International Journal of Artificial Intelligence, Data Science, and Machine Learning
Grace Horizon Publication | Volume 5, Issue 4, 320-329, 2024
ISSN: 3050-9262 | https://doi.org/10.63282/3050-9262.IJAIDSML-V5I4P128

Original Article

Causal Graph-Based Root Cause Analysis for Distributed Data Integrity Incidents: A Framework for Cross-System Failure Attribution and Resolution

Vineeth Kumar Reddy Mittamidi
Application Support Engineer, TCS, North Carolina, USA.

Abstract *- Distributed data platforms increasingly rely on microservices, event streams, replicated storage, automated deployment pipelines, and heterogeneous integration layers. These environments improve scalability and organizational agility, but they also create a difficult class of incidents: cross-system data integrity failures whose visible symptoms occur far away from their originating cause. A duplicate financial transaction, stale customer profile, corrupted analytical feature, or conflicting replicated record may appear in one system while its causal antecedents lie in delayed message propagation, schema drift, compensating transaction failure, weak consistency, deployment misconfiguration, or an upstream machine-learning service. Conventional monitoring and root cause analysis techniques are often optimized for latency, availability, or infrastructure symptoms, and therefore they struggle to distinguish causal faults from correlated anomalies. This paper proposes CG-RCA-DI, a causal graph-based root cause analysis framework for distributed data integrity incidents. The framework combines system topology, event lineage, observability telemetry, data quality constraints, deployment metadata, and domain policies into a multilayer causal graph. It then applies causal discovery, temporal precedence, counterfactual screening, and graph-based attribution to identify likely root causes and recommend resolution actions. The contribution is a complete reference architecture, an incident model, a causal scoring method, and an evaluation protocol for precision, attribution latency, resolution quality, and auditability. The paper argues that data integrity RCA must move beyond service-level symptom localization toward evidence-grounded causal attribution across application, data, process, and governance layers.*

Keywords *- Causal Graphs, Root Cause Analysis, Data Integrity, Distributed Systems, Microservices, Causal Discovery, Observability, Failure Attribution, Data Lineage, AIops.*

1. Introduction

Distributed software systems now process business-critical data through many loosely coupled components, including microservices, workflow engines, message brokers, data lakes, stream processors, replicated databases, ML services, and third-party APIs. This architectural decomposition allows teams to deploy independently and scale selectively, yet it also creates fragmented responsibility for data correctness. A single logical business fact may pass through authentication, validation, enrichment, persistence, replication, feature generation, reporting, and audit subsystems before it becomes visible to a user or regulator. The practical consequence is that data integrity incidents are rarely local. A stale inventory count may be caused by a delayed event consumer. A duplicate payment may be caused by an idempotency-key regression. A mismatched dashboard may be caused by asynchronous replication combined with a schema migration. Microservice literature has long recognized the operational complexity of independently deployable services, but data correctness adds an additional dimension because the failure object is not merely a service but a relationship among data, time, and business semantics [24].

Traditional root cause analysis tends to begin with alerts and logs, then proceeds through manual triage, dashboards, and engineer intuition. This process can be effective for simple outages, but it becomes unreliable when multiple symptoms co-occur. Distributed data integrity incidents often produce a misleading first alert: a reporting table fails validation, a downstream API returns inconsistent values, or a model produces anomalous predictions. The first visible symptom is not necessarily the originating fault. A reliable RCA method must therefore answer a causal question: which prior event, state transition, deployment, or dependency change most plausibly produced the observed integrity violation? Causal inference provides a formal vocabulary for such reasoning because it distinguishes statistical association from intervention-relevant cause [1].

The challenge is amplified by the semantics of distributed time. In a single-process system, temporal order can often be inferred from wall-clock timestamps. In distributed systems, clock skew, retries, batching, asynchronous queues, eventual consistency, and partial failures make wall-clock order insufficient. Lamport's happens-before relation remains foundational because it represents causal ordering through process order and message exchange rather than absolute time [3]. A data integrity RCA framework should therefore model not only metrics and dependencies but also logical event relations: write-

before-read, publish-before-consume, deploy-before-error, schema-before-deserialization, and compensation-before-finalization.

A second challenge is that modern data platforms operate under consistency trade-offs. High availability, partition tolerance, and low latency frequently require weaker consistency choices, and these choices can permit temporary divergence that is harmless in normal conditions but harmful when combined with retries, cache invalidation defects, or incomplete reconciliation. The formal CAP result explains why distributed web services cannot simultaneously guarantee strong consistency, availability, and partition tolerance under the classic asynchronous failure model [7]. In practice, integrity incidents are often produced by design-time consistency compromises interacting with run-time changes. RCA must therefore reason over both structural design decisions and observed incident evidence.

A third challenge concerns governance. Software lifecycle decisions increasingly involve AI-assisted defect prediction, automated testing, release prioritization, and architecture-centered risk management. Decision intelligence approaches for software governance emphasize that operational and architectural decisions should be connected to measurable risk, traceable evidence, and feedback loops [2]. Data integrity RCA should follow the same principle. It should not merely name a failing service; it should provide an auditable attribution chain linking the symptom to causal evidence and remediation choices.

This paper proposes CG-RCA-DI, a causal graph-based framework for cross-system failure attribution and resolution in distributed data integrity incidents. The framework treats an incident as a set of violated data integrity predicates observed over a distributed execution. It builds a multilayer causal graph whose nodes represent service states, data objects, event transitions, deployment actions, configuration states, consistency mechanisms, and business constraints. Directed edges encode candidate causal relationships derived from topology, traces, event lineage, temporal precedence, known invariants, and learned dependence. The framework then ranks candidate causes using a composite causal attribution score that combines anomaly magnitude, temporal plausibility, structural reachability, counterfactual support, and policy criticality. The final output is a ranked causal explanation with suggested resolution actions and verification tests.

The contribution of this work is fourfold. First, it reframes data integrity incidents as causal attribution problems rather than isolated monitoring anomalies. Second, it introduces a multilayer causal graph model that integrates observability, lineage, consistency semantics, deployment metadata, and governance context. Third, it defines an RCA workflow that separates symptom detection, causal graph construction, candidate generation, counterfactual pruning, and resolution planning. Fourth, it provides an evaluation methodology suitable for research and industrial adoption, including top-k cause accuracy, mean attribution latency, explanation completeness, rollback precision, and post-resolution integrity recovery.

2. Background and Related Work

Causal graph-based RCA draws from three major research traditions: causal inference, distributed systems, and operational diagnosis. Causal inference supplies the conceptual foundation for reasoning about interventions and counterfactuals. The structural causal model perspective represents variables as nodes and causal mechanisms as directed relationships, allowing analysts to ask how an outcome would change under an intervention [1]. Constraint-based causal discovery methods, such as the PC algorithm, infer graph structure from conditional independence relations under assumptions such as causal sufficiency and faithfulness [15]. More recent treatments emphasize the conditions under which causal conclusions can be learned from observational and interventional data, which is important for production environments where randomized fault injection is expensive or risky [21].

Distributed systems research supplies the incident substrate. Logical clocks and happens-before ordering provide the basis for reconstructing causal event order in asynchronous systems [3]. Replicated storage systems such as Dynamo demonstrate how availability-oriented designs may rely on versioning, replication, and application-assisted conflict resolution [4]. Causal consistency systems such as COPS show that stronger ordering guarantees can be achieved at wide-area scale, but only through explicit dependency tracking and design discipline [12]. Session guarantees provide user-centered consistency properties, such as read-your-writes and monotonic reads, that are directly relevant to data integrity incidents in multi-device and multi-region applications [17]. Conflict-free replicated data types offer another integrity mechanism by ensuring convergence through mathematically structured merge operations [9].

Operational RCA research has increasingly moved from static dashboards toward graph-based and causal methods. CauseInfer models performance problems with hierarchical causality graphs and statistical inference, illustrating the value of explicit causal paths in large distributed systems [6]. CloudRanger applies causal analysis to cloud-native systems and highlights how service dependency structure can support root cause identification [16]. Root Cause Discovery for microservices treats failures as interventions and avoids learning an unnecessarily complete graph, which is a valuable scalability insight for large architectures [10]. MicroRank uses trace evidence to localize latency issues, showing that request-flow differences between normal and abnormal traces can be diagnostically useful [22]. Neural Granger-based approaches

further incorporate temporal dependencies in microservice metrics, addressing the limitation of methods that ignore the ordering of time-series changes [25].

Recent multimodal RCA methods are especially relevant because data integrity failures rarely appear in a single telemetry channel. CHASE models traces, logs, and monitoring metrics through a causal heterogeneous graph, demonstrating that graph representations can unify different evidence types for microservice diagnosis [23]. Industrial tracing studies show that observability data is useful but difficult to operationalize because traces, logs, and metrics are collected at different granularities and for different engineering purposes [13]. Dapper's large-scale tracing architecture remains a key reference point because it showed how low-overhead distributed tracing can expose request paths across many services [20]. However, tracing alone usually identifies propagation paths, not necessarily the semantic reason why a data value became invalid.

Software engineering and AI governance research also informs this paper. Machine-learning defect prediction studies show that predictive models can support software quality management, but their usefulness depends on performance evaluation and operational integration [14]. Work on the expanding role of ML models in software development suggests that learning systems are becoming part of the software lifecycle rather than external analytical tools [5]. Architecture-centered decision intelligence frameworks connect agile governance, automated testing, and defect prediction into lifecycle decision processes [8]. AI-based systems engineering paradigms extend this view by linking lifecycle governance, predictive quality assurance, automation economics, and cybersecurity intelligence [18]. Converged AI architectures for innovation and cybersecurity risk mitigation further motivate treating RCA as a cross-functional capability rather than a narrow monitoring feature [11].

Despite this progress, existing RCA methods often focus on performance failures, availability incidents, or service-level anomalies. Data integrity incidents require additional reasoning. They involve semantic constraints, business invariants, reconciliation policies, temporal correctness, and audit obligations. A service can be healthy while emitting incorrect data. A database can be available while containing divergent replicas. A trace can complete successfully while violating a business rule. CG-RCA-DI addresses this gap by making data integrity predicates first-class graph objects and by connecting them to system-level, data-level, and governance-level causes.

3. Problem Definition

A distributed data integrity incident is an observed violation of expected data correctness across one or more systems. The violation may concern accuracy, uniqueness, completeness, timeliness, referential consistency, causal consistency, schema validity, authorization-dependent visibility, or business-rule compliance. Unlike a simple outage, the incident may not prevent the system from responding. Indeed, many severe integrity failures occur while the system appears operational: the wrong account balance is displayed, a customer is charged twice, an outdated feature vector influences a model, or a report aggregates conflicting records.

Let a distributed system be represented as a set of services, data stores, queues, jobs, APIs, and governance controls. Let an execution produce events such as writes, reads, publishes, consumes, deployments, schema changes, retries, cache refreshes, reconciliations, and alerts. Let an integrity predicate be a rule that evaluates whether a set of data states and transitions satisfies expected correctness. An incident occurs when one or more predicates are violated over a bounded interval. The RCA objective is to identify the smallest plausible set of antecedent causes that explains the violation and supports effective resolution.

This definition deliberately separates symptoms from causes. A violated predicate is an observation, not a root cause. For example, a duplicate-order predicate may fail because a payment service retried a request without preserving an idempotency key. It may also fail because a message broker redelivered an event after a consumer checkpoint was lost. It may also fail because a compensating transaction was applied twice. Treating the violated predicate as the root cause would produce an inadequate response. The framework must examine candidate causal paths that cross service, storage, event, and governance boundaries.

The problem has five hard constraints. First, observability is incomplete. Some services may lack traces, some logs may be sampled, and some third-party systems may expose only coarse audit events. Second, causality is partially latent. Important variables such as engineer intent, undocumented dependencies, or hidden batch ordering may not be directly observed. Third, correlation is abundant. A single incident may trigger many downstream anomalies, most of which are effects rather than causes. Fourth, temporal ordering is noisy because wall-clock timestamps can conflict across systems. Fifth, remediation must be actionable. An RCA method that produces a mathematically plausible but operationally unusable explanation will not reduce mean time to resolution.

CG-RCA-DI addresses these constraints through layered evidence. Topology constrains possible propagation. Logical event relations constrain temporal plausibility. Data lineage constrains which transformations could affect a corrupted value. Deployment metadata constrains recent changes. Domain rules constrain which anomalies matter. Counterfactual tests

constrain whether removing or reverting a candidate cause would plausibly eliminate the symptom. The result is not a guarantee of perfect causality; rather, it is a structured process for producing better, auditable, and faster attribution than manual triage.

4. Framework Overview: CG-RCA-DI

CG-RCA-DI consists of six stages: integrity predicate monitoring, evidence ingestion, causal graph construction, candidate root cause generation, causal attribution scoring, and resolution orchestration. Each stage is designed to be independently deployable so that organizations can adopt the framework incrementally.

The first stage defines integrity predicates. These predicates express expected data properties, such as “an order identifier must be globally unique,” “a customer profile read after update must reflect the user’s last committed change,” “a debit and credit pair must balance,” or “a feature value must be derived from source records no older than the freshness threshold.” Predicates can be implemented as streaming checks, database constraints, reconciliation jobs, model-monitoring checks, or audit queries. Their role is to convert vague data quality concerns into incident observables.

The second stage ingests evidence. Evidence includes distributed traces, logs, metrics, event streams, schema registry changes, deployment records, configuration diffs, feature store lineage, database replication metadata, access-control policy changes, incident tickets, and business process events. The framework does not assume all evidence is available. Instead, each evidence item is assigned a confidence score based on freshness, completeness, and source reliability. This design allows the graph to represent uncertainty explicitly rather than hiding it.

The third stage constructs a multilayer causal graph. The graph contains five layers. The service layer represents services, APIs, batch jobs, and external providers. The data layer represents tables, topics, objects, keys, features, and derived views. The event layer represents reads, writes, publishes, consumes, and retries, deployments, rollbacks, and reconciliation runs. The policy layer represents constraints, consistency expectations, access rules, and compliance obligations. The human-governance layer represents change approvals, incident annotations, ownership, and release decisions. Edges connect nodes across and within layers. A publish event can cause a consume event. A deployment can cause a schema mismatch. A cache policy can cause stale reads. A consistency setting can permit divergent replicas.

The fourth stage generates candidate causes. Candidate generation starts from violated predicates and performs backward traversal through the graph. It prioritizes nodes that are temporally prior, structurally upstream, anomalous relative to baseline, and connected to the violated data object. This step deliberately avoids ranking every system component. Instead, it narrows attention to nodes with plausible causal paths to the integrity failure. If the incident concerns a stale profile, the framework inspects profile writes, replication lag, cache invalidation, session routing, schema changes, and feature derivation before inspecting unrelated infrastructure metrics.

The fifth stage computes causal attribution scores. The score combines five components: anomaly strength, temporal precedence, structural reachability, counterfactual support, and business criticality. Anomaly strength measures how unusual the node’s behavior is relative to normal operation. Temporal precedence measures whether the node changed before the violation in logical or physical time. Structural reachability measures whether a directed path connects the node to the violated predicate. Counterfactual support estimates whether disabling, reverting, replaying, or substituting the candidate would prevent the violation. Business criticality weights the score by domain impact so that high-risk causes receive attention even if their telemetry is sparse.

The sixth stage recommends resolution. Resolution actions may include rollback, replay, compensation, cache invalidation, schema repair, reprocessing, replica reconciliation, idempotency repair, consumer checkpoint restoration, feature recomputation, access-policy correction, or deployment freeze. Each recommendation is attached to verification tests. For example, if the suspected cause is a broken idempotency key, the verification plan should include duplicate replay tests, audit-log inspection, and predicate re-evaluation after remediation. This action-oriented design connects RCA to recovery rather than stopping at diagnosis.

5. Causal Graph Model

The CG-RCA-DI graph is a directed, typed, attributed graph. Nodes are typed by operational meaning rather than merely by metric name. A service node represents a deployable unit. A data node represents a materialized or logical data object. An event node represents a time-bounded transition. A predicate node represents an integrity condition. A policy node represents a design or governance rule. This typing is essential because data integrity incidents involve semantic relationships that cannot be captured by homogeneous metric graphs.

Edges are also typed. Dependency edges represent structural relationships such as service-to-service calls. Lineage edges represent derivation relationships between data objects. Temporal edges represent happens-before relations. Consistency edges

represent replication or session guarantees. Deployment edges represent change relationships. Policy edges represent rule enforcement. Evidence edges represent observed correlations or learned causal dependencies. Typed edges allow the framework to distinguish "service A calls service B" from "table X derives from topic Y" or "deployment D changed schema S." Without this distinction, RCA may over-rank noisy infrastructure correlations.

Graph construction uses a hybrid strategy. Some edges are imported from service catalogs, infrastructure-as-code, schema registries, API gateways, and lineage systems. These edges are relatively stable but may be incomplete. Other edges are inferred from traces and event metadata. These edges capture runtime behavior but may be affected by sampling. Additional edges are learned from statistical dependence and causal discovery over time-series data. These learned edges are useful for detecting hidden coupling, but they must be constrained by domain knowledge to avoid spurious causal claims.

Temporal modeling is central. Each event node stores both wall-clock time and logical ordering evidence. Logical evidence includes trace parent-child relationships, message offsets, transaction identifiers, version vectors, request identifiers, session identifiers, and read/write dependencies. When wall-clock time conflicts with logical evidence, the framework prefers logical precedence for causal attribution. This prevents false conclusions in cases where an upstream publish appears later than a downstream consume due to clock skew or delayed log ingestion.

Integrity predicates are represented as graph sinks. A predicate node receives edges from data nodes, policy nodes, and event nodes that can affect its truth value. For example, a "no duplicate invoice" predicate connects to invoice table keys, payment events, retry policies, idempotency-key generation, and reconciliation jobs. When the predicate fails, backward traversal from this node identifies possible cause paths. This design makes the RCA graph incident-specific while preserving reusable system knowledge.

Uncertainty is represented through edge weights and evidence provenance. Each edge stores a confidence value, a source, and a validity interval. A service catalog edge may have high confidence but stale validity if the deployment topology changed recently. A learned causal edge may have moderate confidence if based on limited incidents. A manual incident annotation may have high semantic value but lower statistical reliability. During attribution, the framework combines these confidences rather than treating the graph as a perfect representation of reality.

6. Cross-System Failure Attribution

Failure attribution begins when one or more integrity predicates fail. The framework creates an incident subgraph by selecting all nodes that can reach the violated predicate within a bounded causal horizon. The horizon is not only a time window. It is also a dependency window, lineage depth, deployment window, and policy scope. For example, a batch-derived analytical feature may require a longer time horizon than an API freshness violation because the relevant transformations may have occurred hours earlier.

The first attribution step is symptom clustering. Multiple predicate failures may share a cause. A schema drift may produce invalid transformations in several downstream tables. A replication lag may produce stale reads across multiple regions. A broken access policy may expose inconsistent data views to a subset of users. The framework clusters symptoms by shared upstream paths, overlapping data lineage, similar temporal onset, and common deployment context. Symptom clustering reduces alert noise and prevents engineers from investigating each violated predicate as a separate incident.

The second step is causal path extraction. For each candidate upstream node, the framework extracts directed paths to the violated predicate. Paths are scored by path confidence, temporal coherence, and semantic compatibility. A path is semantically compatible if the source node can plausibly affect the predicate type. A CPU spike may affect timeliness but may be less plausible for a deterministic referential-integrity violation unless it caused timeout-induced retries. A schema change may affect completeness, validity, and transformation correctness. A replay job may affect duplication and ordering.

The third step is counterfactual pruning. The framework asks whether the symptom would likely disappear if a candidate cause were removed, reverted, or corrected. In production, counterfactuals may be approximated using canary comparison, replay in a staging environment, shadow reads, historical incident analogies, or invariant-preserving simulation. For example, if a consumer checkpoint rollback is suspected, replaying the same message range in an isolated environment should reproduce duplicates. If a cache invalidation bug is suspected, bypassing the cache should restore read freshness. Counterfactual pruning is important because high correlation alone often selects downstream symptoms rather than originating causes.

The fourth step is root cause set selection. Some incidents have a single root cause, while others require conjunctive explanation. A duplicate transaction may require both a retry storm and missing idempotency enforcement. A stale profile may require both regional replication lag and a routing policy that sends reads to a lagging replica. The framework therefore supports minimal sufficient cause sets rather than forcing a single-cause answer. A cause set is preferred when it explains more violated predicates with fewer assumptions and fewer unsupported edges.

The fifth step is explanation generation. Each explanation contains the violated predicate, the ranked cause or cause set, the causal path, supporting evidence, contradictory evidence, uncertainty, recommended action, and verification test. Explanation quality is a first-class output. In regulated or safety-sensitive environments, it is not enough to repair data silently. Teams must know why the incident occurred, what data was affected, and how future recurrence will be prevented.

7. Resolution and Recovery Workflow

Root cause attribution is valuable only if it improves recovery. CG-RCA-DI therefore links each cause class to resolution playbooks. A schema drift cause maps to schema rollback, compatibility patching, reprocessing, and contract-test reinforcement. A duplicate-event cause maps to idempotency repair, deduplication, offset audit, and replay controls. A stale-read cause maps to cache invalidation, session routing correction, replica health checks, and read-your-writes verification. A broken reconciliation cause maps to compensating transaction audit, ledger repair, and reconciliation rerun. A model-feature integrity cause maps to feature recomputation, model input quarantine, and prediction backfill.

The workflow distinguishes containment, correction, and prevention. Containment stops ongoing damage, such as disabling a faulty consumer or blocking duplicate writes. Correction repairs affected data, such as replaying events or applying compensating entries. Prevention changes the system so the causal pattern is less likely to recur, such as adding contract tests, stronger idempotency constraints, lineage monitoring, or release gates. This three-phase structure prevents teams from confusing short-term mitigation with permanent resolution.

Resolution actions must preserve auditability. For every data repair, the framework records the affected object set, original values, repaired values, causal explanation, operator approval, and verification result. This is especially important when integrity incidents affect financial, healthcare, identity, or compliance data. A repair that destroys evidence can create a second incident. The causal graph therefore remains attached to the incident record as an explanatory artifact.

The framework also supports feedback learning. After an incident is resolved, engineers mark the confirmed root cause, rejected candidates, effective actions, and missed evidence. This feedback updates edge weights, playbook mappings, and predicate definitions. Over time, the RCA system becomes better at ranking causes in the local environment. This learning loop aligns with decision intelligence principles because operational decisions become traceable, measurable, and continuously improved [2].

A key design principle is that the framework should recommend reversible actions before destructive actions. If rollback is safer than data rewrite, rollback should be preferred. If shadow replay can validate a hypothesis before production replay, it should be used. If a cache bypass can confirm staleness without modifying records, it should be attempted. RCA systems can cause harm when they automate aggressive remediation from uncertain attribution. CG-RCA-DI therefore separates high-confidence automated actions from lower-confidence human-approved actions.

8. Illustrative Incident Scenario

Consider an e-commerce platform with services for orders, payments, inventory, customer profiles, recommendations, and analytics. The platform uses an event broker for order events, a replicated key-value store for session state, a relational database for financial records, and a feature store for ML recommendations. A new release modifies the payment service to include a revised idempotency-key format. Shortly after deployment, the data quality monitor detects duplicate completed-payment records for a subset of users. A dashboard alert also shows increased latency in the order service, and a feature monitor flags abnormal purchase-frequency features.

A conventional RCA process may focus on the order-service latency because it is the most visible operational alert. However, the integrity predicate that matters is duplicate completed payments. CG-RCA-DI begins from the duplicate-payment predicate and traverses upstream through the payment table, payment-completed events, retry policy, idempotency-key generation, deployment metadata, and broker redelivery logs. The incident subgraph shows that duplicate payment events share a common deployment window and a common idempotency-key transformation. It also shows that the order-service latency began after duplicate processing increased, making it more likely to be an effect than a cause.

The framework then extracts causal paths. One path links the payment deployment to changed idempotency-key generation, then to failed duplicate suppression, then to repeated payment writes, then to the violated uniqueness predicate. A second path links broker redelivery to repeated payment-completed events. A third path links order-service latency to user retries. Counterfactual pruning tests these paths. In shadow replay, old idempotency-key generation suppresses duplicates, while the new format fails for a subset of retry requests. Broker redelivery alone does not produce duplicates when the old key format is used. The framework ranks the payment deployment and idempotency-key defect as the minimal sufficient cause set.

The recommended containment action is to disable the new key format and route requests through the prior idempotency logic. The correction action is to deduplicate affected payment records using transaction identifiers and issue compensating

reversals where necessary. The prevention action is to add a release gate that checks idempotency invariants under retry and redelivery conditions. The verification plan reruns the duplicate-payment predicate, replays affected event ranges in a staging environment, and checks downstream feature recomputation. This example illustrates why data integrity RCA must evaluate causal relationships among deployments, retries, data writes, and business predicates rather than simply ranking anomalous service metrics.

9. Evaluation Methodology

A rigorous evaluation of CG-RCA-DI should use both benchmarked fault injection and retrospective incident analysis. Fault injection creates labeled incidents under controlled conditions. Retrospective analysis tests whether the framework can recover known root causes from historical evidence. The combination is important because synthetic experiments provide ground truth, while historical incidents reveal messy operational realities such as incomplete telemetry and ambiguous human decisions.

The first evaluation metric is top-k root cause accuracy. For each incident, the confirmed cause should appear within the top k ranked candidates. Top-1 accuracy measures immediate precision, while top-3 or top-5 accuracy reflects practical triage usefulness. Because some incidents require cause sets, evaluation should include set-level accuracy: whether the framework identifies all necessary components of a conjunctive cause. This avoids rewarding systems that identify only a retry storm while missing the missing idempotency control.

The second metric is attribution latency. Attribution latency measures the time from predicate violation to a ranked causal explanation. For production use, a moderately accurate explanation produced quickly may be more valuable than a perfect explanation produced after manual triage has already resolved the incident. However, latency should not be optimized by ignoring evidence quality. Evaluation should therefore report accuracy-latency trade-offs under different graph sizes and evidence completeness levels.

The third metric is explanation completeness. A complete explanation should include the violated predicate, causal path, supporting evidence, uncertainty, recommended action, and verification test. Explanation completeness can be scored by expert reviewers using a rubric. This metric is important because engineers need understandable causal narratives, not only ranked node identifiers. RCA tools often fail in adoption when their output cannot be trusted, audited, or converted into action.

The fourth metric is resolution effectiveness. A successful attribution should lead to faster containment, lower recurrence, and fewer harmful remediation actions. This can be measured by mean time to containment, mean time to data correction, post-resolution predicate recovery, and recurrence rate. For high-risk domains, evaluation should also track whether data repair preserved audit evidence. A framework that identifies causes accurately but recommends unsafe repair actions is incomplete.

The fifth metric is robustness to missing evidence. Production systems frequently lack perfect traces, complete logs, or clean topology maps. Evaluation should therefore remove evidence channels systematically and measure performance degradation. A robust framework should degrade gracefully: it may lower confidence, but it should not hallucinate certainty. This is where explicit edge confidence and provenance become critical.

The sixth metric is governance value. Because the framework records causal explanations and post-incident feedback, it should improve organizational learning. Governance value can be evaluated by the number of new tests, policy changes, contract updates, or architecture decisions produced after incidents. This metric connects technical RCA with software lifecycle governance and quality assurance, which is increasingly important in AI-driven engineering organizations [8].

10. Discussion

The central claim of this paper is that distributed data integrity incidents require causal, semantic, and cross-system reasoning. Observability tools expose what happened. Dependency graphs expose where systems are connected. Data lineage exposes how values are derived. Governance metadata exposes why changes occurred. None of these alone is sufficient. CG-RCA-DI combines them into a causal graph that can support attribution and resolution.

The framework also clarifies why purely metric-driven RCA is insufficient. A metric anomaly may be a cause, a symptom, or an unrelated coincidence. For example, high CPU may cause delayed consumers, but it may also be caused by duplicate processing after an upstream defect. Without causal direction and data semantics, RCA can invert the explanation. Causal graphs reduce this risk by requiring temporal, structural, and counterfactual support before ranking a candidate highly.

Another important implication is that data integrity predicates should be treated as observability primitives. Many organizations monitor latency, error rate, saturation, and availability but under-monitor semantic correctness. This creates a blind spot: a system can satisfy service-level objectives while violating business-level correctness. Predicate monitoring closes that gap by making integrity violations explicit incident triggers.

The framework is compatible with existing microservice and AIOps practices. It can ingest traces from distributed tracing systems, logs from centralized logging platforms, metrics from monitoring tools, lineage from data catalogs, deployment records from CI/CD systems, and policies from governance repositories. The framework is not a replacement for observability; it is an attribution layer above observability. Its novelty lies in organizing evidence around causal questions rather than around dashboards.

The approach also supports human-machine collaboration. Automated graph ranking can reduce search space, but human expertise remains important for interpreting domain rules, approving remediation, and validating causal hypotheses. This is particularly true when evidence is incomplete or when remediation has legal or financial consequences. The framework should therefore be deployed as an explainable decision-support system before it is allowed to trigger autonomous repair in high-impact settings.

Finally, CG-RCA-DI encourages a shift from incident response to incident learning. Every confirmed cause updates the graph, predicates, playbooks, and release gates. Over time, RCA evidence becomes a governance asset. The organization learns which architectural patterns produce integrity risk, which tests prevent recurrence, and which systems require stronger consistency or observability. This feedback loop transforms RCA from a reactive practice into an architecture-improvement mechanism.

11. Threats to Validity and Limitations

The first limitation is causal identifiability. Observational telemetry cannot always distinguish cause from correlation. Hidden confounders, missing logs, undocumented dependencies, and simultaneous deployments may produce ambiguous explanations. CG-RCA-DI mitigates this through counterfactual pruning and evidence confidence, but it cannot guarantee causal truth in every case. In high-stakes incidents, the framework's output should be treated as ranked evidence rather than unquestionable proof.

The second limitation is graph maintenance. Distributed systems change rapidly. Service topology, data schemas, queues, feature definitions, and deployment policies may drift faster than documentation. If the causal graph becomes stale, attribution quality will decline. Automated ingestion from runtime traces, schema registries, and deployment systems can reduce this risk, but human stewardship remains necessary.

The third limitation is scalability. Large enterprises may have thousands of services, millions of data objects, and billions of events. A complete causal graph over all entities may be infeasible. CG-RCA-DI addresses scalability through incident subgraphs, bounded horizons, typed edges, and localized traversal. Nevertheless, efficient indexing, streaming updates, and incremental graph computation are essential for production deployments.

The fourth limitation is predicate design. Poorly specified integrity predicates can produce noisy alerts or miss important failures. Predicate engineering requires domain expertise. A generic framework cannot automatically define all business rules. It can, however, provide reusable predicate templates for uniqueness, freshness, completeness, causal read consistency, referential integrity, and reconciliation balance.

The fifth limitation is remediation risk. An incorrect automated repair can worsen an incident, especially when financial records, medical records, identity data, or compliance logs are involved. CG-RCA-DI therefore emphasizes verification tests, reversible actions, and human approval for destructive changes. Future research should investigate formal safety constraints for automated data repair.

12. Future Research Directions

Future work should explore stronger integration between causal RCA and data contracts. Data contracts can define producer-consumer expectations, schema compatibility, freshness guarantees, and quality thresholds. If contract violations become graph nodes, RCA can more directly connect producer changes to consumer failures. This would also support pre-deployment risk analysis by identifying which contracts are likely to be affected by a change.

A second direction is causal benchmarking for data integrity incidents. Existing microservice RCA benchmarks often emphasize latency or service failure. The field needs benchmark suites containing duplicate writes, stale reads, schema drift, reconciliation defects, feature corruption, cache inconsistency, access-policy divergence, and cross-region replication anomalies. Such benchmarks should include ground-truth causal paths and realistic missing telemetry.

A third direction is integration with formal methods. Some integrity predicates can be specified as temporal logic, invariants, or transaction properties. Combining formal specifications with causal graph evidence could improve both detection and attribution. Formal models may also help identify impossible causal paths, reducing false positives.

A fourth direction is adaptive causal discovery under concept drift. Distributed systems evolve continuously, and causal relationships may change after deployments, migrations, or workload shifts. RCA frameworks need mechanisms to detect when learned causal edges are stale and to relearn them without excessive false alarms. Online causal discovery and drift-aware graph maintenance are promising areas.

A fifth direction is governance-aware automation. As AI-based engineering tools become more common, RCA systems should integrate with automated testing, release management, risk scoring, and compliance review. However, automation should be bounded by explainability and approval controls. The goal is not blind self-healing but accountable, evidence-based recovery.

13. Conclusion

Distributed data integrity incidents represent a growing reliability challenge because they arise at the intersection of microservices, replicated data, asynchronous events, weak consistency, automated deployment, and business semantics. Conventional RCA methods often localize service anomalies but fail to explain how a data value became incorrect across system boundaries. This paper proposed CG-RCA-DI, a causal graph-based root cause analysis framework for cross-system failure attribution and resolution. The framework models services, data objects, events, policies, deployments, and integrity predicates as a typed causal graph. It uses temporal reasoning, structural reachability, anomaly evidence, counterfactual pruning, and business criticality to rank root causes and recommend resolution actions.

The key argument is that data integrity RCA must be causal, semantic, and actionable. It must begin with violated predicates, trace backward through lineage and system dependencies, distinguish causes from symptoms, and connect explanations to safe remediation. By integrating observability, causal inference, distributed systems semantics, and governance feedback, CG-RCA-DI offers a foundation for more reliable and auditable incident response in complex data-intensive architectures. The framework does not eliminate uncertainty, but it makes uncertainty explicit and operationally useful. In doing so, it advances root cause analysis from dashboard-driven investigation toward evidence-grounded causal attribution and continuous architectural learning.

References

[1] J. Pearl, *Causality: Models, Reasoning, and Inference*, 2nd ed. Cambridge, U.K.: Cambridge University Press, 2009.

[2] Gunda SK, Yettapu SDR, Bodakunti S, Bikki SB. Decision Intelligence Methodology for AI-Driven Agile Software Lifecycle Governance and Architecture-Centered Project Management, 2023 Mar. 30;4(1):102-8. https://doi.org/10.63282/3050-9262.IJAIDSML-V4I1P112

[3] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978. https://doi.org/10.1145/359545.359563

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, 2007, pp. 205–220. https://doi.org/10.1145/1323293.1294281

[5] Gunda, S. K. G. (2023). The Future of Software Development and the Expanding Role of ML Models. International Journal of Emerging Research in Engineering and Technology, 4(2), 126-129. https://doi.org/10.63282/3050-922X.IJERET-V4I2P113

[6] P. Chen, Y. Qi, P. Zheng, and D. Hou, "CauseInfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems," in *IEEE INFOCOM 2014—IEEE Conference on Computer Communications*, 2014, pp. 1887–1895. https://doi.org/10.1109/INFOCOM.2014.6848128

[7] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002. https://doi.org/10.1145/564585.564601

[8] Sivva, S. D., Thalakanti, R. R., Bandari, S. S. G., & Yettapu, S. D. R. (2023). AI-Driven Decision Intelligence for Agile Software Lifecycle Governance: An Architecture-Centered Framework Integrating Machine Learning Defect Prediction and Automated Testing. International Journal of Emerging Trends in Computer Science and Information Technology, 4(4), 167-172. https://doi.org/10.63282/3050-9246.IJETCSIT-V4I4P118

[9] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems*, Lecture Notes in Computer Science, vol. 6976, 2011, pp. 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[10] M. A. Ikram, S. Chakraborty, S. Mitra, S. Bagchi, and M. K. Reiter, "Root cause analysis of failures in microservices through causal discovery," in *Advances in Neural Information Processing Systems*, vol. 35, 2022, pp. 31158–31170.

[11] Balerao, M. (2023). A converged artificial intelligence architecture for innovation, software lifecycle optimization, and cybersecurity risk mitigation. International Journal of Multidisciplinary Futuristic Development, 4(1), 117–120. https://doi.org/10.54660/IJMFD.2023.4.1.117-120

[12] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, 2011, pp. 401–416. https://doi.org/10.1145/2043556.2043593

[13] B. Li, X. Peng, Q. Xiang, H. Wang, T. Xie, J. Sun, and X. Liu, "Enjoy your observability: An industrial survey of microservice tracing and analysis," *Empirical Software Engineering*, vol. 27, no. 1, article 25, 2022. https://doi.org/10.1007/s10664-021-10063-9

[14] S. K. Gunda, "Analyzing Machine Learning Techniques for Software Defect Prediction: A Comprehensive Performance Comparison," 2024 Asian Conference on Intelligent Technologies (ACOIT), KOLAR, India, 2024, pp. 1-5, https://doi.org/10.1109/ACOIT62457.2024.10939610.

[15] P. Spirtes, C. Glymour, and R. Scheines, *Causation, Prediction, and Search*, 2nd ed. Cambridge, MA, USA: MIT Press, 2000.

[16] P. Wang, J. Xu, M. Ma, W. Lin, D. Pan, Y. Wang, and P. Chen, "CloudRanger: Root cause identification for cloud native systems," in *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2018, pp. 492–502. https://doi.org/10.1109/CCGRID.2018.00076

[17] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch, "Session guarantees for weakly consistent replicated data," in *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, 1994, pp. 140–149. https://doi.org/10.1109/PDIS.1994.331722

[18] Sivva, S. D. (2023). An end-to-end AI-based systems engineering paradigm for lifecycle governance, predictive quality assurance, automation economics, and cybersecurity intelligence. Journal of Frontiers in Multidisciplinary Research, 4(1), 600–604. https://doi.org/10.54660/.JFMR.2023.4.1.600-604

[19] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," *Proceedings of the VLDB Endowment*, vol. 7, no. 3, pp. 181–192, 2013. https://doi.org/10.14778/2732232.2732237

[20] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Google Technical Report, 2010.

[21] J. Peters, D. Janzing, and B. Schölkopf, *Elements of Causal Inference: Foundations and Learning Algorithms*. Cambridge, MA, USA: MIT Press, 2017.

[22] G. Yu, P. Chen, H. Chen, Z. Guan, Z. Huang, L. Jing, T. Weng, X. Sun, and X. Li, "MicroRank: End-to-end latency issue localization with extended spectrum analysis in microservice environments," in *Proceedings of The Web Conference 2021*, 2021, pp. 3087–3098. https://doi.org/10.1145/3442381.3449905

[23] Z. Zhao, T. Zhang, Z. Shen, H. Dong, X. Ma, X. Liu, and Y. Yang, "CHASE: A causal heterogeneous graph based framework for root cause analysis in multimodal microservice systems," arXiv:2406.19711, 2024.

[24] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds. Cham, Switzerland: Springer, 2017, pp. 195–216. https://doi.org/10.1007/978-3-319-67425-4_12

[25] C.-M. Lin, C. Chang, W.-Y. Wang, K.-D. Wang, and W.-C. Peng, "Root cause analysis in microservice using neural Granger causal discovery," arXiv: 2402.01140, 2024.