Journal of Information Systems Engineering and Management
2024, 9(4)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

The Incident Memory Layer: An Architectural Component for Lakehouse Reliability

Mogana Kumaran Sivaraman
Sr Staff Software Engineer

ARTICLE INFO

Received: 01 Sept 2024

Revised: 20 Oct 2024

Accepted: 29 Oct 2024

ABSTRACT

Lakehouse platforms support diverse analytical and machine learning workloads. Operational troubleshooting, however, remains fragmented across monitoring dashboards, log streams, ticketing platforms, and undocumented team experience. This study proposes a searchable incident memory layer as a dedicated architectural component for modern lakehouse operations. The layer is a persistent, queryable store of structured incident records positioned between observability infrastructure and operational decision-making. The architecture comprises five components: incident ingestion, canonical representation, indexing, similarity search, and retrieval interfaces. The memory layer fills a specific gap in the lakehouse operational stack: observability tools support real-time inspection but do not retain structured incident knowledge; ticketing systems store resolution records but lack telemetry integration; log analysis tools detect patterns but do not connect detected anomalies to historical remediation outcomes. The proposed layer bridges these systems by providing a unified, searchable knowledge base that links failure signatures to resolution histories. A comparative analysis shows that the memory layer addresses capabilities absent from each existing category. Its inclusion enables historical reasoning over operational failures. Diagnosis becomes more consistent across teams. The layer also provides the knowledge substrate for future agentic systems that operate over incident histories.

Keywords: incident memory layer, lakehouse operations, operational architecture, failure diagnosis

1. Introduction

The operational picture is fragmented today. Modern lakehouse platforms manage analytical pipelines and ML training on shared infrastructure. Streaming ingestion and ad hoc queries run alongside on the same fabric. These workloads fail regularly. Jobs exhaust memory. Schema changes propagate errors. Orchestration schedules conflict, and permission configurations break. Operations teams respond using monitoring dashboards, log inspection tools, alerting services, and ticketing systems. Each tool addresses a narrow slice of the diagnostic process. None retains structured knowledge of how past failures were identified, diagnosed, and resolved.

This fragmentation produces a characteristic pattern. When a failure occurs, an engineer consults real-time metrics and searches logs. Ticketing platforms are checked for prior reports. Colleagues are queried as a final resort. If the same failure occurred last month and was resolved by someone else, that resolution is likely recorded only in a closed ticket narrative or chat thread. The diagnostic effort is repeated from scratch. This echoes the foundational CBR observation that reusing solutions from similar past problems is more efficient than solving each independently [1].

The core issue is not a lack of operational data. Lakehouse platforms generate extensive telemetry: execution logs, runtime metrics, resource traces, orchestration events, and system tables [2]. Workload characterization studies show task failures are frequent and clustered [3], [4]. The pattern

is structurally recurring rather than exceptional. The issue is that this data is organized for current-state inspection rather than historical reasoning: observability answers 'what is happening now?' but not 'what happened before under similar circumstances, and how was it resolved?'

This study argues that the missing piece is an explicit architectural component: a searchable incident memory layer. The memory layer is a persistent store of structured incident records that sits between observability infrastructure and operational workflows. It ingests failure episodes from telemetry and represents them in a canonical format. It indexes them for retrieval. It exposes search interfaces that connect new failures to historically similar incidents and their resolutions. Just as database systems include a buffer layer between storage and query processing, the incident memory layer mediates between raw operational data and the higher-level reasoning required for troubleshooting.

The contributions of this study are:

1. An architectural specification of the incident memory layer as a distinct component in the lakehouse operational stack.
2. A component-level design covering ingestion, representation, indexing, search, and retrieval interfaces.
3. A comparative analysis positioning the memory layer against three categories of existing operational infrastructure: observability tools, ticketing systems, and log analysis platforms.
4. A discussion of the memory layer as a prerequisite for agentic operational systems.

Section 2 reviews related work. Section 3 presents the architecture with a worked example. Section 4 provides the comparative analysis. Section 5 discusses design considerations, agentic integration via RAG, and data governance. Section 6 identifies limitations. Section 7 concludes.

2. Related Work

2.1 Lakehouse Architecture and Operations

The lakehouse paradigm unifies data warehousing and advanced analytics on open storage formats [2]. Transactional storage layers such as Delta Lake provide ACID guarantees over cloud object stores [5]. This convergence simplifies data management but concentrates diverse workloads on shared infrastructure, increasing the variety and frequency of operational failures. Pipelines for batch ETL and streaming ingestion coexist on the same compute clusters and storage layers. Feature engineering and model training run alongside. Each has distinct failure modes and resource profiles.

Operational practices in lakehouse environments rely on platform-native system tables. Third-party monitoring integrations supplement these. Manual inspection workflows handle the rest. These tools provide visibility into current execution state but do not maintain structured records of past failure episodes in a form suitable for comparison or retrieval.

2.2 Observability and Monitoring

Observability systems collect logs, metrics, and traces to support real-time operational awareness. Tools such as Prometheus, Grafana, and Datadog aggregate time-series data. They trigger alerts on threshold violations. Platform-native capabilities in lakehouse environments expose job execution histories and cluster utilization. Query performance statistics are also surfaced.

These systems are optimized for recency. They answer questions about active failures and ongoing performance degradation. Live resource consumption is also surfaced. They are not designed to answer questions about past incidents in aggregate, compare the current failure against prior episodes, or surface remediation knowledge associated with historically similar conditions.

2.3 Incident Management Platforms

Ticketing and incident management systems such as PagerDuty, ServiceNow, and Jira track incident lifecycle from detection through resolution. They capture human-authored narratives along with severity classifications. Assignees and timelines are also recorded. These records are valuable for

audit. The bigger limitation is that they are disconnected from the telemetry that characterizes each failure.

Searching a ticketing system for incidents similar to a current failure requires the operator to formulate a keyword query over free text, a process that depends on how the original ticket was written. There is no systematic mechanism to match a current failure's telemetry profile against previously resolved incidents.

2.4 Log Analysis and Anomaly Detection

Log-based approaches to operational intelligence include template extraction methods such as Drain [6], sequence-based anomaly detection such as DeepLog [7], and noise-resistant parsing such as LogRobust [8]. These methods process log streams to identify anomalous patterns. They extract templates and classify failure types.

Log analysis contributes to detection. However, it does not address the downstream questions. Has this pattern been seen before? What was the root cause, and what remediation was applied? Connecting a detected log anomaly to a prior incident's resolution requires a separate mechanism that log analysis tools do not provide.

2.5 AIOps and Intelligent Operations

AIOps spans several detection and triage methods. The AIOps literature addresses automated detection, classification, and root cause analysis of operational failures. Dang et al. [9] identify real-world challenges spanning anomaly detection. Failure prediction and root cause analysis are also covered. Systems such as Xpert [10] apply automated reasoning to recommend diagnostic queries and guide operators through triage.

Workload characterization studies provide empirical grounding. Reiss et al. [3] analyzed cluster traces and found task failures are frequent and clustered. Many failures are also repeated. Di et al. [4] characterized cloud workloads and demonstrated that failure patterns exhibit temporal and structural regularity. These findings confirm that operational failures are structurally recurring, making a memory-based approach both feasible and well-motivated.

Despite this breadth, existing AIOps approaches predominantly operate at the point of failure occurrence. They detect, classify, or triage the current incident but do not systematically connect it to a corpus of resolved historical incidents. That is the gap the memory layer addresses.

2.6 Case-Based Reasoning and Knowledge Reuse

CBR offers the conceptual scaffold here. The memory layer's design draws conceptually from case-based reasoning (CBR), a paradigm in which new problems are solved by retrieving and adapting solutions from similar past cases [1]. The CBR cycle of retrieve, reuse, revise, and retain maps directly onto the operational workflow. A new failure triggers retrieval of similar historical incidents. The operator reuses or adapts the prior resolution, revises it as needed, and the outcome is retained. The memory layer operationalizes this cycle in the specific context of lakehouse incident management.

2.7 Positioning

The memory layer is not a replacement for observability, ticketing, or log analysis. It is a distinct architectural component that consumes outputs from these systems and provides a capability none offers individually: structured, searchable, telemetry-grounded incident knowledge.

3. Architecture of the Incident Memory Layer

The design has five components in total. The incident memory layer is an architectural component that sits between the observability infrastructure of a lakehouse platform and the operational interfaces used by engineers and automated systems. Its purpose is to capture, structure, store, and serve incident knowledge. The architecture comprises five components: ingestion, canonical

Journal of Information Systems Engineering and Management
2024, 9(4)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

representation, indexing, similarity search, and retrieval interfaces. Figure 1 illustrates the overall design.

Figure 1 : The incident memory layer as the missing layer in the lakehouse operational stack.

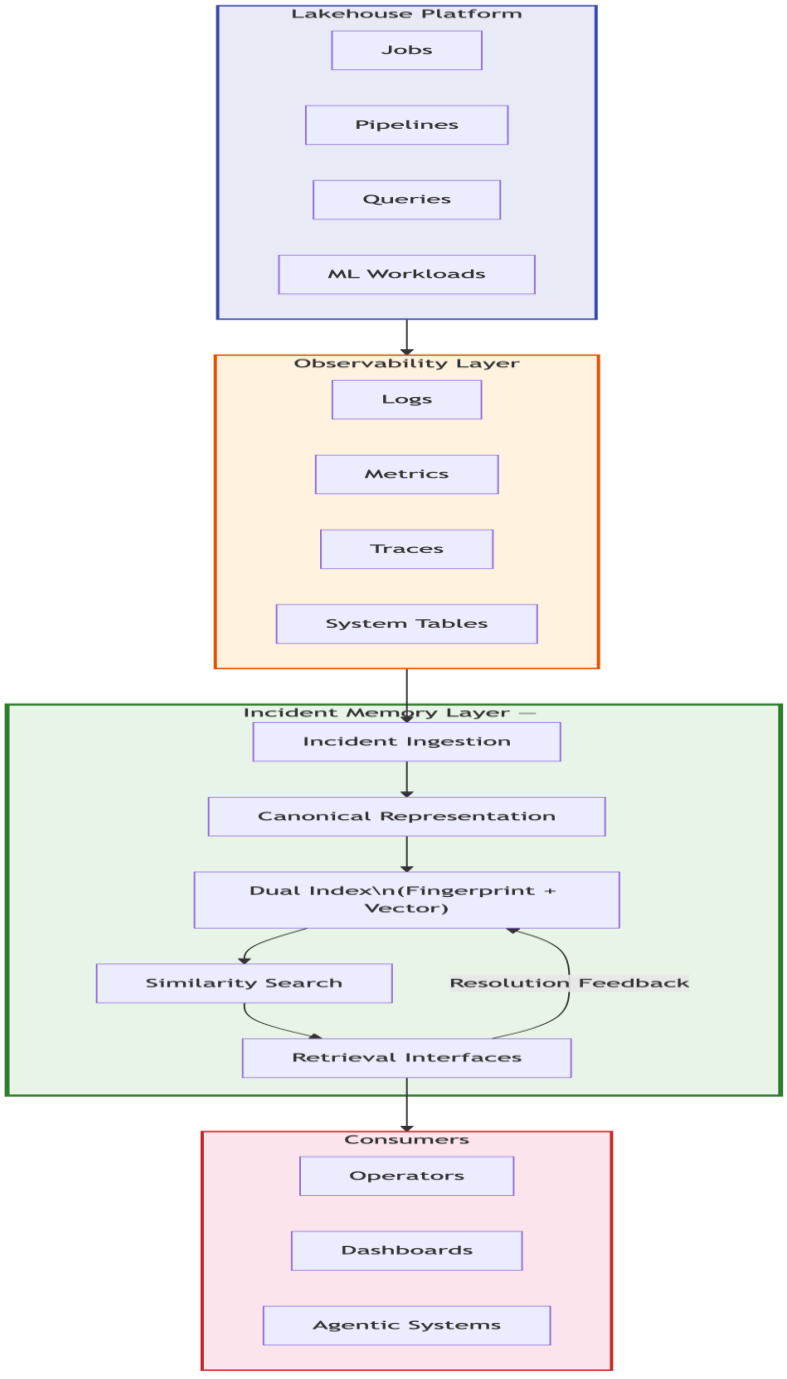


Figure 1. Incident memory layer in the lakehouse operational stack. *The observability layer captures logs, metrics, traces, and system-table signals from lakehouse workloads. The memory layer converts failure signals into canonical incident records, indexes them with fingerprint and vector indices, and exposes similarity-based retrieval to operators, dashboards, and agentic*

Journal of Information Systems Engineering and Management
2024, 9(4)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

systems. A resolution feedback loop improves the memory layer over time, closing the gap between raw observability data and intelligent operational decision-making.

3.1 Ingestion

Ingestion happens at the start. The ingestion component collects failure-relevant data from platform sources at the time of an incident. It operates on event triggers: a job failure, a task retry breach, an SLA violation, or an alert firing. Upon triggering, it assembles a raw incident bundle from:

- Execution logs from the failed job or task.
- Runtime metrics captured during the failure window (CPU, memory, I/O, shuffle statistics).
- Orchestration metadata (schedule, dependencies, retry history).
- Error codes, exception types, and stack traces.
- Cluster and environment configuration at the time of failure.

The ingestion component does not interpret the failure. Its role is to collect raw observational data for the downstream representation component. Automated ingestion is essential. Manual construction of incident records after the fact is slow and inconsistent. It is also prone to information loss.

Table 1. *Components of the incident memory layer*

Component	Input	Output	Function
Ingestion	Platform telemetry, logs, metrics, alerts, tickets	Raw incident bundle	Collects failure-relevant data upon event trigger
Canonical Representation	Raw incident bundle	Structured incident record	Normalizes and structures data into a comparable format
Indexing	Structured incident record	Fingerprint hash + vector embedding	Creates retrieval-ready indices for exact and approximate matching
Similarity Search	Query incident record, index structures	Ranked list of historical matches	Compares new failures against stored incidents
Retrieval Interfaces	Search results, incident records	Formatted incident context	Serves results to operators, dashboards, APIs, and agents

3.2 Canonical Representation

Raw incident data varies in format and granularity. Completeness also differs across workloads, clusters, and platform versions. The representation component transforms each raw bundle into a canonical incident record with four standardized facets:

1. Telemetry features. Normalized execution metrics covering resource utilization and task timing. Data volumes and performance indicators are also included. Normalization is workload-relative: each metric is expressed as a deviation from the historical baseline of the same pipeline or job type, accounting for substantial differences in normal operating ranges across workload categories.
2. Execution context. Metadata describing the workload and its environment includes job type, pipeline identifier, and cluster configuration. Data sources, scheduling parameters, and dependency chain position complete the picture.

3. Failure signature. The signature comprises error code and exception class. Failure stage (read, compute, write, commit), retry count, and log template identifiers extracted via methods such as Drain [6] complete the record.
4. Remediation record. The record captures how the incident was resolved. Entries are added automatically when the platform records the fix, or through operator annotation. Standard fields include root cause label, action taken, time to resolution, and free-text notes. This facet may be incomplete at ingestion and is populated through the feedback loop.

The canonical representation serves as the unit of storage and comparison. Its structure is fixed across incidents, enabling systematic comparison even when the underlying failures involve different workloads or platform components.

3.3 Indexing

The indexing component creates retrieval structures over the corpus of canonical incident records. Two complementary index types are maintained:

Deterministic fingerprints. A fingerprint is a hash computed from selected attributes. Typical inputs are error code, failure stage, job identifier, and cluster type. Fingerprint matches identify strict recurrences where the observable failure signature is identical. Lookup is inexpensive and returns high-precision results for exact recurrence.

Vector embeddings. The full canonical representation, or a subset, is encoded as a dense vector and indexed in a store supporting approximate nearest-neighbor search. This enables retrieval of incidents that share operational characteristics without matching on surface-level attributes. Two failures may have different error codes but arise from the same underlying resource contention pattern. Embeddings can range from hand-engineered feature vectors to learned representations; contrastive learning methods [11], [12] produce embeddings that cluster incidents sharing root causes even when surface signatures differ.

The dual-index design reflects operational reality. Some recurrences are exact replays. Others share root causes but differ in error presentation.

3.4 Similarity Search

Search proceeds in two stages overall. When a new failure occurs, the search component constructs its canonical representation. It then queries the index in two stages:

1. Fingerprint lookup. The system computes the fingerprint of the new incident and checks for exact matches. If found, the corresponding historical records and their remediation information are returned immediately.
2. Vector search. If no fingerprint match is found, the system performs an approximate nearest-neighbor search over the vector index. The top-k most similar historical incidents are returned with similarity scores.

3. The search component may filter by time window, workload category, or platform region. It may also combine fingerprint and vector results when both are available, presenting exact matches first followed by approximate matches ranked by similarity.

3.5 Retrieval Interfaces

The retrieval interface component exposes the memory layer's contents to operational consumers. Three interface categories are supported:

Operator-facing interfaces. Dashboard integrations and CLI tools present matched historical incidents alongside the current failure context. These display the historical incident's telemetry summary and failure signature. The remediation record and similarity score are presented alongside. Contextual differences are highlighted to help the operator assess whether the prior resolution applies.

Programmatic APIs. REST or gRPC endpoints that allow external systems (orchestrators, alerting, custom tooling) to query the memory layer, enabling integration with existing workflows without requiring operators to switch contexts.
Agent interfaces. Structured query and response formats for automated reasoning systems. These interfaces are forward-looking: they anticipate AI agents that operate over incident histories to perform diagnosis, recommend actions, or orchestrate remediation.

3.6 Feedback Loop

A feedback mechanism connects retrieval interfaces back to the index. When an operator resolves an incident, resolution details enrich the corresponding record's remediation facet. Relevance signals capture whether a surfaced match was relevant. They can refine the vector embeddings. They can also adjust similarity ranking over time. The feedback loop transforms the memory layer from a static archive into a learning system.

3.7 Worked Example: Spark OOM Failure Through the Memory Layer

To make the architecture concrete, this subsection traces a single incident through each component. The scenario is a Spark out-of-memory (OOM) failure in a nightly ETL pipeline. This is a common failure mode in lakehouse environments. The cause is the interaction between data skew, executor memory limits, and shuffle operations.
Triggering event. At 02:17 UTC, a nightly pipeline transforming clickstream events into session-level aggregates fails with `java.lang.OutOfMemoryError: Java heap space` during a shuffle-write stage. The orchestrator marks the job as failed after its third retry and fires an alert.
Ingestion. The component activates on the alert and assembles: (a) Spark driver and executor logs, including the OOM stack trace and preceding GC pressure warnings; (b) runtime metrics, including executor memory at 98.7% of 8 GB, shuffle write 47 GB (vs. 30-day average 29 GB), GC time 38% of task duration; (c) orchestration metadata — pipeline identifier, schedule, upstream dependency completion, three retries; (d) cluster configuration — 12 workers with 8 GB executor memory; (e) error code and exception class.
Canonical representation. Telemetry normalization records shuffle volume as 1.62x the 30-day baseline, memory above the 95th percentile, and GC fraction. Execution context captures pipeline, cluster, and dependency chain. Failure signature records error code `OOM`, `java.lang.OutOfMemoryError`, stage `ShuffleWrite`, retry 3, and the Drain-extracted log template [6]. Remediation is empty pending resolution.
Indexing. A deterministic fingerprint is computed from (error code: `OOM`, failure stage: `ShuffleWrite`, pipeline template: `clickstream_sessionize_*`, cluster type: `standard-8g`), and a vector embedding is generated from the full record.
Later retrieval. Three weeks later, a different pipeline fails with `java.lang.OutOfMemoryError` during a `ShuffleRead` stage on a different cluster. The fingerprint does not match because the failure stage and pipeline template differ, but vector search returns the earlier incident as the second-ranked result (similarity 0.87). The telemetry profile (shuffle deviation, memory saturation, elevated GC pressure) is structurally similar.
The operator sees the historical match, reads the remediation record (increased executor memory to 16 GB and enabled adaptive query execution), and applies an analogous fix. Resolution time drops from an estimated 90 minutes of investigation to ~15 minutes of guided remediation. The new resolution feeds back through the loop, strengthening the vector index for future shuffle-related OOM retrievals.
This walkthrough illustrates the core value proposition: the memory layer converts isolated failure episodes into organizational knowledge that accumulates and compounds over time.

Journal of Information Systems Engineering and Management
2024, 9(4)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Figure 2. Incident flow through the memory layer, illustrated with a Spark OOM failure.

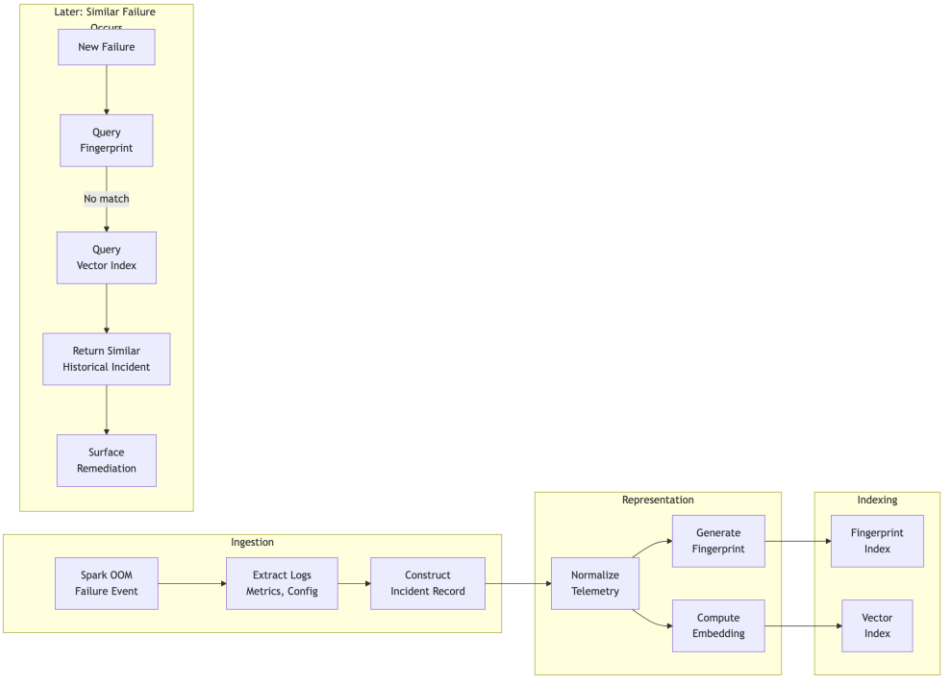


Figure 2. Incident flow through the memory layer, illustrated with a Spark OOM failure. The failure is ingested, normalized, fingerprinted, and embedded into the dual index. When a similar failure occurs later, the memory layer retrieves the historical incident and surfaces its remediation context.

4. Comparative Analysis

The incident memory layer occupies a position in the operational stack that is distinct from existing tool categories. Table 2 compares the memory layer with three categories of existing operational infrastructure across six capability dimensions.

Table 2. Comparison of the incident memory layer with existing operational tool categories

Capability	Observability Tools	Ticketing Systems	Log Analysis Tools	Incident Memory Layer
Current-state visibility	Yes	No	Partial	No (not its role)
Structured incident storage	No	Partial (unstructured text)	No	Yes (canonical records)
Telemetry-grounded records	N/A (raw data, not incident records)	No	Partial (log patterns only)	Yes (multi-signal)
Historical comparison	No	Keyword search only	Template matching only	Yes (fingerprint + vector)
Remediation knowledge linkage	No	Yes (in free text)	No	Yes (structured facet)
Agent-ready query interfaces	No	Limited	No	Yes (by design)

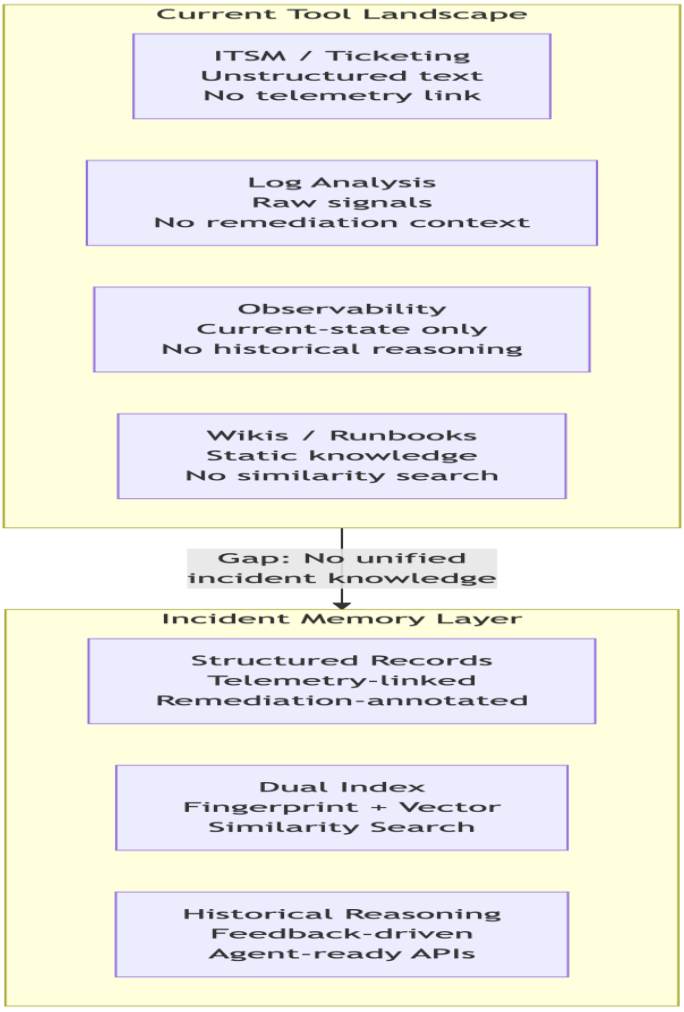
Journal of Information Systems Engineering and Management
2024, 9(4)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Several distinctions matter for this comparison. Several distinctions are worth highlighting. Observability tools excel at real-time visibility but do not retain incident-level knowledge. Their data model is time-series oriented, not incident oriented. Querying an observability platform for 'failures similar to this one' requires the operator to manually translate the current failure into metric queries. The operator then inspects the results. They must infer whether a similar pattern occurred. The process is ad hoc and does not scale. Ticketing systems retain resolution narratives in unstructured form. A ticket may describe a root cause and fix in natural language, but this information is not linked to the telemetry that characterized the failure. Keyword matching over free text is sensitive to terminology variation and inconsistent documentation practices.

Log analysis tools identify patterns within log streams but operate at the signal level rather than the incident level. They can detect that a log template is anomalous but do not connect that detection to a complete incident record with context, root cause, and remediation. The memory layer is not a competitor to these systems. It consumes their outputs and provides a capability they collectively lack: structured, searchable, telemetry-grounded incident knowledge. This positioning as a missing layer, not a replacement, is the central architectural claim.

Figure 3. The incident memory layer addresses gaps left by existing operational tools.



Journal of Information Systems Engineering and Management
2024, 9(4)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Figure 3. *The incident memory layer addresses gaps left by existing operational tools. Unlike ticketing systems, log analyzers, observability platforms, and static knowledge bases, the memory layer provides structured, telemetry-linked incident records with similarity search and remediation context in a single queryable layer.*

4.1 Detailed Platform Comparison

To sharpen the comparative analysis, Table 3 evaluates the incident memory layer against four specific categories of existing platforms across five operational dimensions relevant to incident knowledge management.

Table 3. Detailed comparison of the incident memory layer with four categories of existing operational platforms

Dimension	ITSM Platforms (ServiceNow, Jira)	Log Analysis (ELK Stack, Splunk)	Observability Platforms (Datadog, Grafana)	Manual Knowledge Bases (Confluence, Wikis)	Incident Memory Layer
Structured incident capture	Partial. Metadata fields exist but diagnostic substance is in free-text descriptions with no enforced schema for failure characteristics.	No. Log entries are raw or semi-parsed strings; aggregation produces counts, not incident-level records.	No. Metrics and dashboards describe system state, not incident episodes. Alerts fire on thresholds but do not produce structured incident records.	No. Articles are authored retrospectively for high-severity incidents only; coverage is selective and inconsistent.	Yes. Automatic capture via the ingestion pipeline into the four-facet canonical representation.
Telemetry integration	Minimal. Operators paste log snippets into tickets; no programmatic linkage to underlying telemetry.	Logs only. No integration of metrics, traces, or orchestration metadata into a unified incident view.	Strong for real-time signals but ephemeral — correlation during active investigation is not persisted as an incident record.	None. Articles reference telemetry informally through screenshots; connection to source data is not maintained.	Native. Ingestion programmatically collects logs, metrics, traces, and orchestration metadata and binds them into each incident record.
Similarity search	Keyword search over ticket text; depends on terminology consistency across authors.	Pattern matching over log templates only; does not incorporate resource metrics or	Limited. Dashboard queries can be rerun across time ranges, but no mechanism to search for historically	Manual browsing or full-text search only; discovery depends on the reader knowing what to search for.	Dual-mode. Fingerprints for exact-match recurrences; vector similarity for structurally similar incidents with different surface

Journal of Information Systems Engineering and Management
2024, 9(4)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

		resolution outcomes.	similar failure episodes.		presentations.
Remediation reuse	Present but unstructured. Resolution notes in closed tickets require reading narrative text to extract applicable knowledge.	Absent. Identifies patterns but does not record or link remediation actions.	Absent. Supports investigation but does not capture resolution knowledge.	Present but brittle. Runbooks are manually maintained, frequently outdated, and disconnected from the incidents they describe.	Structured. Remediation facet stores root cause labels, actions, resolution time, and notes in queryable format. Feedback loop keeps records current.
Scalability across incidents	Scales in storage but not retrieval quality; keyword search produces increasingly noisy results as volume grows.	Scales for log volume but not incident-level reasoning; connecting log patterns to historical contexts is not supported.	Scales for metric cardinality but retention windows are limited; historical incident comparison is not a design goal.	Does not scale. Manual authorship cannot keep pace with incident volume; knowledge bases become stale.	Designed to scale. Vector indices support sub-linear retrieval; fingerprint indices provide constant-time exact-match lookup. Time-decay weighting manages staleness.

Table 3 *shows that existing platform categories each cover only part of the required capability set: structured capture, telemetry integration, similarity search, remediation reuse, and scalable retrieval. The proposed memory layer unifies these dimensions by consuming outputs from existing systems and organizing them into a reusable incident knowledge store.*

5. Discussion

5.1 The Missing Layer Argument

A gap remains in the operational stack. The lakehouse operational stack has well-defined layers for compute orchestration, storage management, query processing, and observability. It does not have a dedicated layer for operational memory. The absence means each incident is treated as an isolated event, even when the same failure pattern has been resolved repeatedly.

This architectural gap has consequences that scale with organizational size. A small team with a narrow set of pipelines may rely on individual memory. A large organization with hundreds of pipelines, multiple teams, and rotating on-call engineers cannot. Without a memory layer, operational knowledge scales linearly with individual experience but does not accumulate at the organizational level.

The memory layer addresses this gap by providing a shared, persistent, queryable store of incident knowledge. Its value increases with incident volume, remediation quality, and workload breadth. The pattern is analogous to other architectural components that become essential at scale: a data catalog is unnecessary for ten tables but indispensable for ten thousand. Just as transactional storage was added to cloud object stores to provide ACID guarantees that storage alone could not offer [5], the incident memory layer adds structured incident reasoning that observability alone cannot provide.

Journal of Information Systems Engineering and Management
2024, 9(4)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

5.2 Design Trade-offs

Representation fidelity vs. scalability. Richer canonical representations improve matching quality. They also increase storage and indexing costs. The four-facet structure balances expressiveness with tractability. Organizations may extend with domain-specific facets (data quality metrics, lineage) as deployment matures.

Automated vs. human-annotated remediation. Automated capture is faster and more consistent but may miss contextual nuances only the resolving engineer understands. Human annotation is richer but introduces latency. The design supports both, with automated capture as default and human annotation as enrichment.

Cold start. A new memory layer deployment contains no historical records. Utility grows as incidents accumulate. A practical mitigation is to backfill from existing ticketing systems and log archives. The trade-off is accepting less-structured initial records.

Index maintenance. As the corpus grows and the platform evolves, some historical records become stale. Periodic curation or time-decay weighting manages this without requiring manual deletion.

5.3 Toward Agentic Operations

The memory layer is a prerequisite for AI agents that reason over operational history to diagnose failures, recommend actions, and potentially orchestrate remediation. These capabilities do not yet exist at production maturity.

Current agent architectures in AIOps typically operate over real-time signals or static runbooks. They lack access to a structured corpus of resolved incidents that would allow them to ground their reasoning in organizational experience. The memory layer provides this corpus. An agent querying the memory layer can answer: 'Have we seen this failure pattern before? What was the root cause? What fix was applied? Did it hold?'

This capability transforms the agent from a reactive responder into a historically informed reasoner. Retrieval-augmented generation (RAG) offers a natural integration pattern. The agent retrieves relevant historical incidents. It then conditions its diagnostic reasoning on retrieved context [13]. This grounds responses in the organization's actual operational history rather than encoding all incident knowledge into model parameters.

Consider an AI diagnostic agent on a tool-use architecture with access to a metrics API, log search, cluster configuration store, and the incident memory layer. When a new alert arrives, the agent queries the memory layer with the failure's canonical representation. The layer returns the top-k historically similar incidents with their telemetry summary, failure signature, and remediation record. The agent incorporates these into its reasoning context [13] and generates a diagnostic hypothesis grounded in what the organization has observed and resolved.

This retrieval-grounded reasoning has several properties distinguishing it from prompt-only designs. Responses are anchored in verifiable organizational history. Each retrieved incident is a concrete record the operator can inspect, reducing hallucination risk. The agent can reason over remediation effectiveness: if a prior resolution was applied but the failure recurred, it can flag this and suggest alternatives. Iterative tool use lets the agent refine retrieval queries as it gathers context.

Multi-step diagnostic workflows extend this pattern. An agent might proceed step by step. (1) retrieve similar historical incidents. (2) query metrics to compare the current failure's resource profile. (3) check whether cluster configuration has changed since the relevant historical incident. (4) synthesize these into a ranked list of candidate root causes with remediation actions traceable to historical precedents. The memory layer functions as the agent's long-term memory.

Future work should explore specific agent architectures that consume the memory layer's interfaces. The impact of retrieval quality on diagnostic accuracy also merits study. Multi-agent configurations, in which specialized agents share the memory layer as a common knowledge substrate, are another open direction.

Journal of Information Systems Engineering and Management
2024, 9(4)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

5.4 Scope and Boundaries

The memory layer as described in this study is an architectural specification. It does not prescribe specific technologies for storage (relational, document, or graph databases), vector indexing (HNSW, IVF, or product quantization), or embedding computation (hand-crafted features or learned representations). These choices depend on deployment context, scale requirements, and organizational constraints. The contribution is the identification and specification of the component itself, not a particular implementation.

5.5 Data Governance Considerations

The incident memory layer stores detailed operational records that may contain sensitive information. Data governance must be addressed as part of the architectural design rather than deferred to implementation. Personally identifiable information in incident records. Logs may include usernames or email addresses. Query text may reference customer data. Stack traces can also expose service account names or API keys. The ingestion component must apply configurable redaction rules at ingestion time, not at query time, to prevent sensitive data from being persisted. Pattern-based redaction provides a baseline; regulated-data organizations may integrate with existing data loss prevention infrastructure. The canonical representation's structured format aids redaction because sensitive content concentrates in specific facets rather than appearing uniformly.

Retention policies. Regulatory requirements such as GDPR constrain how long operational data may be stored if linkable to identifiable individuals. Beyond regulation, stale records degrade retrieval quality. The memory layer should support three retention mechanisms: time-based expiration, relevance-based decay (deprioritizing records not retrieved within a window), and explicit exclusion by compliance teams. Expired records must be removed from both fingerprint and vector indices. Access control. Incident records may describe failures in systems subject to access restrictions. A failure in a pipeline processing financial data should be visible only to authorized teams. Retrieval interfaces must enforce role-based access control scoped by business unit, data classification, or pipeline ownership, integrating with the organization's IAM infrastructure.

A memory layer that stores sensitive data without adequate redaction, retains records beyond regulatory limits, or exposes restricted details to unauthorized users creates liability that may outweigh operational benefits. Governance must be integral to the architecture.

6. Limitations

The evaluation is based on systematic comparative analysis. Worked examples and architectural assessment supplement the analysis. Large-scale controlled experiments remain future work. Future work should extend evaluation to production-scale deployments. Several limitations follow from this scope, each with a mitigation strategy.

1. No implementation. The memory layer has not been built or deployed. Mitigation: a phased implementation can reduce risk. An initial deployment could implement only ingestion and fingerprint indexing, providing exact-match retrieval over a limited corpus. Vector search and full canonical representation can be added in subsequent phases.
2. No retrieval evaluation. Search precision, recall, and latency are uncharacterized. Mitigation: retrospective evaluation against historical incident data provides a path without requiring a live deployment. Organizations can construct a test corpus from resolved tickets with telemetry. A subset is withheld as queries. Retrieval is then measured using precision@k, recall@k, and mean reciprocal rank. Latency benchmarking against FAISS or ScaNN would characterize scalability.
3. Representation generality. The four-facet canonical representation has not been validated across different platforms, workloads, and organizational contexts. Mitigation: treat the representation

as extensible rather than fixed. The four core facets provide a minimum viable structure, but the schema can accommodate platform-specific extensions. Examples include a data quality facet for schema-drift-prone pipelines, or a cost facet where resource overprovisioning is the primary concern.

4. Remediation quality dependency. The value depends on the quality of remediation records. Mitigation: automated capture from platform signals (detecting that an operator resized a cluster or modified a configuration after a failure) reduces reliance on manual annotation. Structured annotation templates with predefined root cause categories and action types lower manual input effort. Surfacing annotation prompts within the ticketing system at closure can increase compliance.
5. Integration complexity. Connecting the memory layer to diverse observability, ticketing, and orchestration systems requires integration engineering. Mitigation: an adapter-based architecture isolates complexity and allows incremental onboarding. Standard protocols (OpenTelemetry for traces and metrics, syslog or structured JSON for logs) and ticketing APIs provide well-documented integration surfaces.
6. Evaluation methodology. Validating operational impact requires controlled studies in production. Mitigation: A/B testing at the team level, with some on-call rotations using the memory layer and others not, provides comparative evidence. Time-to-resolution, escalations, and repeat incident rates are measurable proxies. Shadow-mode deployment, where the layer runs alongside existing workflows with recommendations logged but not surfaced, enables offline evaluation without disrupting operations.
7. Embedding drift. As the platform evolves, vector embeddings may become less effective. Mitigation: periodic retraining on the growing corpus, combined with time-decay weighting from Section 5.2, addresses distributional shift. Monitoring retrieval quality over time signals when retraining is needed.

7. Conclusions

The contribution is concrete and architectural. This study proposed a searchable incident memory layer as a dedicated architectural component for modern lakehouse operations. The memory layer ingests failure episodes, transforms them into canonical incident records, indexes them for exact and approximate retrieval, and serves historical matches to operators, programmatic clients, and future agent-based systems.

The central argument is that the lakehouse operational stack lacks a memory component. Observability tools provide real-time visibility. Ticketing stores resolution narratives. Log analysis detects anomalous patterns. None provides structured, searchable, telemetry-grounded incident knowledge. The memory layer fills this gap as a distinct architectural layer. The role is not as a replacement; it is the missing intermediary between raw observability data and intelligent operational decision-making.

The design is specified at the component level: ingestion, canonical representation, indexing, similarity search, and retrieval interfaces. A comparative analysis demonstrates that the memory layer addresses capabilities absent from existing tool categories. A feedback loop allows it to improve as it accumulates incidents and resolution feedback.

The memory layer is also positioned as a prerequisite for agentic operational systems. AI agents that reason over incident histories require a structured knowledge substrate current tooling does not provide. The memory layer offers this substrate, creating a foundation for historically grounded automated diagnosis through RAG and related architectures.

Future work should extend evaluation to larger-scale production environments, evaluate retrieval quality and operational impact, and explore agent architectures that consume the memory layer's interfaces for automated diagnostic reasoning.

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.

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, Formal Analysis, Writing — Original Draft, and Writing — Review and Editing.

Data Availability Statement

No new data were created or analyzed in this study. The memory layer architecture and evaluation are based on publicly available literature and systematic analysis of lakehouse platform operational characteristics.

References

[1] 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.
[2] 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.
[3] 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 Comput. (SoCC)*, 2012, Art. 7.
[4] S. Di, D. Kondo, and W. Cirne, "Characterization and comparison of cloud versus grid workloads," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, 2012, pp. 230-238.
[5] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. van Hovell, A. Ionescu, A. Lusber, M. Switakowski, M. Sasaki, X. Li, S. Luo, V. Nham, and M. Zaharia, "Delta Lake: High-performance ACID table storage over cloud object stores," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3411-3424, 2020.
[6] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, 2017, pp. 33-40.
[7] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly detection and diagnosis from system logs through deep learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security (CCS)*, 2017, pp. 1285-1298.
[8] 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 self-supervised log semantics," in *Proc. 27th

ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2019, pp. 508-518.
[9] 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.
[10] N. Jiang et al., "Xpert: Empowering incident management with query recommendations via intelligent automation," in *Proc. 46th Int. Conf. Softw. Eng. (ICSE)*, 2024.
[11] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," in *Proc. 37th Int. Conf. Mach. Learn. (ICML)*, 2020, pp. 1597-1607.
[12] P. Khosla, P. Teterwak, C. Wang, A. Sarna, Y. Tian, P. Isola, A. Maschinot, C. Liu, and D. Krishnan, "Supervised contrastive learning," in *Proc. Adv. Neural Inf. Process. Syst. 33 (NeurIPS)*, 2020, pp. 18661-18673.
[13] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Kuttler, M. Lewis, W. Yih, T. Rocktaschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Proc. Adv. Neural Inf. Process. Syst. 33 (NeurIPS)*, 2020, pp. 9459-9474.