Plan2RCA: Compiling Observability Evidence into Executable Graph Patches for Microservice Root Cause Analysis

Anonymous ACL submission

Abstract

Root cause analysis (RCA) in microservice systems is particularly challenging when failures occur together with topology transitions, such as autoscaling, pod replacement, selector redirection, endpoint removal, and canary deployment. These transitions do not merely add noise to telemetry; they change the semantics of nodes, edges, routes, and traffic weights. Existing graph-based RCA methods typically rely on a static dependency graph or an incident-time snapshot, whereas unconstrained LLM-agent diagnosis is hard to audit and may introduce unsupported dependencies or root-cause claims. We propose Plan2RCA, an LLM-agent that formulates topology-adaptive RCA as evidence-to-graph-patch compilation. Given compact observability evidence, the agent produces a schema-constrained GraphPatchPlan instead of directly predicting a root cause. The patch is validated by deterministic schema, whitelist, integrity, and evidence-grounding checks, executed on the reference topology, and then scored by a fixed-configuration GNN. An evidence-bound reranker finally verifies only the GNN top-k candidates. On dynamic-topology benchmarks, Plan2RCA achieves Acc@1 scores of 0.871 on Online Boutique and 0.823 on TrainTicket, outperforming the strongest baseline by 28.2 percentage points on both benchmarks. Ablation results further show that the full pipeline improves Acc@1 from 0.371 to 0.871 and MRR from 0.573 to 0.905 over the static-graph variant, demonstrating the value of executable, evidence-grounded graph patches for reliable microservice RCA.

1 Introduction

Microservice root cause analysis (RCA) is difficult because failures propagate through service dependencies and often surface as downstream symptoms. The problem becomes more challenging in Kubernetes-based systems (Burns et al., 2016), where the topology may change during an incident: autoscaling adds replicas, pod reconstruction changes runtime identities, selector misconfiguration redirects endpoints, and canary release splits traffic across service versions. In such cases, the topology transition is not merely background noise; it is part of the diagnostic evidence.

Existing graph-based RCA methods typically rank suspicious services on a static dependency graph or on a reconstructed incident-time graph. Both choices are limited for dynamic-topology incidents. The static graph misses runtime mutations, while the incident snapshot loses the semantics of the transition. For example, an added pod may be a benign HPA response, an empty endpoint may indicate blackhole routing, and a selector-to-backend mismatch may directly encode the fault mechanism. These cases require different graph semantics even when their raw node or edge differences appear similar.

LLM agents can integrate heterogeneous observability evidence, but directly asking an LLM to output a root cause is hard to validate. Free-form diagnoses may introduce unsupported dependencies, confuse root causes with propagated symptoms, or make decisions that cannot be traced to metrics, logs, traces, or topology evidence. Operational RCA therefore requires an agent design whose intermediate actions are constrained, executable, and auditable.

We propose Plan2RCA, a topology-adaptive RCA framework that compiles observability evidence into executable graph patches. Instead of predicting a root-cause label, the LLM emits a schema-constrained GraphPatchPlan composed of whitelisted graph operations. The plan is validated, executed on the reference topology, and scored by a fixed-configuration GNN. An evidence-bound reranker then verifies only the GNN top-k candidates. Thus, the LLM changes the RCA graph only through validated operations and cannot introduce unsupported final root causes.

Our contributions are:

1. We formulate dynamic-topology RCA as evidence-to-graph-patch compilation, where topology transitions are represented as typed, executable, and inspectable graph operations.

2. We design GraphPatchPlan, a schema-constrained patch representation that grounds node, edge, weight, and state-transfer operations in endpoint, pod, trace, metric, log, and topology evidence.

3. We combine LLM-planned graph patches with fixed-configuration GNN scoring and evidence-bound reranking, and evaluate the framework on static RCA datasets and dynamic-topology benchmarks.

2 Preliminary and Motivation

2.1 Dynamic-Topology Root Cause Analysis

Let $G_0 = (V_0, E_0)$ denote the reference topology before an incident. Let $G_{\text{obs}} = (V_{\text{obs}}, E_{\text{obs}})$ denote the runtime topology observed after a topology mutation has occurred. The first graph represents the stable dependency structure, whereas the second is an incident-time measurement that may mix benign operational changes, such as autoscaling, with fault-inducing changes, such as selector redirection or endpoint removal. Each experiment also provides telemetry summaries X, including metrics, logs, traces, endpoint states, pod states, and topology-transition evidence.

The objective of dynamic-topology RCA is to rank candidate root-cause entities from a candidate set $\mathcal{C}$, where candidates may include services, pods, version-specific service nodes, and virtual incident nodes created to represent missing or unreachable endpoints:

$$R = [v_1, v_2, \ldots, v_k], \quad v_i \in \mathcal{C}. \tag{1}$$

The true root cause should appear as early as possible in this ranked list.

The key distinction from static RCA is that the transition from G_0 to G_{obs} is itself diagnostic evidence. A removed endpoint, a new pod, and a selector-to-backend mismatch may correspond to blackhole routing, replica scaling, and selector poisoning, respectively. These cases require different causal interpretations even when their raw graph differences appear similar. Dynamic-topology RCA therefore requires a structured transition representation rather than direct replacement of G_0 with G_{obs}.

2.2 Motivation 1: Why Graph Patches Rather than Snapshots?

A static dependency graph misses runtime replicas, endpoint changes, version splits, and blackhole paths, while an incident-time snapshot removes the contrast between the intended topology and the observed topology. This contrast is often the diagnostic signal: HPA scaling may be only a response to load, pod drift requires identity alignment, selector poisoning requires endpoint contradiction or trace mismatch, and canary release requires version-level traffic semantics. These cases motivate explicit graph patches rather than static graphs or raw snapshots.

To preserve this information, Plan2RCA represents topology adaptation as an executable graph patch:

$$\begin{aligned} P &= [op_1, op_2, \ldots, op_m], \\ G^* &= \text{Execute}(G_0, P). \end{aligned} \tag{2}$$

Each operation in P has an explicit type, argument set, and evidence reference. The resulting graph G^* retains the stable reference topology while injecting incident-specific transition evidence as typed nodes, edges, and weights. Since the downstream GNN is fixed across variants, performance differences can be attributed primarily to the representation of topology evidence rather than to model capacity or hyperparameter choices.

2.3 Motivation 2: Why Constrained LLM Actions rather than Free-Form Diagnosis?

LLM agents can connect topology changes with metrics, logs, and traces, but free-form root-cause generation is difficult to audit. It may invent dependencies, over-rank downstream symptoms, or choose services unsupported by the observed topology. Such errors are risky because RCA results may guide rollback, mitigation, or debugging actions.

Plan2RCA therefore uses the LLM as a constrained planner rather than a free-form predictor. The planner outputs a schema-constrained GraphPatchPlan whose operations come from a small whitelist and must pass validation for schema, operation type, node and edge references, graph integrity, and evidence grounding. Invalid edits

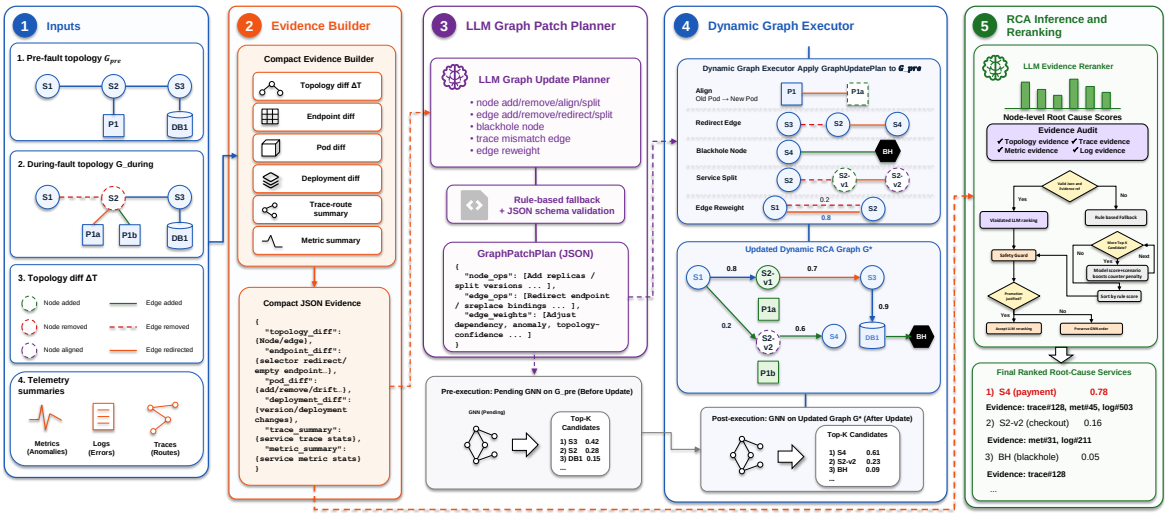


Figure 1: The architecture of Plan2RCA. The LLM agent compiles observability evidence into an executable graph patch, while deterministic validation, graph execution, and fixed RCA inference constrain how the patch can affect the final ranking.

trigger a rule-based fallback. The same principle is used after GNN scoring: the evidence-bound reranker can reorder only the GNN top-k candidates when supported by explicit evidence. Thus, LLM reasoning is restricted to validated patch planning and evidence checking, while root-cause scoring remains assigned to a fixed graph model.

3 Method

Plan2RCA is built around a compiler-style agent architecture for adaptive RCA over dynamic topologies. The main design principle is to separate topology-transition interpretation from root cause scoring, so that adaptation occurs through graph construction rather than through unconstrained diagnosis. The LLM agent converts observability evidence into an executable graph patch, deterministic components validate and execute the patch, and a fixed RCA model performs ranking on the patched graph. This structure allows the LLM to contribute semantic planning capacity without allowing free-form root cause generation. Figure 1 summarizes the architecture.

3.1 Inputs and Evidence Builder

Plan2RCA takes three inputs: the reference topology G_0, the observed runtime topology G_{obs}, and telemetry evidence X. The evidence builder maps these inputs into a compact evidence object:

$$E = \mathcal{B}(G_0, G_{\text{obs}}, X). \tag{3}$$

The evidence object contains topology-transition summaries, endpoint changes, pod identity changes, trace-route mismatches, metric anomalies, and selected log signals. It is intentionally not a full telemetry dump. Its role is to expose the evidence needed for graph-patch planning while removing irrelevant operational noise. It also defines the grounding boundary for the LLM agent: every planned graph patch must be supported by evidence contained in E.

This design is important for dynamic-topology RCA because the observed runtime topology may combine benign platform behavior and fault-inducing changes. For example, autoscaling and canary deployment may be normal operational events, whereas selector redirection and empty endpoints may directly encode the failure mechanism. The evidence builder therefore represents topology change as structured diagnostic evidence rather than as a simple graph snapshot.

3.2 LLM Graph Patch Planner

The LLM graph patch planner formulates topology adaptation as a constrained evidence-to-patch compilation task. Given E, the planner emits a candidate GraphPatchPlan:

$$P = \pi_{\text{LLM}}(E). \tag{4}$$

The patch plan is not a diagnosis and does not contain the final root cause label. Instead, it specifies graph edits that transform G_0 into an RCA graph while preserving the semantics of the observed topology transition.

GraphPatchPlan is a typed intermediate representation built from the compact action space in

Table 1. Each patch links a topology transition to concrete graph edits and the evidence that supports them, so the edits can be validated before RCA inference. The planner therefore acts as a constrained compiler from observability evidence to graph construction instructions, preserving transition semantics that would be lost by directly replacing G_0 with G_{obs}.

3.3 Patch Validation and Safety Control

The candidate patch plan is accepted only after deterministic validation. We denote the validated patch plan by $\tilde{P} = \mathcal{V}(P, E, G_0, G_{\text{obs}})$. The validator enforces four constraints. First, the output must satisfy the GraphPatchPlan schema. Second, every operation must belong to the whitelisted DSL. Third, referenced nodes and edges must exist in G_0, appear in G_{obs}, or be legally creatable by the executor. Fourth, each operation must be grounded in explicit evidence from E.

This validation layer is the main hallucination-control mechanism in Plan2RCA. It prevents the LLM agent from inventing dependencies, editing nonexistent graph elements, or introducing unsupported topology changes. If validation fails, the system falls back to a deterministic rule planner. The fallback keeps the pipeline reproducible and provides a controlled comparison between rule-based and LLM-based graph patching under the same executor.

3.4 Dynamic Graph Executor

The dynamic graph executor applies the validated graph patch to the reference topology:

$$G^* = \text{Execute}(G_0, \tilde{P}). \tag{5}$$

The executor does not merely synchronize G_0 with G_{obs}. It materializes topology transitions as typed causal evidence in the RCA graph. Pod drift is represented through identity alignment, selector redirection through runtime endpoint contradiction, empty endpoints through blackhole routing, and canary deployment through version-level graph decomposition.

Category	Representative actions
Node	Add replicas, align pod identities, split versions, add blackhole nodes.
Edge	Redirect endpoints, replace bindings, add trace-mismatch edges, split traffic.
Weight	Adjust dependency, anomaly, topology-confidence, and change weights.

Table 1: Compact action space of GraphPatchPlan.

The executor also updates edge features and edge weights. For instance, a redirect operation can reduce confidence in the declared dependency while increasing the weight of the observed runtime path. A split operation can distribute traffic across version-specific nodes, which prevents service-level aggregation from hiding version-local anomalies. The resulting graph G^* preserves the stable dependency structure of G_0 while incorporating the diagnostic evidence observed in G_{obs}.

3.5 RCA Inference and Reranking

RCA inference is performed on G^* using a fixed-configuration GNN scorer. The GNN receives the same model configuration across all graph variants, so performance differences reflect graph construction rather than hyperparameter tuning. Weighted-GraphSAGE and EdgeAwareGAT are used as representative small graph models because they can consume edge weights and topology-change features without requiring free-form language reasoning. These choices build on graph neural architectures for node representation learning (Kipf and Welling, 2016; Hamilton et al., 2017; Veličković et al., 2017), while differing from temporal graph models that update representations over graph streams (Pareja et al., 2020; Rossi et al., 2020).

The GNN produces a ranked list of candidate root causes. The evidence-bound reranker then verifies only the top-k candidates. It may reorder candidates when topology, metric, trace, log, or graph-patch evidence supports the change, but it cannot introduce arbitrary new root causes. If the evidence is insufficient, the reranker preserves the GNN order. This final stage uses the LLM agent as a constrained verifier for patch-based RCA rather than as an open-ended diagnostic generator.

4 Experiment

This section is organized into four main parts. In Sections 4.1–4.3, we describe the experimental settings, including the datasets, dynamic-topology scenarios, graph-construction baselines, and evaluation metrics. Section 4.4 reports the main results and further discusses graph-patch validity and scenario-level behavior as diagnostic views of the same evaluation. Sections 4.5–4.6 then present ablation and sensitivity analyses to clarify the contribution of graph construction, LLM-planned patches, evidence-bound verification, and key design parameters.

Scenario	Topology transition	Patch semantics
T1 HPA	Replica scaling	Add runtime replicas and rebalance endpoint or traffic weights.
T2 Pod drift	Pod replacement	Align pod identities while preserving service-level semantics.
T3 Selector poison	Redirected or empty endpoint	Encode endpoint contradiction, trace mismatch, or blackhole routing.
T4 Canary	Version split	Create version nodes and split traffic or anomaly evidence.

Table 2: Dynamic-topology scenarios and their corresponding graph-patch operations.

4.1 Dataset

The evaluation data consist of two complementary parts: conventional static-topology RCA data and newly generated dynamic-topology fault-injection data. The static component includes the public AIOps Challenge dataset and a private project-management platform dataset. These cases represent standard RCA settings in which the service dependency graph is treated as fixed during diagnosis.

The dynamic component is designed to evaluate RCA under runtime topology changes. We inject faults into two microservice benchmarks, Online Boutique and TrainTicket, following prior benchmark and empirical studies of microservice systems (Gan et al., 2019; Zhou et al., 2018), and collect observability evidence before, during, and after each incident. The collected evidence includes metrics, logs, and distributed traces, where tracing follows the general observability model introduced by large-scale tracing infrastructures (Sigelman et al., 2010). The injected incidents cover four scenario families: HPA-driven replica scaling, pod reconstruction and node drift, service selector poisoning, and canary release. These scenarios are summarized in Table 2. Together, they exercise node addition, pod identity alignment, endpoint contradiction, blackhole routing, version-level decomposition, and edge reweighting. Details of the experimental platform and fault-injection protocols are provided in Appendix A and B.

4.2 Baselines

We retain six representative baselines from the original comparison set, covering multimodal RCA, holistic dependency reasoning, and LLM-agent diagnosis. **Eadro** jointly models intra-service behavior and inter-service dependencies from KPIs, logs, and traces (Lee et al., 2023). **HolisticRCA** standardizes heterogeneous observability data and reasons over service dependencies to identify root causes across cloud-native failure scenarios (Han et al., 2024). **Nezha** performs interpretable fine-grained RCA over multimodal observability data (Yu et al., 2023). **TAMO** is a tool-assisted LLM-agent framework that uses multimodal alignment and model tools for fine-grained cloud RCA (Zhang et al., 2025). **RCAgent** is an autonomous tool-augmented LLM agent that iteratively collects observability evidence and synthesizes a root-cause diagnosis (Wang et al., 2024). **mABC** uses multi-agent collaboration for RCA in microservice architectures (Zhang et al., 2024).

4.3 Evaluation Metrics

We evaluate RCA as a ranked retrieval problem over candidate root-cause entities. For an incident i, let r_i denote the rank of the ground-truth root cause in the predicted list, and let N denote the number of evaluated incidents. Accuracy at k measures whether the ground-truth root cause appears within the top-k predictions:

$$\text{Acc@}k = \frac{1}{N}\sum_{i=1}^{N}\mathbb{I}(r_i \le k). \tag{6}$$

We report Acc@1, Acc@3, and Acc@5 to evaluate strict top prediction quality and broader candidate retention. We also report mean reciprocal rank:

$$\text{MRR} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{r_i}. \tag{7}$$

MRR rewards methods that place the correct root cause closer to the top of the ranking. For all reported metrics, higher values indicate better RCA performance.

4.4 Main Result

Tables 3 and 4 report the main comparison on static-topology and dynamic-topology RCA benchmarks, respectively. The static datasets provide a reference setting in which topology adaptation is not required, while the dynamic benchmarks evaluate whether a method can exploit runtime topology changes. On AIOps Challenge 2022, TAMO achieves the best scores across all reported metrics, while Patch2RCA ranks second on Acc@1, Acc@5, and MRR. On the Project Management Platform dataset, TAMO leads on Acc@1, Acc@5, and MRR, whereas Patch2RCA obtains the best Acc@3 score of 0.811 and remains second on the other three metrics. These results show that

Dataset	Method	Acc@1	Acc@3	Acc@5	MRR
AIOps Challenge	HolisticRCA	0.656	0.763	0.817	0.735
	Eadro	0.136	0.370	0.457	0.320
	TAMO	**0.719**	**0.821**	**0.888**	**0.791**
	Nezha	0.523	0.807	0.830	0.667
	RCAgent	0.221	0.284	0.303	0.337
	mABC	0.342	0.421	0.445	0.448
	Patch2RCA	0.692	0.741	0.851	0.755
Project Platform	HolisticRCA	0.611	0.750	0.778	0.702
	Eadro	0.406	0.563	0.719	0.541
	TAMO	**0.722**	0.806	**0.861**	**0.786**
	Nezha	0.495	0.632	0.714	0.606
	RCAgent	0.248	0.316	0.351	0.364
	mABC	0.369	0.452	0.493	0.475
	Patch2RCA	0.703	**0.811**	0.842	0.771

Table 3: Main root cause analysis results on two datasets.

Dataset	Method	Acc@1	Acc@3	Acc@5	MRR
Online Boutique	HolisticRCA	0.431	0.493	0.566	0.527
	Eadro	0.312	0.392	0.519	0.433
	TAMO	0.523	0.630	0.671	0.617
	Nezha	0.268	0.377	0.452	0.398
	RCAgent	0.589	0.716	0.784	0.684
	mABC	0.313	0.410	0.432	0.428
	Patch2RCA	**0.871**	**0.952**	**0.984**	**0.905**
TrainTicket	HolisticRCA	0.401	0.442	0.497	0.459
	Eadro	0.118	0.281	0.346	0.237
	TAMO	0.541	0.576	0.734	0.606
	Nezha	0.311	0.389	0.502	0.397
	RCAgent	0.322	0.417	0.478	0.405
	mABC	0.288	0.391	0.371	0.371
	Patch2RCA	**0.823**	**0.918**	**0.961**	**0.870**

Table 4: Main root cause analysis results on dynamic-topology benchmarks.

Patch2RCA is competitive in static settings, but the benefit of topology adaptation is most visible when runtime topology changes are available as incident evidence.

The advantage becomes larger on the dynamic benchmarks. As shown in Table 4, Patch2RCA reaches Acc@1 scores of 0.871 on Online Boutique and 0.823 on TrainTicket. Against the strongest non-Patch2RCA baseline, the Acc@1 gains are 28.2 percentage points on both benchmarks. Patch2RCA also achieves Acc@5 scores of 0.984 and 0.961, improving over the strongest baseline by 20.0 and 22.7 percentage points, respectively. The gap between static and dynamic settings supports the central claim of this paper: when topology changes are part of the incident evidence, RCA benefits from an executable graph patch rather than from free-form LLM diagnosis or a direct replacement of the topology snapshot.

Graph-patch validity. Figure 2 evaluates whether generated graph patches are valid and executable before RCA inference. Both planners reach 1.000 operation validity and 1.000 execution success, showing that the validation and executor layers reliably filter invalid edits. The advantage of the LLM planner is mainly semantic: it improves node-edit F1 from 0.446 to 0.984, raises version-split recall from 0.667 to 1.000, and improves weight accuracy from 0.853 to 1.000. Alignment remains identical at 0.974, and redirect recall improves slightly from 0.989 to 1.000. These results indicate that rules are sufficient for basic executable patches, while the LLM planner better handles topology changes that require node creation, service decomposition, and traffic-weight estimation. Detailed metric definitions are provided in Appendix F.

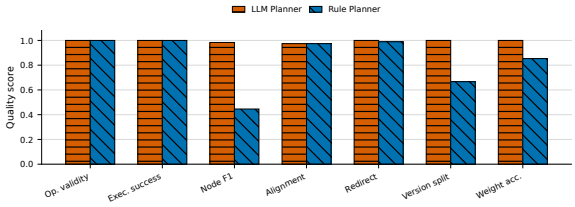
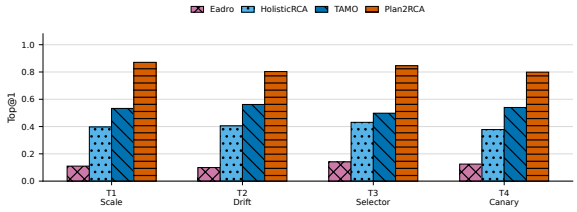
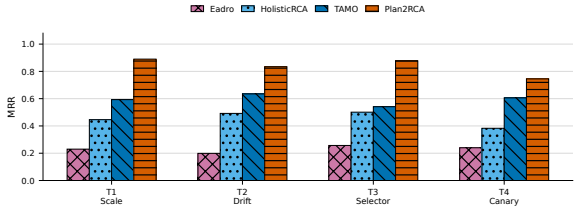


Figure 2: Graph-patch validity of the LLM planner. The metrics summarize whether generated patch operations are legal, executable, topology-aware, and faithful to runtime evidence.

Scenario-level behavior. Figure 3 reports scenario-level Top@1 and MRR across the four dynamic-topology fault families. Plan2RCA is consistently the strongest method, with Top@1 scores of 0.871, 0.803, 0.846, and 0.799 on T1–T4, respectively. Compared with the strongest non-Plan2RCA baseline in each scenario, the Top@1 gains are 33.8 percentage points for T1 autoscaling, 24.1 points for T2 pod drift, 34.8 points for T3 selector poisoning, and 25.9 points for T4 canary release. The MRR results show the same trend: Plan2RCA improves over the strongest baseline by 29.7, 19.9, 33.6, and 13.9 points on T1–T4. The largest gains appear in T1 and T3, where dynamic topology evidence directly changes replica structure or exposes runtime endpoint contradictions. T4 remains the hardest case: although Plan2RCA still achieves the best Top@1 and MRR, its MRR drops to 0.745, indicating that version splitting and traffic-weight estimation introduce more ranking ambiguity. These results support the need for scenario-specific graph-patch semantics rather than a single static dependency representation.



(a) Top@1 by scenario.



(b) MRR by scenario.

Figure 3: Scenario-level RCA behavior on dynamic-topology faults of TrainTicket.

Model	Acc@1	Acc@3	Acc@5	MRR
Full	0.871	0.952	0.984	0.905
w/o reranker	0.839	0.936	0.968	0.876
w/o LLM planner	0.807	0.919	0.952	0.855
w/o full patch semantics	0.758	0.887	0.935	0.822
w/o graph patch	0.597	0.823	0.887	0.715
w/o runtime topology	0.371	0.694	0.790	0.573

Table 5: Ablation study over graph construction and reranking components on dynamic-topology fault injection cases of Online Boutique.

4.5 Ablation Study

Table 5 isolates the effect of each graph-construction choice under the same Weighted-GraphSAGE configuration. All rows use the same model and training setup; therefore, the differences mainly reflect how topology evidence is represented before RCA inference.

Runtime topology. Using runtime topology as an unstructured input raises Acc@1 from 0.371 to 0.597 and MRR from 0.573 to 0.715. This confirms that incident-time topology is useful, but the gap to the full model shows that a raw observed snapshot is still insufficient.

Graph-patch semantics. Typed patch semantics provide the largest structural gain. Removing graph patches reduces Acc@1 from 0.871 to 0.597, while removing full patch semantics reduces it to 0.758, indicating that node, edge, and weight updates should be modeled jointly as executable transition evidence.

LLM planning. Replacing the LLM planner with deterministic rules lowers Acc@1 from 0.839 to 0.807 before reranking, and the full pipeline improves over *w/o LLM planner* by 6.4 points in Acc@1 and 5.0 points in MRR. The LLM agent is therefore useful when it is constrained to produce validated graph patches rather than free-form diagnoses.

Evidence-bound reranking. Removing the evidence-bound reranker decreases Acc@1 from 0.871 to 0.839 and MRR from 0.905 to 0.876. Its role is not to replace graph inference, but to correct ambiguous top-k rankings when the evidence pack provides explicit support; overall, the full pipeline improves over *w/o runtime topology* by 50.0 points in Acc@1 and 33.2 points in MRR.

4.6 Parameter Sensitivity

Plan2RCA remains stable under the tested parameter settings. Across 62 cases per setting, the main configuration obtains 0.8298 Top@1, 0.9460 Top@3, 0.9800 Top@5, and 0.8365 MRR. Removing the target-service prior reduces Top@1 to 0.8015 and MRR to 0.8152, which shows that the prior provides a useful but bounded gain when it is available as an observable alert cue. The learning rate, pairwise ranking coefficient, and topology-confidence coefficient are also stable around the main setting. The best values appear at a learning rate of 0.003, a pairwise ranking coefficient of 1.0, and a topology-confidence coefficient of 0.25; moving away from these values lowers Top@1 by at most 3.88 percentage points. Figure 4 shows the core sensitivity trends, while auxiliary reranker diagnostics are reported in Appendix G and Figure 13.

5 Related Work

Root cause analysis. Root cause analysis in cloud and microservice systems often relies on dependency graphs, causal propagation, statistical correlation, or multimodal telemetry fusion. Early graph-based methods construct impact graphs or attributed dependency graphs to rank suspicious services from anomalous metrics and calls (Wang et al., 2018; Wu et al., 2020). Subsequent systems improve scalability and evidence coverage by analyzing dynamically constructed call graphs, pruning propagation chains, or jointly modeling traces, logs, and KPIs (Liu et al., 2021; Lee et al., 2023). Other lines of work use trace-spectrum analysis, holistic observability fusion, interpretable multimodal reasoning, causal discovery, causal-structure

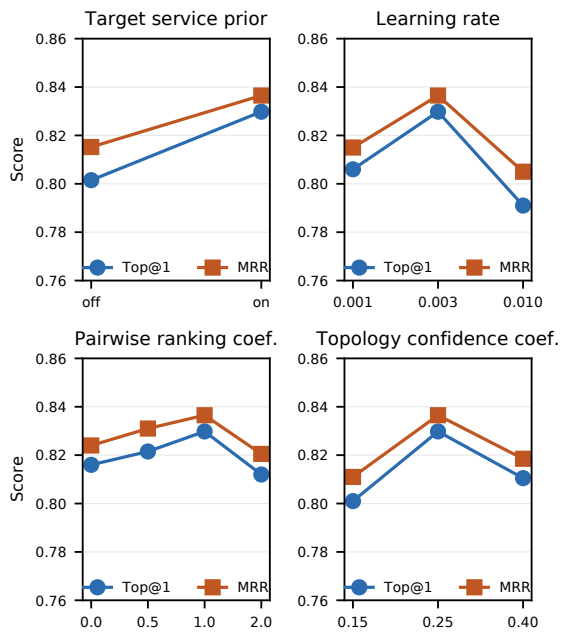


Figure 4: Core parameter sensitivity of Plan2RCA. The figure summarizes Top@1 and MRR under changes to the target-service prior, learning rate, pairwise ranking coefficient, and topology-confidence coefficient.

learning, event graphs, or ML-driven performance debugging (Yu et al., 2021; Han et al., 2024; Yu et al., 2023; Zhang et al., 2023; Ikram et al., 2022; Zheng et al., 2024; Wang et al., 2021; Gan et al., 2021). These methods show that graph structure is central to microservice RCA, but they usually treat topology as a given dependency graph, a reconstructed call graph, or an observed snapshot. Plan2RCA is complementary to this line of work: it focuses on incidents in which the topology transition itself is diagnostic evidence, and represents that transition as an executable graph patch before downstream RCA inference.

LLM agents for RCA. Recent work applies LLM agents to cloud RCA by allowing them to collect evidence, call diagnostic tools, and synthesize root-cause explanations. RCAgent uses a tool-augmented autonomous agent for industrial cloud RCA, TAMO introduces a tool-assisted LLM agent that aligns multimodal observations and invokes specialized localization and classification tools, and mABC studies multi-agent collaboration for microservice RCA (Wang et al., 2024; Zhang et al., 2025, 2024). This direction is related to broader LLM-agent and reasoning techniques, including retrieval-augmented generation, chain-of-thought reasoning, self-consistency, reasoning-and-acting, tool use, reflection, and tree-structured deliberation (Lewis et al., 2020; Wei et al., 2022; Wang et al., 2022; Yao et al., 2022; Schick et al., 2023; Shinn et al., 2023; Yao et al., 2023). These studies demonstrate the potential of LLM agents for heterogeneous evidence integration and interactive diagnosis. However, LLM-agent RCA remains difficult to validate when the agent directly generates diagnoses or explanations from long observability contexts. Plan2RCA instead constrains the agent role to graph-patch planning: the LLM produces a schema-checked, whitelisted, evidence-grounded patch that must pass deterministic validation and execution before it can influence the GNN scorer and evidence-bound reranker.

6 Conclusion

We introduced Plan2RCA, a topology-adaptive framework that constrains LLM reasoning through executable graph patches for microservice RCA. Instead of asking the LLM agent to produce a free-form diagnosis, Plan2RCA uses it to compile compact observability evidence into a schema-constrained GraphPatchPlan that is validated, executed on the reference topology, and scored by a fixed-configuration GNN. An evidence-bound reranker further limits the final LLM intervention to supported reorderings of the top-k candidates. Experiments on Online Boutique and TrainTicket show that this design improves RCA under autoscaling, pod drift, selector poisoning, endpoint removal, and canary release. The ablation study shows that runtime topology evidence, typed patch semantics, LLM-based patch planning, and evidence-bound reranking all contribute to the final ranking quality. These findings indicate that dynamic-topology failures should be modeled as topology-transition evidence rather than as static-graph noise or raw incident snapshots. More broadly, the results support a practical design principle for LLM agents in operational diagnosis: the agent should generate validated intermediate programs that reshape structured model inputs, rather than directly issuing unverifiable root-cause claims.

Limitations

Although Plan2RCA improves RCA under dynamic topology changes, the current study has several limitations.

Benchmark scope and external validity. The evaluation is conducted on controlled deployments of Online Boutique and TrainTicket. These two benchmarks are complementary in scale: Online Boutique provides a compact service graph, whereas TrainTicket contains a larger and deeper microservice topology. However, both systems are still benchmark applications deployed in an instrumented Kubernetes environment. They cannot fully reproduce the heterogeneity of production clusters, such as multi-tenant interference, partial instrumentation, long-lived stateful dependencies, cross-cluster routing, asynchronous messaging, cloud-provider load balancers, or operator interventions during incident mitigation. Therefore, the reported gains should be interpreted as evidence that executable topology-transition modeling is useful under controlled dynamic-topology faults, rather than as a guarantee of the same absolute accuracy in every production setting.

Fault-model coverage. The dynamic benchmark focuses on four topology-changing fault families: HPA-driven replica scaling, pod reconstruction and drift, service selector poisoning or endpoint removal, and canary release. These scenarios cover important classes of node addition, endpoint contradiction, blackhole routing, identity alignment, version splitting, and edge reweighting. Nevertheless, they do not exhaust the space of cloud-native failures. We do not yet evaluate multi-root-cause incidents, concurrent topology changes across several services, rolling upgrades without explicit canary resources, DNS or service-discovery cache faults, network-policy misconfigurations, storage-layer failures, message-queue backpressure, database contention, or failures outside the monitored service graph. The current candidate set also assumes that the root cause is represented as a service, pod, version-specific node, or virtual blackhole node. If the true cause lies outside this candidate space, Plan2RCA can only rank the closest observed symptom.

Dependence on observable topology and telemetry. Plan2RCA assumes that useful topology snapshots and telemetry summaries are available around the incident window. In the experiments, the collector records Kubernetes resources, metrics, logs, traces, and verification artifacts before, during, and after each fault. Production systems may provide weaker evidence: traces may be sampled, logs may be dropped or redacted, endpoint updates may lag behind the actual routing state, and labels or service names may be inconsistent across telemetry backends. This limitation is especially relevant to selector poisoning and endpoint-removal cases. In these cases, the declared service topology can remain visually plausible while runtime routing is wrong or empty, so trace-route evidence becomes crucial. If trace mismatch evidence is missing or delayed, a graph patch may underrepresent the true routing contradiction.

Topology-change detection uncertainty. The current topology verifier does not detect every transition with equal reliability. In particular, selector-related faults are sparse at the graph-difference level because the number of services and pods may remain unchanged even though endpoint semantics are corrupted. Some T3 cases are therefore marked as weak or no visible topology change, while T1, T2, and T4 are more consistently observable. The framework partially addresses this by using traces, endpoint summaries, and blackhole nodes, but the remaining cases show that dynamic-topology RCA should not rely solely on structural diffs. More robust endpoint probing, active reachability tests, and lower-latency service-discovery snapshots would be needed for production-grade coverage.

Expressiveness of the graph-patch language. The restricted GraphPatchPlan language is intentionally small: it allows the LLM planner to add or align nodes, redirect or split edges, insert trace-mismatch or blackhole evidence, and reweight graph relations. This restriction is useful for validation and reproducibility, but it also limits expressiveness. More complex topology transitions may require operations that are not yet implemented, such as temporal edge intervals, probabilistic routing rules, queue-based dependencies, cascading autoscaler actions, cross-namespace service aliases, or simultaneous updates to multiple control-plane resources. In addition, deterministic validation can ensure that a patch is well formed, executable, and grounded in the compact evidence, but it cannot prove that the chosen causal role or edge weight is semantically optimal. A valid patch may still be incomplete or may encode an imprecise causal interpretation.

LLM dependence and reranking constraints. Although Plan2RCA does not allow the LLM to directly output an unconstrained root cause, it still depends on an LLM for graph-patch planning and evidence-bound reranking. Its behavior may vary with the LLM model version, prompt template, evidence compression strategy, decoding settings, API availability, and latency or cost constraints. The rule-based fallback improves reproducibility when LLM output is invalid, but fallback rules are necessarily less adaptive than semantic planning. The reranker is also deliberately restricted to the GNN top-k candidates, except for explicitly introduced special nodes such as version or blackhole nodes. This design reduces hallucinated diagnoses, but it also means that the reranker cannot recover the correct root cause if the GNN fails to include it in the candidate set. Finally, the target-service prior must be interpreted carefully: it is valid when it corresponds to an observable alert entry point, but it should be disabled or separately reported if the evaluation protocol treats it as hidden label information.

Training stability and split sensitivity. The experiments use a fixed GNN configuration to isolate the effect of graph construction, and the sensitivity study varies only a small set of predefined parameters. This design avoids per-method hyperparameter search, but it also leaves residual uncertainty about training variance. The dynamic datasets are stratified by scenario, yet the test partitions remain small relative to the diversity of possible topology transitions. Additional random seeds, repeated train–validation–test splits, leave-scenario-out evaluation, leave-service-out evaluation, and cross-benchmark transfer experiments would provide stronger evidence of robustness. This is particularly important for pairwise ranking loss and topology-confidence weighting, whose optimal settings may depend on graph size, candidate-set size, and the relative strength of metric versus topology evidence.

Metric and explanation limitations. The main evaluation reports Acc@1, Acc@3, Acc@5, and MRR. These metrics capture ranking quality, but they do not fully measure operational usefulness. In real incident response, engineers also care about explanation faithfulness, time to mitigation, rollback recommendations, confidence calibration, and whether the model distinguishes the initiating fault from downstream propagation. The graph-patch validity metrics provide an execution-level view of whether generated operations are legal and scenario-consistent, but they still do not replace a human-centered evaluation of diagnostic explanations. Future work should include operator studies or replay-based incident-response evaluations to measure whether graph patches reduce debugging effort.

Deployment cost and instrumentation assumptions. The current pipeline relies on relatively rich observability, including metrics, logs, traces, Kubernetes snapshots, and service-mesh routing metadata. Collecting and storing this evidence may be expensive in large production environments, especially when trace sampling is high or when the service graph is much larger than the benchmarks. LLM calls also add latency and operational cost. While compact evidence construction reduces the amount of raw telemetry sent to the LLM, the end-to-end system still assumes that the monitoring stack can provide sufficiently complete and timely summaries. Deploying Plan2RCA in resource-constrained or privacy-sensitive environments would require stronger evidence compression, caching, redaction, and possibly on-premise or smaller local planning models.

Privacy and security considerations. The experimental artifacts are sanitized benchmark data, but real observability streams may contain user identifiers, request payload fragments, internal hostnames, secrets accidentally printed in logs, or business-sensitive routing information. Because Plan2RCA builds compact evidence packages for LLM planning and reranking, a practical deployment should include strict redaction, access control, audit logging, and data-retention policies. The current paper focuses on RCA accuracy and graph-patch validity; it does not yet evaluate privacy-preserving evidence construction or adversarial manipulation of telemetry and topology evidence.

Overall, these limitations suggest that Plan2RCA should be viewed as a step toward auditable LLM-assisted RCA rather than a complete production solution. The most important future directions are broader fault coverage, stronger handling of weakly observable selector and blackhole cases, multi-seed and cross-benchmark validation, richer graph-patch operations, prior-free reporting, and deployment-oriented studies of cost, latency, privacy, and operator usefulness.

References

Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade. *Queue*, 14(1):70–93.

Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: practical and scalable ml-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 135–151.

Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, and 1 others. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, pages 3–18.

Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30.

Yongqi Han, Qingfeng Du, Ying Huang, Pengsheng Li, Xiaonan Shi, Jiaqi Wu, Pei Fang, Fulong Tian, and Cheng He. 2024. Holistic root cause analysis for failures in cloud-native systems through observability data. *IEEE Transactions on Services Computing*, 17(6):3789–3802.

Azam Ikram, Sarthak Chakraborty, Subrata Mitra, Shiv Saini, Saurabh Bagchi, and Murat Kocaoglu. 2022. Root cause analysis of failures in microservices through causal discovery. *Advances in Neural Information Processing Systems*, 35:31158–31170.

Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.

Cheryl Lee, Tianyi Yang, Zhuangbin Chen, Yuxin Su, and Michael R Lyu. 2023. Eadro: An end-to-end troubleshooting framework for microservices on multi-source data. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1750–1762. IEEE.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, and 1 others. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474.

Dewei Liu, Chuan He, Xin Peng, Fan Lin, Chenxi Zhang, Shengfang Gong, Ziang Li, Jiayu Ou, and Zheshun Wu. 2021. Microhecl: High-efficient root cause localization in large-scale microservice systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 338–347. IEEE.

Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao Schardl, and Charles Leiserson. 2020. Evolvegcn: Evolving graph convolutional networks for dynamic graphs. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 5363–5370.

Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. 2020. Temporal graph networks for deep learning on dynamic graphs. *arXiv preprint arXiv:2006.10637*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652.

Benjamin H Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903*.

Hanzhang Wang, Zhengkai Wu, Huai Jiang, Yichao Huang, Jiamu Wang, Selcuk Kopru, and Tao Xie. 2021. 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.

Ping Wang, Jingmin Xu, Meng Ma, Weilan Lin, Disheng Pan, Yuan Wang, and Pengfei Chen. 2018. Cloudranger: Root cause identification for cloud native systems. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 492–502. IEEE.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Zefan Wang, Zichuan Liu, Yingying Zhang, Aoxiao Zhong, Jihong Wang, Fengbin Yin, Lunting Fan, Lingfei Wu, and Qingsong Wen. 2024. Rcagent:

Cloud root cause analysis by autonomous agents with tool-augmented large language models. In *Proceedings of the 33rd ACM international conference on information and knowledge management*, pages 4966–4974.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. 2020. Microrca: Root cause localization of performance issues in microservices. In *IEEE/IFIP Network Operations and Management Symposium (NOMS)*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. 2021. Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments. In *Proceedings of the Web Conference 2021*, pages 3087–3098.

Guangba Yu, Pengfei Chen, Yufeng Li, Hongyang Chen, Xiaoyun Li, and Zibin Zheng. 2023. Nezha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data. In *Proceedings of the 31st ACM joint European software engineering conference and symposium on the foundations of software engineering*, pages 553–565.

Shenglin Zhang, Pengxiang Jin, Zihan Lin, Yongqian Sun, Bicheng Zhang, Sibo Xia, Zhengdan Li, Zhenyu Zhong, Minghua Ma, Wa Jin, and 1 others. 2023. Robust failure diagnosis of microservice system through multimodal data. *IEEE Transactions on Services Computing*, 16(6):3851–3864.

Wei Zhang, Hongcheng Guo, Jian Yang, Zhoujin Tian, Yi Zhang, Yan Chaoran, Zhoujun Li, Tongliang Li, Xu Shi, Liangfan Zheng, and 1 others. 2024. mabc: multi-agent blockchain-inspired collaboration for root cause analysis in micro-services architecture. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 4017–4033.

Xiao Zhang, Qi Wang, Mingyi Li, Yuan Yuan, Mengbai Xiao, Fuzhen Zhuang, and Dongxiao Yu. 2025. Tamo: Fine-grained root cause analysis via tool-assisted llm agent with multi-modality observation data in cloud-native systems. *IEEE Transactions on Services Computing*, 18(6):4221–4233.

Lecheng Zheng, Zhengzhang Chen, Jingrui He, and Haifeng Chen. 2024. Mulan: Multi-modal causal structure learning and root cause analysis for microservice systems. In *Proceedings of the ACM Web Conference 2024*, pages 4107–4116.

Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2):243–260.

A Experimental System and Data-Collection Platform

A.1 Platform Overview

The experimental platform is built on a local Kubernetes-in-Docker cluster and is designed to generate dynamic-topology incident data in a repeatable, recoverable, and fully observable manner. The platform contains three groups of components: benchmark microservice applications, a unified observability stack, and fault-injection controllers. The dynamic benchmark suite includes Online Boutique and TrainTicket. For every experiment, the platform stores topology snapshots before, during, and after the incident, together with metrics, logs, distributed traces, Kubernetes events, and manually specified ground-truth labels. To preserve operational privacy, the appendix reports only benchmark-level service names, abstracted topology relations, and aggregate statistics; local paths, hostnames, user identifiers, and raw log payloads are not reported.

Table 6: Experimental infrastructure.

Layer	Stack and role
Runtime	Kind Kubernetes v1.30.0 runs the benchmark applications and exposes pod-, node-, endpoint-, and service-level topology changes.
Routing	Istio v1.29.2, VirtualService, and DestinationRule provide sidecar telemetry, canary routing, delay injection, and abort injection.
Observability	Prometheus, kube-prometheus-stack, Loki, Promtail, Tempo, and Grafana collect metrics, logs, traces, snapshots, and monitoring views.
Intervention and validation	Chaos Mesh injects resource and pod faults; the topology verifier checks topology-change evidence and graph annotations.

A.2 Benchmark Service Dependency Graphs

Figure 5 shows the Online Boutique service-level dependency structure used in the controlled dynamic-topology experiments. The candidate root-cause services are primarily productcatalogservice, cartservice, recommendationservice, checkoutservice, shippingservice, and currencyservice. Other services act as ingress, downstream dependencies, or background services involved in fault propagation.

TrainTicket is processed through the same reference-topology, observed-topology, and graph-patch abstraction. Its monitored graph is larger and structurally deeper than Online Boutique. Figure 6 shows a privacy-preserving service-level topology used for graph construction. The figure omits the common ts- prefix and -service suffix when doing so improves readability. To avoid visual clutter, the complete service-level call graph is summarized through functional dependency groups, dominant inter-group flows, and representative local calls that can be displayed without crossing group boundaries.

A.3 Telemetry and Snapshot Collection

Each experiment collects telemetry and Kubernetes objects over a unified incident window. Prometheus metrics are queried at a 5-second step with 30-second rate windows. Loki is used for full and error-filtered logs. Tempo is used to search trace identifiers by service and then retrieve full trace payloads. Kubernetes snapshots include Pods, Services, Endpoints, Deployments, Events, and HPAs. The collector uses benchmark-aware configuration templates so that the same telemetry schema is applied to both Online Boutique and TrainTicket without hard-coding benchmark-specific namespaces or deployment paths.

The Prometheus files provide request-volume, latency, resource-pressure, replica-count, availability, and response-flag signals. These metrics capture source-destination traffic and status codes, median and tail latency at the destination workload, container CPU pressure, autoscaling behavior, service reachability, and Envoy error categories. The non-metric artifacts are stored in a small set of structured files. Full logs, error-filtered logs, and per-service logs are written under logs/; full trace payloads and flattened trace summaries are written under traces/. Kubernetes snapshots record Pods, Services, Endpoints, Deployments, Events, and HPAs, including labels, selectors, endpoint addresses, replica states, pod identities, node assignments, and restart information. These artifacts provide the raw evidence used by the topology verifier and the evidence builder.

A.4 Benchmark Adaptation Notes

The TrainTicket adaptation keeps the Online Boutique data-collection protocol unchanged while extending the configuration layer to support a larger benchmark graph. The collector selects benchmark-specific service lists, graph edges, query templates, and health thresholds at runtime. This design avoids duplicating the RCA pipeline and ensures that any performance difference between benchmarks is caused by workload and topology complexity rather than by incompatible collection logic.

Table 7: Dynamic benchmark comparison. The TrainTicket deployment contains 41 services, from which 30 core business services are monitored for graph construction.

Dimension	Online Boutique	TrainTicket
Application scale	11 service nodes and 13 service-level dependency edges.	41 deployed services; 30 monitored business services and 78 known service-level dependency edges.
Implementation style	Heterogeneous services implemented in Go, Python, Java, and .NET.	Java and Spring Boot services with a larger service and storage footprint.
Stateful components	One Redis component.	Multiple MongoDB components and one MySQL component.
Call-graph depth	Maximum depth of approximately 3 service hops.	Maximum depth of approximately 6 service hops, with broader fan-out around preserve, rebook, and order workflows.
Traffic generation	Built-in benchmark load generator.	External Locust workload with 20 concurrent users and approximately 8.7 requests per second during collection.
Dynamic experiments	400 valid experiments with 100% execution success.	400 valid experiments with 100% execution success.

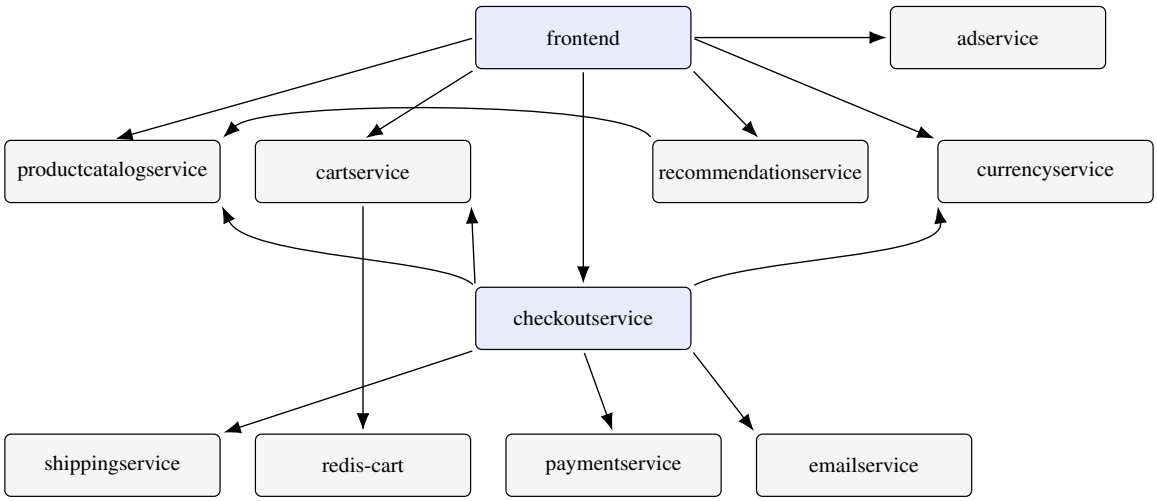


Figure 5: Online Boutique service dependency graph used by the experiments. The RCA graph uses a fixed set of service-level dependency edges while dynamic topology operations add pod, endpoint, version, mismatch, and blackhole evidence as needed.

Several engineering adjustments were required to obtain stable TrainTicket data. First, the stateful services were pinned to a MongoDB version compatible with the database driver used by the benchmark. Second, the larger number of sidecars required higher file-descriptor and inotify limits at the container-runtime level. Third, traffic generation was moved to an external Locust workload because the benchmark does not include the same built-in load generator as Online Boutique. These adjustments are infrastructure-level compatibility changes; they do not alter the graph-patch representation, the evidence schema, or the RCA inference pipeline.

B Dynamic Topology Fault-Injection Scenarios

B.1 Unified Fault-Injection Workflow

All fault scripts follow a six-phase workflow: pre-check, baseline collection, injection, observation, recovery, and post-recovery validation. Figure 7 illustrates the workflow. This design gives all fault types comparable pre/during/post windows, enabling a unified preprocessing pipeline.

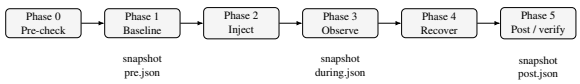


Figure 7: Unified fault-injection workflow. Each script records comparable topology snapshots and telemetry windows around the incident.

The pre-check phase validates cluster health and stores initial parameters. The baseline phase lasts 20 seconds and records the reference topology. The injection phase then creates Chaos Mesh resources, patches Kubernetes Services, or applies Istio canary resources. The observation phase lasts 120 seconds and records incident-time telemetry and topology. Finally, recovery removes injected resources, restores selectors or routing rules, and the

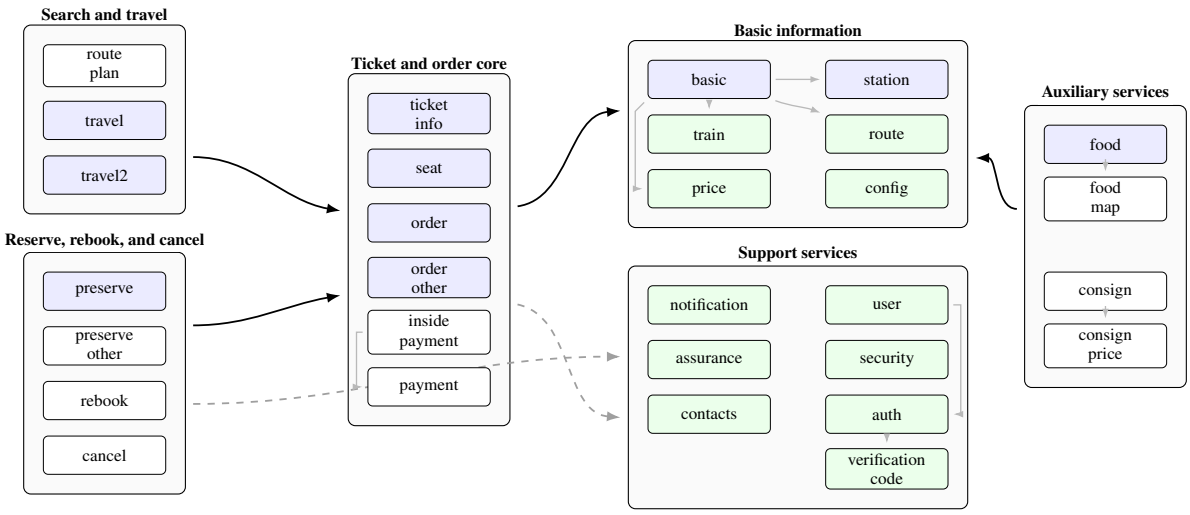


Figure 6: Condensed TrainTicket service dependency graph used by the experiments. Blue nodes denote services used as injected root-cause targets in at least one dynamic-topology scenario. The six group boxes summarize the 30 monitored business services and their 78 directed dependency edges; solid arrows denote dominant inter-group workflow dependencies, dashed arrows denote high-fan-out support dependencies, and thin gray arrows denote representative local service calls.

post-recovery phase records a 90-second validation window. The same workflow is used for Online Boutique and TrainTicket, which makes their dynamic-topology evidence directly comparable despite their different call-graph sizes.

B.2 T1: HPA Autoscaling

T1 injects CPU pressure into a target service using Chaos Mesh StressChaos. The parameter space varies cpu_workers over $\{2, 3, 4, 6\}$ and cpu_load over $[60\%, 90\%]$. The injected pressure triggers the Kubernetes Horizontal Pod Autoscaler (HPA), increasing the number of replicas and changing the service-to-pod endpoint set. A typical run expands one replica to multiple replicas during the incident and scales back after recovery. The topology event is usually an effect of CPU stress rather than the original cause of the CPU anomaly, which makes the causal direction non-trivial for RCA. In TrainTicket, this scenario is applied to travel, order, and preserve workflows.

B.3 T2: Pod Reconstruction and Node Drift

T2 uses Chaos Mesh PodChaos to kill one or two pods of the target Deployment, with repeated trials over five target services per benchmark. The Deployment controller reconstructs new pods, often with new names, UIDs, IP addresses, and possibly different worker nodes. The service-level graph may look unchanged, while endpoint identity changes at the pod level. A typical path replaces service/pod-A with service/pod-B on the same or another worker node. The model must therefore preserve service identity through pod-level alignment rather than treating the reconstructed pod as an unrelated entity.

B.4 T3: Service Selector Poisoning

T3 modifies the selector of a Kubernetes Service. The parameter space includes the redirect and empty variants, where redirect targets are selected from a small benchmark-specific service set. The redirect variant points a Service to pods of a different service, so requests declared for one business function may be served by a semantically different backend. The empty variant removes reachable endpoints and creates blackhole-like runtime behavior. T3 is particularly challenging because application pods and Deployments may remain healthy, while the semantic meaning of service-to-pod routing is corrupted. In this setting, metrics can remain weak or misleading, and trace-level route mismatch is often the most reliable evidence.

B.5 T4: Canary Release

T4 creates a v2 Deployment for the target service and uses Istio VirtualService and DestinationRule resources to split traffic between v1 and v2. The v2 traffic weight varies over $\{10, 20, 30, 40, 50\}\%$, and delay or abort faults are injected into the v2 route. The resulting topology contains version-level entities, typically service:v1 and service:v2.

Because service-level metrics aggregate v1 and v2 traffic, the failure signal may be diluted when the v2 traffic weight is low. The RCA model must therefore localize the cause at the version level, not only at the aggregate service level. TrainTicket uses the same version-split design, with readiness checks adapted to the service ports of the benchmark.

B.6 Scenario Summary

Table 8 condenses the four injected dynamic-topology settings into their diagnostic roles and reports the parallel coverage on the two dynamic benchmarks. The purpose of this summary is to make clear that the scenarios differ not only in injected fault type, but also in the graph operation required for faithful RCA.

The TrainTicket collection follows the same scenario counts as Online Boutique but increases the structural scale of the graph. The 400 TrainTicket runs cover 10 distinct target groups. The execution log reports 400 successful runs and no failed runs, with an average duration of approximately 265 seconds per experiment. The average duration by scenario is approximately 275 seconds for T1, 258 seconds for T2, 266 seconds for T3, and 257 seconds for T4.

Figure 8 complements the tabular summary by showing how each scenario changes the pre-fault graph over time. The figure emphasizes that Plan2RCA models a transition from a reference graph to an incident graph, rather than treating the incident snapshot as a complete replacement.

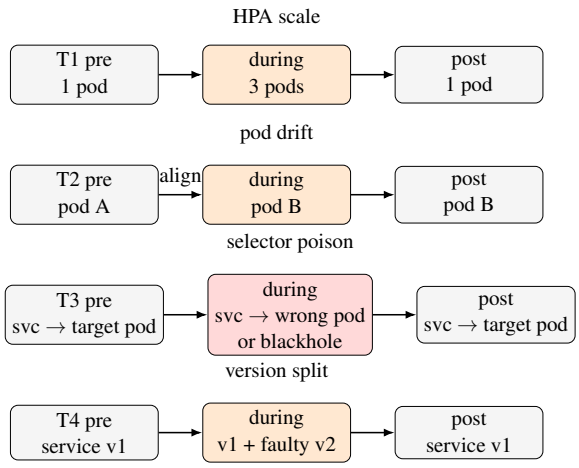


Figure 8: Schematic topology evolution for the four fault classes. The dynamic graph retains pre-fault context while adding change-event evidence.

C Dataset Statistics and Feature Construction

C.1 Dataset Overview

The dynamic-topology dataset contains two benchmark-specific collections. Online Boutique provides a compact service graph with shallow dependencies, while TrainTicket provides a larger and deeper graph with more services, more edges, and longer traces. Both collections contain 400 valid experiments under the same four scenario families. Table 9 reports the parallel dataset scale used by the appendix.

Table 10 provides a parallel view of the collected data volume. The TrainTicket values are reported as collection-level estimates derived from exported artifacts and service-graph scale. The larger size is expected because TrainTicket has 2.7× more monitored service nodes, 6.0× more service-level edges, and deeper request paths.

C.2 Services and Candidate Root Causes

The Online Boutique graph contains 11 services. The injected root-cause services cover the product catalog, cart, recommendation, checkout, shipping, and currency components. The remaining entry, downstream, cache, and background services are retained in the graph because they can still participate in fault propagation, even when they are not used as injected root causes.

The TrainTicket graph contains 30 monitored business services selected from the full benchmark deployment. The injected root-cause target groups cover order, preserve, travel, travel2, basic, seat, station, ticketinfo, food, and order-other workflows. The target distribution is intentionally non-uniform because several scenarios share central services. In the collected TrainTicket runs, order contributes 80 cases; preserve, travel, and travel2 contribute 45 cases each; basic, seat, station, and ticketinfo contribute 35 cases each; food contributes 25 cases; and order-other contributes 20 cases. Non-target services remain in the dependency graph because they are needed for propagation, trace matching, and downstream symptom modeling.

C.3 Scenario and Root-Cause Distributions

Each benchmark collection contains 100 T1 cases for HPA and node-scale changes, 75 T2 cases for pod drift, 100 T3 cases for selector redirect or endpoint removal, and 125 T4 cases for canary version splits. These counts correspond to 25.0%, 18.8%,

Table 8: Fault scenario summary across the two dynamic benchmarks.

Scenario	Online Boutique	TrainTicket	Main topology change	TrainTicket target groups
T1 HPA	100	100	Node scale and endpoint expansion.	travel, travel2, order, order-other, preserve
T2 Drift	75	75	Pod replacement and endpoint identity drift.	station, basic, order, seat, ticketinfo
T3 Selector	100	100	Endpoint redirect or endpoint removal.	basic, seat, order, ticketinfo, station
T4 Canary	125	125	Canary version split and traffic-weight change.	travel, travel2, preserve, order, food
Total	400	400	Four dynamic-topology families.	10 distinct TrainTicket root-cause target groups

Table 9: Dataset overview for the two dynamic-topology benchmark collections.

Dimension	Online Boutique	TrainTicket
Valid experiments	400	400
Execution success	400/400 (100%)	400/400 (100%)
Collection time	43.1 hours	42.8 hours
Service nodes	11	30 monitored business services
Dependency edges	13	78
Root-cause target groups	6	10
Node tensor	$[60, 11, 12]$	$[60, 30, 12]$
Edge tensor	$[60, 13, 4]$	$[60, 78, 4]$
Topology vector	$[7]$	$[7]$
Single-sample scalar count	11,047	35,527

25.0%, and 31.2% of the valid experiments, respectively. The distribution intentionally gives slightly more coverage to canary faults because version-level RCA is the most structurally different setting, while still keeping HPA and selector faults at comparable scale. The two collections therefore share the same scenario-level design but differ substantially in service-graph scale and call-chain complexity.

C.4 Topology-Change Type Distribution

The Online Boutique verifier marks 369 out of 400 experiments as topology-changing, corresponding to a detection rate of 92.2%. The most frequent detected type is edge_redirect, with 143 cases, followed by 126 mixed cases and 99 node_scale cases. There are 31 no_change cases, mostly from T3 parameter combinations that do not produce a visible topology diff, and one standalone edge_split case, because most version-split cases are grouped into the mixed category. This distribution shows that dynamic-topology RCA is not dominated by a single mutation type: selector-related edge semantics, mixed canary or deployment changes, and replica-scale changes all occur frequently enough to stress different parts of the graph-patch planner.

Table 11: Topology-change detection summary by scenario. TrainTicket rates are collection-level estimates from the current exported artifacts.

Scenario	Online Boutique	TrainTicket
T1 HPA	99%	approximately 98%
T2 Drift	100%	approximately 100%
T3 Selector	70%	approximately 65%
T4 Canary	100%	approximately 100%
Overall	92.3%	approximately 90%

C.5 Topology-Change Vector

The 7-dimensional topology-change vector summarizes the structural changes detected in each experiment. Its entries are node additions, node removals, edge additions, edge removals, replica delta, version split, and a normalized change score. The same vector is used for both benchmarks, while the node-change and edge-change masks scale with the benchmark graph size. Online Boutique therefore uses node and edge masks of length 11 and 13, whereas TrainTicket uses masks of length 30 and 78.

For Online Boutique, the vector is sparse but highly diagnostic. node_added is nonzero in 299 cases, with a mean value of 1.337 and a maximum of 6. node_removed is nonzero in 83 cases, while edge_added is always zero in the current service-level graph construction. edge_removed is nonzero in 299 cases, replica_delta is nonzero in 226 cases, and version_split is active in 125 cases. The normalized change_score is nonzero in 369 cases, with a mean of 0.277. The TrainTicket collection fol-

Table 10: Approximate data scale of the dynamic benchmark collections.

Data component	Online Boutique	TrainTicket	Main scaling factor
Metrics per experiment	approximately 700 KB	approximately 2.1 MB	More service-level metric series.
Logs per experiment	approximately 4 MB	approximately 12 MB	More monitored services.
Traces per experiment	approximately 72 MB	approximately 130 MB	Deeper call chains and more spans.
Topology JSON per experiment	approximately 24 KB	approximately 80 KB	Larger node and edge sets.
Metadata per experiment	approximately 12 KB	approximately 20 KB	Larger verification records.
Raw collection total	approximately 33 GB	approximately 58 GB	400 experiments per benchmark.
Preprocessed sample size	approximately 47 KB	approximately 95 KB	Larger node and edge tensors.
Preprocessed total	approximately 19 MB	approximately 38 MB	Same split policy and feature schema.

lows the same feature semantics, but the larger graph makes endpoint replacement, trace mismatch, and version split evidence more frequent at the edge-mask level. Across both benchmarks, T1 is dominated by new pods and replica-count changes, T2 by paired pod additions and removals, T3 by sparse edge-semantic changes, and T4 by explicit version split plus pod additions.

C.6 Node-Change Detection

For Online Boutique, the node-change mask concentrates on the six eligible root-cause services. currencyservice, recommendationservice, checkoutservice, and productcatalogservice have the largest changed counts, with 67, 62, 61, and 60 changed cases respectively. shippingservice appears in 40 changed cases and cartservice appears in 20. The remaining services are included in graph propagation but have no direct changed-mask activations in this dataset. For TrainTicket, changed-mask activations are distributed over the 10 target groups listed in Table 8, with the largest concentration on order-related workflows because they participate in all four scenario families.

C.7 Node and Edge Features

The feature construction follows the same design principle as the graph-patch planner: service behavior, topology state, and runtime communication evidence are kept separate before being fused by the GNN. Table 12 reports the node-level features used to describe service state within each time bin.

Table 12: Twelve node features extracted for each service and time bin.

Dim.	Feature	Source	Description
0	cpu_usage	Metrics	Mean CPU utilization over pods of the service.
1	request_rate_in	Metrics	Inbound request rate where the service is the destination.
2	request_rate_out	Metrics	Outbound request rate where the service is the source.
3	error_rate	Metrics	5xx request ratio.
4	latency_p50	Metrics	P50 latency in milliseconds.
5	latency_p99	Metrics	P99 latency in milliseconds.
6	replica_count	Metrics	Deployment replica count.
7	log_error_count	Logs	Error log count.
8	log_total_count	Logs	Total log count.
9	trace_error_rate	Traces	Fraction of spans with nonzero status code.
10	trace_avg_duration	Traces	Average span duration in milliseconds.
11	trace_span_count	Traces	Number of spans.

Table 13 reports the edge-level features used to describe communication behavior along service dependencies. These features allow the RCA model to distinguish a faulty node from a downstream service that only receives abnormal traffic.

Table 13: Four edge features extracted for each service dependency edge.

Dim.	Feature	Source	Description
0	call_rate	Request metrics	Call rate from source to destination.
1	error_rate	Request metrics	Edge-level 5xx rate.
2	latency	Traces	Span duration for the corresponding source-destination pair.
3	response_flags	Metrics	Envoy response-flag anomaly count.

Together, the node and edge features provide a compact representation of metrics, logs, and traces while leaving topology-change semantics to the dynamic graph executor.

The fixed service-level graphs contain 13 dependency edges for Online Boutique and 78 dependency edges for TrainTicket. Online Boutique is centered on two fan-out services: the frontend calls user-facing downstream services, and checkout calls cart, catalog, shipping, currency, payment, and email services. TrainTicket has a deeper and

broader call graph, with preserve and rebook workflows calling more than ten downstream service groups. Dynamic topology operations augment these fixed graphs with pod-level, endpoint-level, version-level, mismatch, and blackhole evidence.

C.8 Phase Statistics and Root-Cause Feature Shifts

Across Online Boutique services, the during-incident window shows lower inbound request rate, lower total log count, higher trace error rate, and higher average trace duration than the pre-incident window. Root-cause services show stronger shifts: CPU usage increases by 0.032 on average, inbound request rate changes in 99.8% of cases, replica count changes in 81.8%, and average trace duration increases by 5.454 on average with a large maximum shift of 105.149. These statistics support the use of multimodal node features rather than topology evidence alone.

The TrainTicket collection increases the trace scale of the dataset. The exported artifacts contain approximately 1,200 traces per experiment on average, compared with approximately 700 traces per experiment for Online Boutique. The average number of spans per trace increases from approximately 24 to approximately 40, and service coverage per trace increases from more than 6 services to approximately 10–15 services in representative runs. This larger trace footprint provides a more demanding setting for evidence compression and graph-patch planning.

C.9 Dataset Splits

Experiments are split by stratified sampling with seed 42. For each 400-experiment dynamic benchmark collection, the training, validation, and test sets contain 279, 59, and 62 samples respectively. The test set includes 15 T1 cases, 12 T2 cases, 15 T3 cases, and 20 T4 cases, preserving scenario coverage in the final evaluation. The same split ratio is used for Online Boutique and TrainTicket so that benchmark differences reflect topology and telemetry scale rather than different evaluation protocols.

D RCA Differences Between Dynamic and Static Topologies

D.1 Fundamental Challenges Introduced by Dynamic Topology

Static RCA methods typically assume that the service dependency graph is fixed throughout the observation window, that nodes have stable semantics, and that edges represent stable call relationships. Dynamic topology invalidates these assumptions in several ways. The reference graph can become an incomplete baseline when replicas, endpoints, or versions change. A service name can denote different pod sets or version sets over time. A declared dependency may point to a wrong or missing backend at runtime. Moreover, a topology change may itself be a root cause, a symptom, or an intermediate propagation mechanism. These effects make propagation paths unstable and can dilute fault evidence when service-level aggregation hides instance-level or version-level anomalies.

D.2 Topology Dimensions of the Four Scenarios

The four scenarios affect different topology dimensions. T1 mainly changes node count, replica count, and endpoint expansion, and it is visible from Kubernetes and metric signals. T2 has little net service-level change but replaces pod identities and endpoint IPs, so it is only partially visible through Kubernetes APIs and is weak in metrics. T3 has almost no node-level change but corrupts endpoint semantics; traces are therefore essential because they reveal declared-observed route mismatch. T4 adds version-level entities and weighted routing, so topology and trace evidence are both required to separate v1 and v2 behavior.

D.3 Observable-Data Change Matrix

Table 14 summarizes the relative strength of signals by modality. The scale is 0 for no change, 1 for weak change, 2 for significant change, and 3 for severe change.

Table 14: Observable-data change matrix across scenarios.

Observable dimension	T1 HPA	T2 Drift	T3 Selector	T4 Canary
Metrics				
cpu_usage	3	0	0	1
request_rate	2	1	3	1
error_rate	0	0	2	1
latency	0	0	2	2
replica_count	3	1	0	1
Logs				
error_count	0	1	1	0
total_count	1	1	2	1
Traces				
span_count	2	1	2	1
trace_error_rate	0	0	2	1
trace_avg_duration	0	0	2	3
Topology				
node_change	3	2	0	2
edge_change	2	1	2	1
version_split	0	0	0	3

The matrix shows that no modality dominates across all settings. It also explains why the framework uses topology-aware graph construction together with multimodal telemetry, rather than relying on a single source such as metrics or traces.

D.4 Features Unique to Dynamic Topology

Dynamic topology introduces evidence that is absent under a purely static assumption. Replica-count jumps and endpoint IP changes are detected from metrics and Kubernetes snapshots. Pod identity replacement is detected by comparing UIDs, pod names, IP addresses, and node assignments. Version coexistence and traffic-weight routing are detected from pod labels and Istio routing resources. Trace route mismatch is detected by comparing declared service destinations with observed runtime destinations, while blackhole behavior is identified through missing endpoints and reachability evidence. The topology-change score aggregates these resource, graph, weight, and reachability signals into a compact verifier output.

D.5 Multimodal Complementarity

No single telemetry modality is sufficient for all four scenarios. T1 is strongly metric-driven, T2 is primarily topology-driven, T3 depends heavily on traces, and T4 requires topology and trace evidence jointly.

Scenario	Metrics	Topology	Traces	Logs
T1 HPA	dominant	auxiliary	weak	weak
T2 Drift	weak	dominant	auxiliary	weak
T3 Selector	weak	weak	dominant	weak
T4 Canary	useful	dominant	dominant	weak

Figure 9: Qualitative modality complementarity. Dynamic-topology RCA requires metrics, logs, traces, and topology evidence to be fused rather than treated as interchangeable signals.

The qualitative pattern is consistent with the scenario design: T1 is driven by resource pressure, T2 by identity changes, T3 by route inconsistency, and T4 by version-level routing and trace evidence.

E Method Details: Evidence-to-Patch RCA

E.1 Problem Definition

For each experiment, the input consists of the reference topology graph G_0, the observed runtime topology graph G_{obs}, the topology-transition summary ΔT, multimodal telemetry summaries X from metrics, logs, and traces, and a candidate root-cause set $\mathcal{C}$. The output is a ranking over candidate services or version-level entities:

$$R = [v_1, v_2, \ldots, v_k], \tag{8}$$

where v_1 is the predicted most likely root-cause entity.

E.2 End-to-End Pipeline

Plan2RCA has five stages. First, compact evidence is built from topology diffs, endpoint diffs, pod diffs, deployment diffs, trace summaries, and metric summaries. Second, the LLM Graph Patch Planner generates a structured GraphPatchPlan. Third, the Graph Executor applies the patch to G_0 to construct G^*. Fourth, a lightweight GNN assigns root-cause scores to candidate nodes. Finally, the LLM Evidence Reranker checks multimodal evidence and reranks the GNN Top-K candidates.

$$\begin{aligned} G^* &= \text{Executor}(G_0, \text{LLMPlanner}(E)), \\ S &= \text{GNN}(G^*), \\ R &= \text{LLMReranker}(\text{TopK}(S), E, G^*), \end{aligned} \tag{9}$$

where E denotes the compact evidence package.

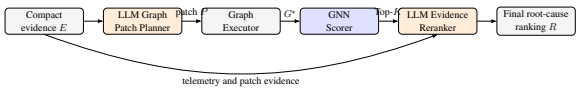


Figure 10: End-to-end Plan2RCA pipeline. The LLM is used to compile observability evidence into executable graph patches and to check evidence, not for unconstrained root-cause prediction.

This pipeline separates graph patching from RCA scoring. As a result, the LLM agent changes the input graph through validated operations, while the downstream GNN remains a fixed scorer across all graph variants.

E.3 Dynamic Attributed-Graph Representation

The system topology is represented as a directed attributed graph:

$$G = (V, E, A_V, A_E), \tag{10}$$

where V includes service, pod, version, and blackhole nodes; E includes dependency, endpoint, redirect, trace-mismatch, and version-split edges; A_V contains node attributes such as service name, pod state, version label, and replica count; and A_E contains edge attributes such as dependency weight, anomaly weight, topology confidence, change weight, and final weight.

Unlike simply using G_{obs} as a replacement snapshot, the dynamic graph G^* retains G_0 as a stable reference and inserts explicit change-event evidence. For example, T1 adds new pod nodes and redistributes service-to-pod edge weights; T2 inserts alignment evidence from old pods to new pods; T3 reduces confidence on corrupted edges and adds trace-mismatch or blackhole evidence; and T4 splits an aggregate service into version-level nodes.

Table 15: Node and edge types used by the dynamic RCA graph.

Type family	Examples	RCA role
Node types	service, pod, version, blackhole	Represent service identity, runtime instances, canary versions, and missing endpoints.
Edge types	dependency, endpoint, redirect, trace mismatch, version split, change event	Encode call dependencies, runtime bindings, topology mutations, and evidence of semantic mismatch.
Node attributes	replica count, ready replica count, pod phase, version label, candidate flag	Provide topology and status evidence to the GNN.
Edge attributes	dependency weight, anomaly weight, topology confidence, change weight, final weight	Provide weighted structural and anomaly evidence to message passing.

E.4 Compact Evidence Package

The system does not feed raw traces or all metric time series directly into the LLM. Instead, it constructs a compact JSON evidence package containing only the summaries needed for graph-patch planning and RCA evidence checking. This reduces cost and constrains the LLM to verifiable evidence.

Table 16: Fields in the compact evidence package.

Field	Purpose
experiment_id	Identifies the experiment and links all artifacts.
scenario_name	Encodes the expected class of topology dynamics.
target_service	Provides the incident entry or target service when used as an observable prior.
topology_diff_summary	Summarizes graph-level node and edge differences.
endpoint_diff_summary	Detects selector redirect, empty endpoint, and endpoint replacement.
pod_diff_summary	Detects pod add/remove events, pod drift, and replica scaling.
deployment_diff_summary	Detects deployment-level changes and version-level resources.
trace_route_summary	Captures declared-versus-observed route mismatches.
metric_summary	Extracts anomaly strength from latency, error, request-volume, and CPU signals.

E.5 GraphPatchPlan and Restricted Operation Space

The LLM Graph Patch Planner is not asked to directly predict a root cause. Instead, it emits a structured GraphPatchPlan. The patch plan must pass JSON schema validation and may only use operations from a restricted operation set. If the patch plan is invalid, incomplete, references nonexistent nodes, uses unknown operations, or if the API is unavailable, the system falls back to a deterministic rule planner.

Table 17: Fields in GraphPatchPlan.

Field	Meaning
experiment_id	Experiment identifier.
change_type	Predicted topology-change category.
confidence	Planner confidence in the patch plan.
causal_role	Whether the topology change is treated as a cause, effect, mixed event, or alignment event.
node_ops	Node-level operations such as add, align, split, or blackhole insertion.
edge_ops	Edge-level operations such as redirect, split, replace endpoint, or trace mismatch insertion.
weight_ops	Edge reweighting operations.
state_transfer	State transfer or identity alignment between old and new nodes.
evidence	Verifiable evidence references supporting the patch plan.

Table 18 specifies the constrained action space available to the LLM planner. The operation grouping is intentionally small so that generated patch

plans remain executable, auditable, and comparable with deterministic fallback rules.

Table 18: Allowed graph operations grouped by patch family.

Family	Whitelisted operations and role
Node	ADD_NODE, REMOVE_NODE, KEEP_NODE, ALIGN_NODE, SPLIT_NODE, MERGE_NODE, and ADD_BLACKHOLE_NODE; used to preserve identity, add runtime entities, and represent version or blackhole nodes.
Edge	ADD_EDGE, REMOVE_EDGE, REDIRECT_EDGE, SPLIT_EDGE, ADD_TRACE_MISMATCH_EDGE, and REPLACE_ENDPOINT_EDGE; used to encode endpoint replacement, route mismatch, redirection, and traffic split.
Weight	REWEIGHT_EDGE; used to update dependency, anomaly, topology-confidence, and change-intensity attributes before computing the final edge weight.

E.6 Prompt Templates for the LLM Agent

The two LLM components use fixed prompt templates and receive only compact evidence serialized as JSON. The planner prompt constrains the agent to emit graph patches, while the reranker prompt constrains the agent to audit and reorder existing candidates under explicit evidence references. The templates are summarized in Figures 11 and 12.

E.7 Rule-Based Fallback Planner

The fallback planner provides a deterministic safety path whenever the LLM output is unavailable, malformed, or inconsistent with the evidence package. It is not designed to maximize flexibility; instead, it encodes conservative scenario-level patches that keep the pipeline executable.

Table 19: Deterministic fallback rules by scenario.

Scenario	Rule summary
T1 node scale	When replicas increase, add pod nodes and service-to-pod endpoint edges, redistribute endpoint weights, and mark the event as effect_or_mixed.
T2 pod drift	When a same-service pod is reconstructed, apply ALIGN_NODE, add an old-to-new change edge, replace the endpoint edge, and mark a topology event.
T3 redirect	When the endpoint backend belongs to another service, apply REDIRECT_EDGE, add runtime and trace-mismatch edges, and mark the change as a fault cause.
T3 empty endpoint	When no backend is reachable, add a blackhole node and a service-to-blackhole edge, and mark the change as a fault cause.
T4 canary split	When v2 or version labels are detected, split the service and upstream traffic, estimate routing weights, and mark the event as cause or mixed.

These rules also define a strong non-LLM reference point. Improvements over this fallback therefore indicate that the LLM planner contributes useful graph-patch composition, rather than merely benefiting from the same executor.

E.8 Graph Patch Executor

The Graph Executor applies a validated patch plan to G_0 and constructs G^*. It validates all operations before execution, records warnings instead of silently dropping invalid operations, preserves stable node IDs, and stores intermediate graph artifacts without modifying raw data.

Table 20: Execution semantics of major graph operations.

Operation	Execution semantics
ALIGN_NODE(old,new)	Preserve both old and new nodes and add an old-to-new change-event edge.
SPLIT_NODE(service)	Preserve the parent service node and add version-level child nodes.
REDIRECT_EDGE	Lower topology confidence on the old edge and add a new runtime edge.
ADD_BLACKHOLE_NODE	Create a blackhole node and add an edge from the affected service to the blackhole.
REWEIGHT_EDGE	Update edge attributes and recompute the final edge weight.
ADD_TRACE_MISMATCH_EDGE	Insert an evidence edge that links the declared destination to the observed runtime destination.

E.9 Edge-Weight Computation

Each edge has a final weight derived from dependency strength, anomaly strength, topology confidence, and topology-change intensity:

$$w_e = 0.30w_{dep} + 0.35w_{anom} + 0.25w_{topo} + 0.10w_{change}. \quad (11)$$

Here, w_{dep} measures dependency or route strength, w_{anom} measures latency, error, response-flag, or trace anomaly strength, w_{topo} measures consistency between declared topology, endpoints, selectors, and traces, and w_{change} measures whether the edge was created or modified by the graph-patch plan. These coefficients are fixed in the main experiments and are not tuned separately for each baseline.

E.10 GNN Node and Edge Features

After graph execution, the dynamic RCA graph is converted into GNN tensors. The feature groups in Table 21 summarize graph identity, Kubernetes status, incident evidence, and edge attributes in a form that can be consumed by both Weighted-GraphSAGE and EdgeAwareGAT.

Prompt Template for the LLM Graph Patch Planner

System role. You are a graph update planner for microservice RCA. Given topology and telemetry summaries, output a valid GraphPatchPlan JSON. Your task is to update graph dependencies, edge types, node identity mappings, and edge weights. Do not directly predict the root cause. Do not tune model hyperparameters, model families, optimizers, training epochs, learning rates, or search spaces. Only output graph operations.

Allowed actions. The planner may use node operations for adding, removing, keeping, aligning, splitting, merging, and inserting blackhole nodes; edge operations for adding, removing, redirecting, splitting, replacing endpoints, and adding trace-mismatch edges; and weight operations for reweighting dependency, anomaly, topology-confidence, change-intensity, and final-weight attributes. Pod node identifiers must use the form pod:service-pod, version nodes must use service:v1 and service:v2, and blackhole nodes must be created through the service field. Unsupported actions, KEEP_EDGE, and model-training keys are invalid.

Scenario guidance. For HPA and node-scale incidents, add replica pod nodes and reweight endpoint edges. For pod drift, align old and new pod identities and replace endpoint edges. For selector redirect, encode trace mismatch and runtime endpoint redirection. For empty endpoints, insert a blackhole node. For canary release, split the service into version nodes and split traffic edges.

User template. Build a GraphPatchPlan JSON for this experiment evidence. The required top-level keys are experiment_id, change_type, confidence, causal_role, node_ops, edge_ops, weight_ops, state_transfer, and evidence. Evidence JSON: {evidence_json}. Return valid JSON only, without markdown, comments, or prose.

Figure 11: Prompt template used by the LLM Graph Patch Planner. The runtime system inserts the compact evidence JSON into the user template before querying the LLM.

Table 21: Representative GNN features on the dynamic RCA graph.

Feature group	Examples
Node type indicators	Service, pod, version, blackhole, split parent, and split child indicators.
Kubernetes status	Replica count, ready replicas, endpoint count, and pod status summaries.
Graph structure	In-degree, out-degree, endpoint out-degree, dependency in-degree, and candidate-mask indicator.
Incident evidence	Maximum incident edge final weight, maximum change weight, minimum topology confidence, trace-mismatch indicator, and blackhole-edge indicator.
Edge attributes	dependency_weight, anomaly_weight, topology_confidence, change_weight, and final_weight.

E.11 WeightedGraphSAGE

WeightedGraphSAGE performs edge-weighted neighbor aggregation. For node v at layer l,

$$h_v^{(l+1)} = \sigma\left(W_l\left[h_v^{(l)} \| \sum_{u\in\mathcal{N}(v)} \alpha_{uv} h_u^{(l)}\right]\right), \tag{12}$$

where α_{uv} is normalized from the edge final weight. The node-level root-cause score is then computed by a linear output layer:

$$s_v = W_o h_v^{(L)}. \tag{13}$$

E.12 EdgeAwareGAT

EdgeAwareGAT uses edge attributes, especially final_weight, as attention-related inputs. Its purpose is to let the model attend more strongly to neighbors connected by edges with high anomaly evidence or strong topology-change evidence.

E.13 Training Objective

The main training objective is node-level cross entropy over the candidate service set:

$$\mathcal{L}_{CE} = -\log \frac{\exp(s_y)}{\sum_{v\in\mathcal{C}} \exp(s_v)}, \tag{14}$$

where y is the ground-truth root-cause node. A pairwise ranking loss can also be used:

$$\mathcal{L}_{rank} = \frac{1}{|\mathcal{C}|-1} \sum_{v\neq y} \max(0, m - (s_y - s_v)). \tag{15}$$

The total loss is

$$\mathcal{L} = \mathcal{L}_{CE} + \lambda \mathcal{L}_{rank}. \tag{16}$$

The main experiments use $\lambda = 1.0$, while sensitivity analysis varies this coefficient.

E.14 LLM Evidence Reranker and Overall Algorithm

After the GNN produces Top-K candidates, the LLM Evidence Reranker checks topology, metrics, logs, traces, and the executed graph patch. It is only allowed to rerank candidates in the GNN Top-K set unless the evidence explicitly introduces special nodes such as blackhole or version nodes. The reranker must output JSON with final ranking, confidence, reason, and evidence references. If the output is invalid, the system uses a rule-based reranker. Algorithm 1 summarizes the complete procedure.

Prompt Template for the LLM Evidence Reranker

System role. You are an RCA evidence checker. You receive the Top-K root-cause candidates from a small GNN model, evidence extracted from topology, metrics, logs, and traces, and a graph patch plan that has already been executed.

Ranking policy. Treat the GNN order as the primary ranking and preserve it by default. Do not introduce new candidates outside Top-K unless the evidence contains an explicit blackhole or version node. Promote a lower-ranked candidate only when it has direct causal evidence that higher-ranked candidates lack, such as selector redirect, empty endpoint, trace mismatch, endpoint replacement, pod alignment, version split, abnormal v2 evidence, or a direct topology-update edge. Generic latency, generic logs, or vague downstream symptoms are insufficient for promotion.

Evidence policy. Do not invent evidence. Use only evidence reference identifiers from the evidence pack in supporting_evidence and counter_evidence. Check whether each candidate explains all available evidence modalities. If the target service appears in Top-K and the target-service prior is enabled, treat it as a strong incident prior when other evidence is ambiguous. If evidence is ambiguous or the GNN top candidate is plausible, keep the original order and set changed_order to false.

Output schema. Return valid JSON only. The output contains final_ranking, where each entry includes candidate, final_score, confidence, reason, supporting_evidence, and counter_evidence, together with changed_order and summary. Evidence pack: {evidence_json}.

Figure 12: Prompt template used by the LLM Evidence Reranker. The reranker is restricted to evidence-bound auditing of the GNN Top-K candidates.

Algorithm 1 Plan2RCA Evidence-to-Patch RCA

Require: Reference topology G_0, observed runtime topology G_{obs}, topology change ΔT, telemetry summaries X, reranking size K
Ensure: Final ranked root-cause list R

```
Build compact evidence E from ΔT, topology snapshots, metrics, logs, and traces.
Prompt the LLM Graph Patch Planner with E.
Parse the response as GraphPatchPlan P.
if P is invalid, unavailable, or fails schema validation then
    P ← RulePlanner(E).
end if
Execute P on G_0: G* ← GraphExecutor(G_0, P).
Compute edge weights w_e = 0.30 w_dep + 0.35 w_anom + 0.25 w_topo + 0.10 w_change.
Convert G* into model tensors: node features, edge index, edge attributes, and candidate mask.
Train or load the fixed-configuration GNN model.
Compute node root-cause scores s_v ← GNN(G*).
Select Top-K candidates C_K.
Build a candidate-specific evidence package.
Prompt the LLM Evidence Reranker with C_K and the evidence package.
if the reranker output is invalid then
    R ← RuleReranker(C_K, E).
else
    R ← validated LLM ranking.
end if
return R.
```

F Evaluation Metrics, Baselines, and Training Setup

F.1 Graph-Construction and Patch Baselines

The ablation variants are designed to isolate the source of performance gains reported in Table 5. They progressively remove runtime topology, explicit graph patches, full patch semantics, LLM planning, and evidence-bound reranking while keeping the downstream GNN configuration fixed.

Table 22: Ablation variants compared in the main experiment.

Variant	Description
w/o runtime topology	Uses only the reference topology and does not introduce incident-time topology evidence.
w/o graph patch	Uses the observed runtime topology as an unstructured snapshot, without typed patch semantics.
w/o full patch semantics	Removes part of the typed patch semantics and keeps a reduced graph-update representation.
w/o LLM planner	Uses deterministic rule-based graph patches under the same validation and executor constraints.
w/o reranker	Uses the LLM-planned graph patch and GNN scorer but removes the evidence-bound reranking stage.
Full	Uses the complete Plan2RCA pipeline with LLM patch planning, deterministic validation, graph execution, GNN scoring, and evidence-bound reranking.

F.2 Models and Fixed Training Configuration

The evaluation uses two fixed GNN scorers. WeightedGraphSAGE performs edge-weighted neighbor aggregation and outputs node-level root-cause scores. EdgeAwareGAT uses edge attributes and final_weight as attention-related inputs, also producing node-level root-cause scores. Both models are trained under the same configuration so that differences across graph variants mainly reflect the effect of graph construction.

Table 23: Fixed training configuration for the main experiments.

Model		Optimization	
Hidden dim	128	Learning rate	3×10^{-3}
Layers	2	Weight decay	1×10^{-5}
Dropout	0.1	Epochs	200
Batch size	1 graph	Early stop	30
Seed	42	Ranking coef.	1.0

F.3 RCA Ranking Metrics

RCA performance is evaluated as node-level service root-cause ranking. The main metrics are Top@1, Top@3, Top@5, and mean reciprocal rank (MRR). For a test sample with ground-truth root-cause rank r, Top@K is 1 if $r \leq K$ and 0 otherwise. MRR is

$$\mathrm{MRR} = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{r_i}. \tag{17}$$

F.4 Graph-Patch Quality Metrics

The graph-patch metrics evaluate whether a generated patch is structurally valid before its downstream RCA accuracy is considered. This separation is important because a high ranking score would be less meaningful if the intermediate graph patch were invalid or unsupported by evidence.

Table 24: Graph-patch quality metrics.

Metric	Meaning
Operation validity rate	Fraction of generated graph-patch operations that satisfy the operation schema, contain all required fields, and refer to valid or legally creatable nodes and edges. Higher values indicate fewer structurally invalid operations.
Plan execution success rate	Fraction of GraphPatchPlans that can be applied by the graph executor without runtime errors or inconsistent graph states. Higher values indicate that the generated plans are executable rather than merely well-formed.
Node add/remove F1	F1 score for detecting node additions and removals, including new pods, removed pods, and version-level nodes. This metric is most relevant to node scaling and canary splitting.
Node alignment accuracy	Accuracy of mapping an old pod to the corresponding reconstructed pod under the same service or deployment identity. This metric mainly evaluates pod-drift cases.
Edge redirect recall	Recall for detecting selector-induced runtime redirection and generating the corresponding redirect or runtime dependency edge. This metric mainly evaluates selector-redirect cases.
Blackhole recall	Recall for identifying empty or unreachable backends and constructing a virtual blackhole node with an associated service-to-blackhole edge. This metric mainly evaluates endpoint-removal cases.
Version split recall	Recall for decomposing a service into version-level nodes and creating the corresponding split edges. This metric mainly evaluates canary cases.
Edge weight MAE	Mean absolute error between generated edge weights and expected routing proportions, such as canary traffic ratios. Lower values indicate better weight estimation.
Weight accuracy	A normalized high-is-better view of edge-weight quality, defined as $1 - \mathrm{EdgeWeightMAE}$. This transformation is used only for unified visualization.
Trace mismatch recall	Recall for generating a trace-mismatch edge when trace evidence shows a discrepancy between the declared destination and the observed runtime destination. This metric mainly evaluates selector poisoning.

These metrics cover syntax, execution, and scenario-specific patch behavior. They therefore provide an execution-level view of hallucination control that cannot be obtained from free-form LLM diagnoses. For scenario-level interpretation, T1 node F1 measures whether new replicas or pods and their service-to-pod edges are correctly represented. T2 alignment measures whether reconstructed pods are linked to the correct pre-fault identity. T3 redirect recall measures whether the planner identifies a mismatch between declared topology and runtime routing. T4 version recall measures whether the planner creates version-level nodes and split traffic edges, while T4 weight accuracy measures whether the generated edge weights match the expected canary routing ratios.

G Sensitivity Analysis

The sensitivity experiments are not hyperparameter optimization. They vary a small set of pre-

specified design parameters to understand the stability of the pipeline. The main parameter trends are shown in Figure 4. Figure 13 reports auxiliary reranker diagnostics, including the change from GNN-before-rerank scores to final reranked scores and the number of rows handled directly by the LLM reranker versus deterministic fallback. Each setting is evaluated on 62 cases.

G.1 Target-Service Prior

The target-service prior has a visible but bounded effect on the reranker. Without the prior, the model reaches 0.8015 Top@1, 0.9230 Top@3, 0.8152 MRR, and a mean rank of 2.0800. With the prior, the corresponding scores increase to 0.8298 Top@1, 0.9460 Top@3, 0.8365 MRR, and a mean rank of 1.8600. The prior therefore improves Top@1 by 2.83 percentage points and MRR by 2.13 percentage points. The prior is valid if it comes from an observable alert entry point; otherwise, the prior-off setting should be used as the conservative evaluation.

G.2 Learning Rate

A learning rate of 0.003 is the strongest among the tested values, reaching 0.8298 Top@1, 0.9460 Top@3, 0.9800 Top@5, and 0.8365 MRR. A lower learning rate of 0.001 reduces Top@1 to 0.8060 and MRR to 0.8150, while a higher learning rate of 0.010 reduces Top@1 to 0.7910 and MRR to 0.8050. These results indicate that the model benefits from a moderate learning rate, but the performance changes remain limited.

G.3 Pairwise Ranking Coefficient

The pairwise ranking coefficient has a small but consistent effect. Coefficients 0.0, 0.5, 1.0, and 2.0 obtain Top@1 scores of 0.8160, 0.8215, 0.8298, and 0.8120, respectively, with MRR scores of 0.8240, 0.8310, 0.8365, and 0.8205. The main value of 1.0 is the best setting, while an overly large coefficient slightly degrades ranking quality.

G.4 Topology-Confidence Coefficient

The topology-confidence coefficient is stable in the tested range, but the main value of 0.25 remains strongest. Coefficients 0.15, 0.25, and 0.40 obtain Top@1 scores of 0.8010, 0.8298, and 0.8105, respectively, and MRR scores of 0.8110, 0.8365, and 0.8185. Increasing topology confidence from 0.15 to 0.25 improves ranking, but further increasing it to 0.40 reduces the contribution of anomaly evidence and lowers performance.

G.5 Evidence-Bound Reranker Diagnostics

The evidence-bound reranker consistently improves the GNN-before-rerank scores. Across the pairwise-ranking and topology-confidence settings, Top@1 increases by 4.70–5.20 percentage points, Top@3 increases by 4.50–5.60 points, and MRR increases by 3.25–3.85 points. The LLM reranker handles 50–55 of the 62 cases per setting, while deterministic fallback handles 7–12 cases. The main setting has the smallest fallback count, with 55 LLM-handled cases and 7 fallback cases. These results show that reranking contributes a stable final verification gain without replacing the graph-patch and GNN stages.

H Failure Cases, Threats, and Limitations

H.1 T3 Selector Redirect and Endpoint Removal

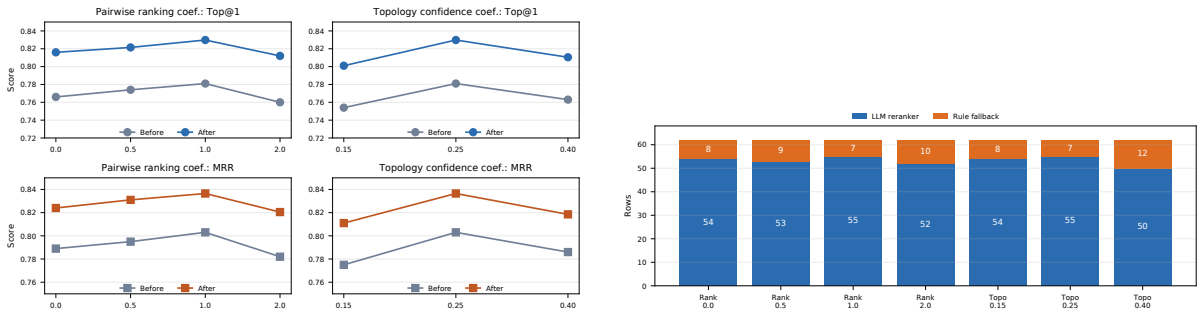
T3 is the most difficult scenario because the declared service topology can remain visually plausible while runtime routing is corrupted. A redirect can send traffic to a wrong backend, and an empty endpoint can behave like a blackhole. When topology diffs are sparse, traces become the main evidence source. The model may still rank an affected propagation service above the actual misconfigured service if trace mismatch evidence is weak or not encoded explicitly.

H.2 Low-Weight T4 Canary Traffic

T4 remains difficult at the ranking level even when the graph-patch planner detects version splits correctly. In the scenario-level results, Plan2RCA reaches the best Top@1 on T4, but its MRR is lower than in the other dynamic-topology scenarios. When v2 traffic weight is small, service-level latency and error-rate metrics are diluted by healthy v1 traffic, which can make the faulty version harder to rank first. This motivates finer version-level feature extraction and stronger route-weight evidence in future experiments.

H.3 Interpretation of the LLM Reranker Prior

The best reranker result uses target_service as an observable incident prior, reaching 0.8298 Top@1 and 0.8365 MRR. This is appropriate when the



(a) Scores before and after LLM reranking.

(b) LLM-handled rows and fallback rows.

Figure 13: Auxiliary sensitivity diagnostics for evidence-bound reranking. The figure distinguishes GNN-before-rerank behavior from final reranked scores and reports how often deterministic fallback is used under parameter variation.

target service corresponds to an alerting entry point or externally observed incident scope. If a protocol treats target service as hidden label information, the prior-off result of 0.8015 Top@1 and 0.8152 MRR should be reported as the conservative result.

H.4 Training Variance and Additional Validation

The pairwise-ranking sensitivity is moderate, with the best result at coefficient 1.0 and lower scores at coefficient 2.0. Future evaluation should include multiple random seeds, leave-scenario-out validation, and additional train/test splits.

In summary, the main limitations are sparse topology diffs in T3, diluted version-level evidence in low-weight T4 canary cases, the protocol-dependent use of the target-service prior, residual small-graph training variance, and limited blackhole-specific breakdowns. The corresponding mitigation directions are active endpoint probing, stronger trace-mismatch extraction, finer version-level trace features, explicit prior-on and prior-off reporting, multi-seed validation, scenario-wise cross-validation, and refined blackhole labels.