Journal of Information Systems Engineering and Management
2022, 7(1)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ **Research Article**

Historical Incident Reuse for Data Platform Reliability

Mogana Kumaran Sivaraman
Staff Software Engineer

ARTICLE INFO

Received: 03 Jan 2022

Revised: 14 Feb 2022

Accepted: 26 Feb 2022

ABSTRACT

The problem is familiar in practice. Repetitive incident diagnosis is a chronic issue in enterprise data platforms. Operational teams frequently re-encounter failures that resemble previously resolved issues. This study proposes a framework that turns historical incident knowledge into a reusable reliability resource. Each failure episode is represented using normalized telemetry, execution context, error attributes, and associated remediation knowledge, then stored in a searchable repository. When a new failure occurs, the framework compares it against historical entries to identify relevant analogs and support faster diagnosis. Five components organize the design: incident capture, canonical representation, repository indexing, similarity retrieval, and remediation surfacing. Conventional observability systems emphasize current-state visibility. This work, by contrast, treats prior operational knowledge as a first-class reliability mechanism that supports reuse across jobs, workflows, and execution environments. Comparative analysis shows that the framework addresses gaps left by pure log analysis, ticket-based search, runbook lookup, and ML-based anomaly detection, particularly in structured retrieval with integrated remediation knowledge. The result is a path toward shorter troubleshooting times. Incident handling becomes more consistent. Institutional learning strengthens in cloud and lakehouse data platforms.

Keywords: incident reuse, data platform reliability, operational knowledge, incident repository ,data platform

1. Introduction

The story repeats itself often. Operational failures in enterprise data platforms are rarely unique. Jobs fail because of resource exhaustion, schema drift, orchestration conflicts, and permission errors. These same failure categories recur across pipeline runs, teams, and execution windows. Failure distributions in production clusters are heavily skewed [1], [2]. A small number of root-cause categories account for the majority of incidents. The disconnect is quite striking. Despite this regularity, operational response remains largely reactive. Engineers diagnose each failure from scratch. They consult fragmented dashboards and log streams, supplemented by undocumented team knowledge, only to reach conclusions others have already reached.

Recurring failure modes in lakehouse and cloud platforms span a wide range. Schema drift, where an upstream producer changes a column type or drops a field without coordinating with consumers, causes ingestion pipelines to fail silently or with cryptic deserialization errors; out-of-memory failures during Apache Spark shuffle operations also halt jobs mid-execution and trigger cascading retries. Permission errors arise when service principals lose access to storage after credential rotation, a problem that manifests identically each cycle but is diagnosed anew by whichever engineer is on-call; similarly, orchestration race conditions, in which two workflows write to the same Delta Lake table concurrently, produce transaction conflict exceptions that require identifying the conflicting writer and adjusting scheduling. Each failure follows a recognizable pattern. Yet the on-call engineer typically arrives at a diagnosis the organization has already produced multiple times. The economic cost of repetitive troubleshooting is substantial. Industry analyses estimate that unplanned downtime

Copyright © 2022 by Author/s and Licensed by JISEM. This is an open access article distributed under the Creative Commons Attribution License which permitsunrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Journal of Information Systems Engineering and Management
2022, 7(1)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ **Research Article**

costs large enterprises between several hundred thousand and several million dollars per hour depending on system criticality [3]. Even without reaching full-outage severity, recurring failures erode engineering productivity: a team handling twenty incidents per week at a 90-minute average resolution time devotes thirty engineer-hours weekly, and if half resemble previously resolved issues, fifteen of those hours are spent rediscovering known solutions. Beyond direct labor, repeated failures erode SLA compliance. Downstream analytics also slip. Stakeholder confidence in the data platform takes a hit over time. Modern platforms generate extensive observability data. The volume is striking. Logs, metrics, traces, and execution metadata are captured at scale, and monitoring tools provide real-time visibility into system health [4], [6]. Yet observability today answers what is happening now. It does not say what happened before under similar circumstances. The gap between observing a failure and connecting it to prior operational experience remains unaddressed.

This disconnect has measurable consequences in production. Resolution times suffer first. Incident resolution times remain high even for recurring failure patterns. Analysis of large-scale outages at major cloud providers shows that a substantial fraction of incidents share root causes with prior events, yet the time to diagnose them does not decrease proportionally [5]. Diagnostic consistency varies across teams and shifts. Knowledge gained from resolving one incident is seldom captured in a form that future operators can retrieve and apply. The result is a persistent loss of operational learning.

Here is the core proposal. This study presents a framework for historical incident reuse in enterprise data platforms. The central idea is to capture each failure episode as a structured operational record, store these records in a searchable repository, and enable comparison between new failures and previously resolved incidents, thereby shifting incident handling from isolated troubleshooting toward a cumulative process that builds on prior experience.

The contributions of this study are:

1. A formal definition of the incident reuse problem in data platform operations, distinguishing it from failure detection, classification, and ticket management.
2. An architectural specification for a historical incident repository comprising five components: incident capture, canonical representation with workload-relative normalization, dual-index storage (fingerprint and vector), similarity retrieval, and remediation surfacing.
3. An analysis of design trade-offs including representation granularity, normalization strategy, cold-start mitigation, and feedback-driven refinement, grounded in the operational constraints of enterprise lakehouse environments.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 presents the proposed framework. Section 4 discusses design considerations and practical implications. Section 5 identifies limitations. Section 6 concludes.

2. Related Work

2.1 Observability in Data Platforms

Observability is well established today. Modern data-platform observability systems collect logs, metrics, traces, and execution metadata to support real-time monitoring and alerting. Tools such as Prometheus, Grafana, and Datadog provide visibility into job execution alongside platform-native telemetry [4]. Resource usage and error states are also surfaced. The lakehouse architecture unifies data warehousing and advanced analytics [6], and its system tables expose job-level telemetry suitable as raw input for incident capture.

Several telemetry sources matter here. Within the Databricks lakehouse ecosystem, several relevant sources can support incident capture. System tables record job-run metadata. Standard fields include start time, termination state, and cluster identifier, with error messages alongside. Unity Catalog audit logs track data access and permission changes. Cluster event logs capture node additions, driver

Copyright © 2022 by Author/s and Licensed by JISEM. This is an open access article distributed under the Creative Commons Attribution License which permitsunrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Journal of Information Systems Engineering and Management
2022, 7(1)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ **Research Article**

failures, and autoscaling decisions with timestamps. Spark executor metrics report per-stage and per-task resource consumption including executor memory, GC time, and shuffle volumes. Together these sources provide a multi-layered telemetry surface for constructing incident records. Each, however, uses a different schema, retention policy, and access mechanism. No existing tool integrates them into a unified incident representation suitable for historical comparison.
These systems handle current-state inspection well. They are not designed to support historical reasoning. Observability data is time-series oriented and optimized for recency rather than structured comparison across historical incidents. Reiss et al. [1] characterized workload patterns across a Google production cluster of over 12,000 machines. Di et al. [2] compared failures across cloud and grid environments. Failure rates and distributions differ substantially by platform type. These findings underscore the need for workload-aware normalization in any incident comparison scheme.

2.2 Incident Management and AIOps

AIOps spans several methods today. The AIOps literature addresses failure detection, anomaly identification, root cause analysis, and automated remediation [7]. DeepLog [8] applies sequence models to log streams to detect deviations from normal execution. The literature extends further still. Drain [9] extracts log templates using a fixed-depth tree parser. LogRobust [10] uses semantic embeddings that remain stable across log format changes. Zhou et al. [11] surveyed fault analysis and debugging in microservice systems, highlighting the challenges of diagnosing failures that propagate across service boundaries.

The focus of most AIOps work is narrow. Most research targets detecting or classifying failures at the point of occurrence. Comparatively little attention has been given to systematic reuse of resolved incidents as a resource for future diagnosis. Chen et al. [5] analyzed outage patterns at Meta and found that many incidents recur in recognizable forms, yet the processes for capturing and reusing resolution knowledge remain ad hoc. Incident management platforms such as PagerDuty and ServiceNow store ticket histories, but these records are typically unstructured text and are not directly connected to the telemetry signals that characterize each failure.

Enterprise incident management platforms serve an important coordination function but are architecturally limited as knowledge retrieval systems. PagerDuty orchestrates alert routing and escalation. ServiceNow provides a structured workflow for ticket lifecycle management. Opsgenie offers alerting and on-call scheduling with integration hooks into monitoring tools. These platforms share a design centered on the current incident as a workflow item rather than as a knowledge artifact. Ticket descriptions are free-form text. Resolution notes are optional and unstructured. Search operates over keyword matching rather than semantic or telemetry-aware retrieval. An engineer searching a ticketing system for a prior Spark-shuffle OOM must formulate the right keyword query. Hundreds of results may need scanning. Analogy must be assessed manually. The absence of structured telemetry linkage means two incidents with identical root causes but different surface descriptions may never be connected.

2.3 Case-Based Reasoning

CBR offers a well-known approach. Case-based reasoning (CBR) provides a well-established framework for solving new problems by retrieving and adapting solutions from similar past cases. The cycle has four phases. Aamodt and Plaza [12] formalized the CBR cycle as follows: retrieve similar cases, reuse the solution from the best-matching case, revise the proposed solution if necessary, and retain the new experience for future use. Kolodner [13] provided a comprehensive treatment of CBR foundations, including case representation, indexing strategies, and adaptation mechanisms.

Copyright © 2022 by Author/s and Licensed by JISEM. This is an open access article distributed under the Creative Commons Attribution License which permitsunrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Journal of Information Systems Engineering and Management
2022, 7(1)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ **Research Article**

The mapping is direct in this case. The retrieve-reuse-revise-retain cycle maps directly onto incident resolution. In the retrieve phase, a new failure triggers a search for historical incidents with similar characteristics; similarity criteria may include error type, affected component, resource profile, and execution context. In the reuse phase, the remediation steps from the most relevant historical incident are proposed as a starting hypothesis. In the revise phase, the operator adapts the proposed remediation to account for differences between the historical and current incidents, such as a changed cluster configuration or an updated library version. In the retain phase, the resolved incident, including any adaptations, is stored as a new case available for future retrieval. This cycle ensures the knowledge base grows richer with each resolved incident.

CBR has a long applied track record. It has been used in several technical domains with structural similarities to data platform operations. In manufacturing fault diagnosis, CBR systems match new equipment failures against historical repair records, reducing mean time to repair [14]. IT helpdesk systems have used CBR to suggest resolutions for user-reported issues, indexing cases by symptom descriptions and system configurations [15]. Medical decision support systems retrieve treatment plans from similar patient histories [16]. These applications share a common pattern. Problems recur in recognizable but not identical forms. Expert knowledge is valuable yet difficult to codify as explicit rules. The solution space is also large enough that exhaustive enumeration is impractical.
Its application to data platform operations has been limited, partly because operational incidents have not traditionally been captured in the structured, comparable format that CBR requires. The framework proposed in this study addresses this gap by providing the canonical representation and indexing infrastructure needed to operationalize CBR principles in a platform reliability context.

2.4 Gap and Positioning

The picture is fragmented across the literature. The three bodies of work reviewed above address complementary but disconnected concerns. Observability systems capture telemetry but do not organize it for historical comparison. AIOps methods detect and classify failures but do not systematically connect new incidents to prior resolutions. CBR provides the reasoning framework for experience reuse but has not been applied to data platform operations due to the absence of structured incident representations.
The framework bridges these gaps. The framework proposed in this study defines a structured mechanism for capturing, representing, storing, and retrieving operational incidents. The mechanism connects platform telemetry to actionable historical knowledge through the CBR paradigm. The specific contribution relative to existing work is the integration of workload-aware canonical representation with dual-index retrieval in a design tailored to the operational characteristics of lakehouse and cloud data platforms.

2.5 Knowledge Management in Operations

Most practice today remains informal. Operational knowledge sits in informal forms across most teams. Across enterprise data platform teams, several semi-structured mechanisms now capture it. Runbooks document step-by-step procedures for handling known failure scenarios, typically maintained as wiki pages or version-controlled Markdown files. Post-mortem reports provide detailed retrospective analyses of significant incidents. Standard contents include timelines and root cause findings. Corrective actions are recorded too. Internal wikis accumulate troubleshooting tips and configuration notes. Tribal knowledge contributed by individual engineers builds up over time. Chat transcripts in tools such as Slack or Microsoft Teams preserve real-time diagnostic conversations, though these are rarely indexed or curated for future retrieval.
Each source has limitations that prevent effective reuse. Runbooks assume the failure matches a predefined scenario and become stale as platforms evolve. Post-mortem reports are produced only for high-severity incidents, leaving routine failures undocumented. Wikis suffer from fragmentation and

Copyright © 2022 by Author/s and Licensed by JISEM. This is an open access article distributed under the Creative Commons Attribution License which permitsunrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Journal of Information Systems Engineering and Management
2022, 7(1)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ Research Article

inconsistent formatting. Chat transcripts are conversational and lack the context needed for a newcomer to extract actionable information. The common deficiency is the absence of structured, telemetry-linked representations that support similarity-based retrieval. An engineer facing a new failure cannot query a runbook collection by telemetry profile, nor search post-mortem archives using the specific combination of error code, resource metrics, and execution context that characterizes the current incident. The framework proposed here addresses this gap by converting operational knowledge into structured incident records indexed for both exact and approximate retrieval.

3. Proposed Framework

The proposed framework consists of five components: incident capture, canonical representation, repository storage, similarity retrieval, and remediation surfacing. Figure 1 illustrates the overall architecture.

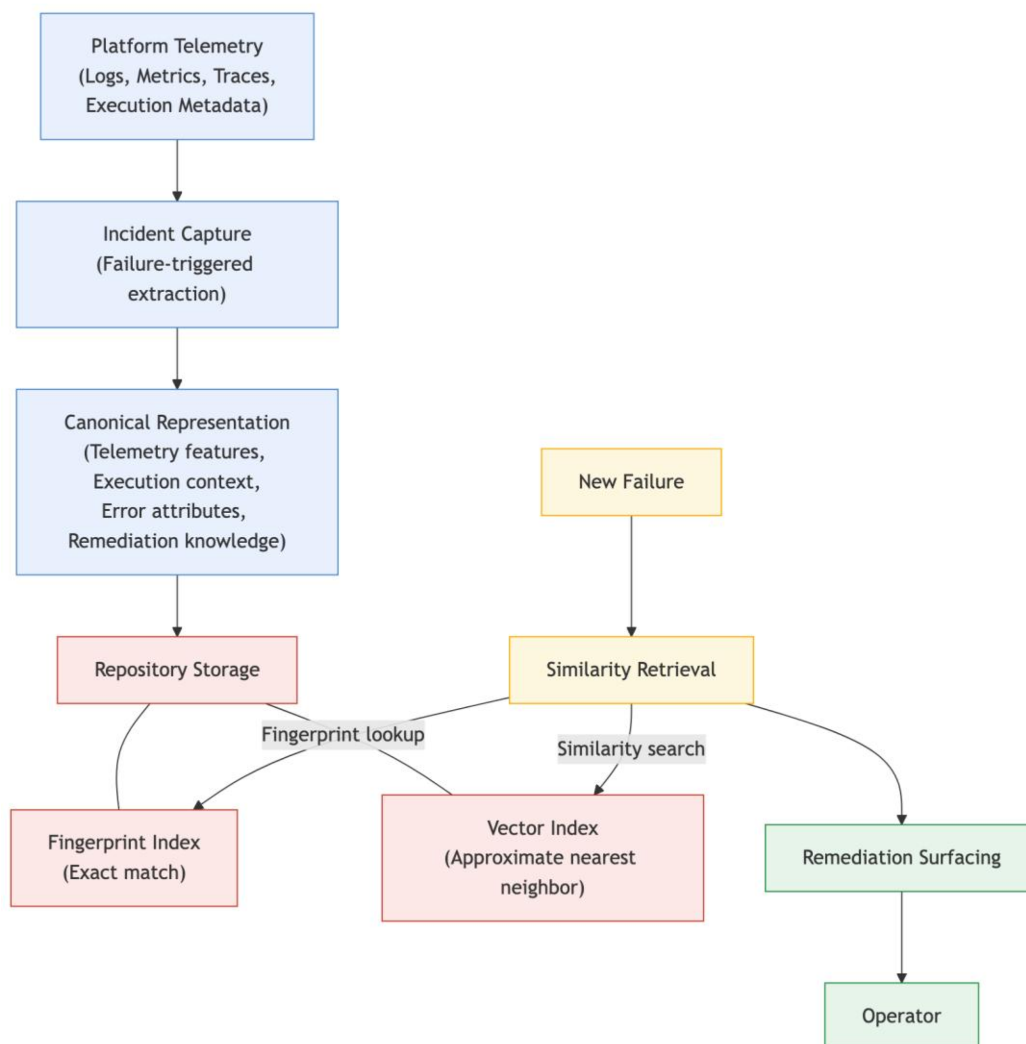

Figure 1. Architecture of the historical incident reuse framework. Platform telemetry feeds into incident capture, which produces canonical representations stored in a dual-indexed repository. When a new failure occurs, similarity retrieval queries both the fingerprint index (for exact matches) and the vector index (for approximate matches), then surfaces relevant historical remediation knowledge to the operator.

Copyright © 2022 by Author/s and Licensed by JISEM. This is an open access article distributed under the Creative Commons Attribution License which permitsunrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

2022, 7(1)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ Research Article

3.1 Incident Capture

Capture happens at the start. The first component extracts failure-relevant information from platform telemetry at the time of an incident. Data sources include execution logs and runtime metrics. Error codes and stack traces are pulled in too. Resource utilization records and orchestration metadata complete the bundle. In lakehouse environments, Spark's event logging and metrics system provides structured access to job-level telemetry [4]. Available signals include cluster configuration and task-level timing. I/O statistics round out the picture. The capture process is triggered by failure events such as job termination, task retry exhaustion, or SLA violation.

Spark workloads need special handling. For Apache Spark workloads, the capture component extracts telemetry signals diagnostic of common failure modes. Executor memory metrics matter most for identifying out-of-memory conditions. Peak heap, off-heap consumption, and the storage-versus-execution fraction together expose how memory is being spent. Shuffle spill volumes indicate memory pressure during wide transformations such as joins and aggregations. Stage-level task duration distributions reveal skew conditions where a few tasks take disproportionately longer than peers, often signaling partition imbalance. Stage failure counts and retry histories locate the failure in the execution DAG. Garbage-collection time provides an early signal of memory exhaustion before an explicit OOM error. Input and output record counts support detection of data-volume anomalies, such as an upstream source producing an order of magnitude more records than historical norms.

Capture must be timely and consistent. If incident records are constructed only after manual triage, critical contextual information may be lost or inconsistently recorded. The framework therefore assumes automated or semi-automated extraction from the observability layer, with human annotation added post hoc where needed.

3.2 Canonical Representation

Raw telemetry is highly heterogeneous. It varies in format, scale, and granularity across workloads and platforms. To enable meaningful comparison, each incident must be transformed into a canonical representation. The proposed representation includes four components:

Table 1. Components of the canonical incident representation

Component	Description	Examples
Telemetry features	Normalized execution metrics	CPU utilization, memory peak, shuffle bytes, task duration
Execution context	Workload and environment metadata	Job type, cluster configuration, data source, schedule
Error attributes	Failure classification signals	Error code, exception type, failure stage, retry count
Remediation knowledge	Resolution information	Root cause label, fix applied, time to resolution, operator notes

The key step is normalization. It is essential for cross-workload comparability. The framework applies workload-relative normalization rather than fixed global thresholds, so metric values reflect deviation from expected behavior for each pipeline or job type. Empirical studies show that metric distributions differ substantially across workload classes and platform types [2], [1], supporting this choice.

The choice of percentile-based normalization reflects specific properties of data platform telemetry. Z-score normalization assumes approximately Gaussian distributions, an assumption that frequently fails: shuffle-spill volumes, for example, follow a heavy-tailed distribution where most successful runs

Copyright © 2022 by Author/s and Licensed by JISEM. This is an open access article distributed under the Creative Commons Attribution License which permitsunrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Journal of Information Systems Engineering and Management
2022, 7(1)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ **Research Article**

produce zero spill while failing runs may produce gigabytes. A z-score would assign moderate scores to clearly anomalous values because the high variance absorbs the deviation. Min-max normalization maps values to a fixed range but is sensitive to outliers; a single extreme value compresses the normalized range for all subsequent observations. Percentile-based normalization expresses each metric value as its rank position within the historical distribution of the same workload. It conveys operational significance regardless of absolute scale, cluster size, or extreme outliers. This property is valuable in heterogeneous environments, where the same logical pipeline may execute on clusters ranging from four to sixty-four nodes. Figure 2 illustrates how the four component groups combine into a single canonical record. Figure 3 traces the incident lifecycle from initial failure detection through resolution and retention, with the feedback loop returning resolved cases to the matching stage.

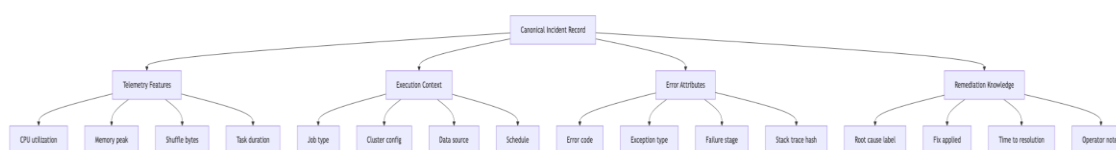

Figure 2. Structure of the canonical incident representation. Each record combines four component groups that together capture the telemetry signature, operational context, failure identity, and resolution knowledge of an incident.

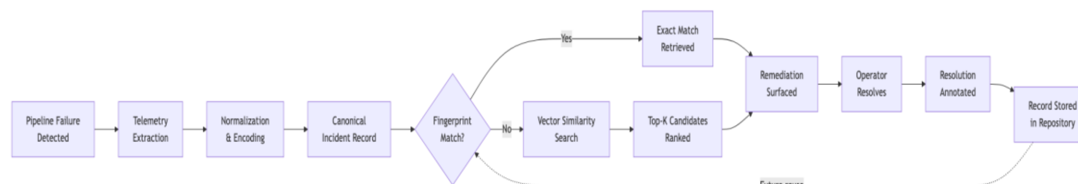

Figure 3. Incident lifecycle from initial failure detection through resolution and retention. The feedback loop from stored records back to the matching stage enables cumulative knowledge reuse.

3.3 Repository Storage and Indexing

Storage is the next concern. Canonical incident records are stored in a persistent repository that supports both exact and approximate lookup. The repository maintains two index structures:

1. Deterministic fingerprints for exact matching. A fingerprint is a hash computed from a subset of incident attributes (e.g., error code, failure stage, job identifier). Exact fingerprint matches identify strict recurrences where the failure signature is identical.
2. Vector index for similarity-based retrieval. The canonical representation, or a learned embedding derived from it, is indexed in a vector store that supports approximate nearest-neighbor search. This enables retrieval of incidents that are operationally similar but not identical in surface attributes.

Vector index choice is a three-way trade-off. The choice of vector index algorithm involves trade-offs between recall, latency, and memory. Hierarchical Navigable Small World (HNSW) graphs offer high recall with logarithmic query time. They suit repositories in the tens of thousands to low millions of incidents. Their incremental insertion matches continuous repository growth. Inverted File (IVF) indices partition the vector space into clusters and search only the nearest ones. They use less memory per vector. The trade-off is periodic centroid retraining as the distribution shifts. HNSW provides a practical default for slowly growing repositories; IVF with periodic retraining may fit environments generating thousands of incidents per day. Embedding dimensionality affects performance: higher dimensions capture finer distinctions but increase storage and can degrade nearest-neighbor accuracy

Copyright © 2022 by Author/s and Licensed by JISEM. This is an open access article distributed under the Creative Commons Attribution License which permitsunrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

due to distance concentration. Empirical tuning of dimensionality against retrieval precision on a held-out set is recommended at deployment.
The dual-index design allows the system to handle both exact recurrences and approximate matches. This distinction is important because many recurring failures share root causes without producing identical error signatures.

3.4 Similarity Retrieval

Retrieval works in two stages. When a new failure occurs, the framework constructs its canonical representation and queries the repository. The retrieval process proceeds in two stages:

1. Fingerprint lookup. The system first checks for exact fingerprint matches. If a match is found, the corresponding historical incident and its remediation record are returned immediately.
2. Similarity search. If no exact match is found, the system performs a nearest-neighbor search over the vector index, returning the top-k most similar historical incidents along with their similarity scores.

The similarity function has two options. It may operate over raw normalized features such as cosine similarity. Alternatively, learned embeddings can capture higher-order operational relationships. This two-stage design follows the retrieve phase of the CBR cycle [12], where the goal is to identify the stored cases most relevant to the current problem.

3.5 Remediation Surfacing

The final step is presentation. Retrieved historical incidents are shown to operators alongside their associated remediation knowledge. The surfacing layer provides:

- The matched historical incident record, including telemetry summary and error attributes.
- The remediation action taken previously, including root cause label and resolution steps.
- A similarity or confidence score indicating how closely the historical incident matches the current failure.
- Contextual differences between the historical and current incidents, to help the operator assess whether the prior resolution applies.

To illustrate remediation surfacing, consider the following scenario. A nightly ETL job that joins a large fact table with a dimension table fails with `java.lang.OutOfMemoryError: Java heap space` during the shuffle read phase of stage 7. Incident capture extracts the relevant signals. Peak executor memory reaches 14.2 GB against a 15 GB allocation. Shuffle spill totals 48 GB. GC time runs at 38% of task execution. Three prior task retries have already been recorded. The canonical representation places these metrics at the 97th, 94th, 96th, and 89th percentiles for this workload class. The fingerprint finds no exact match because the job was recently migrated to a new cluster type. The vector similarity search returns a historical incident from four months earlier with cosine similarity 0.91. That earlier incident captures the same logical join on a prior cluster configuration with comparable metrics, and its remediation record identifies the root cause: an un-bucketed join key was producing skewed shuffle partitions. The resolution was to repartition the fact table by the join key and increase `spark.sql.shuffle.partitions` from 200 to 800. The surfacing layer presents this case alongside a diff of configuration differences, enabling the engineer to confirm the partition-skew diagnosis in minutes, apply the fix, and retain an updated record covering both cluster configurations.
The framework does not automate remediation. The role is supportive only. It surfaces relevant historical context to help human decision-making, reducing the time and effort required to diagnose familiar failure patterns. This corresponds to the reuse phase of the CBR cycle, where the operator adapts prior solutions to the current context rather than applying them mechanically [13].

Copyright © 2022 by Author/s and Licensed by JISEM. This is an open access article distributed under the Creative Commons Attribution License which permitsunrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

2022, 7(1)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

4. Discussion

4.1 Design Considerations

Three considerations stand out. Several design decisions shape the practical viability of the framework.
Representation granularity. The canonical representation must be detailed enough to support meaningful comparison but compact enough to scale across thousands of incidents. Overly detailed representations increase storage and retrieval cost without proportional improvement in matching quality. The four-component structure proposed here balances expressiveness with tractability.
Normalization strategy. Workload-relative normalization is preferred over global normalization because metric distributions vary substantially across job types, cluster sizes, and execution patterns [2]. A percentile-based approach, where each feature is normalized against the historical distribution of the same workload, preserves interpretability while reducing sensitivity to scale differences.
Cold start. The repository starts empty. The framework requires an initial population of historical incidents before it can provide useful retrieval. In early deployment, the repository may contain too few records to support reliable matching. A phased rollout strategy, beginning with the highest-frequency failure categories, can mitigate this limitation. Cluster trace studies suggest that a small number of failure types account for a disproportionate share of incidents [1], so even a modest initial repository may cover the most common cases.
Feedback and refinement. Operator feedback closes the loop. Retrieval quality improves when operators provide feedback on the relevance of surfaced incidents. A feedback mechanism that captures whether the suggested historical analog was helpful allows the system to refine its similarity model over time. This corresponds to the revise and retain phases of the CBR cycle [12].

4.2 Practical Implications

Several practical payoffs follow from this design. The framework addresses several operational pain points.

- Reduced troubleshooting time. By surfacing relevant historical incidents, the framework reduces the diagnostic search space for recurring failures.
- Improved consistency. Structured incident records and standardized retrieval reduce variability in how different teams or operators handle similar failures.
- Institutional learning. The repository accumulates operational knowledge over time. Ephemeral troubleshooting experience becomes a persistent organizational resource.
- Foundation for automation. A populated incident repository provides the knowledge base needed for future intelligent operational assistants that can reason over historical failure patterns.

4.3 Relation to Existing Systems

Existing tools have a place. The proposed framework is complementary to, not a replacement for, existing observability and incident management tools. Observability systems provide the raw telemetry from which incident records are constructed. Apache Spark platform telemetry [4] and log parsing tools such as Drain [9] supply the structured inputs for incident capture and canonical representation. Ticketing systems may serve as a source of remediation annotations. The framework adds a structured retrieval layer between these existing systems and the operator, filling a gap that current tooling does not address.

4.4 Comparison with Related Approaches

To situate the proposed framework relative to existing operational strategies, Table 2 compares it against four commonly used approaches for handling recurring failures in data platform environments: pure log analysis, ticket-based search, runbook lookup, and ML-based anomaly

Copyright © 2022 by Author/s and Licensed by JISEM. This is an open access article distributed under the Creative Commons Attribution License which permitsunrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Journal of Information Systems Engineering and Management
2022, 7(1)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ **Research Article**

detection. Each approach is evaluated across five dimensions that are relevant to effective incident reuse.

Table 2. Comparison of approaches for recurring failure handling

Dimension	Pure Log Analysis	Ticket-Based Search	Runbook Lookup	ML-Based Anomaly Detection	Proposed Framework
Structured retrieval	Low; keyword/regex over raw logs	Low; free-text search over ticket fields	Medium; structured by scenario but manual selection	Low; outputs anomaly scores, not retrievable cases	High; dual-index retrieval over canonical representations
Remediation reuse	None; logs record symptoms, not fixes	Partial; resolution notes are optional and unstructured	High for known scenarios; absent for novel failures	None; detection only, no remediation linkage	High; each incident record includes structured remediation knowledge
Telemetry integration	High; operates directly on log data	None; tickets are disconnected from telemetry	None; runbooks reference expected metrics but do not query live data	High; models are trained on metric and log data	High; canonical representation integrates telemetry, context, and error attributes
Learning over time	None; static log corpus	Passive; tickets accumulate but are not curated for reuse	Manual; runbooks require explicit authoring and updates	Partial; models retrain on new data but do not retain case-level knowledge	Active; the retain phase of the CBR cycle adds each resolved incident to the repository
Handling of novel failure variants	Poor; regex patterns miss variations	Poor; keyword queries miss semantically similar but lexically different tickets	Poor; runbooks cover predefined scenarios only	Moderate; models generalize across anomaly patterns	Good; vector similarity retrieval identifies operationally similar incidents despite surface-level differences

Logs come first in any incident. Pure log analysis is the most immediate tool available to an on-call engineer. By searching log files for error messages, stack traces, and exception types, the engineer can identify what went wrong in the current execution. However, logs record symptoms rather than diagnoses. They do not capture the reasoning that led a prior engineer to a root cause, nor do they record the remediation that resolved the issue. Logs record symptoms, not diagnoses. Searching for a prior occurrence of the same error message may locate the timestamp of a previous failure. It provides

Copyright © 2022 by Author/s and Licensed by JISEM. This is an open access article distributed under the Creative Commons Attribution License which permitsunrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

no guidance on how it was resolved. Tickets are the next layer to consider. Ticket-based search, using platforms such as ServiceNow or Jira, offers access to resolution notes when they have been recorded. In practice, resolution notes are frequently absent, terse, or ambiguous. Furthermore, ticket search relies on keyword matching over free-text fields, which fails when different engineers describe the same failure using different terminology. A ticket titled "Spark job crash on join" and another titled "OOM during shuffle phase" may describe the same root cause, but a keyword search for either term will miss the other.

Runbooks have a clear niche. Runbook lookup is effective for well-characterized, high-frequency failure scenarios. A runbook for handling Delta Lake transaction conflicts, for example, can guide an operator through the diagnostic and resolution steps in a standardized manner. The limitation of runbooks is their rigidity: they cover only scenarios that have been explicitly documented, and they do not adapt to variations in execution environment, data volume, or platform version. Maintaining runbooks is labor-intensive, and coverage gaps are common for lower-severity or less frequent failure modes.Anomaly detection is a different tool. ML-based anomaly detection identifies deviations from normal behavior in metric streams and log sequences. These systems are valuable for alerting operators to potential problems, but they do not provide remediation guidance. An anomaly detector may flag that executor memory usage exceeded the historical norm, but it does not indicate whether the appropriate response is to increase memory allocation, repartition the data, or fix an upstream schema change. Moreover, anomaly detection models do not retain case-level knowledge; they generalize statistical patterns but discard the specifics of individual incidents.

The proposed framework addresses the limitations of each approach by combining structured telemetry integration with case-level remediation knowledge and similarity-based retrieval. Its principal advantage is that it treats each resolved incident as a reusable knowledge artifact, indexed for both exact and approximate retrieval, rather than as a disposable workflow item.

5. Limitations

Several caveats apply to this work. The evaluation in this study is based on case-based analysis and systematic comparison rather than large-scale controlled experiments. Future work should extend the evaluation to larger-scale production deployments. Several specific limitations should be noted:

1. Scale of evaluation. The framework has been evaluated through case-based analysis and systematic comparison rather than large-scale controlled experiments with production incident data. Retrieval precision, recall, and latency at production scale remain to be established. A controlled evaluation using anonymized incident logs from a production lakehouse environment would measure precision at varying k, resolution-time reduction when historical analogs are available, and operator satisfaction with surfaced recommendations.
2. Representation assumptions. The four-component canonical representation is proposed as a general design, but its suitability may vary across platforms, workload types, and organizational contexts. Organizations with highly specialized workloads, such as streaming pipelines with sub-second latency requirements, may require additional representation components that capture temporal dynamics not addressed by the current design. A modular representation schema that supports domain-specific extensions could mitigate this limitation.
3. Scalability. The dual-index retrieval strategy has not been evaluated at the scale of large enterprise environments with thousands of daily incidents. At high ingestion rates, the vector index may require partitioning or hierarchical organization to maintain acceptable query latency. Periodic index maintenance, including compaction and rebalancing, would add operational overhead that must be weighed against the benefits of incident reuse.

Copyright © 2022 by Author/s and Licensed by JISEM. This is an open access article distributed under the Creative Commons Attribution License which permitsunrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

4. Remediation quality. The value of surfaced remediation knowledge depends on the quality and completeness of the original annotations. Poorly documented resolutions will limit the utility of the reuse mechanism. Encouraging thorough remediation documentation requires organizational incentives and tooling that lowers the annotation burden, such as pre-populated templates that extract key fields from the telemetry and present them for operator confirmation rather than requiring free-form entry.
5. Maintenance burden. The repository requires ongoing curation to handle schema evolution, deprecated failure patterns, and changing platform configurations. An expiration or archival policy for outdated incidents, combined with periodic review of retrieval accuracy, can prevent the repository from accumulating stale records that degrade matching quality.

6. Conclusions

The contribution is concrete and actionable. This study proposed a framework for historical incident reuse in enterprise data platforms. The framework captures failure episodes as structured operational records. It stores them in a searchable repository with dual indexing. When new failures occur, it retrieves relevant historical analogs for comparison. The design treats prior operational knowledge as a first-class reliability mechanism, operationalizing the case-based reasoning cycle of retrieve, reuse, revise, and retain [12] in the context of data platform operations.
The primary contribution is architectural in nature. The specification connects existing observability data to actionable historical knowledge through canonical representation and similarity-based retrieval. The framework is intended to reduce troubleshooting time. It also improves diagnostic consistency. Institutional memory grows stronger in cloud and lakehouse environments as a side effect. The implications go further. Beyond immediate operational benefits, the framework reshapes how data engineering organizations manage reliability knowledge. As platforms grow in complexity, the cost of unstructured knowledge management compounds. Each departure or team rotation risks losing accumulated expertise. Platform migrations carry the same risk. The fix is structural in nature. A structured incident repository converts this perishable knowledge into a durable organizational asset. These principles generalize. Canonical representation, telemetry-aware indexing, and similarity-based retrieval are not specific to any single platform, and they could be adapted to other operational domains.
Several research directions remain open. Future work should extend evaluation to larger-scale production environments while also measuring retrieval quality under realistic operational conditions. Learned similarity models merit investigation. Several directions merit investigation. First, integration with intelligent operational agents could enable natural-language querying of the incident repository. Operators could describe a failure conversationally. The system would return structured historical analogs. Second, real-time retrieval triggered by streaming failure signals could reduce latency between failure detection and remediation surfacing. Third, cross-platform federation through a shared canonical schema could enable knowledge reuse across environments that currently operate in isolation.

Declarations

Conflict of Interest

The author declares no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Funding Statement

This research received no external funding.

Copyright © 2022 by Author/s and Licensed by JISEM. This is an open access article distributed under the Creative Commons Attribution License which permitsunrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Ethics Approval

Not applicable. This study involves no human subjects, animal experiments, or sensitive data.

Author Contributions

The author is solely responsible for all aspects of this work, including Conceptualization, Methodology, Investigation, Formal Analysis, Writing — Original Draft, and Writing — Review and Editing.

Data Availability Statement

No new data were created or analyzed in this study. The framework design and evaluation are based on publicly available literature and architectural analysis of enterprise data platform operations.

References

[1] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in Proc. 3rd ACM Symp. Cloud Computing (SoCC), 2012, Art. 7.

[2] S. Di, D. Kondo, and F. Cappello, "Characterizing and modeling cloud applications/jobs on a Google data center," J. Parallel Distrib. Comput., vol. 72, no. 4, pp. 1-12, 2012.

[3] Ponemon Institute, "Cost of data center outages," 2022. [Online]. Available: Ponemon Institute Research Report.

[4] M. Zaharia et al., "Apache Spark: A unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56-65, 2016.

[5] Y. Chen et al., "How Meta manages production incidents at scale," in Proc. 2022 USENIX Annu. Tech. Conf. (USENIX ATC '22), 2022.

[6] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia, "Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics," in Proc. CIDR, 2021.

[7] Y. Dang, Q. Lin, and P. Huang, "AIOps: Real-world challenges and research innovations," in *Proc. 41st IEEE/ACM Int. Conf. Software Engineering: Companion (ICSE-Companion)*, 2019, pp. 4-5.

[8] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly detection and diagnosis from system logs through deep learning," in Proc. 2017 ACM SIGSAC Conf. Comput. Commun. Security (CCS), 2017, pp. 1285-1298.

[9] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in Proc. 2017 IEEE Int. Conf. Web Services (ICWS), 2017, pp. 33-40.

[10] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, J. Chen, R. Zhang, S. Tao, P. Sun, and R. Zhou, "LogRobust: Robust anomaly detection through template-aware log feature extraction," in Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE), 2019, pp. 508-518.

[11] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Trans. Software Engineering*, vol. 47, no. 2, pp. 243-260, 2019.

[12] A. Aamodt and E. Plaza, "Case-based reasoning: Foundational issues, methodological variations, and system approaches," AI Commun., vol. 7, no. 1, pp. 39-59, 1994.

[13] J. L. Kolodner, Case-Based Reasoning. San Mateo, CA: Morgan Kaufmann, 1993.

[14] P. Cunningham, A. Tsymbal, and S. Puuronen, "Maintaining case-based reasoning systems," Knowl. Eng. Rev., vol. 18, no. 2, pp. 153-157, 2003.

Copyright © 2022 by Author/s and Licensed by JISEM. This is an open access article distributed under the Creative Commons Attribution License which permitsunrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Journal of Information Systems Engineering and Management
2022, 7(1)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ Research Article

[15] I. Watson, "Case-based reasoning is a methodology not a technology," Knowl.-Based Syst., vol. 12, no. 5-6, pp. 303-308, 1999.

[16] I. Bichindaritz and C. Marling, "Case-based reasoning in the health sciences: What's next?" Artif. Intell. Med., vol. 36, no. 2, pp. 127-135, 2006.

Copyright © 2022 by Author/s and Licensed by JISEM. This is an open access article distributed under the Creative Commons Attribution License which permitsunrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.