International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Self-Healing Cloud Data Pipelines Using AI-Based Failure Detection, Root Cause Analysis, and Automated Recovery

Pavan Kumar Kodati

Abstract: Enterprise cloud data pipelines fail far more often than organizations acknowledge, and the downstream cost of each failure extends well beyond the immediate job. Schema drift, delayed file arrivals, resource exhaustion, API instability, and orchestration defects each produce cascading effects across reporting layers, machine learning features, and regulatory deliverables. Current monitoring approaches detect failure states but stop short of diagnosis and remediation, leaving operational teams to manually cross-reference logs, lineage records, and infrastructure metrics before any recovery can begin. This article proposes a self-healing cloud pipeline framework that closes the gap between detection and resolution. The framework integrates observability telemetry aggregation, machine learning-based failure classification, retrieval-grounded large language model (LLM) root cause interpretation, and policy-driven automated recovery execution within a four-layer architecture. The layers, including observation, intelligence, decisioning, and recovery processes, manage incidents from the initial signal to the resolved state without requiring human intervention in approved scenarios. The approach targets measurable reductions in mean time to recovery (MTTR), lower manual intervention rates, and more consistent recovery behavior across large and diverse pipeline estates. The framework is designed for cloud-native orchestration environments and is assessed against rule-based and machine-learning-only baseline approaches across representative failure scenario categories.

Keywords: *automated recovery, cloud data pipelines, failure detection, root cause analysis, self-healing systems*

1. Introduction

Cloud-native data engineering has fundamentally changed how organizations ingest, transform, and serve data for analytics, compliance, and machine learning. Pipeline estates now span event streams, batch ingestion jobs, transformation engines, cloud object stores, data warehouses, application programming interface (API) integrations, and orchestration systems coordinating thousands of interdependent tasks. The breadth is not uniform; pipelines range from low-latency streaming workflows to multi-hour batch transformations, each carrying distinct service-level obligations and downstream dependencies [1].

Failure in these environments is not rare, and it is rarely simple. Schema changes propagate silently from upstream systems. Files arrive late or not at all. APIs return unexpected payloads. Distributed compute clusters exhaust memory under volume spikes. What makes these failures expensive is not their individual severity; it is the compounding effect. A single missed upstream partition can cascade through transformation layers, invalidate downstream aggregations, delay regulatory deliverables, and consume hours of engineering effort in investigation alone [2].

The operational model in most organizations remains reactive. Threshold-based alerts fire when a job fails; engineers inspect logs, cross-reference metadata, review recent configuration changes, and apply recovery manually based on accumulated tacit knowledge. Consistency and speed depend on who is on shift, their familiarity with the affected pipeline, and whether prior incidents have been documented. The gap between detection and recovery is, at its core, a knowledge problem, and knowledge problems do not scale well [3].

A self-healing pipeline architecture addresses this gap by embedding intelligence between failure detection and recovery execution. The system classifies the failure type, interprets likely root causes from aggregated telemetry, selects an appropriate recovery response, and, where policy permits, executes that response automatically. This article proposes such a framework, designed

Independent Researcher, USA
ORCID ID: 0009-0000-5372-1504

specifically for enterprise cloud data platforms. The contribution has four parts: a structured taxonomy of cloud pipeline failure categories and their detection signals; a four-layer self-healing architecture with cleanly separated concerns; a retrieval-grounded approach to LLM root cause interpretation that constrains hallucination risk; and a policy-aware recovery execution model that preserves governance control. The article proceeds as follows: Section 2 examines the problem scope and background; Section 3 presents the proposed framework; Section 4 describes the implementation design; Section 5 defines the evaluation approach; Section 6 discusses implications and limitations; Chapter 7 concludes.

2. Background and Problem Scope

Pipeline failures in cloud-native environments are distributed across five broad categories. Data-based failures consist of incomplete or late file delivery, format change, null issue, and duplicate entry processing. Logical failures include data transformation problems, SQL errors, and violation of assumptions because of new patterns in data sources. Infrastructural failures occur due to compute memory pressure, network failures, cluster failure, and quota failure. Dependency failure includes the failure of an API call and the failure of a third-party data feed, as well as the expiration of permissions or authentication tokens. Another category of failures is that of policies and permission denial, which is largely left out of traditional failure models [4].

Current monitoring tools aggregate logs, job states, and resource metrics but lack the capacity for contextual synthesis. A failed Spark job may surface an executor error in one log stream, while a storage access log records a missing partition, and a metadata catalog reflects a schema change introduced earlier that day. Human operators must manually assemble this evidence, as there is no structured model to indicate which signals are most diagnostic for each failure class. The problem compounds directly with scale: what is manageable at fifty pipelines becomes unworkable at five thousand [5].

Research in AI for IT Operations (AIOps) has demonstrated that telemetry aggregation and machine learning can improve anomaly detection and incident triage in infrastructure operations. Techniques including log parsing, metric correlation, time-series anomaly detection, and graph-based incident matching have produced measurable improvements in detection speed and triage accuracy in general cloud infrastructure contexts [6]. Data engineering pipelines, however, introduce domain-specific elements that general AIOps frameworks do not address natively: schema lineage, data quality contracts, transformation logic dependencies, and downstream business impact scoring all require purpose-built models and evidence structures [7].

Self-healing principles have been studied extensively in service mesh, microservice, and cloud infrastructure contexts. Common mechanisms include circuit breakers, automatic scaling policies, and restart configurations. These address availability at the infrastructure layer but do not account for the semantic content of data failures. A pipeline that restarts successfully may still produce incorrect outputs if the underlying cause was a schema change rather than a transient resource spike. The distinction matters here: the issue is not service uptime but data correctness and lineage integrity. The proposed framework adapts self-healing principles to the data engineering domain with this distinction at its center [8].

3. Proposed Self-Healing Framework

The proposed framework is organized into four sequential layers: observation, intelligence, decisioning, and recovery execution. Each layer has a defined input, a processing function, and a structured output that feeds the next. Separating concerns in this way, telemetry collection does not determine classification logic, and classification does not constrain recovery options, making each layer independently testable and replaceable without redesigning the full system [9].

The observation layer aggregates heterogeneous telemetry from across the pipeline estate. Sources include orchestration logs, distributed processing metrics, data warehouse query execution history, object storage access events, API response records, data quality rule execution results, schema change event logs, and prior incident resolution records. Raw signals are normalized into structured event records carrying job identifiers, task metadata, timestamps, error codes, resource utilization snapshots, schema version references, lineage context, and recent configuration change history. This normalization step is non-negotiable,the intelligence layer cannot operate reliably on heterogeneous raw formats, and completeness here

determines the quality of every downstream component [10].

The intelligence layer performs two distinct functions. First, it classifies the incident. Supervised classification models trained on labeled historical incidents identify known failure patterns with associated confidence scores. For novel or ambiguous failures, unsupervised clustering and anomaly detection surface patterns that deviate measurably from established norms without requiring prior labeling. Second, a retrieval-augmented reasoning component generates a root cause interpretation. An LLM receives a curated evidence bundle assembled by the retrieval layer, relevant log excerpts, schema difference records, matched historical incidents, and lineage graph context and produces a concise, human-readable root cause summary. Grounding the model in retrieved evidence rather than allowing unconstrained generation reduces hallucination risk and makes explanations traceable to actual telemetry [11].

The decisioning layer selects the appropriate recovery response based on classification output, confidence score, and a policy model. The policy model encodes organizational rules: which failure types may be remediated automatically, which require human approval, and which should always escalate regardless of the factors that shape policy application, including data criticality, regulatory sensitivity of the affected datasets, reversibility of the proposed action, and the business risk of delayed recovery. The recovery execution layer implements the approved action, monitors the outcome, and records the full incident cycle for continuous improvement of both the classifier and the playbook library [12].

Table 1. Taxonomy of failure types in cloud pipelines with primary detection signals and matched recovery actions.

Failure Category	Common Causes	Primary Detection Signals	Matched Recovery Actions
Data-Related	Missing files, format changes, late arrival, null anomalies	File absence alerts, parser exceptions, quality rule failures	Delayed retry, fallback parser, quarantine zone routing
Logic-Related	SQL defects, transformation errors, schema assumption violations	Row count anomalies, typecast errors, assertion failures	Schema refresh, transformation rollback, alternate logic branch
Infrastructure	Memory pressure, network interruptions, quota exhaustion	OOM errors, task timeout, resource metric spikes	Memory scaling, repartitioning, cluster resizing, retry
External Dependency	API outages, third-party disruptions, auth expiry	HTTP 5xx errors, connection timeouts, auth rejection logs	Retry queue, fallback source, credential refresh
Policy / Permission	Access revocation, compliance holds, security rule changes	Permission denied errors, audit log events, policy flag	Escalate to human review, access request, pipeline pause

4. Implementation and Solution Design

A practical implementation begins with a normalized incident model. Each incident record contains job identifiers; task-level metadata; timestamps at both start and failure points; raw error messages and codes; resource utilization snapshots; schema version identifiers for all involved datasets; upstream and downstream lineage references; and a log of code or configuration changes deployed within a configurable lookback window. This model serves two purposes simultaneously: it provides the feature set for the classification models, and it provides the structured evidence bundle for the retrieval component that feeds the reasoning layer. Without a consistent incident model, neither component can operate reliably across a heterogeneous pipeline estate [13].

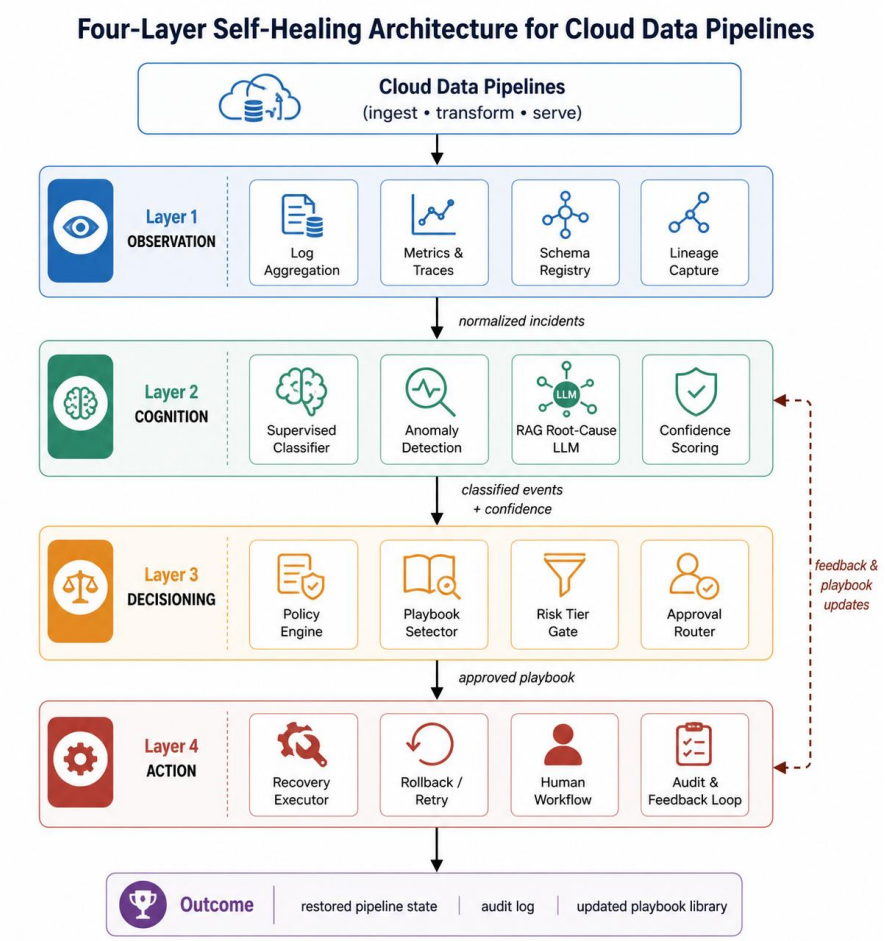

Figure 1. Proposed four-layer self-healing architecture for cloud data pipeline incident management.

Known failure classes can be learned from historical incident records. For schema drift, the classifier may learn to associate a combination of parser exception codes, column count mismatches in schema snapshots, and metadata change events within the lookback window. For timeout-related failures, the model may learn correlations with input volume growth, skewed partition distributions, and downstream warehouse contention metrics. Each classification produces a confidence score and attaches the supporting telemetry features, an interpretable basis for downstream decision-making that reviewers and auditors can examine without needing to interrogate the model directly [14].

Root cause summarization relies on a retrieval layer that assembles curated evidence before invoking the LLM. The retrieval component queries a vector index of historical resolved incidents to surface the most similar prior cases and fetches schema difference records, recent orchestration metadata, and relevant lineage graph paths. The above evidence set is then handed to the LLM in conjunction with a structured prompt that demands a brief diagnosis based on the exact evidence given. It cannot make any assumptions on the cause-and-effect relationship not directly inferred from the evidence at hand, which is the only possible way to verify the validity of the explanation. Example Output: "Pipeline failure was due to the presence of two additional layers of nested objects in the incoming JSON message compared to the schema, which caused a mismatch. There is an alternative schema registered that doesn't sacrifice any data." [15].

The recovery playbooks are formulated as machine-executable workflows tailored to particular failure categories. In the case of a schema mismatch, the next step is to perform a schema registry update, followed by parser recreation and execution of the impacted tasks again. A memory-related failure for Spark jobs will necessitate increasing the memory allocated to the executors or rebalancing the partitioning of the input data before retrying. Repeated API failures route input to a retry queue or quarantine store and notify the responsible data contract owner. Each playbook specifies triggering

conditions, action sequences, rollback procedures, and the governance approval requirement for execution. Integration with orchestration systems such as Airflow, AWS Step Functions, or Databricks Workflows allows playbooks to execute as native workflow steps, preserving existing lineage and audit infrastructure [16].

Automated recovery playbooks also include a safeguards framework around playbook execution. This includes checks on incident classification confidence, dataset criticality, policy eligibility, health of playbook dependencies, and whether or not any prerequisites for executing a playbook are met. The framework also supports a dry-run mode for medium/high impact recovery actions, to simulate the expected remediation and its effects without making actual changes to production assets. This enables operators and governance teams to analyze the expected impact of a given recovery action before executing it.

Each recovery playbook also includes a rollback procedure to roll back all changes made to the configurations, schema and routes, etc., or to the state's mode of a workflow, or any resource allocation, if the recovery procedure did not succeed. Post-recovery validation controls are then used to determine whether the recovery was successful. This includes, but is not limited to, data quality checks, lineage consistency checks, schema conformity checks, as well as the health of downstream dependencies. When all these validation controls are satisfied, the recovery is successful.

The framework keeps track of recovery results and escalation history, so it doesn't attempt the same automated remediation strategy again if it failed a second time for the same type of incident within a given observation window. This escalation mechanism prevents automation loops, bounds operational risk, and provides a safeguard mechanism for new or complex failures to be analyzed by human experts instead of relying on automation.

Table 2. Recovery playbook matrix showing failure class, trigger condition, automated action, and governance policy tier.

Failure Class	Trigger Condition	Automated Action	Safety Controls	Governance Policy
Schema mismatch	Column mismatch + parser error + schema change event in lookback window	Registry refresh, parser regeneration, controlled re-run	Pre-action schema validation, rollback to previous schema version, post-recovery data quality check	Auto-execute if confidence >= 0.85 and the dataset is non-regulated
Memory / OOM failure	OOM error + high spill ratio + input volume above threshold	Executor memory increase or input repartitioning, then retry	Dry-run resource estimation, automatic rollback of resource changes if retry fails	Auto-execute; notify platform team if repeated within 24h
Missing file / late arrival	File absence alert within defined SLA arrival window	Delayed retry loop up to SLA boundary	SLA validation before retry, escalation after repeated failures	Auto-execute; escalate to on-call if SLA boundary breached
Repeated API failure	HTTP 5xx errors above threshold in lookback window	Route to retry queue or quarantine; notify contract owner	Quarantine validation, rollback of routing changes, human approval checkpoint	Human approval required before quarantine promotion
Permission / policy issue	Permission denied error +	Pause pipeline; trigger access	No automation permitted, mandatory	Always requires human approval; no

Failure Class	Trigger Condition	Automated Action	Safety Controls	Governance Policy
	corresponding audit log event	review workflow; escalate	escalation and audit logging	auto-remediation permitted

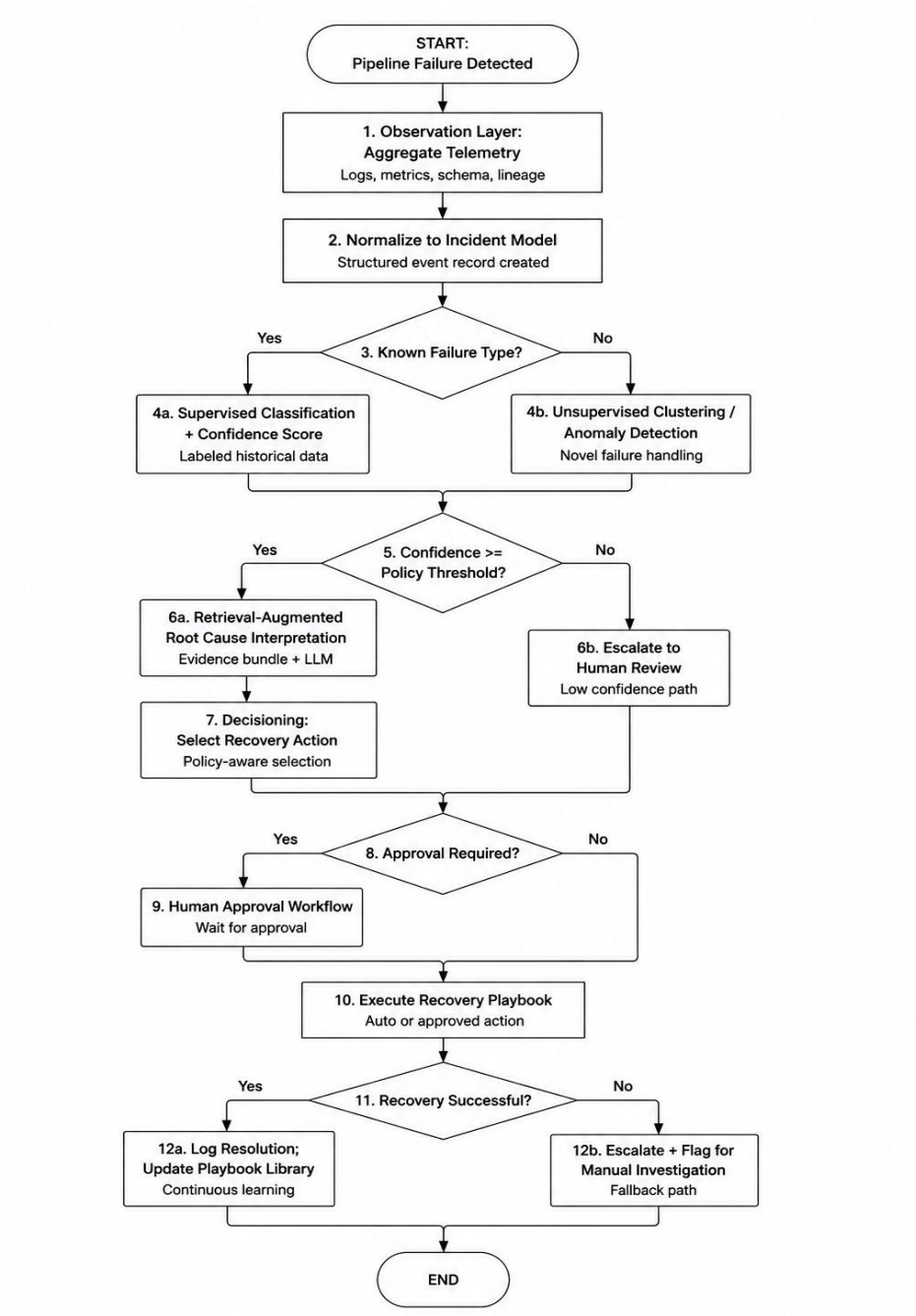

Figure 2. Incident lifecycle flow from initial telemetry signal to resolved state within the self-healing framework.

5. Experimental Evaluation

Evaluation of the proposed framework addresses two dimensions independently before combining results: resilience improvement and operational efficiency. The primary experimental design involves replaying historical pipeline failures in a controlled test environment. A representative failure corpus is assembled from five scenario categories—schema drift, missing files, parser exceptions, resource exhaustion, and API instability—with

multiple instances per category to provide adequate coverage. Three approaches are compared for each incident: deterministic rule-based recovery using fixed threshold triggers, representing the current operational baseline; machine learning classification without the retrieval-grounded reasoning layer; and the proposed hybrid approach combining classification, root cause interpretation, and policy-aware recovery execution [6].

Mean time to recovery, measured from failure detection to confirmed resolution, is the primary metric. Secondary metrics include recovery success rate, false positive recovery rate, classification precision and recall by failure category, reduction in manual intervention time, and incident explanation usefulness as rated by a panel of engineers who review generated root cause summaries against actual incident records. The usefulness rating captures whether the explanation correctly identified the causal factor and whether it provided sufficient context to guide a human reviewer who might override the automated decision [2].

The comparative study is structured to isolate the contribution of each component. Rule-based recovery establishes what the current state delivers. ML-only classification tests whether identification alone, without retrieval-grounded reasoning or structured playbooks, produces adequate recovery. The hybrid approach tests the full system. The reasoning layer is expected to show the largest improvement in novel failure categories, where labeled training data is sparse and retrieval of similar resolved incidents compensates directly for limited classifier coverage. Governance compliance is tested by verifying that no automatic action was executed in any scenario where the policy model required human approval, regardless of classification confidence.

Table 3. Evaluation metrics with baseline descriptions and expected improvement direction for the proposed framework.

Metric	Rule-Based Baseline	ML-Only Approach	Proposed Hybrid Framework
Mean Time to Recovery	High-manual triage and recovery for all cases	Moderate, faster classification, manual recovery still required	Low,classification plus automated playbook execution
Recovery Success Rate	Moderate, dependent on operator knowledge and availability	Moderate, varies by classification accuracy per failure type	High, structured playbooks with confidence gating applied
False Positive Recovery Rate	Low-rule-based triggers are deterministic	Moderately, ML may over-trigger on ambiguous telemetry signals	Low,confidence thresholds and policy gating enforced
Classification Precision / Recall	Not applicable, rule-based, not probabilistic	Varies by failure type and available training data volume	High precision; recall supported by retrieval for novel cases
Manual Intervention Reduction	None; every case requires human action	Partial,classification reduces investigation time only	Substantial,auto-remediation applied for approved policy tiers
Governance Compliance Rate	High, humans always approves before any action taken	Not natively enforced by classification model	High,policy model enforced at the decisioning layer

6. Discussion

The central design tension in a self-healing data platform is not technical; it is about trust. Automatic remediation accelerates recovery and lowers operational burden; the case for it is clear in low-risk, high-frequency failure classes where the action is reversible and the classifier is confident. The

picture changes considerably when the affected dataset carries regulatory sensitivity, when confidence falls below a reliable threshold, or when the proposed action is not easily undone. The policy tier model addresses these issues by making the autonomy boundary a governed design decision rather than an implicit engineering assumption [8].

Table 4. Governance policy tiers defining approval requirements by risk level, action type, and example scenario.

Risk Tier	Action Type	Confidence Requirement	Approval Requirement	Example Scenario
Low	Retry, delay, resource scaling	>= 0.80	Auto-execute; log for review	Memory OOM retry with increased executor allocation
Medium	Schema refresh, fallback routing	>= 0.85	Auto-execute; notify data owner	Schema mismatch on non-regulated silver layer table
High	Quarantine, pipeline suspension	>= 0.90	Human approval required before execution	Repeated API failure on regulated financial data feed
Critical	Any action on regulated or PII datasets	Any confidence level	Always requires human approval	Permission error on GDPR-scoped customer pipeline

Two limitations require direct acknowledgment. First, the classification model faces a cold-start problem for failure types not represented in the training corpus. Novel failures, introduced by a new upstream system or a previously unseen infrastructure configuration, may receive low confidence scores, which correctly triggers escalation rather than automated recovery. This is a safe default, not a design flaw, but it means the system adds less operational value in the early phase of deployment for estates with highly heterogeneous or frequently changing pipeline patterns. Coverage improves as resolved novel incidents are added to the training set over time [9].

Second, the framework depends on telemetry completeness. Pipelines that do not emit structured logs, that lack lineage metadata, or that run in environments with limited observability produce incomplete incident records that reduce classification accuracy and retrieval relevance directly. This is not a problem the framework can solve internally; it requires investment in observability infrastructure before the self-healing layer can operate at full effectiveness. Organizations considering adoption should treat observability maturity as a prerequisite, not a parallel workstream [6].

7. Governance, Auditability, and Enterprise Control

For enterprise self-healing systems, an automated decision must be transparent, explainable and auditable. The proposed framework maintains an incident evidence record for the entire life cycle of an incident. In each case, it includes the history of approvals of the recovery action, including decisions, timestamps, and approvers' identities if human approval is required.

The framework stores the version controlled recovery playbooks and the actual version of the playbook executed in the event of an incident remediation. The audit log stores the model version used for incident classification, the retrieval context used for retrieving relevant passages in the diagnosis phase, and the generated explanation presented to the operators, since classification and root cause analysis depend on ML models and retrieval-grounded LLM components. This aids in reproducibility and future compliance review.

An incident evidence bundle is generated for every recovery event. This bundle includes telemetry, diagnostics artifacts, lineage, root cause summaries, logs of what was executed, and the state of the pipelines before and after recovery. These artifacts can be used by an auditor to verify what operational

impact the automated remediation actions had. Details of recovery outcome, validation, rollback status and any other escalation events also become part of the permanent incident record.

When a human reviewer overrides a machine learning classification recommendation, the system learns from their decisions, explanation comments and override history, to continuously improve the classification models and the recovery policies, while maintaining the accountability, traceability and governance controls mandated by regulated enterprise environments.

8. Conclusion

Self-healing data pipelines represent a concrete advance in the operational maturity of cloud-native data engineering. The framework proposed in this article addresses a gap that traditional monitoring leaves open: the distance between failure detection and recovery execution. By combining structured telemetry aggregation, machine learning-based failure classification, retrieval-grounded root cause interpretation, and policy-aware recovery playbooks, the framework enables a system that learns from its incident history and acts on that knowledge in a governed, auditable way.

Governance is not a constraint added to this design; it is a load-bearing element. Enterprises in regulated environments cannot adopt autonomous remediation systems that act without accountability or explanation. The policy tier approach ensures automation is applied where risk is low and confidence is high, while preserving human judgment for the scenarios where the stakes are greatest. The LLM-generated root cause summaries and the structured playbook audit trail make the behavior transparent to engineers and compliance reviewers alike, which is what makes adoption feasible in practice rather than just theoretically appealing.

Future work extends this framework in two directions. Predictive prevention is the nearer-term extension: risk signals identified before failure occurs, rising memory utilization trends, accumulating schema drift in source systems, and degrading API response quality can trigger preemptive adjustments to scheduling or configuration. Closed-loop optimization is the longer-term direction, where outcome data from executed recovery actions is fed continuously back into the classification model and the playbook library, enabling the system to improve its coverage and accuracy without requiring manual retraining cycles.

References

[1] 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. [Online]. Available: https://dl.acm.org/doi/10.1145/2934664

[2] J. Weng, J. H. Wang, J. Yang, and Y. Yang, "Root cause analysis of anomalies of multitier services in public clouds," in Proc. IEEE/ACM 25th International Symposium on Quality of Service (IWQoS), 2017, pp. 1-6. [Online]. Available: https://ieeexplore.ieee.org/document/7969155

[3] R. Chaiken et al., "SCOPE: easy and efficient parallel processing of massive data sets," Proceedings of the VLDB Endowment, vol. 1, no. 2, pp. 1265-1276, 2008. [Online]. Available: https://doi.org/10.14778/1454159.1454166

[4] F. Psallidas et al., "Provenance for interactive visualizations," in Proc. Workshop on Human-In-the-Loop Data Analytics, 2018, pp. 1-6. [Online]. Available: https://doi.org/10.1145/3209900.3209904

[5] Y. Chen et al., "Outage prediction and diagnosis for cloud service systems," in Proc. The Web Conference 2019, 2019, pp. 2659-2665. [Online]. Available: https://doi.org/10.1145/3308558.3313501

[6] P. Notaro et al., "A survey of AIOps methods for failure management," ACM Transactions on Intelligent Systems and Technology, vol. 13, no. 1, pp. 1-45, 2021. [Online]. Available: https://doi.org/10.1145/3483424

[7] L. Ehrlinger and W. Woss, "Towards a definition of knowledge graphs," in Proc. SEMANTiCS 2016, 2016, pp. 1-4. [Online]. Available: https://ceur-ws.org/Vol-1695/paper4.pdf

[8] M. Grottke and K. S. Trivedi, "Fighting bugs: remove, retry, replicate, and rejuvenate," IEEE Computer, vol. 40, no. 2, pp. 107-109, 2007. [Online]. Available: https://doi.org/10.1109/mc.2007.55

[9] S. He et al., "A survey on automated log analysis for reliability engineering," ACM Computing Surveys, vol. 54, no. 6, pp. 1-37, 2021. [Online]. Available: https://doi.org/10.1145/3460345

[10] J. Soldani and A. Brogi, "Anomaly detection and failure root cause analysis in (micro) service-based cloud applications," ACM Computing Surveys, vol. 55, no. 3, pp. 1-39, 2022. [Online]. Available: https://doi.org/10.1145/3501297

[11] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in Proc. NeurIPS 2020, 2020, pp. 9459-9474. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf

[12] A. Gulenko et al., "Detecting anomalous behavior of black-box services modeled with distance-based online clustering," in Proc. IEEE International Conference on Cloud Computing, 2018, pp. 912-915. [Online]. Available: https://doi.org/10.1109/cloud.2018.00134

[13] M. Bosma et al., "Chain-of-thought prompting elicits reasoning in large language models," in Proc. NeurIPS 2022, 2022, pp. 24824-24837. [Online]. Available: https://doi.org/10.52202/068431-1800

[14] S. He et al., "Towards automated log parsing for large-scale log data analysis," IEEE Transactions on Dependable and Secure Computing, vol. 15, no. 6, pp. 931-944, 2018. [Online]. Available: https://doi.org/10.1109/tdsc.2017.2762673

[15] X. Zhou et al., "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in Proc. 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2019, pp. 683-694. [Online]. Available: https://dl.acm.org/doi/10.1145/3338906.3338961

[16] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot: a technique for cheap recovery," in Proc. 6th Symposium on Operating Systems Design and Implementation (OSDI), 2004, pp. 31-44. [Online]. Available: https://www.usenix.org/legacy/event/osdi04/tech/full_papers/candea/candea.pdf