Explanations over Graphs: An Agent Architecture for IT Enterprise Diagnostic Tasks

Saurabh Jha **Rohan R. Arora** **Bhavya Bhavya** **Noah Zheutlin** **Paulina Toro Isaza**
Laura Shwartz **Yu Deng** **Daby Sow** **Ruchir Puri**
IBM Research

Abstract

LLM agents excel when environments are mostly static and their required information fits in a model's context window, but they struggle with IT enterprise diagnostic tasks—such as incident management, IT vulnerability analysis, and FinOps anomaly explanation—where operators iteratively mine massive observability data over a curated resource graph to identify the origins that explain observed symptoms, enabling correct remediation. The data in these domains is structured with inherent hidden dependencies: entities interact, signals co-vary, and the importance of a fact may only become clear after other evidence is discovered. To cope with bounded context windows, agents must summarize intermediate findings before their significance is known, increasing the risk of discarding key evidence. ReAct-style/CodeAct agents are especially brittle in this regime. Their retrieve-summarize-reason loop makes conclusions sensitive to exploration order and introduces run-to-run non-determinism, producing a reliability gap where Pass@k may be high, but Majority@k remains low. Increasing samples or reasoning lengths cannot guarantee stable results because, in high-risk diagnostic scenarios, consistent accuracy is required over sporadic success. Furthermore, the CodeAct paradigm lacks a mechanism to revise its beliefs as new information becomes available. In addition, the CodeAct paradigm entangles semantic reasoning with controller duties such as tool orchestration and state tracking; execution errors and plan drift degrade reasoning while consuming scarce context. We address these issues by formulating the investigation as abductive reasoning over a dependency graph and proposing EoG (Explanations over Graphs), an agentic architecture where an LLM performs bounded local evidence mining and labeling (cause vs symptom) while a deterministic controller manages traversal, state, and belief propagation to compute a minimal explanatory frontier. On representative ITBench diagnostics tasks, EoG improves accuracy and run-to-run consistency over CodeAct baselines, including a 7× average gain in Majority@k F1 score.

1 Introduction

LLM agents excel when environments are mostly static and their required information fits in the context window. They struggle with diagnostic investigations such as incident management, where operators iteratively mine massive observability data to identify the origins that explain observed symptoms, enabling correct remediation. This challenge cuts across IT enterprise diagnostic tasks—incident management, IT vulnerability analysis, FinOps anomaly explanation, and DevOps release-failure forensics. A unifying property is that each is grounded in a curated *resource graph* (e.g., service-interaction topology, cost-attribution, vulnerability/exposure; full taxonomy in Appendix B), so diagnosis is abductive inference along edges of an already-curated graph rather than free-form exploration. It is particularly acute in *Site Reliability Engineering* (SRE), the discipline responsible for keeping large-scale cloud services reliable by detecting, diagnosing, and remediating production incidents. Operators observe symptoms but must identify the minimal upstream sources to

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

remediate; because acting on an incorrect diagnosis carries operational risk, the goal is inference from observational evidence rather than intervention. More formally, this is *abductive reasoning* over massive observability data from an environment with inherent graphical structure composed of entities connected by dependency edges (§2).

Limits of existing agent patterns: Current agentic approaches struggle with abductive reasoning. For example, on the SRE scenarios of ITBench [20][1] (pertaining to incident management), the best CodeAct agent[2] achieves only 13.81% pass@1 recall (§3). Beyond well-known limitations such as tool invocation failures and redundant or degenerate tool calls [5], CodeAct agents struggle significantly with context gathering and action sequencing under uncertainty. Reasoning is only as sound as the evidence it conditions on: if relevant context is not retrieved, it cannot be reasoned over. In open-ended investigations, evidence relevance is not known *a priori*, and CodeAct agents lack mechanisms for systematic context acquisition. Tool calls are issued opportunistically, missing low-frequency, high-impact signals or retrieving excessive irrelevant data. Summarizing the retrieved evidence before reasoning introduces a critical failure mode that discards information before relevance can be established, often losing critical evidence [28].

Furthermore, open-ended investigations exhibit non-monotonic belief revision behaviors: interpretation of entities with the related signals naturally changes as new evidence emerges elsewhere in the system, making reasoning order-dependent. The sequence of evidence retrieval, summarization, and presentation to the LLM materially affects conclusions. In practice, CodeAct agents show high variance across runs due to planning non-determinism, unstable exploration orderings, and lossy summarization, producing fragile, incomplete, irreproducible explanations. These failures are architectural. The CodeAct paradigm implicitly assumes linear, monotonic reasoning over an accumulating context, whereas in environments with explicit dependency structure, abductive reasoning demands structured context acquisition along those dependencies with explicit belief revision.

Towards graph-guided agentic explanations: We introduce an operational-graph abstraction for investigations under partial observability. We represent the environment as an operational graph and define diagnosis as *explanations over such dynamically evolving operational graphs*: explanations correspond to minimal subgraphs consistent with observed evidence. Framing diagnosis in this way converts an ill-defined exploration problem into structured inference. Our motivation is that the environment for these investigations is not unstructured. Cloud systems are explicitly built as graphs of dependencies; e.g., traces record which services call another, container orchestrators (e.g., Kubernetes) record which deployments manage which pods, etc. Failures propagate along these edges because that is how computation flows. For example, a slow database could make the service that calls it slow, which makes the service that calls *that* slow, and so on, until a symptom surfaces at the edge. The diagnostic task is therefore literally one of tracing effects back to causes along a known (or partially known) graph.

Moreover, diagnosis is inherently non-monotonic. As we gather more evidence, beliefs about an entity's role in these failures get revised. Thus, drawing from belief propagation [36], we model diagnosis as an iterative local belief assignment, revision, and propagation process through the graph. This leads to a disaggregated architecture: local reasoning occurs via LLMs over bounded neighborhoods, while a deterministic controller manages traversal order, context gathering, belief bookkeeping, and termination. This separation enables non-monotonic belief revision through explicit recording and updating of beliefs rather than implicit overwriting in growing CodeAct-style prompts. By externalizing planning and context management, the system keeps context bounded, reduces tool-use failures, and produces reproducible, auditable explanations. This separation also addresses production constraints: bounded per-node context scales to large datasets, while deterministic control minimizes LLM calls and API rate limit exposure. Our contributions:

- **EoG agent with separation of concerns:** We develop EoG (Explanations over Graphs), a disaggregated architecture where a *Deterministic Controller* manages graph traversal, state, and tool orchestration, while a stateless *Abductive Policy* π_{abd} classifies each node as origin, symptom, or healthy based on bounded local evidence (§4). This separation eliminates controller failures, ensures auditability, and enables systematic exploration.

[1] ITBench is an open-source benchmark for testing how well AI agents can diagnose and resolve IT issues.

[2] CodeAct [46] agents subsume the ReAct [50] reason–act loop by using executable code as the action space. Our baseline is a CodeAct agent built on the OpenAI Codex coding harness [35]; we use "ReAct" and "CodeAct" interchangeably to refer to it.

- **Semantic Belief Propagation:** We introduce SBP, a message-passing mechanism inspired by belief propagation [36], to address non-monotonic belief revision. When new evidence changes a node's belief, it broadcasts to neighbors, triggering reconsideration by π_{abd}—correcting early misdiagnoses without restart (§4.3).
- **Validation:** On ITBench SRE scenarios [20], EoG achieves 7× higher Majority@k F1 than CodeAct baselines (§5). We have extended this formalism to SWE tasks to diagnose production bugs [10].

2 Problem Formulation

We model the environment as a *resource graph* $G = (V, E)$ where nodes V are *entities* and edges E encode structural dependencies along which effects propagate. Different IT enterprise tasks instantiate G differently (e.g., services-and-calls for incident management, resources-and-attribution for FinOps, CVEs-and-exposure for vulnerability analysis); see Appendix B for the taxonomy. Entities can originate faults, exhibit symptoms, or propagate effects along domain dependency edges. In this work, we focus on abductive reasoning in SRE to diagnose failures from observed anomalies, where V are services, pods, and configurations, and E are API calls or ownership relations.

An *anomaly* is an observation of abnormal behavior at an entity; in SRE, *alerts* are anomalies that operators configure to receive notifications. Given anomalies $\mathcal{O}$ on entities $V_{\mathcal{O}} \subseteq V$, the task is to *explain them*. Formally, we seek:

1. An *Explanatory Subgraph* $G_S = (V_S, E_S)$: $V_S \subseteq V$ is the active set; E_S are directed edges representing inferred propagation: $u \to v$ means "u explains v".
2. A *Frontier* $\mathcal{F} \subseteq V_S$: the minimal origin set whose faults explain all anomalies on $V_{\mathcal{O}}$.

Our focus is explanatory inference from observational evidence. We avoid interventions since remediation actions carry operational risk. This formulation naturally extends to other IT enterprise diagnostic tasks such as FinOps anomaly explanation, IT vulnerability analysis, and DevOps release-failure investigation.

3 Characterizing Failure Modes

We motivate our approach by characterizing failure modes of CodeAct agents [46] built on OpenAI Codex CLI [35], with sandboxed Python and shell, persistent state, and MCP tool extensibility (Appendix C), on the SRE scenarios of ITBench [20].We focus on SRE scenarios as they require abductive reasoning—inferring root causes from observed symptoms.

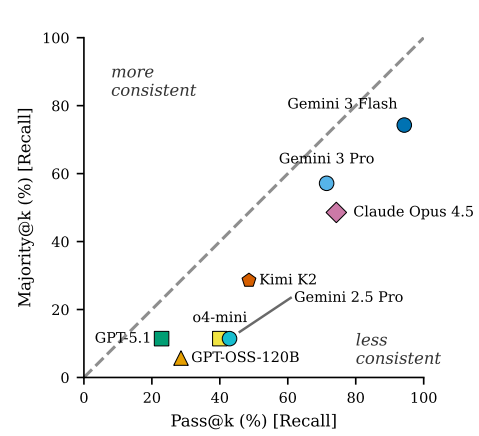
3.1 The Reliability Gap

While most models perform poorly under the CodeAct paradigm, we quantify a more fundamental issue: stochastic success rather than consistent reasoning. We define the *reliability gap* as Pass@k (success in at least one of k runs) minus Majority@k (success in the majority of runs).[3] Figure 1a shows this gap ranging from 14 points for Gemini 3 Flash to 32 points for Gemini 2.5 Pro. Even the best model (Gemini 3 Flash: 89% Pass@k, 74% Majority@k) exhibits significant variance, with only 51% of scenarios yielding consistent correct diagnoses across all runs. This reliability gap must be addressed to ensure successful deployment of agents in enterprise settings [24].

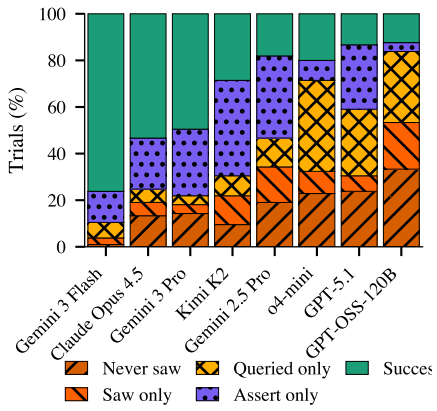
3.2 Exploration Failures

Abductive reasoning reduces to a search over a dependency graph: given observed symptoms, identify the minimal upstream entities whose states explain all observations. The search space is exponential, and the graph constrains tractable exploration. CodeAct agents fail not because they cannot reason, but because they lack structural priors to use the graph.

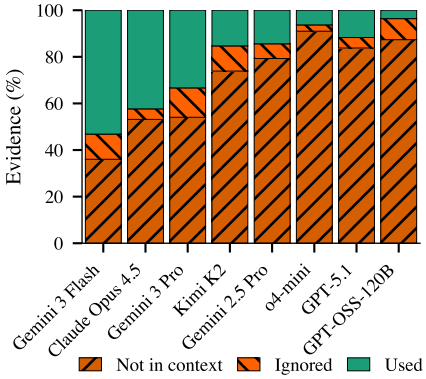
[3]When verification is cheap (e.g., code generation with test suites), one can sample until a correct answer is verified, making Pass@k sufficient for accuracy. However, even then, low Majority@k has consequences for maintainability, latency, and predictability. In incident management, verification requires remediation actions that carry operational risk, so consistent correctness is essential.



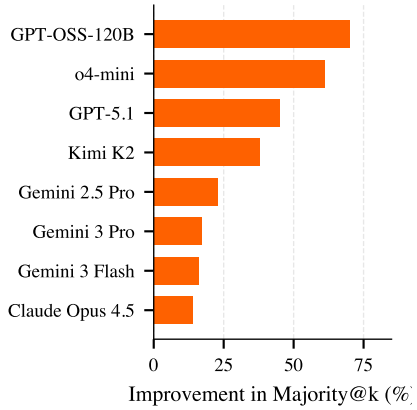
(a) Pass@k vs. Majority@k (reliability gap).



(b) GT discovery funnel.



(c) Evidence discovery failures.



(d) Oracle reordering impact.

Figure 1: CodeAct agent failure modes on ITBench SRE scenarios. (a) Reliability gap: Pass@k (any run succeeds) vs. Majority@k (most runs succeed); deviations from the dotted line indicate stochastic success. (b) Ground-truth discovery funnel: *never saw*, *saw not queried*, *queried not asserted*, *asserted not in output*, *asserted in output*. (c) Evidence discovery: *not in context* (red), *ignored* (orange), *used* (green). (d) Impact of oracle reordering on Majority@k.

Ground truth (GT) entity discovery failures. Figure 1b tracks the GT entity through a discovery funnel. A substantial failure mode is "GT never saw": the GT entity is linked to the symptom via dependency edges, yet the CodeAct agent fails to follow these links.

Evidence discovery failures. Figure 1c shows a high incidence of *Evidence not in context*: supporting data existed but was never fetched, either because the entity was never explored or because what constitutes "relevant" evidence depends on information discovered later (§3.2). The small "evidence ignored" category indicates that modern LLMs can use evidence when it is present; the bottleneck is discovery, not reasoning.

Path dependence. CodeAct agents are highly sensitive to information ordering. In an oracle experiment (Figure 1d), reordering investigation sequences to prioritize paths from symptoms to root causes without adding new information, yields 14–70% improvement in Majority@k across LLMs. This path dependence indicates agents struggle with *non-monotonic reasoning*: early wrong beliefs persist despite emerging evidence.

3.3 Controller Failures

CodeAct agents experience frequent controller failures—plan abandonment, tool repetition, and syntactic errors. We use MAST [42] to characterize these failures on our agent trajectories (Ap-

Algorithm 1: EoG active inference loop. Appendix A formalizes the design rationale under explicit assumptions on π_{abd}.

Initialize: ActiveSet $\leftarrow V_{\mathcal{O}}$; $V_S, E_S, \mathcal{L} \leftarrow \emptyset$; Budget $\leftarrow H_{\max}$
while ActiveSet $\neq \emptyset \wedge$ Budget > 0 **do**
  $v \leftarrow$ ActiveSet.POP(); V_S.add(v)
  $B_v^{old} \leftarrow B_v$ if defined, else $\perp$
  // 1. Fetch Context
  Inbox $\leftarrow$ CollectMessages(v)
  LocalContext $\leftarrow$ CxC.Fetch(v, W)
  Packet $\leftarrow$ (LocalContext, Inbox)
  // 2. Execute Policy
  $(B_v^{new}, \mathcal{P}_v, \mathcal{N}_{next}) \leftarrow \pi_{\text{abd}}$(Packet)
  // 3. Damping (Heuristic)
  if FLIPS(v) $> k_{\text{thresh}}$ **then**
    $B_v^{new}.L \leftarrow$ DEFER
  end if
  // 4. Update Current State
  $\mathcal{L}$.append(v, B_v^{new}); $B_v \leftarrow B_v^{new}$
  $E_S \leftarrow (E_S \setminus \{x \to v : x \in V\}) \cup \mathcal{P}_v$
  for $(u \to v) \in \mathcal{P}_v$ **do**
    if $u \notin V_S$ **then**
      ActiveSet.add(u)
    end if
  end for
  // 5. Propagation
  if $B_v^{new} \neq B_v^{old}$ **then**
    Broadcast($B_v^{new} \to \mathcal{N}(v)$)
    ActiveSet.add($\mathcal{N}(v)$)
  end if
  for $u \in \mathcal{N}_{next}$ **do** ▷ 6. Propose (optional)
    if $u \notin V_S$ **then**
      ActiveSet.add(u)
    end if
  end for
  Budget —
end while
$\mathcal{F} \leftarrow \{v \in V_S : L_v = \text{ORIGIN} \wedge \nexists u \in V_S, L_u = \text{ORIGIN}, u \rightsquigarrow_{E_S} v\}$ ▷ Frontier
Report $\leftarrow \pi_{\text{finalize}}(G_S, \mathcal{F}, \mathcal{L})$

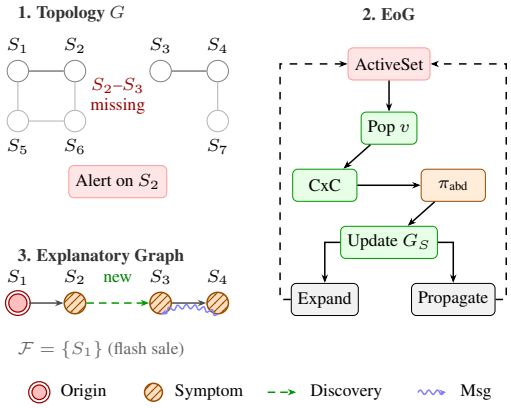


Figure 2: EoG agent architecture, whose inputs are a topology and an alert, and which produces the explanatory graph.

pendix E). The key implication is the separation of deterministic control flow (tool orchestration, state management) from non-deterministic reasoning (causal inference).

These failure modes reinforce one another. Controller brittleness consumes context that could otherwise support broader exploration. Incomplete exploration leads to missed evidence. Path-dependent reasoning prevents recovery when initial assumptions prove incorrect.

4 Methodology

We present **EoG (Explanations over Graphs)**, an agent that traverses the dependency graph G to identify the *frontier* $\mathcal{F}$: the minimal set of origin nodes whose faults explain all observed anomalies (§2). EoG builds an *Explanatory Graph* G_S where each visited node carries a *belief*: its inferred explanatory role (ORIGIN, SYMPTOM, or HEALTHY) and supporting evidence. The agent addresses the limitations of the CodeAct paradigm (§3) through two design principles:

1. **Separation of decision and control.** A *Deterministic Controller* handles graph traversal, state management, and tool orchestration, while a stateless *Abductive Policy* π_{abd} performs local inference over raw evidence (e.g., logs, metrics retrieved via tools) and neighbor messages for each node $v \in V_S$. This separation eliminates controller failures (§3.3), ensures auditability, and enforces systematic graph exploration.
2. **Belief revision via Semantic Belief Propagation (SBP).** Initial beliefs may be incorrect due to incomplete information. For instance, a node with strong failure symptoms (e.g., out-of-memory) may be a victim of upstream problems (e.g., increased load). When new evidence changes a node's belief, it broadcasts to its neighbors, triggering re-evaluation. This message-passing mechanism, inspired by belief propagation [36], addresses path dependence (§3.2), by correcting early misdiagnoses as evidence propagates, without restarting the investigation.

4.1 Architecture Overview

The architecture comprises three modules (Figure 2):

The Deterministic Controller. A symbolic engine that maintains all persistent state: the **Explanatory Graph** $G_S = (V_S, E_S)$ with node beliefs, the **Global Ledger** $\mathcal{L}$ (immutable history of reasoning steps), the **ActiveSet** (nodes pending investigation), and **message queues**, which execute graph traversal, message routing, and termination logic without semantic reasoning. This ensures **safety and reproducibility** (the agent cannot execute arbitrary code; it can only request data via the Context Contract) and **operational efficiency** (bookkeeping is LLM-free, bounding cost to policy invocations).

The Context Contract (CxC). A strict interface that bounds and focuses the information provided to the policy, drawing on principles of active context management [26, 51]. The CxC is implemented as an MCP tool [2] with a simple interface `get_context(entity_id, start_time, end_time)`, abstracting away domain-specific details. When investigating node v, the CxC returns local evidence on that entity (e.g., logs, metrics, traces in IT systems) and topological context (immediate neighbors). The Controller bundles this with the Inbox to form the **Context Packet**.

Relevance Filtering: Domain experts can equip the CxC with specialized tools (e.g., "fetch top-k log anomalies") to surface the most relevant signal without context overflow.

The Abductive Policy (π_{abd}). A stateless, LLM-parameterized policy that performs local abductive inference. Given a bounded Context Packet for a node v, π_{abd} produces: (i) a local belief B_v, (ii) propagation claims $\mathcal{P}_v$, and (iii) a set of candidate entities $\mathcal{N}_{next}$ for further investigation. While the Controller governs execution, π_{abd} determines the epistemic trajectory of the search. We use "policy" in the decision-theoretic sense: a mapping from local observations to beliefs.

4.2 State Formalism

EoG maintains the following state variables:

1. The Global Ledger ($\mathcal{L}$): an append-only log of (v, B_v, t) tuples recording which node was evaluated, its belief, and when—a scratchpad [34] for replay, debugging, and auditability.

2. Local Belief (B_v): stored per node in G_S, updated after each policy execution. $B_v = (L_v, \mathcal{E}_v)$ where:

- $L_v \in \{\text{HEALTHY}, \text{ORIGIN}, \text{SYMPTOM}, \text{DEFER}\}$ is the explanatory label:
  - HEALTHY: Node is operating normally within the incident window.
  - ORIGIN: Node has a citable change (config, deployment, resource) explaining the incident.
  - SYMPTOM: Node is degraded, but evidence points to an upstream cause.
  - DEFER: Evidence is inconclusive—either absent or unrecognized by the policy.
- $\mathcal{E}_v$ is the evidence summary.

3. Messages ($m_{u \to v}$): stored in per-node Inboxes. A message contains the sender's belief B_u at the time of broadcast. When node v is evaluated, the policy reads its Inbox to reason about neighbor states, exploiting the Local Markov Property [36].

4.3 Algorithm

Algorithm 1 presents the EoG agent, interleaving exploration (discovery of new nodes) with propagation (belief refinement). The Controller bootstraps the investigation by setting the time window W (incident start/end), seeding the ActiveSet with alerting entities $V_{\mathcal{O}}$, initializing $V_S, E_S, \mathcal{L} \leftarrow \emptyset$, and capping the search at a budget of investigation hops. Here E_S denotes the current explanatory graph: when v is re-evaluated, step 4 replaces the incoming explanatory claims for v with $\mathcal{P}_v$, while the append-only ledger $\mathcal{L}$ preserves the revision history. The core loop processes nodes in the ActiveSet until convergence (no new messages) or budget exhaustion. The loop terminates when no more messages are pending (fixed point) or the budget is exhausted. The Frontier $\mathcal{F}$ is computed *after* convergence, and a finalize policy π_{finalize} synthesizes findings into a structured report (Appendix D.7). Additional details on correctness properties, termination guarantees, and handling of inconclusive results are provided in Appendix D.10.

4.4 Example scenario

Consider a service chain: S_1 (frontend) $\to$ S_2 (gateway) $\to$ S_3 (processor) $\to$ S_4 (database). Crucially, **the edge $S_2 \to S_3$ is not in the topology graph**.S_2 calls S_3 dynamically based on request type, and

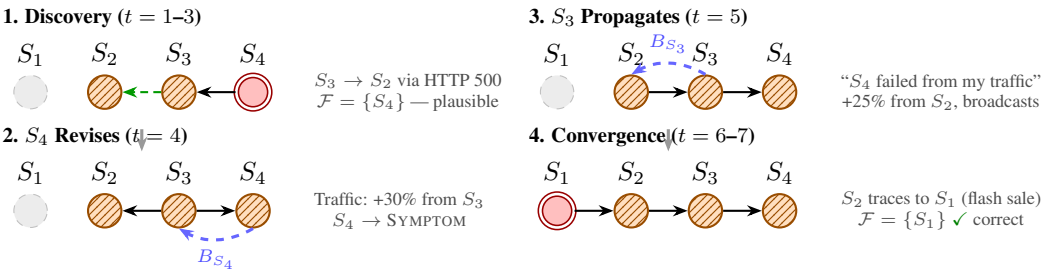


Figure 3: Edge discovery and belief revision sequence. (1) $S_2 \rightarrow S_3$ discovered. (2) S_4 revises belief due to high load. (3) S_3 propagates message. (4) S_1 identified as root cause.

this dependency was never registered. During a flash sale, traffic increases. Each hop amplifies the load slightly (retry storms, fan-out), and S_4, finally exhausts memory. The failure propagates back: $S_4 \rightarrow S_3 \rightarrow S_2$. An alert fires on S_2 (users see 500 errors).

The diagnostic trap. S_4's Out Of Memory (OOM) error is *strong local evidence* of being the origin. Without belief messages, the algorithm would terminate with S_4 as origin, leading operators to restart S_4 (without addressing the upstream traffic pattern). This example shows both **edge discovery** (finding the hidden $S_2 \rightarrow S_3$ dependency) and **belief revision** (correcting the S_4 misdiagnosis).

1. $t = 1$**, Process** S_2 *(Alg. Steps 1–2, 4, 6)*: Alert node. CxC returns HTTP 500 errors showing failed calls to S_3. **Edge discovery**: $S_2 \rightarrow S_3$ was not in the topology graph, but traces reveal this dependency. The policy adds edge $S_3 \rightarrow S_2$ to E_S and proposes S_3. $B_{S_2} =$ SYMPTOM of S_3.
2. $t = 2$**, Process** S_3: Logs show timeouts calling S_4. Proposes S_4. $B_{S_3} =$ SYMPTOM of S_4.
3. $t = 3$**, Process** S_4: Logs show `OutOfMemoryError`. **Strong local evidence of being the origin** points to memory exhaustion, that is a classic root cause.
   At this point: $\mathcal{F} = \{S_4\}$. Single origin, plausible diagnosis. Without belief messages, **this would be the final step of the process** pointing to S_4's memory issue.
4. $t = 4$**,** S_4 **re-enters ActiveSet** *(Alg. Step 5: Propagation)*: CxC now includes traffic metrics (not just error logs). Policy sees: incoming request rate from S_3 is elevated (+30% above baseline). Re-reasons: "My OOM was caused by traffic overload, not internal failure." $B_{S_4} \rightarrow$ SYMPTOM of S_3. **Broadcasts to** S_3**.**
5. $t = 5$**,** S_3 **receives the message** *(Alg. Step 5: Propagation continues)*: Inbox contains B_{S_4}: "I failed from *your* traffic." S_3 checks its own incoming traffic: +25% from S_2. Propagates message upstream. **Broadcasts to** S_2**.**
6. $t = 6$**,** S_2 **receives the message**: Inbox contains B_{S_3}: "The cascade started from *your* traffic." S_2 checks incoming: +20% from S_1. Proposes S_1.
7. $t = 7$**, Process** S_1: Logs show flash sale triggered traffic spike. $B_{S_1} =$ ORIGIN.
8. **Convergence** *(Alg. Post-processing)*: $\mathcal{F} = \{S_1\}$—correct. The flash sale is the true origin.

Key insight. This example demonstrates: (1) **structure learning**: the hidden from topology graph $S_2 \rightarrow S_3$ edge was discovered from observability data [16]; and (2) **belief revision**: S_4's plausible OOM diagnosis was corrected via cascading messages. Without SBP, operators would restart S_4, providing temporary relief until memory fills again. The upstream traffic pattern is never addressed.

5 Implementation & Results

Implementation. We implement EoG and a CodeAct baseline on the Codex agent framework [35] (~5,000 lines of Rust) and evaluate on ITBench abductive-reasoning tasks (§3). The CodeAct baseline incorporates best practices from prior LLM-based diagnostics work [37], exceeding published ITBench baselines; details in Appendices C and D.

We implement domain-specific MCP servers exposing the operational graph and evidence data (e.g., logs, traces, and metrics in ITBench), including distributed traces [23]; the tool set is summarized in Appendix C.4 (Table 6).

For each entity v, the Context Packet satisfies the Local Markov Property: it contains entity metadata, 1-hop neighbors, time-windowed observability data, and the current Inbox. Large contexts (>80K tokens) are chunked with overlap and classified via map-reduce. Domain expertise is encoded in

Table 1: EoG vs CodeAct on ITBench. Each cell shows a 95% Wilson confidence interval as p^{+u}_{-l} (point estimate with asymmetric upper/lower offsets in percentage points). Each of the n=35 scenarios contributes a single binary outcome per cell — a run passes an F1 column if F1$\geq$0.5 and an R column if it recalls the ground-truth (GT) root-cause entity; a scenario *passes* pass@3 if at least one of its 3 runs passes, and majority@3 if at least two do. Every scenario has a single GT entity, so Recall$\in\{0,1\}$ and the R columns are invariant to the recall threshold — so the relevant sample size for each CI is n=35 Bernoulli outcomes. Per-model token usage is reported separately in Appendix D.9.

		RC Entity ↑				RC Reas. ↑
Agent	**Model**	**pass@3 F1**	**pass@3 R**	**maj@3 F1**	**maj@3 R**	**maj@3 R**
CodeAct	GPT-5.1	$22.9^{+16.1}_{-10.8}$	$22.9^{+16.1}_{-10.8}$	$8.6^{+13.8}_{-5.6}$	$11.4^{+14.6}_{-6.9}$	$11.4^{+14.6}_{-6.9}$
EoG	GPT-5.1	$\mathbf{88.9^{+6.6}_{-14.9}}$	$\mathbf{91.4^{+5.6}_{-13.8}}$	$\mathbf{86.1^{+7.6}_{-15.5}}$	$\mathbf{88.6^{+6.9}_{-14.6}}$	$\mathbf{83.0^{+8.9}_{-15.7}}$
CodeAct	GPT-OSS-120B	$28.6^{+16.5}_{-12.3}$	$28.6^{+16.5}_{-12.3}$	$5.7^{+12.9}_{-4.1}$	$5.7^{+12.9}_{-4.1}$	$5.7^{+12.9}_{-4.1}$
EoG	GPT-OSS-120B	$\mathbf{84.1^{+7.8}_{-16.8}}$	$\mathbf{94.3^{+4.1}_{-12.9}}$	$\mathbf{79.4^{+10.6}_{-15.3}}$	$\mathbf{88.6^{+6.9}_{-14.6}}$	$\mathbf{85.0^{+8.7}_{-14.4}}$
CodeAct	Gemini 3 Flash	$88.6^{+6.9}_{-14.6}$	$94.3^{+4.1}_{-12.9}$	$74.3^{+11.5}_{-16.4}$	$74.3^{+11.5}_{-16.4}$	$74.3^{+11.5}_{-16.4}$
EoG	**Gemini 3 Flash**	$\mathbf{94.5^{+3.9}_{-13.1}}$	$\mathbf{97.1^{+2.4}_{-11.6}}$	$\mathbf{92.1^{+4.9}_{-14.5}}$	$\mathbf{94.3^{+4.1}_{-12.9}}$	$\mathbf{90.2^{+6.8}_{-12.6}}$
CodeAct	Kimi K2	$48.6^{+15.8}_{-15.6}$	$48.6^{+15.8}_{-15.6}$	$28.6^{+16.5}_{-12.3}$	$28.6^{+16.5}_{-12.3}$	$25.7^{+16.4}_{-11.5}$
EoG	Kimi K2	$\mathbf{85.3^{+8.4}_{-14.7}}$	$\mathbf{91.4^{+5.6}_{-13.8}}$	$\mathbf{83.0^{+8.9}_{-15.7}}$	$\mathbf{88.6^{+6.9}_{-14.6}}$	$\mathbf{81.2^{+8.8}_{-17.1}}$
EoG	Minimax M2.1	$72.9^{+12.9}_{-15.0}$	$88.6^{+6.9}_{-14.6}$	$64.2^{+12.6}_{-17.9}$	$77.1^{+10.8}_{-16.1}$	$68.0^{+13.4}_{-16.0}$
EoG	Mistral Large	$76.6^{+11.3}_{-15.6}$	$91.4^{+5.6}_{-13.8}$	$67.2^{+14.2}_{-15.2}$	$80.0^{+10.0}_{-15.9}$	$76.6^{+11.3}_{-15.6}$

CodeAct baselines for Minimax M2.1 and Mistral Large are omitted: the Codex framework failed with these models.

Table 2: CodeAct prompted with EoG algorithm (Recall). Cells show 95% Wilson CIs as p^{+u}_{-l} over n=35 scenario-level Bernoulli outcomes.

Model	**pass@3↑**	**maj@3↑**
GPT-5.1	$40.0^{+16.4}_{-14.4}$	$2.9^{+11.6}_{-2.4}$
Gemini 3 Flash	$71.4^{+12.3}_{-16.5}$	0.0*
Kimi K2	$42.9^{+16.2}_{-14.9}$	0.0*

CodeAct prompted with EoG algorithm instructions. Compare with EoG in Table 1 (e.g., GPT-5.1: 91.4/88.6 vs. 40.0/2.9 here). * Only 1 trial/scenario was available for these configurations; maj@3 is structurally unreachable, so the value is a definitional 0 rather than a Bernoulli observation, and a frequentist CI does not apply.

Table 3: SBP ablation (maj@3 Recall, RC Entity). EoG-SBP cells show 95% Wilson CIs as p^{+u}_{-l} over n=35 scenarios.

Model	**EoG-SBP**	**ΔSBP↑**
GPT-5.1	$82.9^{+9.0}_{-15.6}$	+6.9%
GPT-OSS-120B	$74.3^{+11.5}_{-16.4}$	+19.2%
GPT-OSS-20B	$40.0^{+16.4}_{-14.4}$	**+64.3%**
Gemini 3 Flash	$85.7^{+8.0}_{-15.1}$	+10.0%
Kimi K2	$77.1^{+10.8}_{-16.1}$	+14.9%
Minimax M2.1	$60.0^{+14.4}_{-16.4}$	+28.5%
Mistral Large	$71.4^{+12.3}_{-16.5}$	+12.0%

EoG-SBP = Deterministic Controller only. ΔSBP = % relative improvement with SBP.

structured prompts organized by phase (bootstrap, explore, finalize), with the core algorithm logic remaining domain-agnostic.

Experiments. 35 ITBench scenarios spanning misconfigurations, resource exhaustion, and cascading failures, are used for evaluation, with 3 runs per scenario to measure both accuracy and consistency. All experiments use high reasoning effort settings. Notably, CodeAct has access to *more* tools (all MCP tools plus shell utilities and Python code execution) while EoG uses only the Context Contract per entity, and controls the investigation structure algorithmically.

We compute two metrics: **RC Entity** (correctness of the identified root-cause entity, e.g., `otel-demo/Pod/cartservice-xyz`) and **RC Reasoning** (correctness of the causal mechanism, e.g., "Pod crashed due to incorrect OS architecture"); see Appendix G. All 35 scenarios have a single GT root-cause entity (in the 3 multi-fault scenarios, all faults map to the same entity); Recall is therefore binary, and RC Entity R reports whether the GT entity was identified, while F1 additionally penalizes false-positive entities. Both metrics are scored by an LLM judge; against SRE expert's labels, judge agreement is 90.6% (Cohen's κ=0.808) when every self-inconsistent judgment counts as a disagreement, and 98.8% (κ=0.975) under majority vote over 5 judge runs (SRE scenarios only; Appendix G).

EoG vs CodeAct. Table 1 compares EoG against CodeAct baselines. The results directly address the *reliability gap* identified in §3.1: while CodeAct exhibits a large variance between pass@3 and majority@3 (indicating stochastic success), EoG substantially narrows this gap. For example, GPT-5.1 under CodeAct achieves only 22.9% pass@3 and 8.6% majority@3 F1 (RC Entity); under EoG, these rise to 88.9% and 86.1% respectively, near-closing the reliability gap. Against GraphRAG [14]

given the same dependency graph and Context Contract (Gemini 3 Flash, all three query modes), the strongest mode (drift) trails EoG on every metric: RC Entity maj@3 F1 60.0 vs. 92.1 (1.53×) and RC Reasoning maj@3 R 80.0 vs. 90.2 (1.13×); local and global modes reach at most 37.1 (Appendix F.1). The gain is thus attributable to belief revision and controlled traversal, not to the graph interface.

Ablating distributed traces showed no statistically significant change, consistent with the redundancy of observability data—the same evidence is typically inferable from logs and metrics.

Prompting Ablation. We quantify a CodeAct agent prompted to follow the EoG algorithm (§4.3) without the external Deterministic Controller. Table 2 shows majority@3 collapses to near zero despite non-trivial pass@3: the cognitive load of jointly managing exploration state, message passing, and causal reasoning overwhelms instruction-following.

SBP Ablation. Table 3 isolates the contribution of SBP: structured traversal alone captures most gains over CodeAct, while SBP adds 7–64% via post-hoc belief correction (§3.2), with the largest gains on weaker models where early misclassifications are common. At n=35, marginal Wilson intervals on EoG-SBP overlap the EoG cells in Table 1 (e.g., GPT-5.1: $82.9^{+9.0}_{-15.6}$ vs. $86.1^{+7.6}_{-15.5}$)—expected for marginal intervals on a small benchmark since they ignore within-scenario pairing; the signal is the same-direction ΔSBP across all seven models.

Paired Belief-Revision Analysis. To test whether SBP changes decisions in the right cases, we instrument every classification decision (GPT-OSS-120B) and score the same runs with and without belief revision. SBP revises a label in 28% of runs and *fixes half of the scenarios that fail without it while breaking none that succeed*. Of revisions to the GT root cause, 72% move it toward ORIGIN, 85% of these from SYMPTOM—an entity that first reads as a victim of its own blast radius until neighbor attribution reverses the direction. The other 28% never lose the answer: message passing re-promotes the entity, or the origin relocates upstream along the causal chain (e.g., from `Service/shipping` to the `Deployment` whose spec change caused the failure), and the frontier retains it (Appendix F.3).

Generalization. Two further domains, run independently by teams applying EoG in their own settings, compare against the same Codex CodeAct baseline; only the graph and Context Contract change:

(a) **Code bugs** (K8s.af real-world failure reports, n=16, Claude Sonnet 4.5; graph and Context Contract: hammock blocks [22]): EoG improves majority@3 Recall by 20.5 to 88% (95% Wilson CI [64.0, 96.5]); the pass@3–majority@3 gap collapses from ≈29 to ≈5 points.
(b) **FinOps anomaly explanation** (ITBench, n=14, Gemini 3.1 Flash; resource graph defined with domain experts in under one person-week, Context Contract: resource manifest and 7-day spend): EoG improves majority@3 Recall by 29.6 to 98.5%; the gap collapses from ≈15 to 0 points.

Both reproduce the SRE signature—majority@3 rises and the reliability gap closes—but at n=16 and n=14 we make no per-domain significance claim; we treat them as directional confirmation of the SRE effect.

Robustness. We perturb the graph and observability on all 35 SRE scenarios (GPT-OSS-120B; Appendix F.2). Random removal of causal edges degrades consistency, not discovery: dropping 10% is within noise (-3.5% relative maj@3 Recall), dropping 50% costs -10.7% and doubles the reliability gap, and severing every edge incident to the root-cause node costs only -1.5%, because the policy re-discovers missing edges from evidence. Adding 30% spurious edges leaves recall unchanged (at up to 2× tokens).

Cost. EoG consumes 2.4–7.6× more input tokens than CodeAct for three of four models (0.63× for Gemini 3 Flash) but 1.8–10.6× fewer output tokens. Priced per correct diagnosis (3×cost per trial/maj@3 F1, OpenRouter rates), EoG is cheaper on every shared backbone: 4.6× on GPT-5.1, 2.0× on GPT-OSS-120B and Gemini 3 Flash, and 1.15× on Kimi K2 (Appendix D.9).

Table 4: Comparison of LLM/agent baselines.

	CodeAct [46]	Graph-RAG [14]	RCA [37]	STRATUS [7]	GALA [40]	**EoG (Ours)**
Graph as mutable state	✗	○	○	○	○	✓
Hypotheses explicit	✗	✗	○	○	○	✓
Belief revision	✗	✗	✗	✗	✗	✓
Propagation	glob	g.sum	glob	glob	hyb	msg
Controller invariants	✗	✗	○	○	○	✓
Data dynamicity	epis	epis	str	str	str	str

✓=yes, ✗=no, ○=partial. epis=episodic, str=streaming.
Propagation: how updates spread—glob=entire context passed each step, g.sum=global with summarization, hyb=mix of global/local, msg=local message passing.

6 Related Work

Our work relates to LLM-based investigative agents performing multi-step reasoning over structured environments [50, 3, 37, 48, 6, 40, 38, 31]. Recent work introduces belief-centric abstractions [27, 25, 17], but treats beliefs as prompt-level artifacts rather than mutable state. These systems assume monotonic updates and lack explicit mechanisms for belief revision in presence of contradictory evidence. Causal discovery and effect estimation [44, 43, 49] are related but target different outputs under static, fully-observed data and do not model revisable hypotheses under streaming evidence.

Graph-RAG systems [14, 18, 30, 33, 9, 4, 38, 31, 21] enable multi-hop reasoning over knowledge graphs. Theorem-of-Thought [1] uses Bayesian belief propagation to aggregate parallel reasoning attempts; EoG instead uses message passing to correct sequential mistakes as new evidence emerges.

Diagnostic agents for cloud operations [37, 48, 52, 12, 7, 40, 32] generate explanatory hypotheses from partial evidence. Graph-based systems [45, 41, 53, 39, 29, 47] use dependency graphs to localize faults. All these rely on *append-only* inference without explicit revision or localized propagation, making them brittle under evolving evidence in unbounded graphs.

Table 4 summarizes the comparison of EOG with representative CodeAct, Graph-RAG, and diagnostic agents, and ablate revision and propagation, demonstrating that gains arise from state semantics rather than domain-specific engineering.

7 Conclusion

Diagnostic investigations over massive, heterogeneous data with hidden dependency structure require consistent correctness; stochastic success is insufficient when verification carries risk. We introduced EoG, a disaggregated architecture that separates local abductive reasoning via LLMs from deterministic graph traversal and belief propagation, achieving 7× higher Majority@k F1 on ITBench scenarios. This separation of concerns generalizes across IT enterprise diagnostic tasks—incident management, IT vulnerability analysis, and FinOps anomaly explanation—where explanations must be derived from environments with inherent graphical structure.

8 Limitations and Broader Impact

The main limitation is that empirical validation is restricted to the 35 SRE incident scenarios in ITBench [20]; we do not claim the reported gains transfer without adaptation to other IT enterprise diagnostic tasks (FinOps anomaly explanation, IT vulnerability analysis, DevOps), which would typically require re-tuning the abductive policy, adapting the context contract, and supplying a task-specific resource-graph schema. More broadly, we do not advocate for deployment of such agents in production settings; autonomous diagnostic systems acting on safety-critical infrastructure carry risks that warrant careful consideration.

Acknowledgments and Disclosure of Funding

We thank the IBM Instana team—Ameet Rahane, Ragu Kattinakere, Dishant Kaushik, Neel Bhavsar, Shivangi Pathak, Nevil Kandathil Sintho, Adwan Syed, and Marc Palaci-Olgun—for building a commercial solution in Instana and providing feedback to the team. We thank Ruchi Mahindru and

Nick Fuller for their guidance and support, and the entire ITBench team for building the benchmark that enabled this work.

References

[1] Samir Abdaljalil, Hasan Kurban, Khalid Qaraqe, and Erchin Serpedin. Theorem-of-thought: A multi-agent framework for abductive, deductive, and inductive reasoning in language models. *arXiv preprint arXiv:2506.07106*, 2025.

[2] Anthropic. Introducing the model context protocol, 2024. URL `https://www.anthropic.com/news/model-context-protocol`. Accessed: 2026-05-07.

[3] Yuanchen Bei, Weizhi Zhang, Siwen Wang, Weizhi Chen, Sheng Zhou, Hao Chen, Yong Li, Jiajun Bu, Shirui Pan, Yizhou Yu, et al. Graphs meet ai agents: Taxonomy, progress, and future opportunities. *arXiv preprint arXiv:2506.18019*, 2025.

[4] Maciej Besta, Lorenzo Paleari, Jia Hao Andrea Jiang, Robert Gerstenberger, You Wu, JĂłn Gunnar Hannesson, Patrick Iff, Ales Kubicek, Piotr Nyczyk, Diana Khimey, et al. Affordable ai assistants with knowledge graph of thoughts. *arXiv preprint arXiv:2504.02670*, 2025.

[5] Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent llm systems fail?, 2025. URL `https://arxiv.org/abs/2503.13657`.

[6] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, et al. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 674–688, 2024.

[7] Yinfang Chen, Jiaqi Pan, Jackson Clark, Yiming Su, Noah Zheutlin, Bhavya Bhavya, Rohan Arora, Yu Deng, Saurabh Jha, and Tianyin Xu. Stratus: A multi-agent system for autonomous reliability engineering of modern clouds. *arXiv preprint arXiv:2506.02009*, 2025.

[8] Zehui Chen, Weihua Du, Wenwei Zhang, Kuikun Liu, Jiangning Liu, Miao Zheng, Jingming Zhuo, Songyang Zhang, Dahua Lin, Kai Chen, and Feng Zhao. T-eval: Evaluating the tool utilization capability of large language models step by step, 2024. URL `https://arxiv.org/abs/2312.14033`.

[9] Erica Coppolillo. Injecting knowledge graphs into large language models, 2025. URL `https://arxiv.org/abs/2505.07554`.

[10] Shengkun Cui, Rahul Krishna, Saurabh Jha, and Ravishankar K Iyer. Praxis: Integrating program analysis with observability for root-cause analysis. In *2026 56th Annual IEEE International Conference on Dependable Systems and Networks (DSN)*, pages 52–66. IEEE, 2026.

[11] Datadog. Datadog infrastructure and apm. `https://www.datadoghq.com`, 2025. Cloud monitoring platform with service map discovery.

[12] Darshan Deshpande, Varun Gangal, Hersh Mehta, Jitin Krishnan, Anand Kannappan, and Rebecca Qian. Trail: Trace reasoning and agentic issue localization, 2025. URL `https://arxiv.org/abs/2505.08638`.

[13] Dynatrace. Dynatrace software intelligence platform. `https://www.dynatrace.com`, 2025. AI-powered observability with automatic dependency mapping.

[14] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*, 2024.

[15] Shubham Gandhi, Jason Tsay, Jatin Ganhotra, Kiran Kate, and Yara Rizk. When agents go astray: Course-correcting swe agents with prms, 2025. URL `https://arxiv.org/abs/2509.02360`.

[16] Alex Havrilla, David Alvarez-Melis, and Nicolo Fusi. Igda: Interactive graph discovery through large language model agents. *arXiv preprint arXiv:2502.17189*, 2025.

[17] Guoxiu He, Meicong Zhang, Tiancheng Su, Li Ma, and Xiaomin Zhu. Enhancing belief consistency of large language model agents in decision-making process based on attribution theory. *Expert Systems with Applications*, page 129273, 2025.

[18] Xiaoxin He, Yijun Tian, Yifei Sun, Nitesh V Chawla, Thomas Laurent, Yann LeCun, Xavier Bresson, and Bryan Hooi. G-retriever: Retrieval-augmented generation for textual graph understanding and question answering. *arXiv preprint arXiv:2402.07630*, 2024.

[19] IBM. Ibm instana observability. `https://www.ibm.com/products/instana`, 2025. Enterprise observability platform with automatic service topology discovery.

[20] Saurabh Jha, Rohan R Arora, Yuji Watanabe, Takumi Yanagawa, Yinfang Chen, Jackson Clark, Bhavya Bhavya, Mudit Verma, Harshit Kumar, Hirokuni Kitahara, et al. Itbench: Evaluating ai agents across diverse real-world it automation tasks. In *Forty-second International Conference on Machine Learning*, 2025.

[21] Jinhao Jiang, Kun Zhou, Zican Dong, Keming Ye, Wayne Xin Zhao, and Ji-Rong Wen. Structgpt: A general framework for large language model to reason over structured data. *arXiv preprint arXiv:2305.09645*, 2023.

[22] Richard Johnson, David Pearson, and Keshav 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 (PLDI)*, pages 171–185, 1994.

[23] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th symposium on operating systems principles*, pages 34–50, 2017.

[24] NVJK Kartik, Garvit Sapra, Rishav Hada, and Nikhil Pareek. Agentcompass: Towards reliable evaluation of agentic workflows in production, 2025. URL `https://arxiv.org/abs/2509.14647`.

[25] Minsoo Kim, Jongyoon Kim, Jihyuk Kim, and Seung-won Hwang. Qube: Question-based belief enhancement for agentic llm reasoning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 21403–21423, 2024.

[26] Mo Li, LH Xu, Qitai Tan, Ting Cao, and Yunxin Liu. Sculptor: Empowering llms with cognitive agency via active context management. *arXiv preprint arXiv:2508.04664*, 2025.

[27] Aly Lidayan, Jakob Bjorner, Satvik Golechha, Kartik Goyal, and Alane Suhr. Abbel: Llm agents acting through belief bottlenecks expressed in language. *arXiv preprint arXiv:2512.20111*, 2025.

[28] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024. doi: 10.1162/tacl_a_00638. URL `https://aclanthology.org/2024.tacl-1.9/`.

[29] Wei Liu, Chao Peng, Pengfei Gao, Aofan Liu, Wei Zhang, Haiyan Zhao, and Zhi Jin. Graphlocator: Graph-guided causal reasoning for issue localization. *arXiv preprint arXiv:2512.22469*, 2025.

[30] Zichen Liu et al. Polyg: Adaptive graph traversal for diverse graphrag questions. *arXiv preprint arXiv:2504.02112*, 2025.

[31] Linhao Luo, Yuan-Fang Li, Gholamreza Haffari, and Shirui Pan. Reasoning on graphs: Faithful and interpretable large language model reasoning. *arXiv preprint arXiv:2310.01061*, 2024.

[32] Xuyan Ma, Xiaofei Xie, Yawen Wang, Junjie Wang, Boyu Wu, Mingyang Li, and Qing Wang. Diagnosing failure root causes in platform-orchestrated agentic systems: Dataset, taxonomy, and benchmark, 2025. URL `https://arxiv.org/abs/2509.23735`.

[33] Vasilije Marković, Lazar Obradović, Laszlo Hajdu, and Jovan Pavlović. Optimizing the interface between knowledge graphs and llms for complex reasoning, 2025. URL https://arxiv.org/abs/2505.24478.

[34] Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

[35] OpenAI. Codex cli: Lightweight coding agent for the terminal. https://github.com/openai/codex, 2025. Version 0.76.

[36] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[37] Devjeet Roy, Xuchao Zhang, Rashi Bhave, Chetan Bansal, Pedro Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. Exploring llm-based agents for root cause analysis. In *Companion proceedings of the 32nd ACM international conference on the foundations of software engineering*, pages 208–219, 2024.

[38] Jiashuo Sun, Chengjin Xu, Lumingyuan Tang, Saizhuo Wang, Chen Lin, Yeyun Gong, Heung-Yeung Shum, and Jian Guo. Think-on-graph: Deep and responsible reasoning of large language model on knowledge graph. *arXiv preprint arXiv:2307.07697*, 2024.

[39] Lun Tang, Enqiao Kou, Weili Wang, and Qianbin Chen. A root cause analysis framework for iot based on dynamic causal graphs assisted by llms. *IEEE Internet of Things Journal*, 2025.

[40] Yifang Tian, Yaming Liu, Zichun Chong, Zihang Huang, and Hans-Arno Jacobsen. Gala: Can graph-augmented large language model agentic workflows elevate root cause analysis? *arXiv preprint arXiv:2508.12472*, 2025.

[41] Andrea Tonon, Meng Zhang, Bora Caglayan, Fei Shen, Tong Gui, MingXue Wang, and Rong Zhou. Radice: Causal graph based root cause analysis for system performance diagnostic. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 465–475. IEEE, 2025.

[42] UC Berkeley. Mast: Multi-agent system trajectories analysis tool. https://ucb-mast.notion.site/, 2025. Tool for analyzing agentic system failure modes.

[43] Matthew J Vowels, Necati Cihan Camgoz, and Richard Bowden. D'ya like dags? a survey on structure learning and causal discovery. *ACM Computing Surveys*, 55(4):1–36, 2022.

[44] Guangya Wan, Yunsheng Lu, Yuqi Wu, Mengxuan Hu, and Sheng Li. Large language models for causal discovery: Current landscape and future directions. *arXiv preprint arXiv:2402.11068*, 2024.

[45] Hanzhang Wang, Zhengkai Wu, Huai Jiang, Yichao Huang, Jiamu Wang, Selcuk Kopru, and Tao Xie. Groot: An event-graph-based approach for root cause analysis in industrial settings. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 419–429. IEEE, 2021.

[46] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024.

[47] Yong Xiang, Charley Peter Chen, Liyi Zeng, Wei Yin, Xin Liu, Hu Li, and Wei Xu. Simplifying root cause analysis in kubernetes with stategraph and llm. *arXiv preprint arXiv:2506.02490*, 2025.

[48] Junjielong Xu, Qinan Zhang, Zhiqing Zhong, Shilin He, Chaoyun Zhang, Qingwei Lin, Dan Pei, Pinjia He, Dongmei Zhang, and Qi Zhang. Openrca: Can large language models locate the root cause of software failures? In *The Thirteenth International Conference on Learning Representations*, 2025.

[49] Liuyi Yao, Zhixuan Chu, Sheng Li, Yaliang Li, Jing Gao, and Aidong Zhang. A survey on causal inference. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 15(5):1–46, 2021.

[50] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=WE_vluYUL-X`.

[51] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618*, 2025.

[52] Xiao Zhang, Qi Wang, Mingyi Li, Yuan Yuan, Mengbai Xiao, Fuzhen Zhuang, and Dongxiao Yu. Tamo: Fine-grained root cause analysis via tool-assisted llm agent with multi-modality observation data in cloud-native systems. *arXiv preprint arXiv:2504.20462*, 2025.

[53] Yuhan Zhu, Jian Wang, Bing Li, Xuxian Tang, Hao Li, Neng Zhang, and Yuqi Zhao. Root cause localization for microservice systems in cloud-edge collaborative environments. *arXiv preprint arXiv:2406.13604*, 2024.

A Formalization of EoG

The paper's thesis is that Semantic Belief Propagation (SBP) unlocks behaviour that diagnosis without it cannot achieve. This appendix formalises that thesis in two results. After fixing notation and the frontier definition (§A.1), §A.2 establishes a deterministic separation between EoG-with-SBP and the corresponding SBP-free variants of Algorithm 1 (step 5 fully disabled, and auto-enqueue retained without broadcast) on a constructed family of cascade instances, under the same locality+sufficient-context policy class for both sides; §A.3 bounds EoG's probability of returning the correct frontier in terms of the policy's per-call accuracy, and locates the regime in which EoG provably dominates the SBP-free upper bound.

We distinguish *controller-level conditions* (C.x), enforceable by the implementation, from *oracle-level assumptions* (B.x), which constrain the LLM's behaviour and are not derived here. The results therefore support the design rationale—why SBP and a deterministic controller are needed—rather than certify the correctness of the deployed LLM-based system, whose policy is not guaranteed to satisfy (B.x).

A.1 Setup and Frontier Definition

Graphs. A diagnostic instance specifies a finite directed *topology graph* $G = (V, E)$ whose vertices represent system entities. EoG maintains during execution an *explanatory subgraph* $G_S = (V_S, E_S)$ with $V_S \subseteq V$, where edges in E_S are the policy's current best causal explanations. The cascade defines a *true explanatory subgraph* $G^\star = (V^\star, E^\star)$. Edges follow the cause-to-effect convention: $u \to v$ means u explains v. Vertices in $V \setminus V^\star$ have ground-truth role HEALTHY. We write $u \rightsquigarrow_{E_S} v$ for the existence of a *positive-length* directed path from u to v in E_S (so $v \rightsquigarrow_{E_S} v$ holds only if E_S contains a directed cycle through v); intermediate-vertex labels do not affect this relation.

Controller state. For each $v \in V_S$, the controller maintains: a label $L_v \in \{\text{ORIGIN}, \text{SYMPTOM}, \text{HEALTHY}, \text{DEFER}\}$, an evidence set $\mathcal{Y}_v$ accumulated by the Context Contract over a window W, and an inbox Inbox_v of broadcast objects from neighbours, retained per-sender by Last-Write-Wins (LWW). Globally: an ActiveSet of pending vertices and a Global Ledger logging each invocation. The Context Packet at v is $C_v = (\mathcal{Y}_v, \mathrm{Inbox}_v)$.

Policy. The abductive policy π_{abd} is a *stochastic* mapping from context packets to outputs $O_v = (B_v, \mathcal{P}_v, \mathcal{N}_{\mathrm{next},v})$: a broadcast object $B_v = (L_v, \mathcal{E}_v)$, a set of in-edges $\mathcal{P}_v \subseteq V \times \{v\}$ proposed for E_S, and a set of vertices proposed for the ActiveSet. We write $O_v \sim \pi_{\mathrm{abd}}(C_v)$ and treat the LLM as a black-box conditional distribution; the policy's per-call accuracy is bounded by Assumption 1 below, not assumed deterministic. The well-formedness $\mathcal{P}_v \subseteq V \times \{v\}$ says each invocation at v

proposes only v's own causes — a single-writer-per-edge invariant: each edge $u \to w$ is written only at the invocation at w.

Controller-level conditions.

(C.1) Append-only evidence. $\mathcal{Y}_v(t) \subseteq \mathcal{Y}_v(t')$ whenever $t \le t'$.
(C.2) LWW inbox. Inbox_v retains the most recent broadcast from each sender that has ever broadcast to v.
(C.3) Per-target edge replacement. At each invocation at v, $E_S(t^+) = (E_S(t^-) \setminus \{x \to v : x \in V\}) \cup \mathcal{P}_v$. The vertex set V_S is monotone non-decreasing.
(C.4) Idempotent CxC. Evidence retrieval is a deterministic function of (v, W).
(C.5) Fair scheduling. Every vertex enqueued in the ActiveSet is eventually popped.

Frontier. The frontier of G_S is the set of ORIGIN-labelled vertices with no ORIGIN ancestor in E_S:

$$\mathcal{F}(G_S) \;=\; \big\{\, v \in V_S : L_v = \textsc{Origin},\ \nexists\, u \in V_S \text{ with } L_u = \textsc{Origin} \text{ and } u \rightsquigarrow_{E_S} v \,\big\}. \qquad (1)$$

The ground-truth frontier $\mathcal{F}^\star$ is the analogous set in $G^\star$. For $\mathcal{F} = \mathcal{F}^\star$, the labels of G_S must be correct on the cascade and the origin-to-origin reachability in E_S must match that in $E^\star$. Under a single-origin cascade (Assumption (B.v) below), the second condition is vacuous since at most one ORIGIN exists.

Operational completion. For each $v \in V$ and time t, $\mathrm{Complete}^{\mathrm{op}}(v, t)$ holds when v's ancestor configuration in $G^\star$ is recoverable from $C_v(t)$. For HEALTHY vertices the ancestor set is empty and completion is immediate from first visit; for cascade vertices, completion requires upstream broadcasts to have arrived. The mechanism by which broadcasts come to encode upstream summaries is Assumption (B.iii) below.

Termination. Algorithm 1 halts in at most $H_{\max}$ policy invocations, since the loop decrements a budget initialised to $H_{\max}$ once per iteration and exits when the budget reaches zero or the ActiveSet empties. Under a per-node visit cap $k_{\max}$ (§D.3), this tightens to $N_{\mathrm{EoG}} \le k_{\max} \cdot |V_S|$.

A.2 Why SBP Is Necessary

When EoG visits v for the first time, C_v contains $\mathcal{Y}_v$ alone — the inbox is empty of upstream summaries because none of v's ancestors has been visited yet. The policy's output is shaped only by what v can see locally, which may not be enough to identify v's role. As ancestors are processed and broadcast, the inbox fills, and the natural label may flip from a first-visit guess to a post-context assignment driven by upstream evidence. *SBP is the system waiting for the evidence it needs*, not correcting a miscalibrated inference.

The flash-sale walkthrough in §4.4 illustrates the mechanism. At $t = 3$, S_4's first-visit packet contains the local OOM evidence alone and the policy commits $L_{S_4} = \textsc{Origin}$ (the wrong label given the cascade). At $t = 4$, broadcasts from S_3 summarising S_2, S_1 arrive; the packet becomes operationally complete and the policy updates L_{S_4} to SYMPTOM. With SBP, completion drives the frontier to $\{S_1\}$.

A procedure that takes the policy's first-visit commitment and never revisits it cannot exploit this transition. We make the gap precise on a constructed family, comparing Algorithm 1 with full SBP against the same algorithm with SBP disabled (step 5 entirely no-op).

The family $\mathcal{F}_n$. For $n \ge 2$ and each $k \in \{1, \dots, n\}$, instance $\mathcal{I}_k$ has chain topology $S_1, S_2, \dots, S_n$, alert $V_{\mathcal{O}} = \{S_n\}$, hidden origin $h^\star = S_k$, and per-vertex observables $\mathcal{Y}_v = (\sigma_v, t_v)$ with $\sigma_v = \mathbb{1}[v \ge k]$ and $t_v = v - k$ for $v \ge k$ (else $\perp$). The cascade flows $S_k \to S_{k+1} \to \cdots \to S_n$; in the cause-to-effect convention $V^\star = \{S_k, \dots, S_n\}$, $E^\star = \{S_{v-1} \to S_v : k < v \le n\}$, and $\mathcal{F}^\star = \{S_k\}$. CxC returns $\mathcal{Y}_v$ as the local context.

Routing convention for Algorithm 1. Algorithm 1, step 5, broadcasts and re-enqueues to a neighbourhood $\mathcal{N}(v)$ that we take as

$$\mathcal{N}(v) \;:=\; \mathcal{N}_G(v) \cup \mathcal{N}_{G_S}(v),$$

the union of the (undirected) topology-graph neighbourhood $\mathcal{N}_G(v) = \{u : (u, v) \in E \text{ or } (v, u) \in E\}$ and the (undirected) explanatory-subgraph neighbourhood $\mathcal{N}_{G_S}(v) = \{u \in V_S : (u, v) \in E_S \text{ or } (v, u) \in E_S\}$. This supports topology-driven discovery while E_S is empty and edge-driven propagation once E_S has been built up. In the chain, $\mathcal{N}_G(S_v) = \{S_{v-1}, S_{v+1}\}$ for interior v.

Behavioural assumptions on π_{abd}. Three properties of the LLM policy when invoked on packets from $\mathcal{F}_n$. We use m to denote the scalar evidence summary that broadcasts carry; (BC) below gives its concrete recursive form, and (LL)/(SC) refer to it. *Notation:* in (LL)/(SC) below we display the policy output as $(L_v, \mathcal{P}_v, \mathcal{N}_{\mathrm{next},v})$ — the broadcast object B_v's evidence component $\mathcal{E}_v = m_v$ is supplied by (BC) and is omitted from the (LL)/(SC) tuples for readability.

(LL) Locality on insufficient context. Given $C_v = (\mathcal{Y}_v, \mathrm{Inbox}_v)$:
- If $\sigma_v = 0$: $\pi_{\mathrm{abd}}(C_v) = (\textsc{Healthy}, \emptyset, \emptyset)$ regardless of Inbox_v.
- If $\sigma_v = 1$, $t_v > 0$, and no broadcast in Inbox_v carries $m = 0$: $\pi_{\mathrm{abd}}(C_v) = (\textsc{Origin}, \emptyset, \emptyset)$.

(SC) Sufficient-context correctness. :
- If $\sigma_v = 1$, $t_v = 0$ (regardless of Inbox_v): $\pi_{\mathrm{abd}}(C_v) = (\textsc{Origin}, \emptyset, \emptyset)$.
- If $\sigma_v = 1$, $t_v > 0$, and some broadcast in Inbox_v carries $m = 0$: $\pi_{\mathrm{abd}}(C_v) = (\textsc{Symptom}, \{S_{v-1} \to S_v\}, \{S_{v-1}\})$.

(BC) Broadcast content. On $\mathcal{F}_n$, set $\mathcal{E}_v := m_v$, where m_v is propagated recursively along inbox chains:

$$m_v \;:=\; \min\Big(\{t_v\} \cup \{\, m_u : B_u \in \mathrm{Inbox}_v \text{ at invocation} \,\}\Big),$$

with $\min$ taken over values $\neq \perp$. So $B_v = (L_v, m_v)$ on this family, and $m_v = 0$ iff some chain of broadcasts in the current inbox state leads back to S_k.

(LL) and (SC) cover all $(\sigma_v, t_v, \text{has-min-0})$ combinations on $\mathcal{F}_n$ exhaustively. (LL) captures the LLM's behaviour on insufficient evidence: given a smoking-gun-like local observable and no corroborating inbox content, the policy commits to the locally-implied label and proposes no further exploration. (SC) captures correct behaviour on operationally complete packets. (BC) instantiates (B.iii)(a) for $\mathcal{F}_n$.

Theorem 1 (SBP separation)**.** *For $k \sim \mathrm{Unif}\{1, \ldots, n\}$:*

(i) *Algorithm 1 with SBP disabled (step 5 no-op), run with π_{abd} satisfying (LL) and (SC)(1), produces $\mathcal{F}^{\mathrm{term}} = \{S_n\}$ deterministically. Hence $\Pr_{k\sim\mathrm{Unif}}[\mathcal{F}^{\mathrm{term}} = \mathcal{F}^\star] = 1/n$.*

(ii) *Algorithm 1 with full SBP, under (C.1)–(C.5), the routing convention above, and an idealised regime ($H_{\max}$ non-binding, no premature damping), run with π_{abd} satisfying (LL), (SC), (BC), reaches $\mathcal{F}^{\mathrm{term}} = \{S_k\} = \mathcal{F}^\star$ in finitely many iterations under any fair schedule.*

Proof. Proof of (i). Initial state: $\mathrm{ActiveSet} = \{S_n\}$, $V_S = E_S = \emptyset$. Pop S_n (line 14). Step 1 (line 17): $\mathrm{Inbox} = \emptyset$ (no broadcasts have ever been emitted). Step 2 (line 21): $\pi_{\mathrm{abd}}(\mathcal{Y}_{S_n}, \emptyset)$. If $k < n$, (LL)'s second clause applies ($\sigma_n = 1$, $t_n > 0$, no min-0 in empty inbox): output $(\textsc{Origin}, \emptyset, \emptyset)$. If $k = n$, (SC)'s first clause applies ($t_n = 0$): same output. Step 4 (line 28): E_S's in-edges of S_n become $\emptyset$; the for-loop over $\mathcal{P}_{S_n} = \emptyset$ adds nothing. Step 5: no-op (SBP disabled). Step 6 (line 38): for-loop over $\mathcal{N}_{\mathrm{next},S_n} = \emptyset$ adds nothing. ActiveSet is empty; loop exits.

Terminal: $V_S = \{S_n\}$, $E_S = \emptyset$, $L_{S_n} = \textsc{Origin}$. The frontier formula (1): S_n has no E_S-ancestor (in-edges empty), so $S_n \in \mathcal{F}$. $\mathcal{F}^{\mathrm{term}} = \{S_n\}$, equal to $\mathcal{F}^\star = \{S_k\}$ iff $k = n$. Under uniform k, $\Pr = 1/n$.

Proof of (ii). Assume $k < n$ (case $k = n$: (SC)(1) labels $S_n = S_k$ correctly Origin; broadcast enables exploration of healthy S_{n-1} via $\mathcal{N}_G$; $\mathcal{F} = \{S_n\}$ trivially). We establish six lemmas, all under the idealised regime ($H_{\max}$ non-binding, no premature damping) and (C.5) (fair scheduling).

Lemma 1 (Eventual visit). *Every $v \in \{S_n, S_{n-1}, \ldots, S_k\} \cup \{S_{k-1} : k > 1\}$ is eventually popped.*
Proof. $S_n \in V_{\mathcal{O}}$, so eventually popped. On its first pop, by (LL) or (SC), L_{S_n} changes from $\perp$ to a concrete label, so step 5 fires and enqueues $\mathcal{N}_G(S_n) = \{S_{n-1}\}$ (since $\mathcal{N}_{G_S}(S_n) = \emptyset$ initially). Inductively, each S_{n-j}'s first pop changes its label from $\perp$ and enqueues $\mathcal{N}_G(S_{n-j}) \supseteq \{S_{n-j-1}\}$ when $n - j > 1$. By induction down to $j = n - k$, S_k is reached, and if $k > 1$, the same step also enqueues S_{k-1}. Re-pops can fire step 5 again if m_v decreases (a new lower m_u arrives via the inbox), but the *label* component is a function of the input class $(\sigma_v, t_v, \text{has-min-0 in inbox})$, and (LL)/(SC) give the same label whenever this class is unchanged — so labels do not flip on repops alone. Step 5

firings induced by m-only updates enqueue neighbours but cannot create new vertices outside the chain.

Lemma 2 (No min-0 broadcast before S_k is popped). *Let T_k be the time of S_k's first pop. For all $t < T_k$ and all $v \in V$, no broadcast in $\mathrm{Inbox}_v(t)$ carries $m = 0$. Proof.* By (BC), $m_u = 0$ requires either $t_u = 0$ (so $u = S_k$, the only zero-t vertex) or some $B_w \in \mathrm{Inbox}_u$ with $m_w = 0$. The latter unfolds inductively along the broadcast chain into a vertex with $t_\cdot = 0$ that has already broadcast — again S_k. Before T_k, S_k has not been popped and so has issued no broadcast, so no $m = 0$ value exists anywhere.

Lemma 3 (Discovery labels). *Before T_k, every cascade vertex $v > k$ that is popped is labelled* ORIGIN *via (LL)(2). At T_k, S_k is labelled* ORIGIN *via (SC)(1) and broadcasts $m = 0$. Proof.* For $v > k$ popped at $t < T_k$: $\sigma_v = 1$, $t_v > 0$, no $m = 0$ in Inbox_v by Lemma 2. (LL)(2) applies. At S_k: $t_k = 0$, (SC)(1) applies. (BC) gives $m_{S_k} = \min\{t_k, \ldots\} = 0$.

Lemma 4 (Healthy stays healthy). *Every $v < k$ is eventually popped, labelled* HEALTHY*, and never relabelled. Proof.* By Lemma 1, S_{k-1} is enqueued; on its first pop, $\sigma_{k-1} = 0$ activates (LL)(1) and the label flips $\perp \to$ HEALTHY, so step 5 fires and enqueues $\mathcal{N}_G(S_{k-1}) \ni S_{k-2}$. Iterating, every $v < k$ is eventually popped. (LL)(1) ignores the inbox, so re-pops keep $L_v =$ HEALTHY.

Lemma 5 (First propagation step). *After T_k, S_{k+1} is eventually re-popped with a min-0 broadcast in its inbox, and revises to* SYMPTOM *with $\mathcal{P}_{S_{k+1}} = \{S_k \to S_{k+1}\}$. Proof.* At T_k, step 5 fires (belief change $\perp \to$ ORIGIN): broadcast emitted to $\mathcal{N}_G(S_k) \ni S_{k+1}$, S_{k+1} enqueued. By (C.2) (LWW inbox), B_{S_k} persists in $\mathrm{Inbox}_{S_{k+1}}$; by (BC), every broadcast S_k ever emits carries $m_{S_k} = 0$ (since $t_{S_k} = 0$ is in the minimand), so any LWW overwrite from S_k also has $m = 0$. (C.5) ensures S_{k+1} is eventually popped. At that pop: $\sigma_{k+1} = 1$, $t_{k+1} = 1 > 0$, inbox carries $m = 0$. (SC)(2) applies. Step 4 sets S_{k+1}'s in-edges in E_S to $\{S_k \to S_{k+1}\}$ via (C.3). Belief change ORIGIN $\to$ SYMPTOM triggers step 5: S_{k+1}'s broadcast carries $m_{S_{k+1}} = \min\{t_{k+1}, m_{S_k}\} = 0$ by (BC).

Lemma 6 (Inductive propagation). *For $r = 1, \ldots, n-k$, S_{k+r} is eventually relabelled* SYMPTOM *with $\mathcal{P}_{S_{k+r}} = \{S_{k+r-1} \to S_{k+r}\}$. Proof.* Induction on r. Base case $r = 1$: Lemma 5. Inductive step: assume S_{k+r} has been revised, broadcasts $m = 0$ to $\mathcal{N}(S_{k+r}) \ni S_{k+r+1}$, and enqueues S_{k+r+1}. By the same argument as Lemma 5 with indices shifted, S_{k+r+1} is eventually popped with min-0 in inbox and revised to SYMPTOM.

Termination and frontier. After $r = n - k$ inductive steps, S_n has been revised. All vertices have stable labels: $L_{S_k} =$ ORIGIN, $L_v =$ SYMPTOM for $v \in \{S_{k+1}, \ldots, S_n\}$, $L_v =$ HEALTHY for $v < k$. By (BC), m_v takes integer values in $[0, n] \cup \{\perp\}$ and is monotone non-increasing per vertex (the recursive $\min$ only decreases as new m_u's arrive); so m_v changes at most $n + 1$ times per vertex, and once m_v is stable everywhere, B_v is stable, step 5 stops firing, and ActiveSet drains.

The frontier formula (1): the only ORIGIN-labelled vertex in V_S is S_k. The exclusion clause requires some $u \in V_S$ with $L_u =$ ORIGIN and $u \rightsquigarrow_{E_S} S_k$. The only candidate u is S_k itself, but $S_k \rightsquigarrow_{E_S} S_k$ requires a directed cycle in E_S, and $E_S \subseteq \{S_v \to S_{v+1} : k \le v < n\}$ is acyclic. So the exclusion is vacuous, $S_k \in \mathcal{F}$, and no other vertex is. $\mathcal{F}^{\mathrm{term}} = \{S_k\} = \mathcal{F}^\star$. Total pops are bounded: each vertex experiences at most 2 label flips ($\perp \to$ ORIGIN/HEALTHY, then optionally ORIGIN $\to$ SYMPTOM) and at most $n + 1$ m_v updates, so step 5 fires $O(n)$ times per vertex, giving $O(n|V|)$ ActiveSet additions in total. □

The $1/n$ rate in (i) is the best any policy in the (LL)+(SC)(1) class can achieve when SBP is disabled: although $\mathcal{Y}_v$ encodes $t_v = v - k$ (so S_n's packet does in principle carry k), (LL)(2) forces the policy to commit to ORIGIN for every $t_v > 0$ value, collapsing the t_v-channel. The information distinguishing instances reaches S_n *usefully* only via the inbox routing that SBP provides — as part (ii) shows, full SBP transports S_k's broadcast back to S_n via $S_{n-1}, S_{n-2}, \ldots$, at which point (SC)(2) fires and the wrong commitment is rewritten.

Remark 1 (Exploration alone is not enough)**.** *The same $1/n$ rate holds for an exploration-keeping baseline that retains step 5's auto-enqueue but disables broadcast: every S_v is eventually popped, but inboxes stay empty, so (LL)(2) labels every S_v with $v \ge k$ as* ORIGIN *and proposes no parents, giving $E_S = \emptyset$ and $\mathcal{F} = \{S_k, \ldots, S_n\}$, which matches $\mathcal{F}^\star = \{S_k\}$ only at $k = n$. SBP — broadcast carrying m together with the (SC)(2) trigger that consumes it — is what drives the frontier from $\{S_k, \ldots, S_n\}$ down to $\{S_k\}$, not exploration of the chain.*

A.3 Probabilistic Correctness

Theorem 1 used a deterministic oracle policy. A real LLM policy is approximately calibrated; we now bound EoG's probability of correctness in terms of that calibration. Three things have to go right: EoG must explore enough of the cascade, broadcasts must carry enough information for downstream completion, and the policy must label complete packets correctly with high probability. We make all three precise as assumptions, then bound the probability that the realised frontier matches ground truth via a union bound.

Assumptions.

(B.i) Idealised regime. (C.5) holds, $H_{\max}$ is non-binding, and damping does not absorb any cascade vertex before its packet is operationally complete.
(B.ii) Discovery surfaces $G^\star$. At termination, $V^\star \subseteq V_S^\star$ and $E^\star \subseteq E_S^\star$. Each true explanatory edge $u \to w \in E^\star$ is present in E_S before any forwarding broadcast emitted by u that should traverse it (*dynamic edge availability*).
(B.iii) Broadcasts are summary-carrying and monotone. (a) When $u \in V^\star$ broadcasts, B_u encodes a summary statistic of u's context that, with u's inbox at broadcast time, suffices for downstream operational completion. (b) On any invocation at u where the inbox has been augmented by a new upstream summary, the resulting evidence component differs from the previous one and triggers a fresh broadcast. (c) The sequence of broadcasts u emits is monotone in the upstream-information order: any later broadcast carries at least as much upstream information as any earlier one, so LWW retention is non-erasing and operational completion at downstream vertices, once achieved, persists.
(B.iv) Broadcast routing. Step 5 broadcasts (and re-enqueues) to $\mathcal{N}(v) := \mathcal{N}_G(v) \cup \mathcal{N}_{G_S}(v)$, the union of the topology-graph neighbourhood and the explanatory-subgraph neighbourhood. This supports both topology-driven discovery while E_S is empty and edge-driven propagation once E_S is built up.
(B.v) Single Origin. There is exactly one ground-truth ORIGIN vertex $h^\star$.

Assumption 1 (Two-rate calibration)**.** *There exist $p, q \in (0, 1]$ such that for every realised terminal $V_S^\star$ arising with positive probability:*

- *For every $v \in V^\star$ and every C_v on which $\mathrm{Complete}^{\mathrm{op}}(v)$ holds:*

$$\Pr\big[\, L_v = L_v^\star \wedge \mathcal{P}_v \cap (V^\star \times \{v\}) = E^\star \cap (V^\star \times \{v\}) \,\big|\, V_S^\star,\, C_v \textit{ complete} \,\big] \;\geq\; p.$$

- *For every $v \in V_S^\star \setminus V^\star$:* $\Pr[L_v = \textsc{Healthy} \mid V_S^\star] \geq q$.

The first event is what frontier minimality needs (correct labels and in-edges on the cascade); the second prevents healthy false positives. Conditioning on $V_S^\star$ makes the calibration robust to the policy's own exploration history — the visited set is itself a function of the policy's earlier outputs, so unconditional calibration could fail on the trajectories that determine which $V_S^\star$ is realised.

Theorem 2 (Probabilistic correctness)**.** *Under (B.i)–(B.v) and Assumption 1, for any realised terminal $V_S^\star$ on which EoG halts via $\mathrm{ActiveSet} = \emptyset$:*

$$\Pr\big[\, \mathcal{F} = \mathcal{F}^\star \,\big|\, V_S^\star \,\big] \;\geq\; 1 \;-\; |V^\star|\,(1-p) \;-\; |V_S^\star \setminus V^\star|\,(1-q).$$

Proof sketch. Bridge from controller to policy. Since EoG halted via $\mathrm{ActiveSet} = \emptyset$, Inbox_v does not grow after v's last pop: any later broadcast arriving at v would, by step 5's auto-enqueue, re-add v to ActiveSet and contradict $\mathrm{ActiveSet} = \emptyset$ at termination. Combined with (C.1) (append-only $\mathcal{Y}_v$, hence stable post-halt) and (B.iii)(c) (prior LWW overwrites at v were non-erasing), v's terminal output is a realisation of π_{abd} at its last invocation on the terminal packet C_v^{term}.

Operational completion on $V^\star$. For $v \in V^\star$, every ancestor in $G^\star$ lies in V_S at termination by (B.ii)(a). Each ancestor's broadcast carries summary information sufficient for downstream completion (B.iii)(a) and triggers on the arrival of a new upstream summary (B.iii)(b); the broadcast traverses the relevant true edge in E_S once available (B.ii)(b)) and is routed to v by (B.iv). By induction on the longest ancestor-to-v directed path in $G^\star$, a recoverable summary of v's ancestor configuration reaches v in finitely many iterations under any fair schedule. For $v \in V_S^\star \setminus V^\star$, the ancestor set is empty and completion is immediate.

Per-vertex bound and union. Apply Assumption 1 to each invocation under operational completion. Let A_v ($v \in V^\star$) and H_v ($v \in V_S^\star \setminus V^\star$) denote the per-vertex success events. When all A_v and all

H_v hold simultaneously, by (B.v) only $h^\star$ carries the ORIGIN label in $V_S^\star$ and the frontier formula reduces to $\{h^\star\} = \mathcal{F}^\star$ regardless of intermediate-vertex labels on paths in $E_S^\star$. The conditional union bound gives the stated inequality. □

Remark 2 (Multi-origin extension)**.** *Without (B.v), the bound extends to $|\mathcal{F}^\star| \geq 2$ under a no-spurious-origin-path condition stated over the* full *terminal $E_S^\star$: for every pair u, v of vertices labelled* ORIGIN *at termination, $u \rightsquigarrow_{E_S^\star} v$ implies $u \rightsquigarrow_{E^\star} v$. The full-graph form is needed because a path Origin → Healthy → Origin in $E_S^\star$ would still place an Origin ancestor at the second Origin and remove it from the frontier — intermediate-vertex labels do not block reachability.*

When does EoG dominate the SBP-free baseline? By Theorem 1(i)'s mechanics, the SBP-free variant terminates with $V_S = \{S_n\}$, so its frontier can match $\mathcal{F}^\star$ only when $k = n$. Independent of the policy, the SBP-free success rate on $\mathcal{F}_n$ under uniform k is therefore at most $1/n$. To prove EoG strictly dominates this upper bound, it suffices that EoG's lower bound from Theorem 2 exceeds $1/n$:

$$|V^\star|\,(1-p) \;+\; |V_S^\star \setminus V^\star|\,(1-q) \;<\; 1 - 1/n. \tag{2}$$

Equation (2) is permissive across most of $(p, q) \in (0, 1]^2$: at moderate calibration ($p = q = 0.9$, $n = 10$, $|V^\star| = |V_S^\star \setminus V^\star| = 1$), the LHS is 0.2 and the RHS is 0.9. The empirical $7\times$ Majority@k F1 result on ITBench (§5) operates well inside this region.

B Resource Graphs in IT Enterprise Diagnostic Tasks

Each IT enterprise diagnostic task is grounded in one or more curated resource graphs that operators maintain as part of normal operations. Table 5 summarizes the dominant graphs, their entities and edges, and the diagnostic tasks they support. EoG is agnostic to which graph G is supplied: the Context Contract (§4) abstracts the underlying graph, so the same Deterministic Controller and Abductive Policy operate over any of them.

Table 5: Resource graphs underlying IT enterprise diagnostic tasks.

Resource graph	**Nodes**	**Edges**	**Diagnostic task**
Infrastructure topology	hosts, VMs, containers, network segments	runs-on, routes-via	Incident management, capacity
Service-interaction topology	services, pods, APIs	calls, depends-on (traces)	Incident management, SLO regressions
Control-plane ownership	Deployments, ReplicaSets, Pods, ConfigMaps	manages, mounts (Kubernetes)	Release / config incidents
Cost-attribution graph	resources, owners, products, accounts	billed-to, tagged-with	FinOps anomaly explanation
Vulnerability / exposure graph	CVEs, components, hosts, network paths	affects, reachable-from	IT vulnerability analysis
Identity & access graph	principals, roles, permissions, resources	grants, assumes	CISO investigations, breach blast-radius
CI/CD release graph	commits, builds, artifacts, environments	produced-by, deployed-to	DevOps release-failure forensics

C CodeAct Agent Implementation Details

We provide implementation details for the CodeAct baseline agent [46] used in our evaluation. The baseline leverages OpenAI Codex CLI (v0.76) [35], an open-source coding agent that runs locally and supports extensible tool integration via the Model Context Protocol (MCP).

C.1 OpenAI Codex CLI Architecture

Codex CLI implements a **disaggregated architecture** where the LLM reasoning loop is separated from tool execution [35]. The core is written in Rust (`codex-rs`) and provides:

- **Sandboxed Execution**: Shell and code execution occur in isolated subprocesses with configurable permission levels (read-only, write workspace, full access). The sandbox restricts filesystem writes, network access, and system modifications by default.

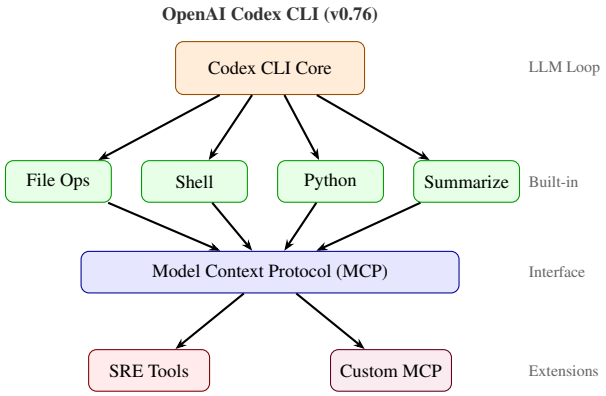


Figure 4: OpenAI Codex CLI architecture. The core agent loop interfaces with built-in tools through the Model Context Protocol. Domain-specific MCP servers extend capabilities.

- **MCP Tool Servers**: Tools run as separate processes communicating via the Model Context Protocol. This separation enables: (1) crash isolation—a failing tool does not crash the agent; (2) resource limits—each tool server can be independently constrained; (3) extensibility—domain tools are added without modifying the core.
- **Automatic Context Summarization**: When tool outputs exceed context limits, Codex summarizes content to preserve important information while staying within token bounds.
- **State Persistence**: Python execution maintains state across invocations; file operations are transactional within the workspace.

C.2 CodeAct Agent Prompt

The CodeAct baseline operates as a Site Reliability Engineering (SRE) support agent designed for offline incident analysis. The agent follows a structured investigation workflow with explicit phases for context discovery, symptom analysis, hypothesis generation, evidence collection, and causal chain construction. The full system prompt is provided below, with environment variables (`$SNAPSHOT_DIRS`, `$WORKSPACE_DIR`) populated at runtime.

CodeAct System Prompt

Task: You are an expert SRE (Site Reliability Engineer) and Kubernetes SRE Support Agent investigating a production incident from OFFLINE snapshot data.
You are a highly capable tool-using agent able to: Diagnose Kubernetes failures; Correlate alerts, events, traces, and metrics; Identify contributing factors (not just root cause); Perform data analysis using Python when useful.

INCIDENT SNAPSHOT DATA LOCATION
Your incident data is located in `$SNAPSHOT_DIRS` (READ-ONLY). Do NOT search the filesystem for additional data. Start by listing the contents of these directories. Working directory: `$WORKSPACE_DIR`. Application topology available at `$WORKSPACE_DIR/app.json`.

FINAL OUTPUT FORMAT (MANDATORY)
Generate a JSON diagnosis identifying all Kubernetes entities: entities that CAUSED the incident (`contributing_factor=true`), entities IMPACTED but did not cause it (`contributing_factor=false`), and the propagation chain showing how the incident spread. Requirements: explain all firing alerts, provide reasoning/evidence for every entity, construct fault propagation chain, use Python for data analysis when necessary. Write diagnosis to `$WORKSPACE_DIR/agent_output.json`. Validate JSON using `jq` after writing.

ENTITY NAMING CONVENTION
All entities MUST use format: `namespace/Kind/name`. Examples: `otel-demo/Deployment/ad`, `otel-demo/Service/frontend`. Do NOT include UIDs.

PROPAGATION CHAIN (MANDATORY)
Construct chain: Root Cause $\rightarrow$ Intermediate Services $\rightarrow$ Impacted Services. Each link includes: `source` (causing entity), `target` (affected entity), `condition` (what state caused propagation), `effect` (observable effect on target).

RULES FOR CONTRIBUTING FACTORS
`contributing_factor=true` (IRREDUCIBLE): Mark ONLY if entity is an independent cause not fully explained by another contributing factor. Use "irreducibility test": if failure is explained by "upstream X failed" with propagation edge $X \rightarrow$ entity, then entity is NOT irreducible $\rightarrow$ set `false`.
`contributing_factor=false` (DERIVED): Mark entities that are downstream effects/symptoms caused by another contributing factor. IMPORTANT: Do NOT mark both cause and derived symptom as `true`. Multiple `contributing_factors` allowed ONLY if truly independent.

DEBUGGING PRINCIPLES
(1) Differential Observability: Compare replicas and time windows. (2) Occam's Razor: Choose simplest explanation consistent with evidence. (3) Duration Matching: Theory must explain entire incident duration. (4) Follow Breadcrumbs: Let alerts/errors guide investigation. (5) No Jumping to Conclusions: Validate every hypothesis with evidence. (6) Chaos Files $\neq$ Active Chaos: Verify experiment was running AND time-aligned. (7) Semantic Name Normalization: Try variations, strip suffixes, search partial matches.

Table 6: MCP tools for SRE domain.

Tool	Description
`alert_summary`	Active alerts with severity
`get_context_contract`	Context Packet for entity
`topology_analysis`	Dependency graph queries
`metric_analysis`	Metrics with anomaly detection
`log_analysis`	Service logs with filtering
`event_analysis`	Events in time window
`get_trace_error_tree`	Trace error propagation
`spec_change_analysis`	Config/deployment diffs

CodeAct uses all tools plus shell utilities and Python; EoG uses `alert_summary`, `spec_change_analysis`, and `event_analysis` to seed the ActiveSet, then invokes `get_context_contract` during entity visits.

INVESTIGATION WORKFLOW (DO NOT SKIP)
Phase 1 – Context Discovery: List available files; read topology.md to understand dependencies.
Phase 2 – Symptom Analysis: Read all alert files; compute start/end time, duration, frequency; build Alerts Table.
Phase 3 – Hypothesis Generation: Create initial hypotheses; create validation plan for each.
Phase 4 – Evidence Collection Loop: Use tools and Python to gather log/event/metrics/trace evidence; validate/refute hypotheses; explain firing alerts.
Phase 5 – Causal Chain Construction: Build chain: [Config Error] → [CrashLoop] → [Service Down] → [Frontend 5xx].
Phase 6 – Conclusion: Ensure all alerts explained, all entities included, contributions correctly labeled; output final JSON.

PYTHON ANALYSIS (ENCOURAGED)
Write Python snippets to: parse alerts/events/metrics/traces, join datasets by timestamp or UID, extract failing spans, compute metrics deltas, identify patterns (CrashLoopBackOff, OOMKilled, 5xx spikes). Persist snippets in Python files.

PROHIBITED ACTIONS
NEVER read ground_truth.yaml; NEVER output anything other than final JSON; NEVER hallucinate K8s objects; NEVER leave alerts unexplained.

C.3 Expected Output Schema

Both the CodeAct baseline and EoG agent must produce a structured JSON diagnosis with three key components:

```
{
  "entities": [{
    "name": "namespace/Kind/name",
    "contributing_factor": true|false,
    "reasoning": "Explanation for involvement",
    "evidence": "Supporting facts summary"
  }],
  "propagations": [{
    "source": "namespace/Kind/source-name",
    "target": "namespace/Kind/target-name",
    "condition": "What caused propagation",
    "effect": "What effect was observed"
  }],
  "alerts_explained": [{
    "alert": "<alert name>",
    "explanation": "Why the alert fired",
    "explained": true|false
  }]
}
```

C.4 SRE Domain Tools (MCP Server)

We implement a custom MCP server providing specialized SRE investigation tools. Table 6 summarizes the tools available to both CodeAct and EoG agents; Table 7 below provides detailed function signatures for the primary tools.

Operational Graph Construction. The `build_topology` tool constructs the operational graph G by combining static information (application architecture, service dependencies declared in configuration) with dynamic runtime state (e.g., pod-to-node bindings, replica counts, endpoint mappings from Kubernetes). Such operational graphs are readily available through commercial observability platforms—including IBM Instana [19], Dynatrace [13], and Datadog [11]—which

Table 7: SRE Utility Tools Available via MCP

Tool	Description
`build_topology`	Build operational topology graph from application architecture and Kubernetes objects
`topology_analysis`	Analyze topology for dependencies, service context, infrastructure hierarchy
`metric_analysis`	Analyze metrics with filtering, grouping, derived metrics (SQL-like interface)
`get_metric_anomalies`	Detect anomalies in time-series metrics
`event_analysis`	Analyze K8s events with filtering, grouping, aggregation
`get_trace_error_tree`	Analyze distributed traces with error trees and latency percentiles
`alert_summary`	High-level alert overview: type, entity, duration, frequency
`alert_analysis`	Detailed alert analysis with filtering and grouping
`k8s_spec_change_analysis`	Track K8s object spec changes over time
`get_k8_spec`	Retrieve K8s spec for a specific resource
`get_context_contract`	Aggregated context for an entity (events, alerts, traces, metrics, spec changes)

automatically discover and maintain service topologies through agent instrumentation and API introspection.

Key Tool Signatures. `metric_analysis`: SQL-like interface for metric analysis supporting batch queries, derived metrics, and flexible grouping.

```
def metric_analysis(
    base_dir: str,              # Metrics directory path
    k8_object_name: str = None, # Optional: specific object
    object_pattern: str = "*",  # Glob pattern (e.g., 'pod/*')
    metric_names: List[str] = [],  # Metrics to analyze
    eval: str = None,           # Derived metric expression
    group_by: str = None,       # Grouping column
    agg: str = "mean",          # Aggregation function
    start_time: str = None,     # Time range start
    end_time: str = None        # Time range end
) -> dict
```

`get_trace_error_tree`: Analyzes distributed traces to find critical paths with regressions.

```
def get_trace_error_tree(
    trace_file: str,            # Path to traces TSV
    service_name: str = None,   # Filter by service
    pivot_time: str = None,     # Timestamp for before/after
    delta_time: str = "5m",     # Comparison window size
    error_threshold_pct: float = 10,   # Error change threshold
    latency_threshold_pct: float = 10  # Latency change threshold
) -> dict
```

`get_context_contract`: The primary tool for EoG, returning full operational context for an entity.

```
def get_context_contract(
    k8_object: str,             # K8s object (Kind/name format)
    snapshot_dir: str,          # Snapshot directory path
    topology_file: str = None,  # Optional topology JSON path
    start_time: str = None,     # Time range start
    end_time: str = None,       # Time range end
    page: int = 1,              # Page number
    deps_per_page: int = 3      # Dependencies per page
) -> dict
```

D EoG Implementation Details

This section provides complete architectural and prompt specifications for the EoG (Explanations over Graphs) agent, sufficient for reproduction.

D.1 Architecture Overview

Figure 5 illustrates the complete EoG execution flow, from initialization through the event-driven inference loop to final output generation.

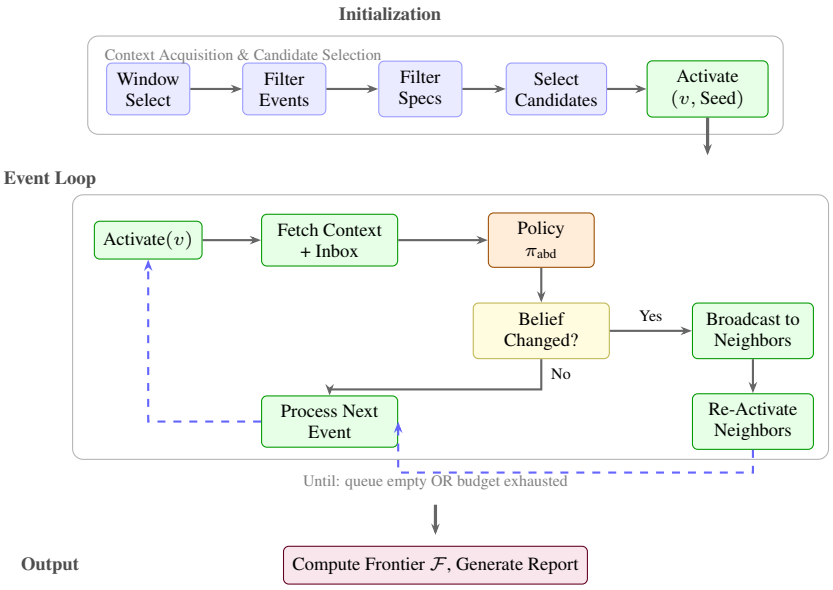


Figure 5: EoG implementation architecture. **Initialization**: Multi-phase context acquisition filters events and spec changes, then bootstraps initial candidates. **Event Loop**: Each `Activate` event triggers context fetch, policy inference, and conditional belief broadcast. The loop continues until the queue empties or budget exhausts. **Output**: The frontier $\mathcal{F}$ is computed and a structured report is generated.

Table 8: Event Types and Semantics

Event	Semantics
`Activate(v, reason)`	Evaluate node v; equivalent to `ActiveSet.pop(v)`
`Message(u, v,` B_u`, t)`	Deliver belief B_u to v's inbox
`BeliefChanged(v)`	Broadcast B_v to neighbors; triggers re-activation

D.2 Event-Driven Architecture

The implementation realizes the ActiveSet abstraction from §4.3 as an event-driven message queue. This architectural choice enables concurrent node evaluation without changing the algorithm's semantics—each event type corresponds directly to operations in the abstract algorithm.

The event queue processes `Activate` events by: (1) checking visit eligibility, (2) collecting inbox messages, (3) fetching local context, (4) invoking the abductive policy, (5) applying damping if oscillating, (6) emitting `BeliefChanged` if belief changed, and (7) emitting `Activate` for discovered or proposed nodes. This is semantically equivalent to the ActiveSet-based loop but enables the actor model for concurrent processing.

D.3 Termination Safeguards

Beyond the damping heuristic (k_{thresh}) described in §4.3, the implementation adds visit limits for robustness:

- **Max Visits**: Each node may be evaluated at most $k_{\max}$ times (default: 5), regardless of belief stability.
- **Cooldown**: Re-evaluation requires at least k_{cool} other evaluations since last visit (default: 2), preventing rapid re-activation loops.

Together with damping, these bound total inference to $O(\min(k_{\text{thresh}}, k_{\max}) \cdot |V_S|)$.

D.4 Checkpointing

The implementation supports fault-tolerant checkpointing at quiescent points between event processing. Checkpoints capture node beliefs, inbox contents, pending events, and the explanatory graph (V_S, E_S), enabling recovery without restarting the investigation.

D.5 Initialization Prompts

The investigation begins with a multi-phase initialization that sets the algorithm variables from §4.3: `select_investigation_window` determines time window W; `filter_events` and `filter_spec_changes` acquire bounded context; `bootstrap` selects initial candidates $V_{\mathcal{O}}$ for the ActiveSet.

D.5.1 Investigation Window Selection

The first step determines the temporal scope by identifying "golden signal" alerts.

select_investigation_window

```
You are an SRE agent. Your goal is to choose an investigation time window [start_time, end_time] that
best captures the onset and peak of the incident, using golden signal alerts as anchors.
Golden signals include alerts indicating: Errors (request error rate, 5xx), Latency (p95/p99),
Traffic (throughput drop, no requests).
Ignore noisy platform alerts (watchdog, inhibitors) unless they directly explain golden-signal
symptoms.
Input: Alert Summary (JSON with names, entities, timestamps, severity), Application Architecture
(service dependencies).
Output: JSON with start_time, end_time, anchor_alerts, golden_signal_alerts, reasoning.
Rules: Prefer application services over infrastructure alerts. start_time ≤ end_time. Use
timestamps from alert summary.
```

D.5.2 Event Filtering

Kubernetes clusters generate thousands of events per hour. We filter events in pages, selecting those relevant to the investigation.

filter_events

```
You are filtering Kubernetes events to identify those relevant to an ongoing incident investigation.
Relevance Criteria -- INCLUDE:
• Warning/Error type events
• Events on alerting entities or their dependencies
• Pod lifecycle failures (CrashLoopBackOff, OOMKilled, Evicted)
• Scheduling failures (FailedScheduling, FailedMount)
• Probe failures (Unhealthy, ProbeWarning)
• Resource issues (FailedCreate, QuotaExceeded)
• Deployments/restarts preceding alerts
EXCLUDE: Normal successful operations (Scheduled, Started, Created) unless systematic churn or
temporally preceding alerts.
Prioritization: Return at most {{max_per_page}} events. Rank by: (1) events preceding alerts, (2)
events on alerting entities, (3) failure events, (4) config changes. Prefer semantic diversity--avoid
redundant events conveying the same information.
Output: JSON with relevant_events array, total_relevant_before_limit, reasoning.
```

D.5.3 Spec Change Filtering

Configuration changes are among the most common root causes. We filter spec changes with similar pagination.

filter_spec_changes

```
You are filtering Kubernetes spec changes to identify those relevant to an ongoing incident
investigation. Configuration changes are one of the biggest failure causes.
Prioritization Guidance:
1. Highest: Changes just before alerts (within minutes)--prime root cause suspects
2. Medium: Changes during incident window
3. Lower: Changes well before/after incident
Key Question: "Did this change happen before the symptoms started?" If yes, strong candidate for
root cause.
Output: JSON with relevant_changes array (max {{max_per_page}}), total_relevant_before_limit,
reasoning.
```

D.5.4 Candidate Selection (Bootstrap)

With filtered events and spec changes, the bootstrap prompt selects initial investigation candidates.

bootstrap

```
You are an SRE agent. Review active alerts and select initial candidate entities for investigation.
Strategy (Golden Signal Priority):
1. Identify minimum set of alerts that cannot be explained by other alerts/events
2. Ignore positive goodput/watchdog alerts
3. Infrastructure alerts (OOM, throttling) help explain golden signals
Strict Ordering by Causal Evidence Strength:
• STRONGEST (FIRST): Entity with spec change preceding alert start
• STRONG: K8s event (OOMKilled, CrashLoopBackOff) on alerting entity
• MEDIUM: Change on entity with known dependency relationship
• WEAKEST: Alerting service with NO associated change--likely symptom
Causal Path Requirement: A spec change is only relevant if there's a causal path to the alerting
application (dependency, traffic source, config, fault injection).
Output: JSON with initial_candidates array (node_id, start_time, end_time, alert_type,
reason_for_selection, identified_facts, identified_events), reasoning.
```

D.6 Abductive Policy Prompt

The explore prompt implements local inference, reasoning over both local evidence and neighbor messages.

explore (Abductive Policy π_{abd})

```
You are an SRE support engineer doing RCA using the EoG algorithm.
Objective: Help the controller converge to a small root-cause frontier. Do NOT enumerate all
impacted entities.
Definitions:
• Healthy (L=0): Operating normally in the alert window
• Symptom (L=0): Degraded, but evidence points to upstream cause
• Origin (L=1): Has citable change (config, deployment, resource) that temporally precedes and
  explains incident
• Defer (L=2): Evidence inconclusive
Messages from Neighbors (Inbox): When neighbor claims SYMPTOM, consider if YOUR component caused
their failure. When neighbor claims ORIGIN, consider if YOUR issues are symptoms of their fault.
Core Investigation Principles:
1. "What Changed" Discipline: A node cannot be Origin if nothing changed within it
2. Transitive Traffic Amplification: Resource exhaustion without local change → suspect traffic
   surge
3. Differential Observability: Compare replicas, time windows
4. Occam's Razor + Multiple Causes: Prefer unified explanation, accept independent causes when
   evidence supports
Hard Rules:
• Evidence required for every decision (cite logs/events/metrics/traces)
• Never revisit entities in Visited set
• Max 2 next_candidates per step
• One direction at a time: upstream | downstream | ownership | infrastructure
Output: JSON with direction, local_origin (Healthy|Symptom|Origin|Defer), reasoning, propagations
array, next_candidates array.
```

D.7 Finalize Prompt

When the event queue empties or budget exhausts, the finalize prompt synthesizes findings.

finalize

```
Summarize the investigation, identifying Root Causes (Frontier) and Causal Graph (G_S). If no
frontier found, make the most educated guess given all evidence.
Critical Distinction:
• Root Cause (contributing_factor: true) = Entity that CAUSED the problem: config changes,
  deployments, infrastructure failures, fault injections, traffic pattern changes
• Symptom (contributing_factor: false) = Entity that EXHIBITED the problem: elevated error rates,
  failures due to upstream causes
Validation Checkpoint: For each contributing_factor: true entity, answer: (1) What specifically
changed? (2) When did it change? (3) How does this change explain the symptoms? If you cannot
answer all three, it's likely a symptom.
```

Table 9: Per-trial token usage (averaged across 35 scenarios). Reasoning tokens excluded.

Agent	Model	Input Tok.	Output Tok.
CodeAct	GPT-5.1	779K	9.3K
EoG	GPT-5.1	1837K	1.7K
CodeAct	GPT-OSS-120B	230K	3.8K
EoG	GPT-OSS-120B	1751K	2.1K
CodeAct	Gemini 3 Flash	5376K	21.2K
EoG	Gemini 3 Flash	3390K	2.0K
CodeAct	Kimi K2	1021K	8.8K
EoG	Kimi K2	2641K	1.9K
EoG	Minimax M2.1	1881K	2.1K
EoG	Mistral Large	1630K	1.9K

```
Contributing Factor Rules:
• True (Irreducible): Origin in Frontier with no incoming causal edge from another Origin
• False (Derived): Symptom affected by propagation; failure explained by upstream
Output: JSON with entities array (name, contributing_factor, reasoning, evidence), propagations array
(source, target, condition, effect), alerts_explained array.
```

D.8 Implementation Notes

The system comprises approximately 3,500 lines of Rust with prompts maintained as Handlebars templates. Key implementation choices:

- **Temperature Zero:** All LLM calls use temperature 0 for determinism.
- **Structured Output:** All prompts request JSON validated against schemas. Invalid responses trigger a single retry with error correction.
- **Token Budgeting:** The Context Contract implements token-aware pagination. When filtered events/changes exceed budgets, tournament-style reduction selects the most relevant items.
- **Entity Normalization:** All prompts mandate `namespace/Kind/name` format for consistent deduplication and graph construction.

D.9 Token Usage

Table 9 reports the per-trial input and output token counts averaged across the 35 ITBench scenarios for each agent–model pair shown in Table 1. Reasoning tokens are excluded because providers do not consistently report them. CodeAct entries for GPT-OSS-20B, Minimax M2.1, and Mistral Large are omitted: those configurations could not complete due to Codex framework failures.

Cost per correct diagnosis. Tokens alone understate the operational trade-off: a misdiagnosis also drives up incident mean time to resolution (MTTR). Table 10 prices each trial at OpenRouter rates and normalizes by correctness,

$$\text{Cost per correct} = \frac{3 \times \text{cost per trial}}{\text{Maj@3 F1}},$$

using the RC Entity maj@3 F1 pass rate from Table 1. EoG is cheaper per correct answer on every shared backbone: 4.6× on GPT-5.1, 2.0× on GPT-OSS-120B and Gemini 3 Flash, and 1.15× on Kimi K2.

D.10 Properties and Correctness

D.10.1 Minimality and Correctness

The algorithm seeks a *Frontier* $\mathcal{F}$ of irreducible origins.

$$\begin{aligned}\mathcal{F} = \{v \in V_S : L_v = \textsc{Origin} \wedge \nexists u \in V_S \\ \text{s.t. } L_u = \textsc{Origin} \wedge u \rightsquigarrow_{E_S} v\}\end{aligned} \tag{3}$$

Table 10: Cost per trial and per correct diagnosis (OpenRouter rates, $ per million input/output tokens).

Model ($/M in, out)	Agent	In/trial	Out/trial	Cost/trial	Maj@3 F1	Cost/correct
GPT-5.1 (1.25, 10)	EoG	1837K	1.7K	$2.31	86.1%	$8.06
	CodeAct	779K	9.3K	$1.07	8.6%	$37.22
GPT-OSS-120B (0.037, 0.17)	EoG	1751K	2.1K	$0.07	79.4%	$0.25
	CodeAct	230K	3.8K	$0.01	5.7%	$0.48
Gemini 3 Flash (0.50, 3.00)	EoG	3390K	2.0K	$1.70	92.1%	$5.54
	CodeAct	5376K	21.2K	$2.75	74.3%	$11.11
Kimi K2 (0.57, 2.30)	EoG	2641K	1.9K	$1.51	83.0%	$5.46
	CodeAct	1021K	8.8K	$0.60	28.6%	$6.31
Minimax M2.1 (0.30, 1.20)	EoG	1881K	2.1K	$0.57	64.2%	$2.65
Mistral Large (2.00, 6.00)	EoG	1630K	1.9K	$3.27	67.2%	$14.60

By definition, $\mathcal{F}$ is minimal relative to the discovered graph G_S. If no node is labeled ORIGIN (i.e., $\mathcal{F} = \emptyset$), the system falls back to best-effort ranking of candidates based on accumulated evidence. The correctness relies on three assumptions:

1. **Completeness**: Relevant data exists and is retrievable via CxC.
2. **Soundness**: The policy π_{abd} correctly infers local dependencies.
3. **Connectivity**: Origins are reachable from anomalous entities $V_{\mathcal{O}}$ via the topology or discovered edges.

If these hold, the message passing ensures that evidence from origins drives the observed symptoms, correcting any initial misclassifications caused by the exploration order.

On the role of the deterministic controller. The Soundness assumption merits clarification: EoG does not eliminate LLM stochasticity—belief assignments remain LLM-dependent. The deterministic controller ensures systematic exploration, belief bookkeeping, and propagation logic, but does not guarantee that local inferences are correct. The architecture's value lies in *bounding* the scope of each LLM decision to local context and *enabling revision* when contradictory evidence emerges via message passing. Reliability gains (Majority@k improvements) arise from this structured exploration and revision mechanism, not from eliminating stochasticity in individual inferences.

D.10.2 Termination

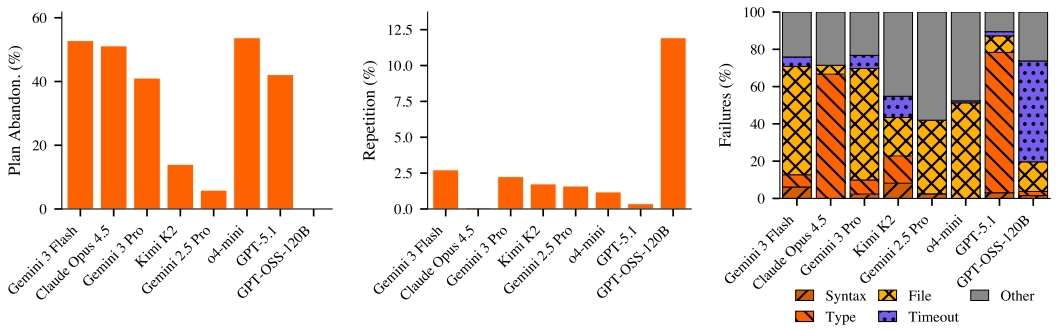
Termination is guaranteed by the damping heuristic. If a node oscillates labels more than k_{thresh} times, it is forced to DEFER (an absorbing state). This bounds the total number of inference steps to $O(k_{\text{thresh}} \cdot |V_S|)$. Cycles in the topology are allowed, but the explanatory graph E_S is directed by evidence. When evidence is conflicting, oscillation can occur; damping converts this to the absorbing DEFER state and guarantees a fixed point when no messages remain.

D.10.3 Handling Inconclusive Results

The agent may not always produce a definitive diagnosis. Two failure modes are possible: (1) *evidence absence*—the relevant observability data is missing; and (2) *evidence blindness*—the policy π_{abd} lacks the domain knowledge to recognize available evidence as diagnostically relevant. In either case, nodes are labeled DEFER. If no ORIGIN is identified at termination (i.e., frontier $\mathcal{F} = \emptyset$), the Controller falls back to a best-effort ranking of candidate nodes based on accumulated evidence, explicitly signaling uncertainty to the operator.

E Analyzing Controller Failures

The CodeAct paradigm conflates *reasoning* (what to investigate) with *execution* (how to invoke tools). This overloading of the LLM leads to measurable controller instability. We analyze three failure modes—plan abandonment, tool repetition, and syntactic failures—that motivate the separation of concerns in EoG.



(a) Plan abandonment rate. **(b)** Tool repetition rate. **(c)** Tool failure categories.

Figure 6: Controller brittleness in CodeAct agents. (a) Plan abandonment rate: frequency with which agents discard stated investigation plans. (b) Tool repetition rate: percentage of tool calls that duplicate previous invocations. (c) Tool failure categories: distribution of syntactic vs. semantic errors.

Plan abandonment. Figure 6a quantifies how often agents abandon their stated investigation plans. o4-mini abandons plans in over 50% of cases; even stronger models like Claude Opus 4.5 exhibit abandonment rates over 50%. Plan abandonment occurs for three reasons: (1) agents fail to revise plans as new evidence emerges, leaving stale steps that no longer apply [15]; (2) agents forget earlier plan steps as context grows (context rot) [28]; and (3) agents prematurely conclude they have found the cause and skip remaining steps, converging to a minimal explanation before sufficient evidence is gathered. This instability undermines auditability: the agent's stated reasoning diverges from its actual behavior.

Tool repetition. Agents exhibit degenerate loops where they repeatedly invoke the same tool calls. Figure 6b shows that GPT-OSS-120B has a repetition rate of about 12%, consuming context budget without advancing the investigation. Tool repetition can occur for two reasons: (1) re-executing a failed tool call without correcting the arguments or function signature; or (2) re-fetching data to analyze it differently in light of new evidence. The latter is potentially useful—revisiting data with updated hypotheses is a valid diagnostic strategy—but we find no evidence of productive re-analysis in our trajectories; the vast majority of repetitions are failed-execution loops. This redundancy compounds with trajectory length, leaving less capacity for novel exploration.

Syntactic failures. Figure 6c categorizes tool invocation failures. A substantial fraction are strictly syntactic—malformed arguments, type errors, invalid parameter combinations—rather than semantic reasoning failures [8]. These errors indicate that LLMs are suboptimal drivers for precise tool orchestration when simultaneously reasoning about complex domains. In contrast, conventional non-deterministic programs tend to excel in tool invocation and execution when they have access to the right contextual information.

E.1 MAST Failure Mode Taxonomy

We use MAST (Multi-Agent System Trajectories) [42] to systematically categorize controller failures in our agent trajectories. MAST provides a structured taxonomy that classifies failures across three inter-agent conversation stages: Pre-Execution, Execution, and Post-Execution.

Figure 7 shows the taxonomy with prevalence rates across our trajectories:

System Design Issues (44.2%). The largest category encompasses architectural failures:

- **1.1 Disobey Task Specification (11.8%)**: Agent ignores explicit task instructions.
- **1.2 Disobey Role Specification (1.5%)**: Agent acts outside its designated role.
- **1.3 Step Repetition (15.7%)**: Agent repeats previously executed steps without progress.
- **1.4 Loss of Conversation History (2.8%)**: Agent loses track of prior context.
- **1.5 Unaware of Termination Conditions (12.4%)**: Agent fails to recognize when task is complete.

Inter-Agent Misalignment (32.3%). Communication and coordination failures:

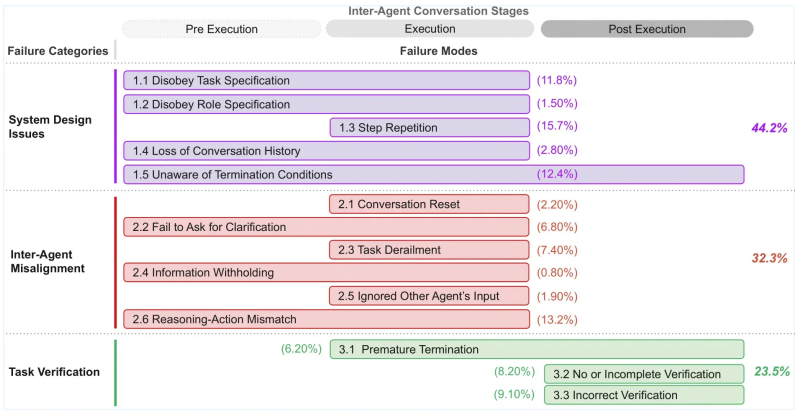


Figure 7: MAST failure mode taxonomy. Failures are categorized into three main categories: **System Design Issues** (44.2%)—failures stemming from how the agent system is architected; **Inter-Agent Misalignment** (32.3%)—failures in communication and coordination; and **Task Verification** (23.5%)—failures in validating task completion.

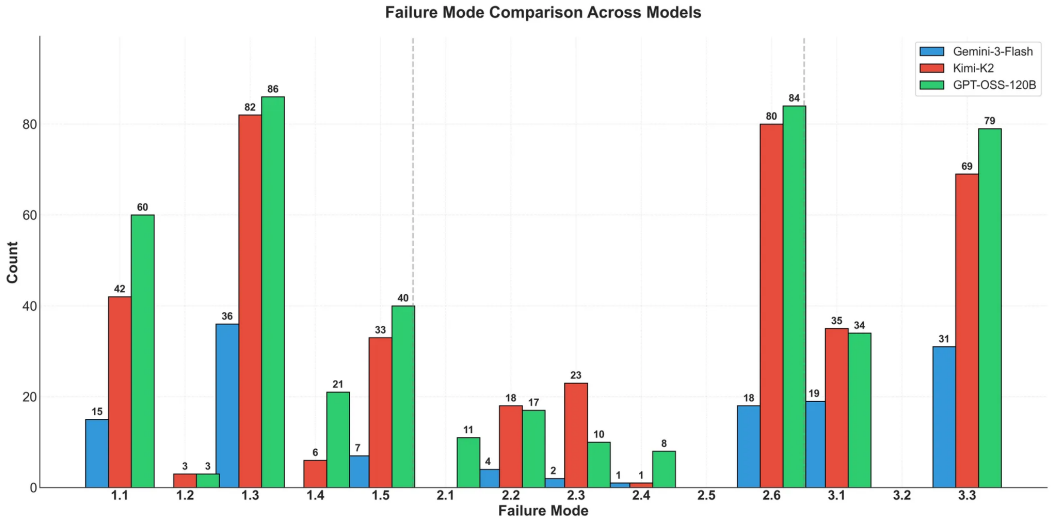


Figure 8: Failure mode counts across models. GPT-OSS-120B and Kimi-K2 exhibit higher failure counts across most categories compared to Gemini-3-Flash, with Step Repetition (1.3) and Reasoning-Action Mismatch (2.6) being the most prevalent failure modes.

- **2.1 Conversation Reset (2.2%)**: Agent restarts conversation inappropriately.
- **2.2 Fail to Ask for Clarification (6.8%)**: Agent proceeds despite ambiguity.
- **2.3 Task Derailment (7.4%)**: Agent deviates from the assigned task.
- **2.4 Information Withholding (0.8%)**: Agent fails to share relevant information.
- **2.5 Ignored Other Agent's Input (1.9%)**: Agent disregards feedback.
- **2.6 Reasoning-Action Mismatch (13.2%)**: Agent's stated reasoning contradicts its actions.

Task Verification (23.5%). Failures in validating task completion:

- **3.1 Premature Termination (6.2%)**: Agent stops before task completion.
- **3.2 No or Incomplete Verification (8.2%)**: Agent fails to verify its output.
- **3.3 Incorrect Verification (9.1%)**: Agent incorrectly validates erroneous output.

Figure 8 compares failure mode prevalence across three models. Key observations: (1) Step Repetition (1.3) is the most common failure across all models, with GPT-OSS-120B showing 86 instances versus 36 for Gemini-3-Flash. (2) Reasoning-Action Mismatch (2.6) shows similar patterns, indicating a fundamental limitation of the CodeAct paradigm where LLMs must simultaneously plan and execute. (3) Gemini-3-Flash exhibits consistently lower failure counts, correlating with its higher diagnostic accuracy.

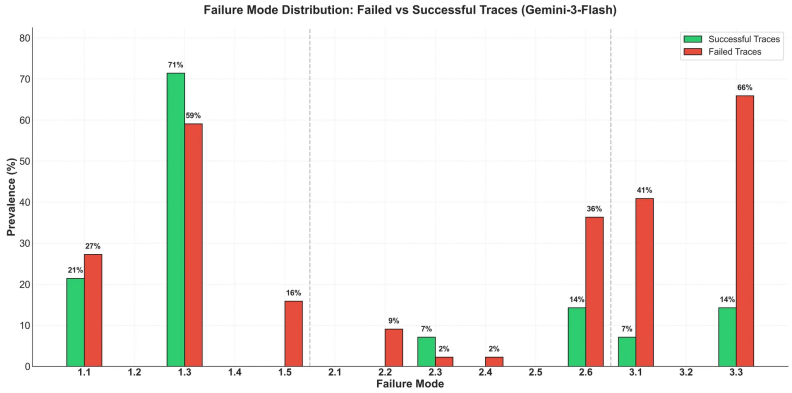


Figure 9: Failure mode distribution in successful vs. failed traces for Gemini-3-Flash. Green bars show prevalence in successful traces; red bars show prevalence in failed traces.

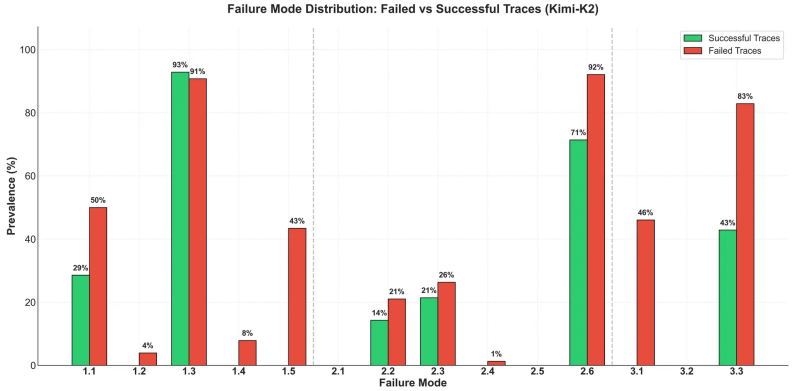


Figure 10: Failure mode distribution in successful vs. failed traces for Kimi-K2. Green bars show prevalence in successful traces; red bars show prevalence in failed traces.

Figures 9–11 reveal which failure modes differentiate successful from failed traces:

- **Discriminative failures**: Incorrect Verification (3.3) shows the largest gap between failed and successful traces across all models (66% vs 14% for Gemini-3-Flash, 83% vs 43% for Kimi-K2, 94% vs 50% for GPT-OSS-120B). This indicates that verification failures are strongly predictive of task failure.
- **Non-discriminative failures**: Step Repetition (1.3) appears at high rates in both successful and failed traces (71% vs 59% for Gemini-3-Flash), suggesting that some repetition may be a natural part of iterative exploration rather than a direct cause of failure.
- **Model-specific patterns**: GPT-OSS-120B shows elevated failure rates across nearly all categories in failed traces, while Gemini-3-Flash maintains lower baseline rates even in failed cases.

These findings motivate the core architectural decision in EoG: by separating deterministic control (state management, termination detection, verification) from non-deterministic reasoning (causal inference), we eliminate the System Design Issues (44.2%) and Task Verification failures (23.5%) that account for over two-thirds of observed failures.

F Additional Results

F.1 GraphRAG Baseline

GraphRAG [14] receives the same dependency graph and the same engineered Context Contract for every node. We run all three query modes (local, drift, global) on all 35 SRE scenarios with 3 trials each, using Gemini 3 Flash, EoG's best-performing model (Table 11). Drift is the strongest mode but trails EoG on every metric (1.53× on RC Entity maj@3 F1, 1.13× on RC Reasoning maj@3 R), so EoG's gain is not explained by the graph and Context Contract alone.

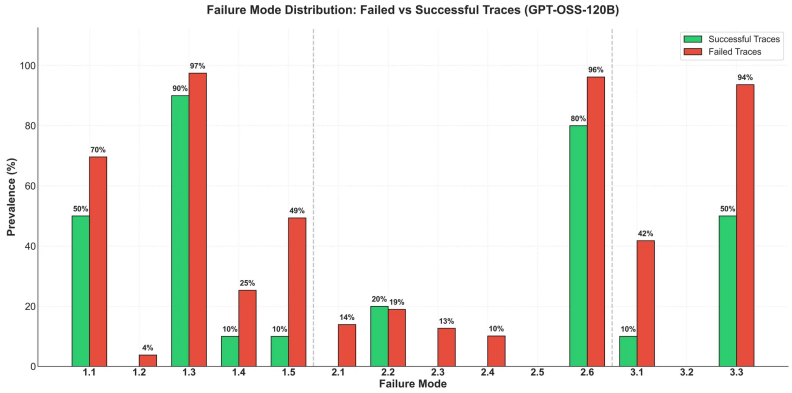


Figure 11: Failure mode distribution in successful vs. failed traces for GPT-OSS-120B. Green bars show prevalence in successful traces; red bars show prevalence in failed traces.

Table 11: GraphRAG vs. EoG (Gemini 3 Flash, n=35). Cells are 95% Wilson CIs as p^{+u}_{-l}; token columns are per trial.

Mode	RC Entity pass@3 F1	RC Entity pass@3 R	RC Entity maj@3 F1	RC Entity maj@3 R	RC Reas. maj@3 R	In tok	Out tok
local	$34.3^{+16.6}_{-13.5}$	$37.1^{+16.5}_{-14.0}$	$31.4^{+16.6}_{-12.9}$	$37.1^{+16.5}_{-14.0}$	$37.1^{+16.5}_{-14.0}$	13.4K	1.3K
drift	$77.1^{+10.8}_{-16.2}$	$91.4^{+5.6}_{-13.8}$	$60.0^{+14.4}_{-16.4}$	$82.9^{+9.0}_{-15.5}$	$80.0^{+10.0}_{-15.9}$	368.8K	393.1K
global	$5.7^{+12.9}_{-4.1}$	$5.7^{+12.9}_{-4.1}$	$5.7^{+12.9}_{-4.1}$	$5.7^{+12.9}_{-4.1}$	$5.7^{+12.9}_{-4.1}$	14.6K	49.0K
EoG	$\mathbf{94.5^{+3.9}_{-13.1}}$	$\mathbf{97.1^{+2.4}_{-11.6}}$	$\mathbf{92.1^{+4.9}_{-14.5}}$	$\mathbf{94.3^{+4.1}_{-12.9}}$	$\mathbf{90.2^{+6.8}_{-12.6}}$	3390K	2.0K

F.2 Robustness to Imperfect Graphs and Observability

We run controlled perturbations on all 35 SRE scenarios with GPT-OSS-120B; Table 12 reports changes relative to the GPT-OSS-120B row of Table 1.

Table 12: Robustness to graph and observability perturbations (GPT-OSS-120B, n=35). Δ is relative to the unperturbed run; the reliability gap is pass@3 − maj@3, as a multiple of the unperturbed gap.

Perturbation	Δ maj@3 R	Gap ($\times$ clean)
Drop 10% of causal edges (`depends_on/calls`)	-3.5%	$1.0\times$
Drop 50% of causal edges	-10.7%	$2.0\times$
Sever all edges incident to root-cause node	-1.5%	$1.1\times$
Graph built without event-derived entities	-32.1%	$2.7\times$
Context Contract: events section hidden	-17.9%	$2.3\times$
Context Contract: dependencies section hidden	-7.2%	$1.3\times$

Missing edges. Random removal of causal edges degrades consistency, not discovery. At 10% the effect is within noise on every metric. At 50%, maj@3 falls 10.7% and the reliability gap doubles: the correct diagnosis remains reachable but is reached less consistently. Dropping 50% of the non-causal (containment) edges has no effect on any metric, so the cost is attributable to causal structure rather than to graph mutation per se. The attenuation is by design: the policy's propagation claims are written into the explanatory graph and unseen sources are enqueued (Algorithm 1), so EoG maintains its own causal graph and re-discovers missing edges from observational evidence—the missing $S_2 \to S_3$ edge of §4.4, at scale. Severing every edge incident to the root-cause node leaves entity recall statistically unchanged, because the node remains a reachable hypothesis that π_{abd} can convict on local evidence alone.

Events. Events are the most informative input; we ablate them two ways. (i) *Event-derived entities removed from the graph:* building the topology from the object sweep alone omits entities that appear only in the event stream, so they never become graph nodes. This is the largest degradation we induce (-30% entity recall, -32% maj@3, gap $2.7\times$), and it is localized: scenarios whose root-cause entity is in the object sweep are unaffected (-2% vs. clean), while scenarios whose root cause appears only in the event stream lose two-thirds of their recall. EoG requires the graph to contain the causal

entity, not to be completely or correctly wired. (ii) *Removal from the Context Contract:* hiding the events section from every classification costs 8.8% entity recall (not significant) and 17.9% maj@3—material, but far smaller than removing event-derived entities from the graph, because the entity still exists as a hypothesis and alert, metric, and spec evidence partially substitute.

Noise and degraded observability. Adding 30% spurious random edges leaves recall unchanged and precision intact: spurious neighbors are explicitly marked non-contributing, so belief propagation does not convict on topology adjacency without supporting evidence; these edges cost up to 2× tokens. Randomly dropping 50% of alerts or of raw events is within noise, as most events and alerts carry no diagnostic signal; this extends the distributed-trace ablation of §5.

F.3 Paired Belief-Revision Analysis

We instrumented every classification decision of EoG with GPT-OSS-120B and score the same runs with and without belief revision, isolating the mechanism independently of the operating point. SBP revises a label in 28% of runs. On the entity that determines the answer, it corrects far more often than it errs: 72% of revisions to the GT root cause move it toward ORIGIN, and 85% of those arrive from SYMPTOM—the error message passing is designed to repair, where an entity first reads as a victim of its own blast radius until neighbor attribution reverses the direction. In one scenario, two revisions (EXONERATED→ORIGIN, SYMPTOM→ORIGIN) produce the only correct labels ever assigned to the injected entity, which the first-visit classifier never identifies; without belief revision that scenario has zero root-cause recall in every run.

The remaining 28% of root-cause revisions move a label away from ORIGIN. In all of them, the run still terminates with the correct root cause on the frontier and in the reported answer, by two routes. Two recover inside the propagation loop, as continued message passing re-promotes the entity; each flips three times before settling on ORIGIN. The other three relocate the cause to the correct point in the causal chain: from `Service/shipping` to the `Deployment/shipping` whose spec change produced the failure, from a spawned `NetworkChaos` instance to the parent `Schedule` that creates it, and from a saturated service to the upstream chaos experiment that saturated it. In each case the demoted node is a downstream consequence of the node that replaced it, and the frontier retains exactly those ORIGIN nodes that no other ORIGIN reaches. Because the terminal decision is drawn from the frontier and its evidence rather than from any single node's label history, a transient demotion is not an answer-level loss. Consequently, belief revision improves half of the scenarios that fail without it and costs accuracy in none of the scenarios that already succeed.

G LLM-as-a-Judge Evaluation Framework

We employ an LLM-as-a-Judge (LAAJ) evaluation framework to assess agent outputs against ground truth root cause analysis. The judge evaluates two primary metrics—**Root Cause Entity Identification** (Precision, Recall, F1) and **Root Cause Reasoning Accuracy**—in a single LLM call.

Evaluation cost considerations. Each agent run produces a structured diagnosis that must be evaluated against ground truth with semantic matching (entity aliasing, reasoning quality assessment). The LAAJ evaluation requires an LLM call per (scenario, run) pair, incurring non-trivial cost at scale. Due to this cost, we focus our detailed analysis on a representative subset of frontier models (Gemini 2.5 Pro/Flash, Claude Opus 4.5, o4-mini) that span the capability spectrum, while reporting aggregate metrics for additional models.

Judge validation. To validate the Gemini-2.5-Pro judge, an SRE expert independently annotated the root-cause entity judgment for 18 incidents across 5 trials of the GPT-OSS-120B evaluation; we ran the judge 5 times per (incident, trial) pair to measure self-consistency. The judge produced the same score in all five runs on 78/85 pairs (91.8%), and on the remaining 7 pairs agreed on 3.9 of 5 runs on average. Against the human annotations, agreement is 90.6% (Cohen's κ=0.808) under the most conservative treatment, which counts every self-inconsistent pair as a disagreement regardless of its majority verdict, and 98.8% (κ=0.975) under majority vote over the five runs. Both exceed the conventional κ>0.8 threshold for strong agreement. The validation covers the SRE domain only; judge agreement on the Code and FinOps domains (§5) has not been measured.

G.1 Entity Mapping Challenge

To understand why entity normalization and mapping is critical, consider Scenario-1 from our benchmark involving a traffic overload incident.

G.1.1 Ground Truth Structure

The ground truth defines entities using semantic identifiers with regex-based filters:

Listing 1: Ground Truth Entity Groups (Scenario-1)

```
groups:
  - id: load-generator-pod-1
    kind: Pod
    filter: [load-generator-.*]
    namespace: otel-demo
    root_cause: true
  - id: load-generator-deployment-1
    kind: Deployment
    filter: [load-generator\b]
    namespace: otel-demo
    root_cause: true
  - id: flagd-config-1
    kind: ConfigMap
    filter: [flagd-config\b]
    namespace: otel-demo
    root_cause: true
  - id: frontend-proxy-service-1
    kind: Service
    filter: [frontend-proxy\b]
    namespace: otel-demo
```

G.1.2 Alias Groups

Multiple Kubernetes objects represent the same logical service. The ground truth captures this through alias groups:

Listing 2: Ground Truth Aliases

```
aliases:
  - [load-generator-service-1, load-generator-pod-1,
     load-generator-deployment-1]
  - [frontend-proxy-service-1, frontend-proxy-pod-1]
```

If a model identifies `otel-demo/Service/load-generator` as a root cause, it receives credit because it is an alias of `load-generator-pod-1`.

G.1.3 Propagation Chain

The ground truth defines the causal chain showing how the fault propagated:

Listing 3: Ground Truth Propagation Chain

```
propagations:
  - source: flagd-config-1
    target: load-generator-deployment-1
    condition: flagd-config updated to enable higher traffic
    effect: load-generator deployment applies new config
  - source: load-generator-deployment-1
    target: load-generator-pod-1
    condition: updated deployment rolls out pods
    effect: load-generator pod configured with higher users
  - source: load-generator-pod-1
    target: load-generator-service-1
    condition: pod generates high number of requests
    effect: service receives high request volume
  - source: load-generator-service-1
    target: frontend-proxy-service-1
    condition: frontend-proxy overloaded
    effect: Error rate and latency above threshold
```

G.1.4 The Mapping Challenge

Actual Kubernetes object names include generated suffixes. The evaluation must map agent-predicted entities to ground truth semantic IDs using regex filters. For example:

- `load-generator-pod-1` matches `otel-demo/Pod/load-generator-c7cbc4c99-xyz12`
- `flagd-config-1` matches `otel-demo/ConfigMap/flagd-config`
- `load-generator-deployment-1` matches `otel-demo/Deployment/load-generator`

G.2 Judge System Prompt

The LAAJ system prompt defines the evaluation protocol:

LAAJ System Prompt

```
You are an expert AI evaluator specializing in Root Cause Analysis (RCA) for complex software systems.
Your role is to act as a fair, consistent, and precise judge when evaluating RCA model outputs.
You will be provided with: (1) A Ground Truth (GT) JSON object: true root cause(s), failure
propagation path(s), and entity definitions. (2) A Generated Response JSON object: output from
an RCA model to evaluate.
Phase 1: Normalization and Entity Mapping -- Before any scoring, normalize and map entities from
Generated Response to Ground Truth based on explicit evidence from entity metadata. Do not infer
mappings based on position in propagation chain.
Definition of a Confident Match: A Generated Response entity is a confident match ONLY IF its name
field clearly corresponds to the filter and kind of the GT entity.
Entity Name Format: Generated Response entities use format: namespace/Kind/name. Examples:
otel-demo/Deployment/flagd, otel-demo/Service/frontend.
CRITICAL - Alias Handling: If a predicted entity matches an ALIAS of a root cause entity, it MUST be
counted as correct. The GT.aliases field contains arrays of equivalent entity IDs. If ANY entity in
an alias group has root_cause: true, then matching ANY entity in that alias group counts as correctly
identifying the root cause.
Example: Model entity otel-demo/Service/adservice matches GT entity ad-service-1 with filter
[".*adservice\\b"].
```

G.3 Evaluation Metrics

The LAAJ framework evaluates two primary metrics:

- **Root Cause Entity** (Precision, Recall, F1): Match `contributing_factor=true` entities against GT root causes with alias support.
- **Root Cause Reasoning** (Average): Semantic similarity of condition descriptions using 0/0.5/1 scoring.

Pass/fail criterion. Every ITBench SRE scenario used here has a single GT root-cause entity; in the 3 multi-fault scenarios, all faults map to the same entity. Recall is therefore binary: if Recall$=0$, the true-positive count is 0, forcing Precision$=0$ and F1$=0$. A run passes the R columns of Table 1 iff it recalls the GT entity, so these columns are invariant to the recall threshold (a sweep over thresholds yields identical results). The F1 columns retain the F1≥ 0.5 threshold and capture false-positive entities; e.g., EoG with GPT-OSS-120B attains 94.3% pass@3 R but 84.1% pass@3 F1.

G.3.1 Root Cause Entity Identification

Root Cause Entity Scoring

```
Method:
1. Extract all entities with root_cause: true in GT.groups.
2. Handle Aliases: Check GT.aliases for any alias groups containing a root
   cause entity. ALL entities in such groups are valid root cause matches.
3. Extract all entities with contributing_factor: true from Generated Response,
   preserving order.
4. Normalize entities to compare against GT root causes.
5. For EACH predicted entity (in order), determine if it matches any GT root
   cause entity OR any alias.
6. Output per-entity match list showing which predicted entities matched GT
   entities.
7. Calculate Recall = Correct / Total GT (count each alias group only once).
8. Calculate Precision = Correct / Total Predicted.
9. Calculate F1 = (2 × Recall × Precision) / (Recall + Precision).
Output: List of GT entities, ordered list of predicted entities with per-entity
match boolean, Precision, Recall, and F1.
```

G.3.2 Root Cause Reasoning Accuracy

Root Cause Reasoning Scoring

```
Method:
1. Extract the condition of every correct root cause entity from Generated
   Response.
2. Extract the condition of corresponding entities from GT.
3. For each correct root cause entity, compare conditions:
   • Score 1.0: Model's reason is semantically equivalent to GT's condition
   • Score 0.5: Model's condition describes similar resource/action but
     at least one key detail is missing or imprecise (e.g., GT="configmap Y
     featureflag X set", Model="configmap updated")
   • Score 0.0: Incorrect or absent model condition
4. Calculate average of all entity reasoning scores.
5. If no root cause entity correctly identified, score = 0.
Output: Average reasoning score.
```

G.4 Evaluation Prompt Template

Evaluation User Prompt

```
Given the following Ground Truth (GT) and Generated Response, evaluate the
response according to the scoring rubric.
## Ground Truth (GT):
{ground_truth}
## Generated Response:
{generated_response}
## Task:
1. Perform Phase 1: Normalize and map entities.
2. Calculate ROOT_CAUSE_ENTITY metrics (Precision, Recall, F1).
3. Calculate ROOT_CAUSE_REASONING score.
4. IMPORTANT: Use calculator_tool for ALL mathematical calculations.
CRITICAL: Final response MUST be ONLY the JSON object with scores.
```

NeurIPS Paper Checklist

1. **Claims**

Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

Answer: [Yes]

Justification: The abstract and introduction (§1) state three concrete claims, (i) CodeAct agents exhibit a reliability gap, exploration failures, and controller failures on abductive operational diagnosis, (ii) EoG's disaggregated controller + abductive policy with Semantic Belief Propagation addresses these failures, and (iii) EoG yields large gains on ITBench SRE scenarios (e.g., $\sim 7\times$ Majority@k F1 over CodeAct on average). These map directly onto the empirical evidence in §3 (failure characterization), §4–§4.3 (method), and §5 (Tables 1–3).

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: §8 discusses limitations: (i) empirical validation is restricted to 35 ITBench SRE scenarios, and the gains are not claimed to transfer without adaptation (re-tuning the abductive policy, adapting the Context Contract, and supplying a task-specific resource-graph schema) to other IT enterprise diagnostic tasks (FinOps anomaly explanation, IT vulnerability analysis, DevOps); and (ii) we do not advocate deploying such agents in production, as autonomous diagnostic systems acting on safety-critical infrastructure carry risks.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

   Justification: Appendix A formalizes EoG and includes needed proofs.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: §5 specifies the benchmark (35 ITBench SRE scenarios), the number of runs per scenario (3), the metrics (RC Entity / RC Reasoning under pass@3 and majority@3), the definition of binary scenario outcomes (F1 ≥ 0.5 for F1 columns; recall of the single ground-truth entity for Recall columns), and the reasoning effort setting. Algorithm 1 provides the full EoG procedure; Appendix D and C give implementation details for both EoG and the CodeAct baseline; Appendix C.4 (Table 6) lists the MCP tool surface; Appendix D.7 provides the finalize prompt; and Appendix G defines the evaluation protocol.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No]

   Justification: Evaluation uses the publicly available ITBench benchmark [20] and the open-source Codex agent framework (`https://github.com/openai/codex`). The EoG implementation is not released; Algorithm 1 and Appendices C–D specify the procedure, prompts, and tool interfaces.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

Justification: No model training is performed. EoG is an inference-time agent architecture wrapping off-the-shelf LLMs. Inference settings (high reasoning effort, 3 runs per scenario, EoG hop budget $H_{\max}$, damping threshold k_{thresh}, context-window chunking at >80K tokens) are reported in §5 and Appendix D. The set of LLMs evaluated (GPT-5.1, GPT-OSS-120B/20B, Gemini 3 Flash, Kimi K2, Minimax M2.1, Mistral Large) and the tool surface available to each agent are specified in §5.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: All cells in Tables 1, 2, and 3 report 95% Wilson confidence intervals over $n = 35$ scenario-level. The unit of variability (per-scenario binary pass/fail at the stated threshold) and the CI construction are stated in the table captions. §5 (SBP ablation paragraph) explicitly notes that marginal CIs do not exploit within-scenario pairing and discusses the resulting interpretation.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Per-model token usage is reported in Appendix D.9. All LLMs are accessed via hosted inference APIs (no local GPU training or fine-tuning); the agent framework runs on standard CPU hardware.

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The work does not involve human subjects, personally identifiable data, or scraped web content. All evaluations use the publicly released ITBench benchmark [20] on synthetic IT environments, and all LLMs are accessed through their providers' standard terms of use.

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: *Positive impacts:* more reliable and auditable AI-assisted incident diagnosis can reduce outage duration and the operational risk of remediation actions taken on incorrect diagnoses, which is the explicit motivation in §1. The disaggregated controller produces explanations as minimal subgraphs, improving auditability over free-form CodeAct trajectories. *Potential negative impacts:* automating operational diagnosis may reduce demand for some site-reliability roles, and overreliance on the agent in safety-critical infrastructure could mask its failure modes when the topology graph or context contracts are incomplete. These points are discussed in §8.

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper does not release models, datasets, or code.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: ITBench [20] (used for evaluation) and the Codex agent framework [35] (used as the implementation substrate) are cited at first use and used in accordance with their respective open-source licenses. All evaluated LLMs are accessed via their providers' published APIs and cited in §5. The MAST taxonomy [42] used for the controller-failure analysis (Appendix E) is similarly credited.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [N/A]

Justification: The paper does not release new assets.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A]

Justification: The paper does not involve crowdsourcing or research with human subjects. All evaluations are automated on the ITBench benchmark.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: The paper does not involve human subjects, so IRB review does not apply.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: LLMs are a core component of the proposed method: the abductive policy π_{abd} and the finalize policy π_{finalize} in Algorithm 1 are realized by an LLM operating over bounded local context. The specific LLMs used (GPT-5.1, GPT-OSS-120B/20B, Gemini 3 Flash, Kimi K2, Minimax M2.1, Mistral Large), how they are invoked (per-node Context Packets, no fine-tuning, high reasoning effort), and their role within the disaggregated architecture are described in §4–§4.3 and §5.