Nanotechnology Perceptions
ISSN 1660-6795
www.nano-*ntp.com*

Designing Self-Healing Data Pipelines Using Metadata, Observability, And Intelligent Anomaly Detection

Jyoti Maheshwari

Independent Researcher jyotimaheshwariatwork@gmail.com

Data pipelines now sit behind everything from routine reporting to machine-learning products and regulatory submissions. Yet many of their most damaging failures do not appear as failed jobs. A workflow may complete on time and still publish stale, incomplete, duplicated, or structurally inconsistent data. This paper presents a practical framework for making such pipelines more resilient by combining metadata, data observability, lineage, anomaly detection, and controlled remediation. The design treats self-healing as a closed operational loop: detect an abnormal condition, assemble its context, identify plausible causes, choose a permitted response, verify the outcome, and retain the evidence. The detection layer combines explicit data contracts, robust statistical baselines, and an unsupervised model so that both known and unfamiliar failure patterns can be surfaced. The remediation layer is deliberately conservative. Retries, bounded backfills, deterministic deduplication, and quarantine may be automated when they are reversible and well understood; changes with semantic or financial consequences remain subject to approval. A fixed-seed synthetic experiment covering 365 daily pipeline runs was used to test the design. Forty faults were injected across seven categories. The hybrid detector identified all 40 faults and produced three false positives, corresponding to precision of 0.930, recall of 1.000, and an F1 score of 0.964. These results are not evidence of production performance, but they show how complementary detection methods and policy controls can be evaluated together. The paper concludes with an implementation blueprint and a staged adoption path for organizations that want to improve reliability without granting unsafe autonomy to their data platforms.

Keywords: self-healing data pipelines; data observability; metadata; anomaly detection; data lineage; automated remediation; DataOps; reliability engineering.

1. Introduction

1.1 Background and motivation

Modern data platforms are rarely single systems. A business metric may pass through application databases, event streams, third-party files, object storage, transformation jobs, orchestration tools, semantic layers, and dashboards before anyone uses it. Each component can appear healthy on its own while the final result is still wrong. That gap between technical success and trustworthy data is where many expensive incidents begin.

Data observability broadens monitoring beyond infrastructure and task status. It asks whether data arrived when expected, whether volumes changed unexpectedly, whether schemas drifted, whether distributions remain plausible, and which downstream assets may

be affected. Even so, visibility alone does not resolve an incident. Engineers still need to decide what happened, whether the response is safe, and how to prove that the repair worked. The term self-healing is often used too casually. In a production environment, an anomaly score should never be enough to justify rewriting financial data, weakening a contract, or accepting a breaking schema change. In this study, self-healing has a narrower and more useful meaning: a governed process that can detect a problem, gather evidence, select only an approved and bounded action, validate the result, and escalate when uncertainty remains.

1.2 Research problem and questions

Pipeline reliability is usually spread across separate tools and teams. The orchestrator records task state, the catalog stores metadata, the quality platform records validation results, and the incident system contains prior responses. Engineers must often reconstruct the story manually. This study asks how those signals can be brought together into one auditable decision process.

- RQ1. Which metadata is required to support explainable detection, diagnosis, and remediation?
- RQ2. How should deterministic rules, robust statistics, and machine-learning detectors be combined?
- RQ3. How can lineage improve root-cause ranking and downstream-impact analysis?
- RQ4. Which policy controls determine whether an action may run automatically?
- RQ5. How should a self-healing system be evaluated beyond detector accuracy?

1.3 Contributions

1. A six-layer reference architecture that separates the execution plane from a metadata-driven reliability control plane.
2. A minimum viable metadata model covering datasets, jobs, runs, contracts, lineage, incidents, remediation actions, and validation evidence.
3. A transparent hybrid detector and a lineage-aware diagnostic workflow.
4. A risk-based Healing Eligibility Score and policy matrix that limit unsafe autonomy.
5. A reproducible synthetic fault-injection experiment and enterprise adoption roadmap.

2. Related Work and Conceptual Foundations

2.1 Data quality and observability

Data quality has long been described through dimensions such as completeness, validity, consistency, uniqueness, accuracy, and timeliness. At platform scale, those concepts are useful only when they can be measured continuously and tied to ownership and action. Work by Schelter et al. showed that large-scale verification can be expressed as executable constraints rather than periodic manual checks. Data observability builds on that idea by adding runtime behavior, freshness, schema change, distribution, and lineage to the picture. The important shift is operational: quality is treated as a property that must be watched during every run, not inspected after a failure reaches a consumer.

2.2 Anomaly detection

No single anomaly-detection method is sufficient for production data. Explicit rules are excellent for known contracts, such as a missing partition or an invalid null rate, but they cannot anticipate every unusual pattern. Rolling medians and median absolute deviation are useful when data is seasonal or noisy because they are less sensitive to extreme values than means and standard deviations. Unsupervised methods can add another perspective by considering several signals at once, although their outputs are harder to explain and calibrate. For that reason, the proposed design uses machine learning as supporting evidence rather than as the sole authority for an operational action.

2.3 Metadata and lineage

Metadata gives an anomaly meaning. A sudden volume drop is more serious when it occurs in a regulatory dataset than in an experimental table; a retry is safer when the job is idempotent; and a schema change is easier to assess when ownership, compatibility rules, and downstream dependencies are known. Lineage standards such as OpenLineage provide a useful model of datasets, jobs, and runs. In the proposed framework, lineage is not only documentation. It is queried during an incident to trace likely upstream causes and estimate the blast radius downstream.

2.4 Workflow recovery and idempotency

Orchestrators already expose useful recovery mechanisms, including retries, reruns, backfills, and dependency controls. These features are necessary, but they are not self-healing by themselves. A retry may duplicate side effects. A backfill may republish an incorrect transformation. A rollback may restore code while leaving corrupted output in place. Recovery therefore has to be informed by metadata, constrained by policy, and followed by independent validation.

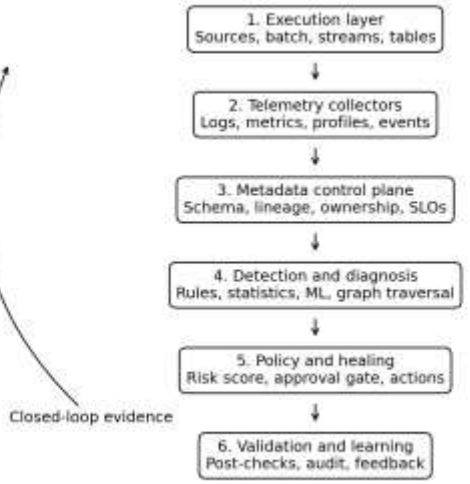
2.5 Research gap

The literature offers mature techniques for data-quality checks, anomaly detection, lineage capture, orchestration, and reliability engineering. What is less developed is the practical decision layer between them: how evidence is combined, how a cause is ranked, how an action is authorized, and how the system proves that the action improved rather than worsened the situation. This paper focuses on that implementation gap.

3. Proposed Self-Healing Framework

The proposed design separates ordinary data processing from reliability control. Pipelines continue to run in the execution plane. A separate control plane consumes their telemetry and metadata, detects unusual conditions, creates an incident context, evaluates the available responses, triggers only those actions allowed by policy, and records the outcome. This separation keeps recovery logic from becoming scattered across individual jobs and makes decisions easier to audit.

Figure 1. Metadata-driven closed-loop architecture for self-healing data pipelines.



3.1 Layer 1: execution plane

The execution plane includes source systems, ingestion services, streaming applications, batch transformations, storage layers, warehouses, semantic models, and serving interfaces. The framework does not depend on a specific vendor or engine. It does, however, require every managed asset and run to have a stable identifier. In practice, that means adopting a consistent naming scheme and emitting run events at the points where data is read, transformed, written, and published.

3.2 Layer 2: telemetry collection

Telemetry collection should be broad enough to reconstruct what happened without forcing operators to search several systems. Four types of evidence are especially useful: operational metrics, data profiles, change events, and validation results. Operational metrics cover state, duration, retries, resource use, queue time, and error signatures. Data profiles include row counts, null and duplicate rates, partition counts, freshness, and selected distribution statistics. Change events capture code, configuration, schema, and ownership changes. Validation results record whether contracts, reconciliations, and business rules passed.

3.3 Layer 3: metadata control plane

The metadata repository acts as the memory of the control plane. It stores more than table descriptions. For each dataset, it records ownership, business importance, expected delivery time, contracts, sensitivity, retention requirements, accepted schema changes, and the remediation actions that are allowed. Run records connect inputs, outputs, code versions, parameters, validation results, and observed incidents. Without that history, the platform can alert, but it cannot make a defensible recovery decision.

Table 1. Minimum viable metadata model.

Entity	Minimum metadata	Operational purpose
Dataset	Identifier, owner, criticality, schema, partitioning, SLO, sensitivity, contract	Determines expected behavior and business risk
Job	Identifier, code version, inputs, outputs, schedule, idempotency class	Supports change correlation and safe actuation
Run	Run ID, interval, state, duration, resources, input/output versions	Provides event-level execution evidence
Quality profile	Counts, nulls, duplicates, quantiles, categories, distribution sketches	Supports rules and adaptive baselines
Lineage edge	Source, target, transformation, observed/declarative flag	Enables cause and blast-radius traversal
Incident	Signals, anomaly scores, suspected cause, affected assets, severity	Creates a unified diagnostic object
Remediation	Action type, parameters, preconditions, reversibility, approver	Constrains automation
Validation	Post-checks, reconciliation, publication state, final disposition	Proves or rejects recovery

3.4 Layer 4: detection and diagnosis

Detection is deliberately multi-method. Deterministic checks cover conditions that are already understood and should never be ambiguous. Robust statistical methods compare a run with its recent history and are well suited to volume, freshness, and distribution changes. An unsupervised detector looks for combinations of signals that are unusual even when no individual threshold is crossed. The output is a set of observations and confidence measures, not a final judgment about business correctness.

Diagnosis starts by asking what changed near the time of the anomaly. The control plane examines recent code deployments, schema revisions, configuration changes, upstream failures, and ownership events. It then walks the lineage graph upstream to find shared causes and downstream to identify affected consumers. Candidate causes are ranked using temporal proximity, dependency distance, supporting error signatures, and similarity to earlier incidents. The ranking is intended to narrow investigation, not replace an engineer's judgment.

3.5 Layer 5: policy and healing

The healing orchestrator translates an incident into a small set of possible responses. Those responses may include a bounded retry, a targeted backfill, quarantine, rollback, resource adjustment, restoration of a known schema mapping, or republishing after deterministic

deduplication. Each action has declared preconditions, a maximum scope, a rollback path, and a validation plan. An action that cannot meet those requirements is not eligible for unattended execution.

A Healing Eligibility Score (HES) summarizes whether automation is acceptable:

HES = 0.30C + 0.20R + 0.15S + 0.15V - 0.10I - 0.10B

Here, C represents confidence in the diagnosis, R the reversibility of the proposed action, S the historical success rate of that action, V the strength of the planned validation, I the likely downstream impact, and B the business criticality of the asset. Each value is normalized between 0 and 1. The weights are starting points rather than universal constants. A finance platform, for example, would normally assign more weight to business criticality and impact than a low-risk analytical sandbox.

Table 2. Risk-based remediation policy.

Risk class	Typical examples	Default disposition
Class A: bounded and reversible	Retry transient failure; backfill one missing partition; resubmit idempotent task	Automatic after precondition check
Class B: deterministic data repair	Deduplicate using an approved business key; restore last valid mapping	Automatic only with quarantine and full validation
Class C: semantic uncertainty	Volume drop; distribution shift; unexpected null growth	Quarantine and human approval
Class D: structural or regulated	Breaking schema; contract relaxation; financial restatement; access-policy change	Block and require authorized approval

3.6 Layer 6: validation and learning

A repair is not complete simply because a command exits successfully. The system must rerun the failed controls, compare the new output with a trusted baseline, reconcile counts or business totals where relevant, confirm that downstream publication is in the expected state, and verify that the action did not introduce a different anomaly. The validation outcome is stored with the incident so that future decisions can use actual recovery history rather than assumptions.

4. End-to-End Operational Workflow

Table 3. Closed-loop operational workflow.

Stage	Required behavior
Observe	Collect run events, profiles, schemas, SLO state, and change events.
Detect	Apply deterministic, statistical, and multivariate detectors.

Stage	Required behavior
Correlate	Group related signals by asset, interval, lineage neighborhood, and time.
Diagnose	Rank probable causes and calculate downstream blast radius.
Decide	Apply hard policies and compute healing eligibility.
Act	Execute a bounded action or quarantine and request approval.
Validate	Run technical and business checks, then republish or roll back.
Learn	Record evidence, action, outcome, and operator feedback.

4.1 Pragmatic incident example

Consider a daily fact table due by 06:00 and normally containing about one million rows. At 06:05, the expected partition is missing. The freshness check fires. Lineage shows that extraction completed, while the transformation failed after a transient resource error. The job is known to be idempotent, the backfill window is limited to one partition, and the same action has succeeded in similar incidents. In that case, the platform can rerun the failed interval, validate counts and business totals, and publish only after those checks pass.

A successful run with a 45% volume drop is different. There may be no technical error to retry. If an upstream application was released shortly before the drop, the data may be semantically incomplete even though the pipeline finished normally. The safer response is to hold publication, preserve the last known good output, assemble the supporting evidence, and request review. Quarantine is therefore treated as a valid healing action, not as a failure to automate.

5. Materials and Methods

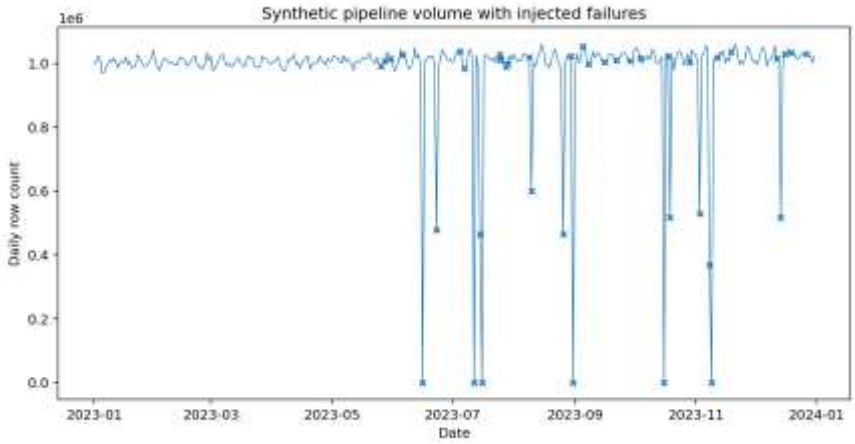
5.1 Research design

The study uses a design-science approach. It begins with a recurring operational problem, constructs an artifact to address it, and evaluates that artifact under controlled conditions. The artifact comprises the reference architecture, metadata model, detector ensemble, policy mechanism, and validation workflow. Production incident data was not available for publication, so the experiment uses synthetic data and makes no claim that its results represent a specific enterprise environment.

5.2 Synthetic pipeline

The experiment models 365 daily pipeline runs beginning on 1 January 2023. Row volume starts near one million records and includes weekly seasonality, a modest upward trend, and random noise. The dataset also includes null rate, duplicate rate, freshness delay, a distribution mean, schema-change indicators, and resource-failure indicators. The goal is not to imitate every feature of a real platform, but to provide enough variation to test how the detection and policy layers behave.

Figure 2. Synthetic daily volume with injected anomalies marked by crosses.



5.3 Fault-injection scenarios

Table 4. Injected failure scenarios.

Failure class	Count	Injected behavior	Expected response
Missing partition	6	Row count set to zero; freshness delay set to 1,440 minutes	Targeted backfill if idempotent
Volume drop	8	Row count reduced to 35-65% of generated value	Quarantine and investigate
Duplicate spike	6	Duplicate rate increased to 3-10%	Deterministic deduplication if contract allows
Null spike	6	Null rate increased to 8-25%	Quarantine and owner review
Distribution shift	6	Mean shifted by 12-25 units in either direction	Semantic validation
Breaking schema	4	Binary breaking-change signal activated	Block publication
Resource failure	4	Failure signal and delay of 180-600 minutes	Bounded retry/resource adjustment

5.4 Detectors

The rule-based detector flags conditions that correspond to explicit contracts: zero rows, null rate above 5%, duplicate rate above 2%, freshness delay above 120 minutes, a breaking

schema event, or a resource failure. These thresholds are easy to understand and audit. They also make the limitation of rules visible: failures that remain inside the predefined boundaries may still be important.

The statistical detector uses the previous 28 observations to calculate a rolling median and median absolute deviation, provided at least 14 earlier observations exist. A robust z-score of 0.6745(x - median)/MAD is then computed. Row-count and distribution-mean changes are flagged when the absolute score exceeds 5. The method was chosen because it tolerates occasional extremes better than a conventional z-score and can adapt to local behavior.

The multivariate component is an Isolation Forest with 300 trees and a contamination setting of 0.03. It is fitted on the first 120 days after the input variables are standardized; skewed rate and delay measures are log transformed. The alert cutoff is set at the 99th percentile of training scores. The hybrid detector raises an incident when any of the three components fires. This favors recall, with policy and validation used later to prevent unsafe action.

5.5 Evaluation metrics

Detector performance is measured on days 121 through 365 using precision, recall, and F1 score. A true positive is an injected fault day that is flagged; a false positive is a normal day that is flagged; and a false negative is an injected fault day that is missed. The evaluation also counts how many incidents the policy would permit to proceed toward automatic remediation, because high detection accuracy alone does not demonstrate operational safety.

5.6 Reproducibility

The data generator and detectors use a fixed random seed of 42. The accompanying files include the full daily dataset and aggregate results. Fault dates, thresholds, training windows, and model parameters are stated explicitly so the experiment can be rerun and challenged. Reproducibility here means that another reader can obtain the same synthetic result, not that the result is universally generalizable.

6. Results

6.1 Detection performance

Table 5. Detection performance on days 121-365.

Detector	Precision	Recall	F1	TP	FP	FN
Deterministic rules	1.000	0.650	0.788	26	0	14
Robust statistical	1.000	0.900	0.947	36	0	4
Isolation Forest	0.400	0.050	0.089	2	3	38
Hybrid union	0.930	1.000	0.964	40	3	0

The deterministic checks produced no false positives but detected 26 of the 40 injected faults. Their recall of 0.650 is unsurprising: they found missing partitions, large null and duplicate spikes, severe freshness delays, breaking schemas, and resource failures, but they

were not designed to recognize contextual volume or distribution changes. The statistical detector covered those contextual cases and detected 38 faults, although it also produced three false positives. The Isolation Forest was weaker on its own. The hybrid approach detected all 40 faults and retained the same three false positives, resulting in precision of 0.930, recall of 1.000, and an F1 score of 0.964.

Figure 3. Precision, recall, and F1 by detector.

6.2 Performance by failure class

Table 6. Detected injected incidents by failure class.

Failure type	Injected	Rules	Statistical	Isolation Forest	Hybrid
missing_partition	6	6	6	0	6
volume_drop	8	0	8	0	8
duplicate_spike	6	6	6	0	6
null_spike	6	6	6	1	6
distribution_shift	6	0	6	0	6
schema_break	4	4	0	0	4
resource_failure	4	4	4	1	4

Results by failure class show why the methods are complementary. Contract violations are handled most cleanly by explicit checks. Volume and distribution shifts are better detected against recent history. The relatively weak standalone machine-learning result is also informative: adding an unsupervised model does not automatically improve a reliability system. Its value depends on feature design, calibration, and how its evidence is combined with more interpretable controls.

6.3 Policy outcomes

The policy layer marked 16 of the 40 injected incidents as candidates for automatic remediation: six missing partitions, six duplicate spikes, and four resource failures. The other 24 incidents were routed to quarantine or approval because the correct response depends on business meaning or structural compatibility. The point of the policy is not to maximize the

percentage of incidents handled without people. It is to automate the small set of actions that are repeatable, bounded, and easy to verify.

7. Discussion

7.1 Interpretation

Three practical lessons emerge from the experiment. First, explicit rules remain essential because they are precise, explainable, and closely tied to contracts. Second, rolling statistical baselines add useful coverage for contextual failures without requiring a labeled incident history. Third, anomaly detection and remediation must be evaluated separately. A detector can be correct while the proposed repair is unsafe, and a conservative quarantine decision can be operationally better than an aggressive automated fix.

This broader view also changes what recovery means. In some incidents, the safest response is not to modify the data at all. Blocking publication, isolating a partition, serving the last known good result, or presenting a ranked set of likely causes may protect consumers more effectively than an uncertain repair. This is particularly important in domains where an incorrect correction can be harder to detect than the original failure.

7.2 Why metadata is the control plane

A detector can say that a row count is unusual. It cannot, by itself, know whether the dataset is critical, whether a backfill is permitted, which code version produced the output, who owns the asset, what depends on it, or what evidence would demonstrate recovery. Metadata supplies those answers. That is why the framework treats metadata as the operational control plane rather than as passive documentation.

7.3 Safety principles

- Fail closed for semantic uncertainty: stop or quarantine publication rather than guessing.
- Require idempotency: no automatic retry or backfill without documented side-effect behavior.
- Bound every action: constrain interval, records, runtime, retries, and resources.
- Separate detection from authorization: a high anomaly score does not grant permission to modify data.
- Validate independently: use controls that are not identical to the detector that triggered the action.
- Preserve evidence: retain pre-action state, parameters, approvals, and validation outcomes.
- Maintain a kill switch: operators must be able to disable automation globally or by asset class.

7.4 Implementation blueprint

Table 7. Technology-neutral implementation blueprint.

Capability	Pragmatic implementation option
Event and lineage capture	OpenLineage-compatible run events emitted by the orchestrator and transformation engine
Metadata storage	Relational metadata store plus graph representation for lineage traversal
Profiles	Incremental SQL aggregates, sketches, and partition-level metrics rather than repeated full scans
Detection	Versioned rule definitions, rolling robust statistics, optional model service
Incident correlation	Stream processor or scheduled service grouping signals into incident objects
Actuation	Orchestrator API for retry/backfill; deployment API for rollback; warehouse procedures for quarantine
Policy	Policy-as-code repository with peer review and asset-level overrides
Audit	Append-only incident and action log with immutable identifiers
Operator interface	Evidence summary, blast radius, recommended action, approval control, and rollback status

7.5 Phased adoption roadmap

Adoption should proceed in stages. The first stage establishes basic observability: stable asset identifiers, ownership, criticality, freshness, volume, schema, and incident history. The second adds lineage and standardized run events. The third automates diagnosis and blast-radius analysis while keeping all actions manual. Only after those foundations are reliable should an organization enable low-risk actions such as bounded retries or isolated backfills. More consequential actions should remain approval-based until repeated evidence supports a change.

7.6 Organizational considerations

The organizational design matters as much as the technical design. Dataset owners define contracts and business validation. Platform teams maintain event collection and recovery actuators. Governance or reliability teams approve policy classes. Security teams control credentials and execution boundaries. Operators review exceptions and failed validations. Without clear accountability, a self-healing platform can quickly become a system in which everyone assumes someone else approved the risk.

8. Limitations and Threats to Validity

The experiment is intentionally small and synthetic. It does not capture the full behavior of skewed partitions, correlated upstream failures, streaming windows, late-arriving events, multi-region outages, or delays introduced by human approval. The injected faults are mostly

isolated point events, while real incidents often unfold gradually or interact with one another. The measured detector scores should therefore be read as evidence that the evaluation is reproducible, not as a forecast of production performance.

The remediation layer is evaluated as a policy classification rather than through live infrastructure recovery. The study does not measure actual recovery time, cloud cost, or downstream side effects. Future work should test the framework on replayed incident histories, compare alternative scoring policies, include gradual and correlated faults, and examine whether operators find the diagnostic evidence useful. A production study should also measure how often automated actions are reversed and how much engineering effort is genuinely reduced.

9. Conclusion

Self-healing data pipelines are best understood as governed reliability systems, not as scripts that react automatically to anomaly scores. A useful design brings together metadata, observability, complementary detection methods, lineage-aware diagnosis, bounded action, and independent validation. The synthetic evaluation showed that a hybrid detector can improve coverage, while the policy layer prevents that wider detection surface from turning into unsafe automation. The practical goal is therefore not full autonomy. It is faster detection, better context, repeatable recovery for low-risk cases, and disciplined escalation when the system does not know enough to act safely.

References

1. Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., & Whittle, S. (2015). 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, 8(12), 1792-1803. https://doi.org/10.14778/2824032.2824076
2. Apache Airflow. (2026). Dag run and backfill documentation. https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/dag-run.html
3. Breunig, M. M., Kriegel, H.-P., Ng, R. T., & Sander, J. (2000). LOF: Identifying density-based local outliers. Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, 93-104. https://doi.org/10.1145/342009.335388
4. Chandola, V., Banerjee, A., & Kumar, V. (2009). Anomaly detection: A survey. ACM Computing Surveys, 41(3), Article 15. https://doi.org/10.1145/1541880.1541882
5. Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. Communications of the ACM, 51(1), 107-113. https://doi.org/10.1145/1327452.1327492
6. Google. (2016). Site Reliability Engineering: How Google Runs Production Systems. O'Reilly Media. https://sre.google/sre-book/table-of-contents/
7. Kleppmann, M. (2017). Designing Data-Intensive Applications. O'Reilly Media.
8. Liu, F. T., Ting, K. M., & Zhou, Z.-H. (2008). Isolation Forest. 2008 Eighth IEEE International Conference on Data Mining, 413-422. https://doi.org/10.1109/ICDM.2008.17
9. OpenLineage. (2023). Announcing OpenLineage 1.0. https://openlineage.io/blog/1.0-release/

10. OpenLineage. (2026). Object model and specification documentation. https://openlineage.io/docs/next/spec/object-model/
11. Page, E. S. (1954). Continuous inspection schemes. Biometrika, 41(1/2), 100-115. https://doi.org/10.2307/2333009
12. Polyzotis, N., Roy, S., Whang, S. E., & Zinkevich, M. (2018). Data lifecycle challenges in production machine learning: A survey. SIGMOD Record, 47(2), 17-28. https://doi.org/10.1145/3299887.3299891
13. Schelter, S., Lange, D., Schmidt, P., Celikel, M., Biessmann, F., & Grafberger, A. (2018). Automating large-scale data quality verification. Proceedings of the VLDB Endowment, 11(12), 1781-1794. https://doi.org/10.14778/3229863.3229867
14. Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., & Dennison, D. (2015). Hidden technical debt in machine learning systems. Advances in Neural Information Processing Systems, 28, 2503-2511.
15. Shkuro, Y. (2022). Positional paper: Schema-first application telemetry. Proceedings of the 2022 ACM SIGMOD Workshop on Data Systems for End-to-End Machine Learning. https://doi.org/10.1145/3544497.3544500
16. Wani, D., Ackerman, S., Farchi, E., Liu, X., Chang, H.-W., & Lalithsena, S. (2023). Data drift monitoring for log anomaly detection pipelines. arXiv:2310.14893. https://doi.org/10.48550/arXiv.2310.14893