augmenting the current dependency graphs with resource/infrastructure and deployment-manifest graphs that explicitly capture those dependencies. Furthermore, PRAXIS will incorporate dynamic program analysis to address static-analysis gaps and enable the diagnosis of concurrency bugs. By leveraging lightweight, strategic runtime instrumentation (e.g., at database calls and gRPC handlers) alongside dynamic tracing with sampling [59]–[61], PRAXIS can capture dynamic bindings and gain crucial visibility into internal microservice states. This runtime context will help reduce false positives for resource, infrastructure, and deployment faults, while simultaneously bringing concurrency bugs into scope. Ultimately, these techniques will ground the static graph traversal in verifiable runtime behavior, enabling more comprehensive incident coverage and precise RCA.

[6]*CI/CD* stands for *continuous integration and continuous deployment* [56].

VII. RELATED WORK

Agentic approach for cloud incident RCA. Recent agentic RCA approaches [15], [17]–[19], [23] adopted the ReAct paradigm [44] to analyze observability data but lack structured reasoning workflow and program context, limiting their effectiveness on software and misconfiguration faults. Prior approaches that incorporate program or code context [16], [20] treat code as plain text in prompts rather than explicitly exploiting its inherent structure, as they do not perform PDG-guided reasoning. PRAXIS advances beyond those efforts by performing LLM-driven reasoning over service dependency and program dependence graphs, unifying incident, observability, cloud, and program context for scalable, end-to-end RCA.

AI/ML for specific RCA workflow automation. AI/ML techniques have long supported incident RCA through specialized models for anomaly detection, fault localization, and diagnosis [28], [62]–[66]. Recent work has employed LLMs for incident querying, understanding, and classification [53], [67]–[72], typically leveraging prior knowledge (e.g., TSGs, SOPs) and requiring SRE supervision. PRAXIS performs automated RCA end-to-end, and those methods can be considered complementary to it.

LLM for graph reasoning and traversal. LLM-driven graph reasoning/traversal is an emerging field [73] that can be applied to knowledge retrieval, question-answering [74]–[77], code search and localization [78], and execution path reconstruction [79]. Though graph models have been established in systems/AIOps [80], LLM-driven graph reasoning in this domain is still underexplored, and we have demonstrated its potential through PRAXIS.

VIII. CONCLUSION

This paper proposes PRAXIS, an agentic approach that reasons over microservice and program dependency graphs for a comprehensive, in-depth root-cause analysis. Our evaluation of PRAXIS on 30 scenarios that span software, configuration, deployment, and resource failures demonstrates that PRAXIS achieves up to $6.3\times$ better root-cause reasoning accuracy, $3.4\times$ higher entity identification accuracy, and a $5.3\times$ reduction in token consumption compared to state-of-the-art agentic baselines.

ACKNOWLEDGMENT

We thank the reviewers, and R. Arora, Bhavya, N. Zheutlin, P. T. Isaza, J. Ahn, A. Paradkar, L. Shwartz, R. K. Kottapalli, A. D. Angelis, A. Patke, P. Cao, Z. Zheng, and J. Applequist for technical input and feedback, and H. C. Fairow, S. Weick, K. Atchley, Y. Deng, R. Pavuluri, M. Vukovic, X. Liu, D. Sow, and N. Fuller for administrative support. This work is supported by the IBM-Illinois Discovery Accelerator Institute (IIDAI) and NSF grant 2530738. Any opinions expressed here are those of the authors and do not necessarily reflect the views of IBM or NSF.

REFERENCES

[1] New Relic, Inc., "New Relic study reveals businesses face an annual median cost of $76 million from high-impact IT outages," Sep. 2025, press release. [Online]. Available: https://newrelic.com/press-release/20250917

[2] ——, "2025 observability forecast report," 2025, report. [Online]. Available: https://newrelic.com/sites/default/files/2025-09/new-relic-2025-observability-forecast-report.pdf

[3] S. Ghosh, M. Shetty, C. Bansal, and S. Nath, "How to fight production incidents? An empirical study on a large-scale cloud service," in *Proceedings of the 13th Symposium on Cloud Computing*, ser. SoCC '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 126–141. [Online]. Available: https://doi.org/10.1145/3542929.3563482

[4] Monzo-engineer, "RESOLVED: Current account payments may fail: Major outage 27/10/2017," Oct. 2017, Monzo Community Forum. Accessed 2025-10-20. [Online]. Available: https://community.monzo.com/t/resolved-current-account-payments-may-fail-major-outage-27-10-2017/26296/95

[5] Reddit-engineer, "You broke Reddit: The Pi-Day outage," Mar. 2023, Reddit Engineering Blog. Accessed: 2025-10-09. [Online]. Available: https://www.reddit.com/r/RedditEng/comments/11xx5o0/you_broke_reddit_the_piday_outage/

[6] Amazon Web Services. (2025) Summary of the Amazon DynamoDB service disruption in the Northern Virginia (US-EAST-1) region. AWS. Post-incident summary of the Oct 19–20, 2025 US-EAST-1 disruption. [Online]. Available: https://aws.amazon.com/message/101925/

[7] M. Prince, "Cloudflare outage on November 18, 2025," https://blog.cloudflare.com/18-november-2025-outage/, Nov. 2025, accessed: 2025-11-20.

[8] R. Johnson, D. Pearson, and K. Pingali, "The program structure tree: Computing control regions in linear time," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI '94. New York, NY, USA: Association for Computing Machinery, 1994, pp. 171–185. [Online]. Available: https://doi.org/10.1145/178243.178258

[9] F. Zhang and E. H. D'Hollander, "Using hammock graphs to structure programs," *IEEE Trans. Softw. Eng.*, vol. 30, no. 4, pp. 231–245, Apr. 2004. [Online]. Available: https://doi.org/10.1109/TSE.2004.1274043

[10] itbench-hub, "ITBench-Scenarios: Kubernetes topology monitor," https://github.com/itbench-hub/ITBench-Scenarios/tree/main/sre/tools/kubernetes-topology-monitor, 2025, code repository as part of ITBench SRE scenarios. Accessed: 2025-11-21.

[11] H. S. Gunawi *et al.*, "Cloud Bug Study (CBS) database," http://ucare.cs.uchicago.edu/projects/cbs/, UCARE Research Group, University of Chicago, 2014, accessed: 2025-06-01.

[12] H. Jacobs, "Kubernetes failure stories," https://codeberg.org/hjacobs/kubernetes-failure-stories, 2023, accessed: 2025-10-07.

[13] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria, "What bugs live in the cloud? A study of 3000+ issues in cloud systems," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 1–14. [Online]. Available: https://doi.org/10.1145/2670979.2670986

[14] H. Liu, S. Lu, M. Musuvathi, and S. Nath, "What bugs cause production cloud incidents?" in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 155–162. [Online]. Available: https://doi.org/10.1145/3317550.3321438

[15] S. Jha, R. R. Arora, Y. Watanabe, T. Yanagawa, Y. Chen, J. Clark, B. Bhavya, M. Verma, H. Kumar, H. Kitahara, N. Zheutlin, S. Takano, D. Pathak, F. George, X. Wu, B. O. Turkkan, G. Vanloo, M. Nidd, T. Dai, O. Chatterjee, P. Gupta, S. Samanta, P. Aggarwal, R. Lee, J. wook Ahn, D. Kar, A. Paradkar, Y. Deng, P. Moogi, P. Mohapatra, N. Abe, C. Narayanaswami, T. Xu, L. R. Varshney, R. Mahindru, A. Sailer, L. Shwartz, D. Sow, N. C. M. Fuller, and R. Puri, "ITBench: Evaluating AI agents across diverse real-world IT automation tasks," in *Forty-second International Conference on Machine Learning*, 2025. [Online]. Available: https://openreview.net/forum?id=jP59rz1bZk

[16] Z. Wang, Z. Liu, Y. Zhang, A. Zhong, J. Wang, F. Yin, L. Fan, L. Wu, and Q. Wen, "RCAgent: Cloud root cause analysis by autonomous agents with tool-augmented large language models," in *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, ser. CIKM '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 4966–4974. [Online]. Available: https://doi.org/10.1145/3627673.3680016

[17] D. Roy, X. Zhang, R. Bhave, C. Bansal, P. Las-Casas, R. Fonseca, and S. Rajmohan, "Exploring LLM-based agents for root cause analysis," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, pp. 208–219. [Online]. Available: https://doi.org/10.1145/3663529.3663841

[18] J. Xu, Q. Zhang, Z. Zhong, S. He, C. Zhang, Q. Lin, D. Pei, P. He, D. Zhang, and Q. Zhang, "OpenRCA: Can large language models locate the root cause of software failures?" in *The Thirteenth International Conference on Learning Representations*, 2025. [Online]. Available: https://openreview.net/forum?id=M4qNIzQYpd

[19] Y. Chen, J. Pan, J. Clark, Y. Su, N. Zheutlin, B. Bhavya, R. Arora, Y. Deng, S. Jha, and T. Xu, "Stratus: A multi-agent system for autonomous reliability engineering of modern clouds," in *Proceedings of the 39th Conference on Neural Information Processing Systems (NeurIPS 2025)*, 2025, accepted; preprint available at arXiv:2506.02009. [Online]. Available: https://arxiv.org/abs/2506.02009

[20] Y. Li, Y. Wu, J. Liu, Z. Jiang, Z. Chen, G. Yu, and M. R. Lyu, "COCA: Generative root cause analysis for distributed systems with code knowledge," in *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE '25)*, 2025. [Online]. Available: https://dl.acm.org/doi/10.1109/ICSE55347.2025.00234

[21] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*, 2016. [Online]. Available: http://landing.google.com/sre/book.html

[22] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987. [Online]. Available: https://doi.org/10.1145/24039.24041

[23] Y. Chen, M. Shetty, G. Somashekar, M. Ma, Y. Simmhan, J. Mace, C. Bansal, R. Wang, and S. Rajmohan, "AIOpsLab: A holistic framework to evaluate AI agents for enabling autonomous clouds," in *Proceedings of MLSys '25*, 2025. [Online]. Available: https://openreview.net/forum?id=3EXBLwGxtq

[24] Prometheus Authors, "Prometheus: Monitoring system & time series database," 2025, open-source systems monitoring and alerting toolkit. [Online]. Available: https://prometheus.io/

[25] Jaeger Project, "Jaeger: Open source, distributed tracing platform," 2025, originally open-sourced by Uber; CNCF project. [Online]. Available: https://www.jaegertracing.io/

[26] ClickHouse, Inc., "ClickHouse: Fast open-source OLAP DBMS," 2025, column-oriented database for real-time analytics. [Online]. Available: https://clickhouse.com/

[27] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "MicroRCA: Root cause localization of performance issues in microservices," in *NOMS 2020: 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, pp. 1–9. [Online]. Available: https://doi.org/10.1109/NOMS47738.2020.9110353

[28] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: Practical and scalable ML-driven performance debugging in microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 135–151. [Online]. Available: https://doi.org/10.1145/3445814.3446700

[29] Datadog, "Datadog cloud monitoring platform," https://www.datadoghq.com/, 2025, accessed: 2025-06-01.

[30] M. Brunsfeld, A. Qureshi, A. Hlynskyi, ObserverOfTime, W. Lillis, J. Vera, dundargoc, P. Turnbull, T. Clem, D. Creager, A. Helwer, R. Rix, D. Kavolis, C. Clason, M. Davis, R. Bruins, A. Delpeuch, Ika, A. Ya, T.-A. Nguyễn, bfredl, S. Brunk, M. Massicotte, N. Hasabnis, J. McCoy, M. Dong, S. Moelius, S. Kalt, and Kolja, "tree-sitter/tree-sitter: v0.25.10," September 2025, software; MIT License. [Online]. Available: https://doi.org/10.5281/zenodo.17180150

[31] R. Krishna, R. Pan, S. Sinha, S. Tamilselvam, R. Pavuluri, and M. Vukovic, "Codellm-Devkit: A framework for contextualizing code LLMs with program analysis insights," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, ser. FSE Companion '25. New York, NY, USA: Association

for Computing Machinery, 2025, pp. 308–318. [Online]. Available: https://doi.org/10.1145/3696630.3728555

[32] OpenTelemetry authors, "OpenTelemetry demo: Astronomy shop microservices," https://github.com/open-telemetry/opentelemetry-demo, 2025, microservice-based distributed system illustrating OpenTelemetry in a near real-world environment. Accessed: 2025-11-28.

[33] T. Ahmed, S. Ghosh, C. Bansal, T. Zimmermann, X. Zhang, and S. Rajmohan, "Recommending root-cause and mitigation steps for cloud incidents using large language models," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, pp. 1737–1749. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00149

[34] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "SWE-agent: Agent-computer interfaces enable automated software engineering," in *Proceedings of the 38th International Conference on Neural Information Processing Systems*, ser. NIPS '24. Red Hook, NY, USA: Curran Associates Inc., 2024.

[35] Google Cloud, "Lifecycle of an incident," https://docs.cloud.google.com/service-health/docs/incident-lifecycle, n.d., accessed: 2025-12-01.

[36] OpenTelemetry Authors, "OpenTelemetry specification: Overview," https://opentelemetry.io/docs/specs/otel/overview/, 2025, overview of the OpenTelemetry project and core concepts. Accessed: 2025-11-28.

[37] H. Yan, Y. Chen, M. Ma, M. Wen, S. Lu, S. Zhang, T. Xu, R. Wang, C. Bansal, S. Rajmohan, C. Zhang, and D. Zhang, "An empirical study of production incidents in generative AI cloud services," *CoRR*, vol. abs/2504.08865, 2025, accepted to ISSRE 2025; preprint on arXiv. [Online]. Available: https://arxiv.org/abs/2504.08865

[38] M. Barletta, M. Cinque, C. Di Martino, Z. T. Kalbarczyk, and R. K. Iyer, "Mutiny! How does Kubernetes fail, and what can we do about it?" in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2024, pp. 1–14.

[39] Discord Engineering, "25% or 6 to 4: The 11/6/23 authentication outage," Nov. 2023, accessed: 2025-12-04. [Online]. Available: https://discord.com/blog/authentication-outage

[40] J. Graham-Cumming, "Details of the Cloudflare outage on July 2, 2019," Jul. 2019, accessed: 2025-12-04. [Online]. Available: https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/

[41] CrowdStrike, "External technical root cause analysis — channel file 291," Aug. 2024, accessed: 2025-12-04. [Online]. Available: https://www.crowdstrike.com/wp-content/uploads/2024/08/Channel-File-291-Incident-Root-Cause-Analysis-08.06.2024.pdf

[42] Spotify Engineering, "Incident report: Spotify outage on april 16, 2025," May 2025, accessed: 2025-12-04. [Online]. Available: https://engineering.atspotify.com/2025/05/incident-report-spotify-outage-april-16

[43] B. Dettelback, "About the Quay.io outage: Post mortem," Aug. 2020, accessed: 2025-12-04. [Online]. Available: https://www.redhat.com/en/blog/about-the-quay.io-outage-post-mortem

[44] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," in *The Eleventh International Conference on Learning Representations*, 2023. [Online]. Available: https://openreview.net/forum?id=WE_vluYUL-X

[45] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[46] LangChain AI, "LangGraph: Build resilient, stateful multi-agent workflows for LLM applications," 2025, open-source framework for long-running, controllable LLM agents; integrates with LangChain. [Online]. Available: https://github.com/langchain-ai/langgraph

[47] TIOBE Software BV. TIOBE index for October 2025. Monthly updated indicator of the popularity of programming languages. [Online]. Available: https://www.tiobe.com/tiobe-index/

[48] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "SWE-bench: Can language models resolve real-world GitHub issues?" in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=VTF8yNQM66

[49] C. Winston and R. Just, "A taxonomy of failures in tool-augmented LLMs," in *2025 IEEE/ACM International Conference on Automation of Software Test (AST)*, 2025, pp. 125–135. [Online]. Available: https://doi.org/10.1109/AST66626.2025.00019

[50] Q. Xiong, Y. Huang, Z. Jiang, Z. Chang, Y. Zheng, T. Li, and M. Li, "Butterfly effects in toolchains: A comprehensive analysis of failed parameter filling in LLM tool-agent systems," in *Findings of the Association for Computational Linguistics: EMNLP 2025*, C. Christodoulopoulos, T. Chakraborty, C. Rose, and V. Peng, Eds. Suzhou, China: Association for Computational Linguistics, Nov. 2025, pp. 16 712–16 729. [Online]. Available: https://aclanthology.org/2025.findings-emnlp.907/

[51] A. Gola, "Multi needle in a haystack," https://blog.langchain.com/multi-needle-in-a-haystack/, Mar. 2024, accessed: 2025-11-23.

[52] K. Hong, A. Troynikov, and J. Huber, "Context rot: How increasing input tokens impacts LLM performance," Chroma, Tech. Rep., July 2025. [Online]. Available: https://research.trychroma.com/context-rot

[53] Y. Chen, H. Xie, M. Ma, Y. Kang, X. Gao, L. Shi, Y. Cao, X. Gao, H. Fan, M. Wen, J. Zeng, S. Ghosh, X. Zhang, C. Zhang, Q. Lin, S. Rajmohan, D. Zhang, and T. Xu, "Automatic root cause analysis via large language models for cloud incidents," in *Proceedings of the Nineteenth European Conference on Computer Systems*, ser. EuroSys '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 674–688. [Online]. Available: https://doi.org/10.1145/3627703.3629553

[54] D. F. Bacon and P. F. Sweeney, "Fast static analysis of C++ virtual function calls," in *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 324–341. [Online]. Available: https://doi.org/10.1145/236337.236371

[55] D. Grove and C. Chambers, "A framework for call graph construction algorithms," *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 6, p. 685–746, Nov. 2001. [Online]. Available: https://doi.org/10.1145/506315.506316

[56] Red Hat, Inc., "What is CI/CD?" https://www.redhat.com/en/topics/devops/what-is-ci-cd, June 2025, accessed: 2026-02-25.

[57] M. A. Hammer, J. Dunfield, K. Headley, N. Labich, J. S. Foster, M. Hicks, and D. Van Horn, "Incremental computation with names," *SIGPLAN Not.*, vol. 50, no. 10, p. 748–766, Oct. 2015. [Online]. Available: https://doi.org/10.1145/2858965.2814305

[58] A. Souter and L. Pollock, "Incremental call graph reanalysis for object-oriented software maintenance," in *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, 2001, pp. 682–691.

[59] H. Liu, G. Li, J. F. Lukman, J. Li, S. Lu, H. S. Gunawi, and C. Tian, "DCatch: Automatically detecting distributed concurrency bugs in cloud systems," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 677–691. [Online]. Available: https://doi.org/10.1145/3037697.3037735

[60] J. Lu, F. Li, L. Li, and X. Feng, "CloudRaid: Hunting concurrency bugs in the cloud via log-mining," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 3–14. [Online]. Available: https://doi.org/10.1145/3236024.3236071

[61] X. Yuan and J. Yang, "Effective concurrency testing for distributed systems," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1141–1156. [Online]. Available: https://doi.org/10.1145/3373376.3378484

[62] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 1285–1298. [Online]. Available: https://doi.org/10.1145/3133956.3134015

[63] H. Ren, B. Xu, Y. Wang, C. Yi, C. Huang, X. Kou, T. Xing, M. Yang, J. Tong, and Q. Zhang, "Time-series anomaly detection service at Microsoft," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '19. New

York, NY, USA: Association for Computing Machinery, 2019, pp. 3009–3017. [Online]. Available: https://doi.org/10.1145/3292500.3330680

[64] Z. Li, J. Chen, R. Jiao, N. Zhao, Z. Wang, S. Zhang, Y. Wu, L. Jiang, L. Yan, Z. Wang, Z. Chen, W. Zhang, X. Nie, K. Sui, and D. Pei, "Practical root cause localization for microservice systems via trace analysis," in *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*, 2021, pp. 1–10. [Online]. Available: https://doi.org/10.1109/IWQOS52092.2021.9521340

[65] S. Jha, S. Cui, S. S. Banerjee, T. Xu, J. Enos, M. Showerman, Z. T. Kalbarczyk, and R. K. Iyer, "Live forensics for HPC systems: A case study on distributed storage systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020. [Online]. Available: https://doi.org/10.1109/SC41405.2020.00069

[66] Y. Zhang, Z. Guan, H. Qian, L. Xu, H. Liu, Q. Wen, L. Sun, J. Jiang, L. Fan, and M. Ke, "CloudRCA: A root cause analysis framework for cloud computing platforms," in *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, ser. CIKM '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 4373–4382. [Online]. Available: https://doi.org/10.1145/3459637.3481903

[67] Y. Jiang, C. Zhang, S. He, Z. Yang, M. Ma, S. Qin, Y. Kang, Y. Dang, S. Rajmohan, Q. Lin, and D. Zhang, "Xpert: Empowering incident management with query recommendations via large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639081

[68] D. Goel, F. Husain, A. Singh, S. Ghosh, A. Parayil, C. Bansal, X. Zhang, and S. Rajmohan, "X-lifecycle learning for cloud incident management using LLMs," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, pp. 417–428. [Online]. Available: https://doi.org/10.1145/3663529.3663861

[69] P. Jin, S. Zhang, M. Ma, H. Li, Y. Kang, L. Li, Y. Liu, B. Qiao, C. Zhang, P. Zhao, S. He, F. Sarro, Y. Dang, S. Rajmohan, Q. Lin, and D. Zhang, "Assess and summarize: Improve outage understanding with large language models," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1657–1668. [Online]. Available: https://doi.org/10.1145/3611643.3613891

[70] X. Zhang, T. Mittal, C. Bansal, R. Wang, M. Ma, Z. Ren, H. Huang, and S. Rajmohan, "FLASH: A workflow automation agent for diagnosing recurring incidents," Microsoft Research Technical Report (preprint), 2024. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2024/10/FLASH_Paper.pdf

[71] Y. Han, Q. Du, Y. Huang, J. Wu, F. Tian, and C. He, "The potential of one-shot failure root cause analysis: Collaboration of the large language model and small classifier," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 931–943. [Online]. Available: https://doi.org/10.1145/3691620.3695475

[72] C. Pei, Z. Wang, F. Liu, Z. Li, Y. Liu, X. He, R. Kang, T. Zhang, J. Chen, J. Li, G. Xie, and D. Pei, "Flow-of-action: SOP enhanced LLM-based multi-agent system for root cause analysis," in *Companion Proceedings of the ACM on Web Conference 2025*, ser. WWW '25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 422–431. [Online]. Available: https://doi.org/10.1145/3701716.3715225

[73] Y. Bei, W. Zhang, S. Wang, W. Chen, S. Zhou, H. Chen, Y. Li, J. Bu, S. Pan, Y. Yu, I. King, F. Karray, and P. S. Yu, "Graphs meet AI agents: Taxonomy, progress, and future opportunities," 2025. [Online]. Available: https://arxiv.org/abs/2506.18019

[74] B. Jin, C. Xie, J. Zhang, K. K. Roy, Y. Zhang, Z. Li, R. Li, X. Tang, S. Wang, Y. Meng, and J. Han, "Graph chain-of-thought: Augmenting large language models by reasoning on graphs," in *Findings of ACL*, 2024, pp. 163–184. [Online]. Available: https://aclanthology.org/2024.findings-acl.11.pdf

[75] J. Sun, C. Xu, L. Tang, S. Wang, C. Lin, Y. Gong, L. Ni, H.-Y. Shum, and J. Guo, "Think-on-Graph: Deep and responsible reasoning of large language model on knowledge graph," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=nnVO1PvbTv

[76] L. Chen, P. Tong, Z. Jin, Y. Sun, J. Ye, and H. Xiong, "Plan-on-Graph: Self-correcting adaptive planning of large language model on knowledge graphs," in *Proceedings of the 38th International Conference on Neural Information Processing Systems*, ser. NIPS '24. Red Hook, NY, USA: Curran Associates Inc., 2025. [Online]. Available: https://doi.org/10.52202/079017-1189

[77] X. Tan, X. Wang, Q. Liu, X. Xu, X. Yuan, and W. Zhang, "Paths-over-Graph: Knowledge graph empowered large language model reasoning," in *Proceedings of the ACM on Web Conference 2025*, ser. WWW '25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 3505–3522. [Online]. Available: https://doi.org/10.1145/3696410.3714892

[78] Z. Chen, R. Tang, G. Deng, F. Wu, J. Wu, Z. Jiang, V. Prasanna, A. Cohan, and X. Wang, "LocAgent: Graph-guided LLM agents for code localization," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, Eds. Vienna, Austria: Association for Computational Linguistics, Jul. 2025, pp. 8697–8727. [Online]. Available: https://aclanthology.org/2025.acl-long.426/

[79] J. Pu, Y. Li, Z. Chen, J. Liu, Z. Jiang, J. Chen, R. Shi, Z. Zheng, and T. Zhang, "ErrorPrism: Reconstructing error propagation paths in cloud service systems," *arXiv preprint arXiv:2509.26463*, 2025. [Online]. Available: https://arxiv.org/abs/2509.26463

[80] L. Pham, H. Ha, and H. Zhang, "Root cause analysis for microservice system based on causal inference: How far are we?" in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 706–715. [Online]. Available: https://doi.org/10.1145/3691620.3695065