NextGen Scientific Publication
American International Journal of Computer Science and Technology
Volume 6 Issue 6, Pg. No. 98-108 AIJCST-V6I6P110, 2024
https://doi.org/10.63282/3117-5481/AIJCST-V6I6P110

Original Article

Machine Learning-Enabled Self-Healing Data Pipelines: An Autonomous Architecture for Failure Detection, Diagnostic Reasoning, and Automated Remediation

***Vineeth Kumar Reddy Mittamidi**

Application Support Engineer, TCS, North Carolina, USA.

Abstract:

Enterprise data pipelines have become essential infrastructure for analytics, artificial intelligence, risk management, digital services, and regulatory reporting. However, their reliability is challenged by issues such as schema drift, source delays, data corruption, orchestration failures, resource constraints, model drift, and downstream inconsistencies. This paper proposes an autonomous architecture for machine learning-enabled self-healing data pipelines that integrates multimodal observability, anomaly detection, diagnostic reasoning, and policy-driven automated remediation. The architecture views data pipelines as continuously monitored socio-technical systems represented through metrics, logs, traces, lineage graphs, data contracts, quality indicators, execution histories, and incident records. The proposed framework consists of five layers: (1) a telemetry and lineage substrate, (2) a feature and knowledge representation layer, (3) a detection layer combining statistical, rule-based, and machine learning techniques, (4) a diagnostic reasoning layer using causal and graph-based methods, and (5) a remediation layer that executes controlled repair actions under governance policies. The study further introduces a fault taxonomy, a remediation safety model, and an evaluation framework based on fault injection experiments. Performance is assessed using metrics such as detection precision, diagnosis accuracy, mean time to recovery, false remediation rate, data quality preservation, and auditability. The paper argues that self-healing should follow a constrained autonomy model, where automated actions are permitted only when supported by sufficient evidence, limited operational risk, validated rollback mechanisms, and explicit policy approval. By integrating AIOps, DataOps, MLOps, and software governance practices, the proposed architecture provides a research-driven blueprint for building resilient data platforms capable of early failure detection, systematic root-cause analysis, automated remediation, and evidence-based escalation.

Keywords:

Self-Healing Data Pipelines, AIOps, DataOps, MLOps, Anomaly Detection, Root Cause Analysis, Automated Remediation, Data Observability, Pipeline Reliability, Diagnostic Reasoning.

Article History:

Received: 29.09.2024

Revised: 30.10.2024

Accepted: 15.11.2024

Published: 24.11.2024

1. Introduction

Modern organizations increasingly depend on data pipelines whose failures can directly affect executive dashboards, fraud systems, personalization engines, regulatory submissions, supply-chain decisions, and machine learning products. The pipeline has moved from a back-office integration mechanism to a production-critical system. In this setting, reliability is no longer defined only by whether a scheduled job finishes. A technically successful run can still deliver late, incomplete, biased, duplicated, semantically invalid, or distribution-shifted data. A pipeline can also satisfy local checks while violating downstream expectations or business semantics. These characteristics make data pipeline reliability a multidimensional challenge involving software engineering, distributed systems,

data management, machine learning operations, and governance. Earlier work on production machine learning emphasized that deployed systems create data-management challenges around validation, cleaning, enrichment, debugging, and understanding, all of which become more difficult when pipelines operate at scale and under continuous change [3].

The practical problem is that conventional monitoring remains too shallow for the failure modes of contemporary data platforms. Job status, CPU utilization, memory consumption, and simple row-count thresholds are useful but insufficient. They do not explain whether an upstream source changed the meaning of a field, whether a new backfill silently overwrote a historical partition, whether a transformation introduced label leakage, whether a late-arriving stream event changed aggregate semantics, or whether a seemingly local task failure is actually caused by a dependency graph conflict. In many organizations, engineers still rely on manual inspection of logs, ad hoc SQL probes, and institutional memory to diagnose incidents. This process increases mean time to detection and mean time to recovery, while also making remediation inconsistent across teams.

The need for self-healing arises from the gap between the complexity of data operations and the limits of human-centered incident handling. Self-healing data pipelines are pipelines that can observe their own behavior, detect abnormal conditions, infer likely causes, choose bounded remediation actions, verify outcomes, and preserve an auditable record of their decisions. The concept is related to autonomous computing, site reliability engineering, AIOps, and MLOps, but data pipelines impose distinctive constraints. Pipeline failures propagate across lineage; data quality errors may be silent; remediation can corrupt analytical truth if applied too aggressively; and incident response must respect governance requirements. These constraints make it inappropriate to equate self-healing with fully autonomous code modification. Instead, the research challenge is to define an architecture that permits automation where evidence is strong and risk is low, while preserving human authority over ambiguous, high-impact, or policy-sensitive cases.

Machine learning is central to this architecture because many pipeline failures are probabilistic, contextual, and multi-signal. A fixed rule can assert that a table should contain a non-null primary key, but it may not detect subtle seasonal deviations, abnormal correlations between features, unusual failure sequences in logs, or emerging performance regressions that precede job failure. At the same time, machine learning alone is not sufficient. Prior research on hidden technical debt in machine learning systems shows that ML components can amplify maintenance risks through entanglement, configuration debt, undeclared consumers, and data dependencies [17]. A self-healing pipeline must therefore combine learned models with contracts, lineage, guardrails, versioning, and explicit policy controls.

This paper proposes a machine learning-enabled architecture for autonomous failure detection, diagnostic reasoning, and automated remediation in data pipelines. The contribution is conceptual and design-scientific: it synthesizes existing research on production ML platforms, anomaly detection, AIOps, defect prediction, data validation, and software lifecycle governance into a unified architecture tailored to DataOps. The work is motivated by three observations. First, many pipeline failures are preceded by weak signals that can be detected by learning from historical execution and data profiles. Second, diagnosis benefits from causal and graph-aware reasoning over lineage, dependencies, telemetry, and incident history. Third, automated remediation is feasible for a meaningful subset of recurring failures if actions are typed, bounded, reversible, and monitored after execution.

The paper makes four contributions. It defines a fault taxonomy that distinguishes infrastructure, orchestration, schema, semantic, dependency, resource, security, and model-adjacent failures. It presents a five-layer autonomous architecture for self-healing data pipelines. It proposes a risk-aware remediation governance model that separates safe automatic repairs from human-approved repairs. Finally, it defines an evaluation protocol using controlled fault injection, offline replay, shadow remediation, and production canarying. The governance perspective is essential because automated pipeline repair can introduce data cascades, a phenomenon in which upstream data problems trigger compounding downstream errors and social-organizational consequences [20].

2. Research Context and Problem Formulation

A self-healing data pipeline can be defined as a pipeline that continuously monitors operational and data-quality state, detects deviations from expected behavior, diagnoses likely causal mechanisms, initiates repair actions within a governed action space, and verifies that the repair restored service without degrading data trust. This definition deliberately includes verification and governance. A retry that masks a transient network issue may be appropriate, but an imputation routine that fills missing values in a risk model may require stronger evidence, downstream impact analysis, and approval. The architecture must therefore combine fast operational automation with explicit decision rights.

The failure surface of modern pipelines is broad. At the ingestion layer, failures include source outages, authentication expiration, API contract changes, file-format changes, broken CDC offsets, and missing partitions. At the transformation layer, failures include SQL logic defects, incompatible library upgrades, skewed joins, incorrect windowing semantics, duplicate records, state-store corruption, and memory pressure. At the orchestration layer, failures include cyclic dependencies, race conditions, schedule drift, partial backfills, task starvation, and dead-letter accumulation. At the quality layer, failures include distribution shift, referential-integrity breaks, unexpected cardinality, semantic inconsistencies, and anomalous feature interactions. At the consumption layer, failures include dashboard breakage, model performance degradation, data contract violations, and stale materialized views. These failures often interact. A credential refresh issue can appear as a freshness anomaly; a source schema change can appear as a downstream model drift problem; a subtle upstream duplication can remain invisible until financial reporting reconciliation.

Traditional pipeline monitoring typically uses thresholds and job-level alerting. This approach has three limitations. First, thresholds are brittle under seasonality, product launches, marketing events, and legitimate growth. Second, job-level alerts do not provide causal evidence. Third, monitoring systems are often separated from remediation systems, creating a gap between detection and action. The architecture proposed here addresses these limitations by integrating observability and remediation through shared representations. Every pipeline run produces telemetry; telemetry becomes features; features feed detectors; detector outputs update a diagnostic graph; the diagnostic graph proposes remediation candidates; remediation execution produces new evidence that updates the system's future behavior.

Production ML platforms provide useful design lessons because they have long required reproducibility, validation, lineage, and deployment discipline. TensorFlow Extended, for example, demonstrates the value of standardizing platform components for production-scale machine learning workflows, including data analysis and validation as first-class concerns [1]. However, the self-healing pipeline problem extends beyond ML model training. It includes generic analytical pipelines, streaming jobs, lakehouse transformations, data marts, reverse ETL, and operational reporting. The proposed architecture therefore treats ML models as one class of consumer and one class of diagnostic method, rather than the only object of governance.

A second research stream concerns data validation. Data validation systems for machine learning demonstrate that schemas, statistics, and anomalies can be used to monitor production data at large scale [8]. In a self-healing setting, validation becomes both a detection mechanism and a remediation boundary. A validation failure can trigger quarantine, rollback, downstream suppression, or human review. Yet validation alone cannot explain why a failure happened. Diagnosis requires linking validation results to lineage, deployment events, upstream incidents, resource metrics, code changes, and historical incident patterns.

A third stream concerns observability. Observability for production ML pipelines emphasizes detection, diagnosis, and reaction across deployed pipelines whose failures may be silent or delayed [6]. This idea directly informs self-healing data pipelines because the reaction stage must be operationalized rather than left as a manual afterthought. Observability must produce actionable evidence: which table changed, which feature shifted, which operator produced the abnormality, which upstream source is implicated, what downstream assets are affected, and which repair options are safe.

A fourth stream concerns anomaly detection and AIOps. Anomaly detection research provides the algorithmic basis for identifying deviations from normal behavior across numerical metrics, event sequences, logs, traces, and multivariate time series [5]. AIOps research adds an operational framing in which detection, failure prediction, localization, diagnosis, and recovery are studied as connected tasks rather than isolated models [9]. Self-healing data pipelines inherit this framing, but their distinctive contribution is to place data semantics and lineage at the center of the reasoning process.

Recent AIOps research also raises the possibility of large language models for incident interpretation, query recommendation, and mitigation synthesis. Such models are useful as constrained reasoning assistants, especially for summarizing evidence and mapping incidents to known playbooks, but they should not be allowed to execute unbounded changes in data systems. Surveys of AIOps in the large language model era identify failure management subtasks such as detection, diagnosis, localization, and remediation, while also highlighting the need for reliability and generality in complex systems [25]. The architecture in this paper therefore treats language models as optional components inside a policy-governed diagnostic layer, not as unrestricted autonomous agents.

Software defect prediction and lifecycle governance are also relevant. Defects in transformation code, orchestration definitions, schema evolution scripts, and infrastructure configuration are major sources of pipeline failures. Comparative work on machine learning models for software defect prediction reinforces the practical importance of selecting models and evaluation metrics according to the characteristics of the target software context [2]. In data pipelines, defect prediction can be applied to code changes, DAG modifications, schema migrations, and configuration updates before deployment, allowing the platform to assign higher risk to changes that resemble historically failure-prone patterns.

A related line of work compares machine learning techniques for defect prediction and emphasizes performance evaluation across classifiers and datasets [16]. This evidence supports the use of model ensembles rather than a single universal detector in self-healing pipelines. A static rule, an isolation-based anomaly detector, a sequence model over logs, and a supervised failure predictor each detect different classes of incidents. Ensemble design should be driven by operational cost, interpretability, and false-positive tolerance.

Governance-oriented research on AI-driven software lifecycle management provides a useful perspective on how predictive models, architectural decisions, and project governance can be integrated into agile software processes [4]. Self-healing data pipelines require similar lifecycle integration: a remediation policy should be versioned with the pipeline, reviewed during architecture governance, tested before production, and monitored after deployment. The pipeline is not only a technical artifact but also a governed product with owners, risk tiers, service objectives, and audit obligations.

The broader expansion of machine learning within software development highlights why data engineering systems increasingly need ML-assisted development, testing, and operational decision support [12]. The same trend creates a paradox. ML can improve pipeline reliability, but it also adds complexity, model maintenance, and decision opacity. The solution is not to avoid ML, but to embed it within transparent operational structures: explicit features, explainable detector outputs, auditable decisions, and fallback mechanisms.

End-to-end AI-based systems engineering further suggests that lifecycle governance, predictive quality assurance, automation economics, and cybersecurity intelligence should be considered together rather than as disconnected concerns [14]. This integrated perspective is especially relevant to self-healing pipelines because a repair action can affect reliability, cost, security, compliance, and data quality simultaneously. A credential rotation repair may improve availability but create audit implications; an automatic backfill may restore completeness but increase cloud spend; an automated schema adaptation may preserve runtime success while violating a downstream contract.

Finally, converged AI architectures for innovation, lifecycle optimization, and cybersecurity risk mitigation illustrate the need to treat automation as part of a broader enterprise architecture rather than a narrow tool feature [10]. The paper's proposed architecture follows that principle. Self-healing is designed as an architectural capability spanning data contracts, observability, reasoning, remediation, policy, security, and human oversight.

3. Fault Taxonomy for Self-Healing Pipelines

A rigorous architecture requires a clear taxonomy of failure. The taxonomy proposed here classifies failures by observable symptoms, likely causal sources, potential blast radius, and remediation admissibility. Infrastructure failures include node loss, container eviction, disk pressure, network partitions, object-store throttling, executor failures, and cloud service disruptions. These are often observable through system metrics and can frequently be remediated through retries, resource scaling, failover, or rescheduling. Orchestration failures include DAG dependency errors, schedule drift, task concurrency conflicts, sensor timeouts, and partial retry loops. These require reasoning about task graphs and run histories. Schema failures include missing columns, changed data types, changed units, renamed fields, new enum values, and incompatible nested structures. Some schema changes are legitimate evolution events, while others are breaking changes. A self-healing system should distinguish additive non-breaking changes from semantic or contract-breaking changes. Remediation can include schema registry update, compatibility shim, quarantine, or escalation. Semantic data failures include abnormal distributions, referential-integrity violations, duplicate keys, impossible values, inconsistent aggregates, and drift in feature relationships. These failures are dangerous because they may not break execution. They require statistical profiles, domain rules, and downstream validation. Dependency failures occur when an upstream asset, library, container image, API endpoint, or credential changes in a way that disrupts a downstream pipeline. Remediation can include rollback, pinning, source failover, or

fallback to a last-known-good dataset. Code and configuration failures include transformation logic defects, wrong parameterization, bad feature flags, and incompatible runtime upgrades. Here, defect prediction and release governance can reduce risk before deployment, while post-deployment self-healing can rollback or disable a change. Security and compliance failures include unauthorized access attempts, unexpected data movement, policy violations, secrets leakage, and privacy-risk signals. These should generally trigger containment and escalation rather than autonomous repair unless approved playbooks exist.

Model-adjacent failures affect pipelines that produce training data, features, embeddings, labels, or inference inputs. Examples include training-serving skew, label delay, concept drift, feature missingness, and invalid embedding versions. These incidents often appear as model performance degradation after a data pipeline change. The self-healing system must therefore support downstream impact analysis and not limit itself to upstream execution states. If a feature table passes schema checks but causes a model monitor to detect unstable prediction distributions, the diagnostic reasoning layer should link the model signal back to recent data changes. Fault severity should be defined by business impact rather than technical symptom alone. A failed experimental pipeline may be low severity; a successful pipeline that publishes materially wrong revenue numbers may be critical. The proposed taxonomy therefore attaches each asset to a criticality tier, service objective, consumer list, compliance tag, and permissible remediation set. This metadata determines which actions can be executed automatically. For example, a non-critical exploratory table can be refreshed from a checkpoint, while a regulated financial table may only be quarantined and escalated.

4. Proposed Autonomous Architecture

The proposed architecture contains five layers: telemetry and lineage, feature and knowledge representation, failure detection, diagnostic reasoning, and automated remediation. The layers are connected by a feedback loop in which every decision and outcome becomes training and governance evidence for future decisions.

4.1. Telemetry and Lineage Layer

The first layer collects operational metrics, logs, traces, data-quality measurements, schema snapshots, lineage relationships, code deployment events, orchestration metadata, access events, cost metrics, and consumer signals. Telemetry must be standardized across batch, streaming, and hybrid workloads. It should include job start and end time, task duration, retry count, input and output row counts, data volume, partition completeness, null ratios, cardinality, freshness, distribution statistics, constraint violations, error messages, resource utilization, and upstream-downstream dependency edges. This layer should also preserve historical run context, because many anomalies are only meaningful relative to seasonality, pipeline version, upstream source version, and workload class. Streaming semantics matter because modern pipelines often process unbounded and out-of-order data. The Dataflow model demonstrates that correctness, latency, and cost must be balanced explicitly in massive-scale unbounded processing [15]. A self-healing system must therefore understand event time, processing time, watermarks, late arrivals, and retractions. Without this context, a detector may misclassify legitimate late events as missing data or treat delayed aggregates as data loss. Telemetry should capture watermark lag, late-event ratio, state-store size, and window completeness so that remediation can respect streaming semantics.

4.2. Feature and Knowledge Representation Layer

Raw telemetry becomes useful when transformed into representations for learning and reasoning. The architecture maintains a pipeline knowledge graph with nodes for datasets, tasks, jobs, services, sources, credentials, code repositories, models, dashboards, owners, incidents, and policies. Edges represent lineage, dependency, ownership, deployment causality, data contracts, and consumer relationships. Feature vectors are derived from this graph and from historical run records. Examples include deviation from seasonal baseline, distance from previous schema, anomaly score for row-count trend, failure frequency for a task, centrality of an asset in lineage, historical repair success rate, and similarity to known incidents.

4.3. Failure Detection Layer

The detection layer combines four families of detectors. Contract detectors evaluate explicit rules such as schema compatibility, not-null constraints, uniqueness, accepted domains, referential integrity, and freshness limits. Statistical detectors compare current profiles to baselines using robust statistics, change-point methods, and distributional distance. Machine learning detectors identify complex multivariate anomalies from metrics, logs, traces, and data profiles. Predictive detectors estimate the likelihood of failure before or during execution based on run context, code change features, resource patterns, and historical incidents. Structured Streaming shows the value of declarative real-time processing for applications that require continuous results and end-to-end reliability [21]. This motivates detectors that work both on batch snapshots and on streaming telemetry.

The detection layer should be ensemble-based because no single model covers all failure types. Thresholds are interpretable and fast, but brittle. Unsupervised detectors can identify novel deviations, but may be difficult to calibrate. Supervised classifiers can predict known failure classes, but depend on labeled incidents. Sequence models can learn log patterns, but may drift when logging changes. The architecture therefore uses detector diversity and evidence aggregation. Each detector emits a typed alert with confidence, supporting evidence, affected assets, severity estimate, and suggested diagnostic features. Alerts are deduplicated and correlated before entering the diagnostic layer.

4.4. Diagnostic Reasoning Layer

Detection answers the question "what looks abnormal?" Diagnosis answers "why is it abnormal, what else is affected, and what evidence supports that explanation?" The reasoning layer uses three complementary mechanisms. First, lineage traversal identifies upstream candidates that can explain downstream symptoms. If ten downstream tables fail after one upstream schema change, the upstream change becomes a high-priority hypothesis. Second, causal event correlation links anomalies to deployments, credential changes, source outages, feature flags, and resource events. Third, incident similarity compares the current evidence vector with historical incidents and successful repairs.

Spark's original design highlights the importance of scalable, fault-tolerant distributed computation for iterative and interactive workloads [23]. Self-healing diagnosis should adopt the same scalability assumption: the reasoning layer must process large telemetry histories and dependency graphs without becoming a bottleneck. In practice, the diagnostic service can use incremental graph updates, indexed event stores, precomputed lineage neighborhoods, and approximate nearest-neighbor search over incident embeddings.

Diagnostic outputs should be ranked hypotheses, not opaque conclusions. A hypothesis record should include suspected root cause, affected assets, confidence, evidence, counterevidence, recommended remediation, expected risk, and verification plan. For example, a diagnosis may state that a downstream null-rate anomaly is likely caused by an upstream schema change because the source introduced a renamed field, downstream transformations mapped the missing field to null, no resource anomalies occurred, and similar incidents were previously resolved by applying a compatibility mapping. This explanation is essential for human trust and auditability.

4.5. Automated Remediation Layer

The remediation layer executes typed actions selected by policy. Remediation actions include retry, exponential backoff, task rescheduling, resource scaling, source failover, checkpoint restoration, partition backfill, dead-letter replay, schema compatibility shim, data quarantine, rollback to last-known-good code, dependency version pinning, credential refresh, downstream notification, circuit breaker activation, and incident escalation. Each action has preconditions, risk tier, allowed asset classes, required evidence, maximum blast radius, rollback method, and verification criteria. Lessons from container-management systems such as Borg, Omega, and Kubernetes show that automated scheduling and resource management can support reliable operation when control loops are well designed [19]. Self-healing data pipelines extend this control-loop idea from compute resources to data and semantic correctness.

Remediation must be safe by construction. The architecture separates observation, recommendation, shadow execution, automatic execution, and human-approved execution. Low-risk actions such as retrying a transient failed task can be automatic. Medium-risk actions such as backfilling a non-critical partition can be automatic only with validation and rollback. High-risk actions such as changing financial data, rewriting regulated history, or modifying production transformation logic should require approval. The system should also support hold-down timers, canary repair, two-person approval for high-impact assets, and post-remediation validation.

5. Machine Learning Design

The machine learning design includes feature engineering, model selection, training data management, calibration, explainability, drift control, and feedback learning. Training data is built from historical pipeline runs, incidents, alerts, remediation actions, and outcomes. Each incident is represented as a time-bounded episode containing pre-failure signals, failure symptoms, diagnosis labels, remediation actions, and post-repair validation. Because labels are often incomplete, the system should combine supervised learning with weak supervision and unsupervised anomaly detection.

Log-based detection is especially useful because many pipeline failures first appear in structured and semi-structured execution messages. Deep learning approaches such as DeepLog model system logs as sequences and detect deviations from learned normal patterns [11]. For data pipelines, log templates can be combined with orchestration state and data-profile features. However, log detectors should be robust to logging changes and should not be the sole basis for remediation. They are strongest when used as one signal within an ensemble.

Time-series anomaly detection is also central. Pipeline metrics such as row count, null ratio, task duration, watermark delay, retry count, and output size are time-indexed and often seasonal. Comprehensive evaluations of time-series anomaly detection demonstrate that algorithm performance varies by anomaly type, data characteristics, efficiency, and robustness [24]. This means detector selection should be empirical. A platform may use robust seasonal baselines for stable daily loads, isolation forests for multivariate profile anomalies, recurrent models for sequential resource patterns, and change-point detection for sudden source behavior shifts.

Predictive failure models should include code and configuration signals. Features can include number of changed lines in transformation logic, modified dependencies, schema migration complexity, number of downstream consumers, historical failure rate of touched components, author experience with the asset, and similarity to prior risky changes. The goal is not to block innovation but to route risky changes through additional validation. For example, a high-risk DAG update can trigger extra synthetic data tests, shadow runs, or staged deployment.

Model calibration is crucial because remediation decisions require trustworthy confidence estimates. A detector with high recall but poor calibration may generate many false repairs. Calibration methods should be evaluated with reliability diagrams, expected calibration error, and cost-weighted decision thresholds. Thresholds should be asset-specific: a critical compliance table should require stronger evidence before write-changing remediation, while an internal experimentation table can tolerate more autonomous action. Explainability should be built into detector outputs through feature attribution, nearest historical incidents, rule traces, and graph evidence.

Feedback learning closes the loop. After each incident, the system records whether the alert was true, whether the diagnosis was accepted, which remediation was executed, whether it succeeded, whether it caused side effects, and how long recovery took. This feedback updates detector thresholds, diagnosis ranking, and remediation policy. The ML Test Score framework emphasizes systematic tests and monitoring requirements for production readiness and technical debt reduction [22]. An analogous self-healing readiness score can evaluate whether a pipeline has sufficient telemetry, contracts, lineage, tests, run history, rollback paths, and validated playbooks to permit autonomous remediation.

6. Diagnostic Reasoning and Decision Intelligence

The architecture's diagnostic layer is not merely a classifier. It is a decision-intelligence layer that connects evidence, uncertainty, actions, and governance. Decision intelligence is important because a remediation action is a decision under uncertainty with operational, financial, and compliance consequences. AI-driven agile lifecycle governance research argues for integrating decision intelligence into architecture-centered project management and software lifecycle governance [7]. In self-healing pipelines, this principle becomes concrete: incident decisions should be traceable to architecture metadata, risk tiers, policy constraints, and historical outcomes.

The diagnostic reasoning process begins with alert correlation. Alerts are grouped by time, lineage neighborhood, shared symptoms, and common deployment events. A cluster of freshness alerts across multiple downstream assets is likely different from independent data-quality alerts in unrelated domains. The system then generates hypotheses using a library of causal templates. Example templates include "upstream schema change causes downstream nulls," "resource saturation causes task timeout," "late stream events cause incomplete window aggregates," "credential expiration causes source freshness failure," and "code deployment causes semantic metric drift." Each template defines expected evidence and counterevidence.

Hypotheses are scored using probabilistic and graph-based methods. Evidence can include temporal precedence, lineage reachability, anomaly magnitude, historical similarity, deployment proximity, and detector confidence. Counterevidence can include unaffected downstream assets, successful validation at intermediate nodes, or unrelated resource metrics. The output is a ranked list of

candidate causes. Importantly, the system should preserve multiple hypotheses when evidence is ambiguous. Prematurely selecting one cause can trigger harmful repairs.

The remediation planner maps hypotheses to actions. For each candidate action, it estimates probability of success, expected recovery time, blast radius, reversibility, data-quality risk, compute cost, and approval requirement. The planner then chooses one of four modes: observe, recommend, execute, or escalate. Observe mode is used when the signal is weak or low impact. Recommend mode provides a human operator with evidence and a proposed action. Execute mode is used for low-risk repairs with validated playbooks. Escalate mode is used for high-risk, ambiguous, security-sensitive, or policy-restricted incidents.

Verification is the final step. A remediation is not successful merely because the job stops failing. Verification checks the original symptom, downstream contracts, freshness, distributional profiles, lineage impact, and consumer health. If verification fails, the system either rolls back or escalates. Verification also guards against the common anti-pattern of masking symptoms while leaving the causal problem unresolved. For instance, increasing memory may allow a job to pass but may not correct a skewed join introduced by a code change. The incident record should document whether the repair addressed the root cause or only restored temporary service.

7. Automated Remediation Governance

Autonomous remediation creates value only when it is safe, auditable, and aligned with organizational accountability. Governance begins with action typing. Each playbook should specify the assets it applies to, the required evidence, the maximum number of retries, the maximum amount of data that can be rewritten, the rollback method, the required validation checks, and the approval mode. Playbooks should be version controlled and reviewed like production code. They should also be tested with synthetic failures before being authorized for automatic execution. A remediation policy can be expressed as a matrix. Rows represent asset criticality tiers, and columns represent action classes. For a bronze-tier exploratory asset, the system may automatically retry, reschedule, backfill, and quarantine. For a gold-tier executive reporting asset, it may automatically retry and quarantine but require approval for backfill or schema adaptation. For regulated or customer-facing assets, the system may only isolate, preserve evidence, notify owners, and open an incident unless an approved playbook exists. This matrix prevents the architecture from becoming an uncontrolled automation mechanism.

Human-in-the-loop design should be selective. Requiring human approval for every repair defeats the purpose of self-healing. Allowing automation to rewrite all data is unsafe. The middle ground is graduated autonomy. The system earns autonomy through evidence, tests, and track record. A playbook that has succeeded hundreds of times on low-risk assets can be executed automatically. A new playbook should begin in shadow mode, where it recommends actions but does not execute. After sufficient validation, it can move to supervised execution and then to limited automatic execution. Auditability is not optional. Each incident should produce a decision record containing alerts, evidence, hypotheses, selected action, policy justification, operator approvals if any, execution logs, validation outcomes, rollback status, and downstream notifications. These records support compliance, postmortems, continuous improvement, and model retraining. They also make the system more trustworthy because teams can inspect why automation acted.

Security must be embedded. A self-healing agent often needs permissions to restart jobs, read logs, modify schedules, trigger backfills, update schemas, and write quarantine flags. These permissions should be least-privilege, scoped by asset, and separated across duties. The remediation agent should not possess broad credentials that enable arbitrary data mutation. Sensitive actions should require signed policy approval, and all agent activity should be logged immutably.

8. Evaluation Methodology

A rigorous evaluation should measure both technical effectiveness and operational safety. The proposed evaluation uses four stages: offline replay, fault injection, shadow remediation, and controlled production canarying. Offline replay applies detectors and diagnostic models to historical runs and incidents. It measures detection precision, recall, time-to-detection, diagnosis ranking accuracy, and calibration. Fault injection introduces controlled failures into non-production or staging pipelines. Examples include schema changes, missing partitions, delayed source files, resource throttling, bad transformations, duplicate rows, credential failures, and out-of-order stream events. Shadow remediation allows the system to recommend repairs without executing them, enabling comparison with human operator decisions.

Controlled production canarying is the final stage. Only low-risk playbooks should be enabled initially, and only for selected asset classes. Metrics should include mean time to detect, mean time to diagnose, mean time to recover, percentage of incidents automatically resolved, false alert rate, false remediation rate, rollback frequency, post-repair data-quality score, compute cost impact, and operator satisfaction. The evaluation should also measure negative outcomes: incorrect quarantine, unnecessary backfill, missed downstream impact, repeated failed retries, and automation-induced incidents. Production ML pipeline studies show that real-world pipelines have complex graph structures, repeated subcomponents, and opportunities for optimization across provenance and execution histories [13]. This supports evaluating self-healing systems at graph level rather than only job level. A job-level benchmark may show that an individual task recovered, while graph-level evaluation can reveal whether downstream consumers were protected, whether redundant computation was avoided, and whether the repair preserved lineage correctness.

Human-centered evaluation is also important. MLOps interview research emphasizes that engineers operationalizing machine learning balance velocity, visibility, and versioning while working through continual monitoring and response [18]. Self-healing pipelines should be evaluated against these human workflows. A system that reduces recovery time but produces inscrutable decisions may not be adopted. A system that provides clear ranked evidence and safe recommendations may improve operator trust even before full automation. The evaluation should include ablation studies. Removing lineage evidence tests whether the diagnostic graph improves root-cause ranking. Removing data-quality features tests whether semantic signals improve detection. Removing incident similarity tests whether historical memory improves remediation selection. Comparing rule-only, ML-only, and hybrid detectors tests whether ensemble design improves robustness. Measuring action outcomes by severity tier tests whether policy constraints reduce harmful automation.

9. Implementation Blueprint

A practical implementation can be built incrementally. The first phase establishes observability. Every pipeline run should publish standardized metadata: run identifier, code version, input assets, output assets, schema version, row counts, quality checks, timings, resource metrics, and status. Lineage should be captured at table, column, and task levels where feasible. Data contracts should define expected schema, freshness, ownership, and quality constraints. The second phase introduces detection. Start with contract and statistical detectors because they are interpretable and easy to validate. Add unsupervised anomaly detection for metrics and data profiles. Add log-sequence models and supervised failure predictors once sufficient incident history exists. At this stage, alerts should include evidence and asset impact. The goal is not merely to alert faster but to alert with context.

The third phase introduces diagnostic reasoning. Construct the knowledge graph and map alerts to affected lineage neighborhoods. Add causal templates and incident similarity search. Provide ranked hypotheses to operators. Avoid automatic repairs until diagnosis quality is measured. Collect operator feedback on whether hypotheses were useful and which evidence was missing. The fourth phase introduces remediation in shadow mode. For each incident, the system recommends a playbook but does not execute it. Operators accept, reject, or modify recommendations. This creates labeled data for playbook suitability. Low-risk playbooks can then move to supervised execution. For example, automatic retry with bounded count, automatic quarantine of invalid non-critical partitions, or automatic downstream notification can be enabled before more invasive actions. The fifth phase enables constrained autonomy. Only playbooks with proven success, clear rollback, and asset-level authorization are allowed to execute automatically. The system continuously monitors repair outcomes. If false remediation exceeds a threshold, autonomy is reduced. If an asset becomes more critical or subject to new compliance constraints, policy automatically tightens.

10. Discussion

The proposed architecture reframes data pipeline reliability as a continuous decision system. Detection, diagnosis, and remediation should not be separate tools connected by human improvisation. They should be parts of a feedback loop grounded in telemetry, lineage, policy, and learning. This design offers several benefits. It can reduce alert fatigue by correlating symptoms. It can reduce diagnosis time by ranking evidence. It can reduce recovery time by executing safe repairs automatically. It can improve governance by creating decision records. It can improve engineering productivity by converting recurring incidents into tested playbooks.

However, the architecture also creates risks. Automation can amplify errors if detectors are miscalibrated or if remediation policies are too permissive. ML-based diagnosis can be biased by incomplete incident history. Data-quality repair can preserve syntactic validity while corrupting semantic truth. Lineage graphs can be incomplete. Human operators may overtrust automation or ignore

recommendations if explanations are poor. These risks justify the paper's emphasis on constrained autonomy. The architecture also has organizational implications. Self-healing is not only a platform capability; it requires ownership models, incident labeling discipline, playbook review, governance boards, and cross-functional collaboration. Data producers and consumers must agree on contracts. Platform teams must provide observability. Security teams must approve agent permissions. Compliance teams must define audit requirements. Product teams must define impact tiers. Without these organizational foundations, technical automation will remain fragile.

A major research opportunity is the development of benchmark datasets for data pipeline self-healing. Existing anomaly and AIOps datasets rarely capture the full combination of data profiles, lineage, orchestration metadata, code changes, remediation actions, and downstream impact. Future benchmarks should include controlled failures across batch and streaming pipelines, annotated root causes, remediation candidates, and safety labels. Such datasets would enable meaningful comparison of diagnostic reasoning methods and remediation planners. Another research opportunity is policy-aware learning. Most anomaly detectors optimize statistical metrics, while self-healing systems must optimize operational utility under constraints. A detector for a critical table should be tuned differently from a detector for exploratory data. A remediation recommender should learn not only which action succeeds but also which action is allowed, reversible, and proportionate. Integrating policy into model training and decision thresholds is therefore an important direction.

11. Limitations

This paper presents an architecture and evaluation methodology rather than a production deployment report. It does not claim empirical performance improvements from an implemented system. The proposed design should be validated through staged implementation, controlled experiments, and longitudinal production studies. Another limitation is that the architecture assumes access to rich telemetry and lineage. Organizations with fragmented tooling may need substantial foundational work before advanced self-healing is feasible. The design also assumes that incidents are labeled with sufficient quality to train supervised components. In practice, incident records are often incomplete, inconsistent, or biased toward severe failures. The role of language models requires caution. They can summarize logs, generate hypotheses, and map incidents to documentation, but they may hallucinate causes or suggest unsafe repairs. In the proposed architecture, language models should be constrained by retrieval, policy, typed action schemas, and verification. They should support reasoning but not bypass governance. Finally, the architecture may not apply equally to all domains. Highly regulated environments, safety-critical systems, and low-latency operational systems may require stricter controls than internal analytical pipelines.

12. Conclusion

Self-healing data pipelines are becoming necessary because modern organizations depend on data systems whose failure modes are too numerous, subtle, and interconnected for purely manual operations. This paper proposed a machine learning-enabled architecture for autonomous failure detection, diagnostic reasoning, and automated remediation. The architecture integrates telemetry, lineage, data contracts, anomaly detection, incident similarity, causal reasoning, and policy-governed remediation into a closed operational loop. Its central principle is constrained autonomy: the system should automate only those actions for which evidence is sufficient, risk is bounded, rollback is available, and governance permits execution. The proposed design contributes a fault taxonomy, a five-layer architecture, a machine learning strategy, a diagnostic decision-intelligence model, a remediation governance framework, and an evaluation methodology. Future work should implement the architecture in representative batch and streaming environments, develop public benchmarks for pipeline incidents, and study how human operators interact with autonomous remediation systems over time.

References

[1] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, C. Y. Koo, L. Lew, C. Mewald, A. N. Modi, N. Polyzotis, S. Ramesh, S. Roy, S. E. Whang, M. Wicke, J. Wilkiewicz, X. Zhang, and M. Zinkevich, "TFX: A TensorFlow-Based Production-Scale Machine Learning Platform," in Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, 2017, pp. 1387-1395, https://doi.org/10.1145/3097983.3098021.

[2] S. K. Gunda, "Comparative Analysis of Machine Learning Models for Software Defect Prediction," 2024 International Conference on Power, Energy, Control and Transmission Systems (ICPECTS), Chennai, India, 2024, pp. 1-6, https://doi.org/10.1109/ICPECTS62210.2024.10780167.

[3] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich, "Data Management Challenges in Production Machine Learning," in Proceedings of the 2017 ACM International Conference on Management of Data, Chicago, IL, USA, 2017, pp. 1723-1726, https://doi.org/10.1145/3035918.3054782.

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

[5] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly Detection: A Survey," ACM Computing Surveys, vol. 41, no. 3, Article 15, pp. 1-58, 2009, https://doi.org/10.1145/1541880.1541882.

[6] S. Shankar and A. G. Parameswaran, "Towards Observability for Production Machine Learning Pipelines," Proceedings of the VLDB Endowment, vol. 15, no. 13, pp. 4015-4022, 2022, https://doi.org/10.14778/3565838.3565853.

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

[8] E. Breck, N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich, "Data Validation for Machine Learning," Proceedings of Machine Learning and Systems, vol. 1, pp. 334-347, 2019.

[9] P. Notaro, J. Cardoso, and M. Gerndt, "A Survey of AIOps Methods for Failure Management," ACM Transactions on Intelligent Systems and Technology, vol. 12, no. 6, Article 81, pp. 1-45, 2021, https://doi.org/10.1145/3483424.

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

[11] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 2017, pp. 1285-1298, https://doi.org/10.1145/3133956.3134015.

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

[13] D. Xin, H. Miao, A. Parameswaran, and N. Polyzotis, "Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities," in Proceedings of the 2021 International Conference on Management of Data, Virtual Event, China, 2021, pp. 2639-2652, https://doi.org/10.1145/3448016.3457566.

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

[15] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernandez-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing," Proceedings of the VLDB Endowment, vol. 8, no. 12, pp. 1792-1803, 2015, https://doi.org/10.14778/2824032.2824076.

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

[17] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden Technical Debt in Machine Learning Systems," in Advances in Neural Information Processing Systems 28, 2015, pp. 2503-2511.

[18] S. Shankar, R. Garcia, J. M. Hellerstein, and A. G. Parameswaran, "Operationalizing Machine Learning: An Interview Study," arXiv:2209.09125, 2022.

[19] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," ACM Queue, vol. 14, no. 1, pp. 70-93, 2016, https://doi.org/10.1145/2898442.2898444.

[20] N. Sambasivan, S. Kapania, H. Highfill, D. Akrong, P. K. Paritosh, and L. M. Aroyo, "Everyone Wants to Do the Model Work, Not the Data Work: Data Cascades in High-Stakes AI," in Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems, Yokohama, Japan, 2021, Article 39, pp. 1-15, https://doi.org/10.1145/3411764.3445518.

[21] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia, "Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark," in Proceedings of the 2018 International Conference on Management of Data, Houston, TX, USA, 2018, pp. 601-613, https://doi.org/10.1145/3183713.3190664.

[22] E. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley, "The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction," in Proceedings of the 2017 IEEE International Conference on Big Data, Boston, MA, USA, 2017, pp. 1123-1132.

[23] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing, Boston, MA, USA, 2010, pp. 10-10.

[24] S. Schmidl, P. Wenig, and T. Papenbrock, "Anomaly Detection in Time Series: A Comprehensive Evaluation," Proceedings of the VLDB Endowment, vol. 15, no. 9, pp. 1779-1797, 2022, https://doi.org/10.14778/3538598.3538602.

[25] L. Zhang, T. Jia, M. Jia, Y. Wu, A. Liu, Y. Yang, Z. Wu, X. Hu, P. S. Yu, and Y. Li, "A Survey of AIOps for Failure Management in the Era of Large Language Models," arXiv:2406.11213, 2024.