Svedberg Open
DISSEMINATION OF KNOWLEDGE

International Journal of Artificial Intelligence and Machine Learning

Publisher's Home Page: https://www.svedbergopen.com/



Research Paper Open Access

A Data Pipeline Observability And Reliability Framework For Enterprise Insurance Systems: Real-Time Health Monitoring, Data Quality, And Automated Alerting At Scale

Gowtham Reddy Pappula

Tech Lead Data Engineer Independent Researcher, Email: gowthampappulaconnects@gmail.com

Abstract
Enterprise insurance data platforms move large volumes of sensitive data across many systems on fixed schedules, and the monitoring that watches them almost always watches execution status. Execution status answers whether a job ran. It does not answer whether the data that job produced is complete, current, correctly shaped or safe to publish. A pipeline can finish with a success code while loading a partial file, reprocessing stale source data, duplicating records, or silently dropping a column that an upstream system renamed, and every one of those outcomes reaches downstream reporting with no alert attached to it. This paper presents an observability and reliability framework that treats operational health and data quality as two signal families that must be evaluated together and reduced to a single health score per pipeline. The framework covers ingestion-time validation before processing begins, execution monitoring against historical baselines rather than static thresholds, staged data quality assessment returning pass, warning or fail, metadata and lineage capture for impact analysis, health scoring, and severity-based alert routing with deduplication and correlation. The work is grounded in a production deployment in an enterprise insurance data environment between July 2025 and July 2026, in which a centralised dashboard monitored approximately 12,500 scheduled jobs at an approximately one-minute refresh interval, reduced manual monitoring effort by approximately two to four hours per engineer per day, and made failures and delays visible within one to five minutes of a status change against several hours under manual checking. Those four outcomes are measured. The wider architecture described here, including standardised quality monitoring, health scoring, and the alerting policy, is generalised from that deployment and has not been formally evaluated; its expected benefits are stated as design targets and labelled as such throughout. Section VIII draws the boundary explicitly, and the paper claims no figure that was not recorded.

Keywords: data pipeline observability, enterprise data engineering, data reliability, data quality monitoring, pipeline health monitoring, automated alerting, real-time monitoring, ETL observability, DataOps, enterprise insurance systems, operational dashboards, incident detection, data governance.

I. Introduction

A scheduled job that reports success has told you one thing: it ran to completion without raising an error. It has not told you that the file it read was the one it was supposed to read, that every record arrived, that the figures are current, or that the column a downstream report depends on still exists under the same name.

In large enterprise data platforms this gap is where incidents live. Pipeline failures are frequently silent. Jobs run late. Upstream systems change a schema without notice. A load completes against a partial file. Duplicate records inflate a count that six reports consume. None of these raises an error in a scheduler, because from the scheduler's point of view nothing went wrong. Figure 1 sets the two views against each other.

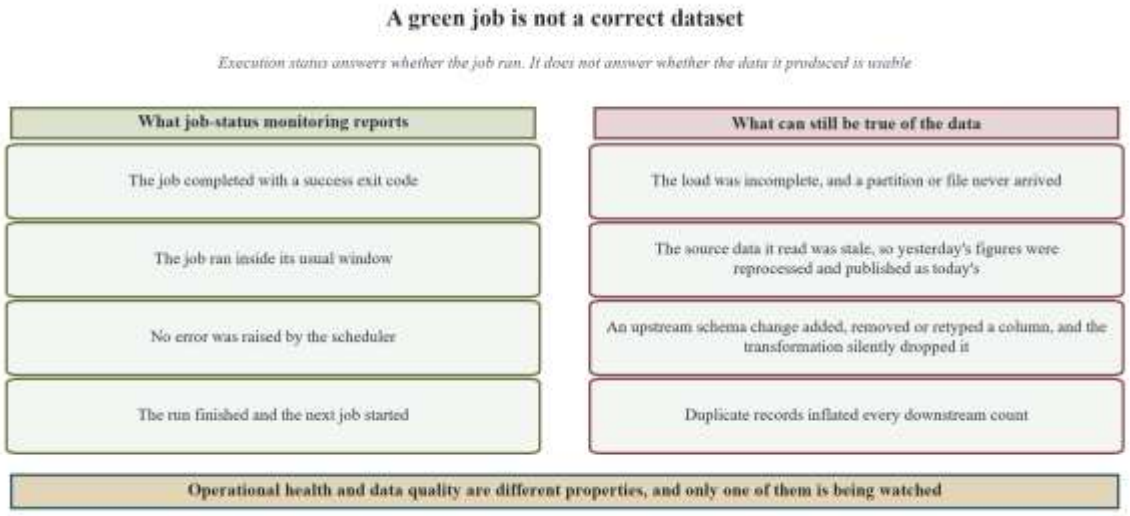


Figure 1. Execution status and data correctness are different properties. Monitoring the first does not establish the second.

A representative incident from the production environment behind this work makes the cost concrete. A scheduled pipeline failed after an upstream dependency changed. Because monitoring relied on manual checks and basic job-status notification, the failure was not detected for several hours. Stale data reached downstream reporting and analytics teams, who consumed it before anyone knew. Recovery required manual investigation, reruns, and a round of data validation. No customer data was lost, but multiple engineering and business teams were delayed, several hours of productivity went into diagnosis, and reporting timelines slipped. The failure was not exotic. It was ordinary, and it was invisible for exactly as long as it took a person to look.

A. Why basic job-status checking does not scale

Traditional monitoring in enterprise environments centres on scheduler state, success, failure or timeout, read from execution logs and infrastructure tooling. At small scale a person compensates for what that misses. At enterprise scale they cannot, for three reasons.

The first is volume. When an estate runs in the tens of thousands of scheduled jobs, reading status by hand is not a check, it is a sampling exercise whose coverage depends on how much time an engineer had that morning.

The second is that the interesting failures do not announce themselves. A job that succeeds while producing incomplete data looks identical, in a scheduler, to a job that succeeds while producing correct data. The distinction is only visible in the data, and nothing in the operational layer is looking there.

The third is fragmentation. Status lives in a scheduler, data lives in a warehouse, resource metrics live in a cloud monitoring service, and the record of who owns a pipeline frequently lives in someone's memory. An engineer answering "is this pipeline healthy" assembles that answer by hand from several systems, which is slow when nothing is wrong and slowest exactly when something is.

B. What this paper contributes

1. A unified signal set (Section IV) that treats operational health and data quality as two families of equal standing, with the measurement and source stated for each signal, rather than treating quality validation as an optional add-on to execution monitoring.
2. An end-to-end monitoring path (Section V) in which validation runs at ingestion, before processing begins, and again after each processing stage, rather than only at the end of a pipeline.
3. A staged quality assessment with a three-way outcome (Section VI), where checks return pass, warning or fail, and only failures raise alerts while warnings accumulate for trend analysis.
4. An alerting policy built around ownership metadata (Section VII), with severity classification, deduplication, correlation, rate limiting and retry awareness, so that the volume of notification does not scale with the volume of telemetry.

5. An explicit evidence boundary (Section VIII) separating four measured production outcomes from the design targets of the generalised framework, including a statement of the quantities that were not recorded and are therefore not claimed.

Section II positions the work. Section III describes the evidence base. Sections IV to VII give the framework. Sections VIII to X give the evaluation, limits and future work.

II. Related work

Four lines of work bear on this framework.

Failure and anomaly detection from logs and telemetry. A substantial literature detects failures by modelling normal behaviour and flagging deviation. DeepLog learns normal log sequences and identifies anomalies against them [1], and earlier empirical work compared supervised and unsupervised log-anomaly methods and released reusable tooling [2]. More recent work fuses logs, metrics and traces for robustness in microservice settings [9]. These methods detect well. They also require training data and ongoing model maintenance, and their outputs can be difficult to act on at the moment an engineer is paged. The framework here uses explicit signals and stated thresholds, so that a flagged pipeline always arrives with a reason attached, and places learned detection in future work (Section X) rather than in the operating design.

Root-cause localization. Trace-based localization for microservice systems is well developed [6], [8], as is causal-inference-based fine-grained localization [7]. This work does not contribute to that line. Where a failure cascades across dependent pipelines, the framework groups and routes rather than diagnoses, and the diagnosis remains human. The author's prior work addresses dependency-aware impact analysis and tiered recovery for the same class of estate [20]; this paper deliberately does not re-develop that material and concentrates instead on the quality and freshness signals that work did not cover.

Data quality and validation. Data quality has a long conceptual literature, including the foundational account of quality as fitness for the data consumer's purpose rather than as an intrinsic property [14], and a survey of assessment and improvement methodologies [15]. On the systems side, declarative unit-test-style assertions over large datasets are well described [4], continuous validation has been integrated into production machine learning pipelines [3], [5], and the practical difficulty of error detection has been assessed empirically [17]. A survey of measurement and monitoring tooling maps the current landscape [16]. This body of work supplies the checks. What it does not generally do is bind them to operational execution state, so that a quality failure and a runtime failure are visible in the same place and scored together. That binding is this framework's concern.

Observability practice. Industrial survey work on tracing and analysis documents the gap between telemetry collected and telemetry that proves actionable [10], and recent stock-takes of monitoring practice describe it as fragmented and short of unified signals [13]. Surveys of failure management note the movement toward autonomous and increasingly model-assisted operations along with its interpretability and data-dependency costs [11]. Established site-reliability practice supplies the service-level vocabulary used here [12]. Provenance and lineage research underpins the metadata layer of Section V-D [18]. Recent work has applied model-driven quality monitoring to enterprise cloud data platforms [19], which is the nearest adjacent approach to this one and differs in relying on learned detection where this framework relies on declared rules.

III. Evidence base

This paper draws on a production deployment and on a framework generalised from it, and the two are not the same thing. Stating which is which is a precondition for the rest of the paper being readable.

What ran in production, between July 2025 and July 2026, in an enterprise insurance data environment, in the author's capacity as a tech lead data engineer: a centralised operational dashboard covering approximately 12,500 scheduled data jobs, refreshing at approximately one-minute intervals, with automated status tracking, failure visibility and operational reporting. A second dashboard monitored engineering workflow compliance and operational metrics. Both were built, deployed and used daily by multiple engineering teams.

What is generalised: the standardised data quality monitoring of Section VI, the health scoring of Section V-E, the alerting policy of Section VII, and the enterprise-wide architecture of Section V as a whole. These extend the deployed components into a complete design. They are described as a specification, not as a running system, and Section VIII separates their expected benefits from the production outcomes.

What exists as artifacts: the production operational dashboard, the live refreshing dashboard, historical execution trend reports, monthly operational summaries, the engineering work-tracking dashboard, workflow

execution logs, scheduler execution history, and documentation of the monitoring processes. No formal research dataset, controlled benchmark or academic evaluation exists, and no reproducible reference implementation accompanies this paper.

The environment is described by function throughout. No employer, vendor, customer or individual is named. Platform names appear only as examples of a category, and the framework is designed to be orchestrator-agnostic.

IV. The signal set

The framework monitors two families of signal. Operational signals describe what the platform did. Data quality signals describe what the data is. Figure 2 states the division and the reason for it.

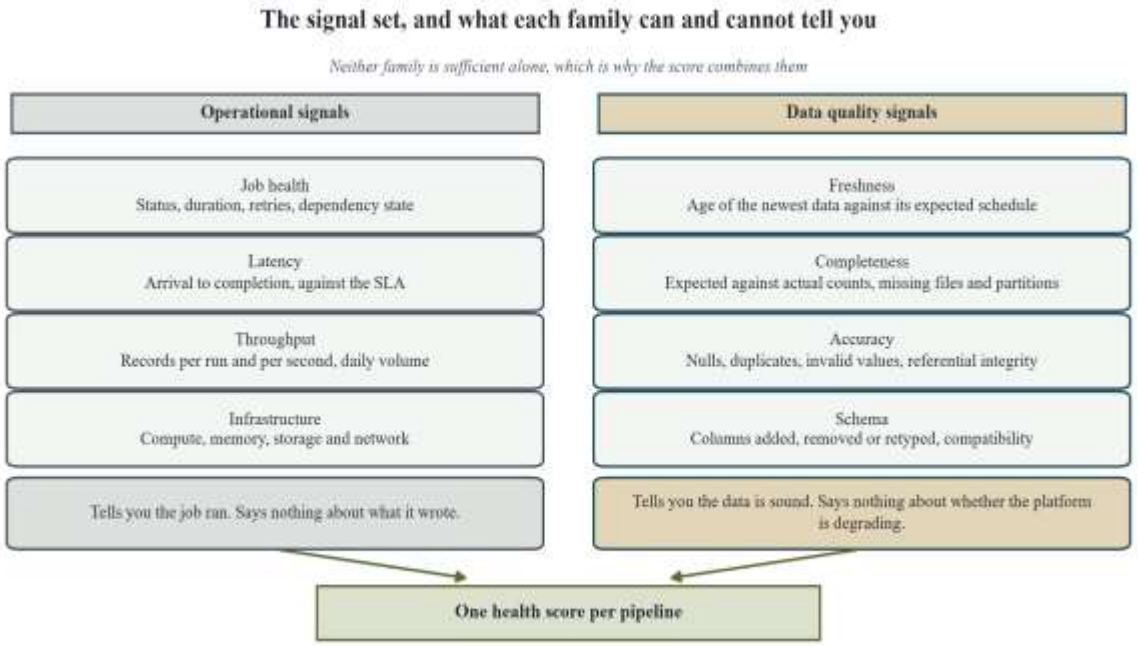


Figure 2. The two signal families. Each is blind to what the other sees, which is why the health score combines them.

Table I gives the signals with the measurement and the source for each.

Table I. Monitored signals, how each is measured, and where it comes from.

Family	Signal	Measured as	Source
Operational	Job health	Status of running, success, failed, cancelled or retry; execution duration; retry count; dependency failures	Scheduler execution logs, orchestration platform
Operational	Pipeline latency	Time from source data arrival to successful completion; SLA compliance; processing delay	Ingestion and completion timestamps, scheduler metadata
Operational	Queue delay	Time a job waits between being due and starting	Scheduler metadata
Operational	Throughput	Records per run, records per second, daily volume	ETL execution logs, processing engine metrics, load statistics
Operational	Infrastructure health	Compute utilisation, memory, storage capacity, network state	Cloud monitoring services, infrastructure logs
Data quality	Freshness	Age of the newest available dataset against its expected refresh schedule	Table and ingestion timestamps, metadata catalog, object metadata

Data quality	Completeness	Expected against actual record counts; missing partitions; missing files; incomplete loads	Target queries, metadata repositories, ingestion validation output
Data quality	Accuracy and integrity	Null percentage, duplicate records, invalid values, referential integrity, business rule results	Automated validation rules, profiling processes
Data quality	Schema	Columns added, removed or retyped; data type changes; compatibility with consumers	Metadata catalogs, schema registries, version-controlled definitions
Both	Alerting and incident metrics	Alert counts by severity, acknowledgement time, resolution time, recurrence	Alerting system, incident tooling, monitoring history

Two properties of this set matter more than its length.
Baselines rather than fixed ceilings. Execution duration is judged against each job's own historical distribution, not one global threshold. A global ceiling either alerts every night on jobs that are naturally long or fails to catch short jobs that have stalled.
Freshness is a data signal, not an operational one. A pipeline can complete on time, within its usual runtime, having read a source that was never updated. Only the freshness check distinguishes that case from a correct run, and no operational signal can.

V. Framework architecture

The framework watches a pipeline from the arrival of source data to the delivery of an alert to the team that owns it. Figure 3 gives the path.

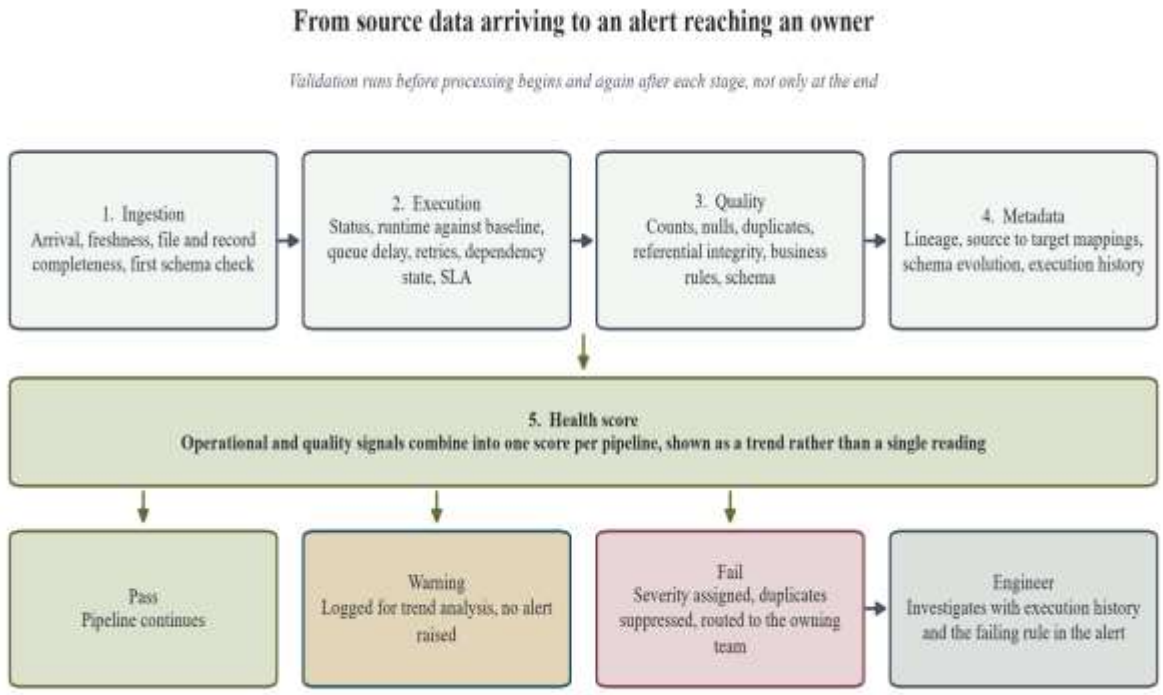


Figure 3. The monitored path. Validation occurs at ingestion and after each processing stage, and the three-way outcome determines whether anything is raised.

A. Ingestion monitoring

Monitoring begins before processing does. As data arrives from source systems, whether transactional databases, interfaces, message queues or batch file transfers, the framework verifies that ingestion succeeded,

records arrival time and compares it against the expected schedule, checks file or message completeness against expected counts, and performs a first schema validation.

Placing these checks before processing is the single most useful structural decision in the design, because a pipeline that is stopped at ingestion has not yet written anything downstream. A pipeline that is caught at the end has.

B. Execution monitoring

As data moves through processing, the framework collects status, execution duration, queue wait, resource utilisation, dependency state and SLA position. The emphasis is on patterns rather than binary outcomes: unusually long runtimes, repeated retries, and upstream dependency failures frequently precede a larger incident, and treating them as signals rather than as noise is what makes detection proactive rather than reactive.

C. Staged quality assessment

After each major processing stage, automated validation rules evaluate the data. Section VI gives the checks. Each returns pass, warning or fail together with a quantitative contribution to the pipeline's score, and the pipeline continues, logs, or halts according to the configured threshold.

D. Metadata and lineage

The framework maintains metadata describing pipeline behaviour: lineage, source-to-target mappings, dataset versions, schema evolution, dependencies and historical execution metrics. This is the layer that makes impact analysis possible. When an upstream system changes, the question that needs answering in minutes is which downstream datasets and reports are affected, and that question is answerable only against recorded lineage [18]. It is also what shortens investigation, since an engineer looking at a failure can see what the pipeline consumed and what consumes it without reconstructing either.

E. Health scoring

Operational and quality metrics are aggregated into one score per pipeline, combining execution success, SLA compliance, latency, completeness, freshness, validation results and infrastructure availability. The score is presented as a trend rather than a single reading, because a pipeline degrading slowly over three weeks is a different operational problem from one that failed this morning, and a point-in-time indicator cannot distinguish them.

The score is a triage aid, not a verdict. Its purpose is to order attention across a large estate, and Section IX states what it cannot do.

VI. Data quality validation

Thirteen checks run against the data at defined stages. Table II groups them by what they establish and states the failure each is there to catch.

Table II. Data quality checks and the failure each is intended to catch.

Group	Check	Failure it catches
Arrival	Missing file detection	A source delivered nothing and the pipeline processed an empty or previous file
Arrival	Missing partition detection	A partition never landed, so a period is absent from an otherwise complete table
Arrival	Data freshness validation	The source was not updated, so current figures are a copy of the previous period
Volume	Record count validation	A partial load completed successfully with a fraction of the expected rows
Volume	Completeness verification	Required fields or segments are absent across an otherwise valid load
Volume	Source-to-target consistency	Counts agree at neither end of a transformation, indicating rows dropped in transit
Content	Null-value percentage	A field that should be populated is systematically empty after a transformation change
Content	Duplicate record detection	Reprocessing or a retry inserted the same records twice, inflating every downstream count
Content	Invalid value and data type validation	Values outside the permitted domain or of the wrong type entered a typed target

Structure	Schema validation	An upstream system added, removed or renamed a column
Structure	Required column validation	A column a downstream consumer depends on is no longer present
Relational	Referential integrity	A record references a key that does not exist in the related dataset
Business	Business rule validation	The data is technically valid and violates a domain rule that defines correctness

The three-way outcome matters as much as the checks. A binary pass or fail forces every deviation into either silence or an alert, and the practical consequence is that thresholds get loosened until the alerts stop. A warning state gives minor deviations somewhere to go: they are recorded, they contribute to the trend, and they do not page anyone. Only a fail raises an alert.

The last row of Table II is the one with the weakest automated coverage, and Section IX returns to it. A business rule can only catch what it was written to catch, and the failures that matter most in an insurance context are frequently the ones where data is structurally perfect and semantically wrong.

VII. Alerting

Alerting is where observability frameworks most often fail in practice, because the volume of signal available scales with the estate while human attention does not. The policy below is deliberately brief; dependency-aware impact analysis and the triage of cascading failures are treated in the author's prior work [20] and are not re-developed here.

Severity is assigned by business consequence, not by component. Critical covers pipeline failure, missing datasets, major quality failures and SLA breaches affecting business operations. High covers performance degradation, repeated retries and upstream dependency failures likely to affect downstream systems shortly. Medium covers moderate latency increases, isolated validation failures and resource warnings. Low covers informational events and successful recoveries.

Routing uses ownership metadata rather than a distribution list. Every pipeline carries an owning engineering team, application, business domain and support group in the observability layer. Infrastructure alerts route to platform teams; quality failures route to the data engineering or governance owners of the affected dataset. The routing is a property of the pipeline record, so it stays correct when teams change and does not depend on a separately maintained list.

Noise control is applied before anything is sent. Alerts arising from one root issue are deduplicated into a single incident. Related failures across dependent pipelines are correlated and grouped. Repeated alerts for an unresolved issue are rate limited. Planned maintenance windows suppress alerting. Transient failures do not alert until automatic retries are exhausted.

Every alert carries the context needed to begin work: the pipeline and execution identifier, the time and stage of failure, the error or the specific validation rule that triggered, recent execution history, upstream dependency state, and links to the relevant logs and dashboards. An alert that states only that something failed transfers the diagnostic burden back to the recipient, which is the behaviour the framework exists to remove.

No measurement of alert volume, false-positive rate or acknowledgement time was recorded in the production deployment. This section is a specification, and Section VIII says so.

VIII. Evaluation

The deployment produced four measured outcomes. The framework as a whole has not been evaluated. Figure 4 draws the line.

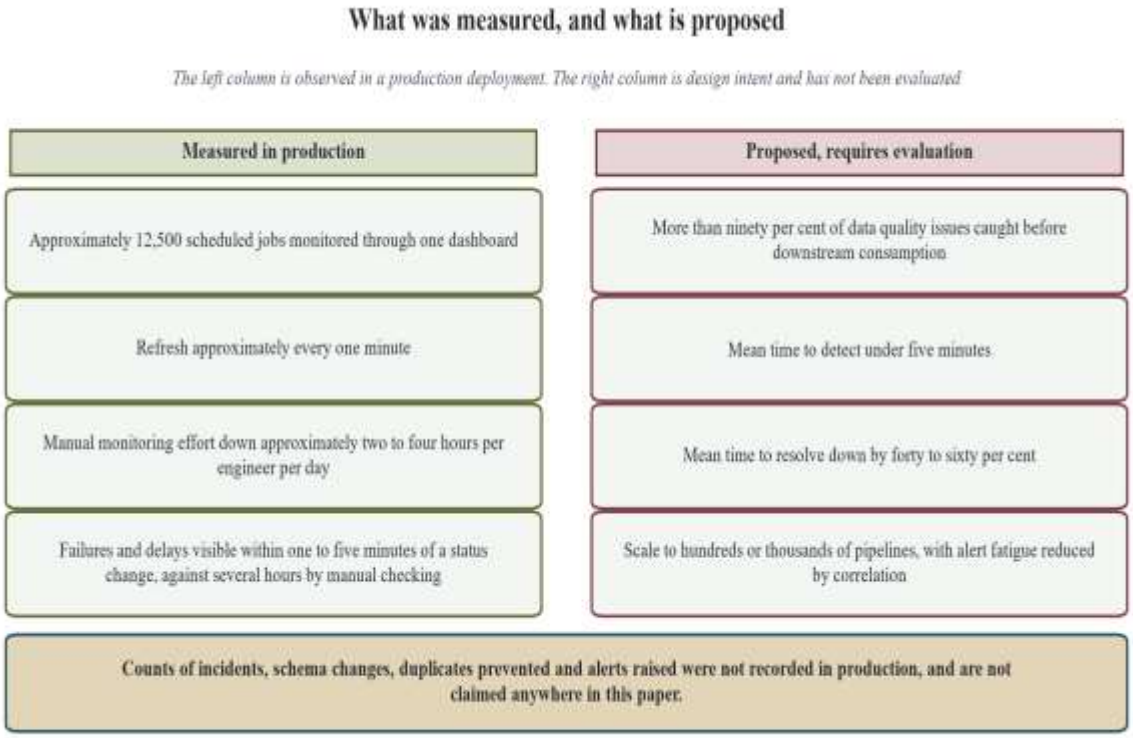


Figure 4. The evidence boundary. Left, observed in production. Right, design targets with no evaluation behind them.

A. Measured in production

Coverage. Approximately 12,500 scheduled data jobs were monitored through a single centralised operational dashboard.

Refresh. Dashboard metrics refreshed at approximately one-minute intervals, giving near real-time operational visibility.

Manual effort. Manual job monitoring effort fell by approximately two to four hours per engineer per day, as repetitive status checks across multiple systems were replaced by centralised dashboards and health indicators.

Detection latency. Pipeline failures and delayed jobs became visible within approximately one to five minutes of a status change. Under the previous manual regime, detection depended on when an engineer next performed a routine check and could take several hours.

One qualitative outcome is worth recording alongside these. Centralised monitoring improved operational awareness across multiple engineering teams by giving them one view of execution status and historical trend, in place of several partial views assembled separately.

B. Proposed, requiring evaluation

The following are the expected benefits of implementing the complete framework. None has been measured. They are stated as design targets so that a future deployment has something to test against.

- Detection of more than ninety per cent of data quality issues before incorrect data reaches downstream reporting and analytics.
- Mean time to detect reduced to under five minutes through continuous health monitoring and automated alerting.
- Mean time to resolve reduced by forty to sixty per cent through contextual alerting and integrated diagnostics.
- Improved pipeline reliability and SLA compliance through proactive detection of latency, dependency failure, schema change and quality violation.
- Monitoring that scales to hundreds or thousands of pipelines on standardised health metrics and metadata-driven ownership.

- Reduced alert fatigue through prioritisation, deduplication and correlation.

C. What was not recorded, and is therefore not claimed

The production deployment did not instrument the following, and no figure for any of them appears in this paper: the number of incidents detected per month, the number of schema changes detected, the number of duplicate data issues prevented, the number of datasets stopped before reaching reporting, the number of alerts generated, the false-positive rate, and the exact uptime improvement.

These are the first things a future evaluation should capture, and their absence is the main reason Section VIII-B is a list of targets rather than results.

IX. Limitations

Semantically wrong data passes every technical check. The framework detects operational failures, latency, schema change, missing data and a wide class of quality violations. It does not detect data that is structurally valid and substantively incorrect, which includes incorrect transformation logic and business rules that are themselves wrongly specified. A report built on a mis-specified rule will be wrong with every indicator green. This is the most important limit in the design and no amount of additional technical monitoring closes it.

Observability has a cost. Collecting execution metrics, retaining logs, running continuous validation and maintaining metadata consume compute, storage and network capacity. Depth of monitoring trades against infrastructure cost, and the trade is managed through sampling rates, retention policies and validation frequency. Monitoring everything at full fidelity is not a neutral choice.

Thresholds require continuous tuning. Fixed thresholds alert unnecessarily during abnormal but expected load and miss gradual degradation. As pipelines and business requirements change, quality rules and alert policies must change with them, and maintaining the monitoring becomes an operational responsibility of its own.

The monitoring layer must not be a single point of failure. If it becomes unavailable, teams fall back to scheduler logs, native platform monitoring and manual validation. The framework is designed to improve visibility, not to become the only route to it.

The evidence is one environment and one deployment. The measured outcomes come from a single enterprise insurance estate. The framework is offered as transferable and its transferability is untested. No controlled comparison against the previous manual regime was run, and no reproducible artifact accompanies this paper. Taken together, these place the framework as a decision-support and operational intelligence layer rather than a replacement for governance, testing and engineering judgement.

X. Future work

The clearest next step is measurement rather than capability. Instrumenting the counts listed in Section VIII-C across a defined period would convert the design targets of Section VIII-B into results, and that work needs no new technology.

Beyond it, four extensions follow naturally from the limits above. Learned anomaly detection would identify unusual pipeline behaviour without depending solely on static thresholds, addressing the tuning burden directly, and predictive failure detection over historical execution patterns would extend that from detection to anticipation. Automated root-cause analysis across dependent pipelines would reduce the diagnostic work that currently remains human. Dynamic alert prioritisation by business impact would let severity reflect consequence at the moment of failure rather than at the moment a rule was written. Automated business-rule validation and trust scoring would begin on the semantic gap of Section IX, which is the one limit that matters most and the one furthest from a solution.

Each of these moves the framework toward the model-assisted operations the surveyed literature describes [11], and each imports the interpretability and maintenance costs that literature also documents. The framework described here is rule-based because an operations team can read, audit and change a rule. That property should be preserved deliberately as learned components are introduced, rather than traded away by default.

XI. Conclusion

The monitoring most enterprise data platforms run answers a question narrower than the one their users are asking. It reports whether a job executed. The business depends on whether the data is complete, current and correct, and those are different properties that require different signals.

This paper has described a framework that treats them as equally weighted families, validates at ingestion before anything is written downstream and again after each stage, reduces both families to a single health score shown as a trend, and routes failures by severity to the team that owns the pipeline, with deduplication and correlation applied before anything is sent.

The production component of this work is narrower than the framework and is reported as such: a centralised dashboard over approximately 12,500 scheduled jobs, refreshing approximately every minute, which removed approximately two to four hours of manual monitoring per engineer per day and brought failure visibility from several hours down to one to five minutes. The remainder is a design, its expected benefits are targets rather than findings, and the quantities that were never recorded are named rather than estimated.

The limit worth restating is the semantic one. Everything here establishes that a pipeline behaved correctly and that its data is structurally sound. Neither establishes that the data is right. Closing that gap needs business-rule validation and trust scoring that do not yet exist in this framework, and until they do, observability reduces the number of ways a platform can fail quietly without reducing it to zero.

References

[1] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly detection and diagnosis from system logs through deep learning," in Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS), 2017, pp. 1285-1298, doi: 10.1145/3133956.3134015.

[2] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience report: System log analysis for anomaly detection," in Proc. IEEE 27th Int. Symp. Software Reliability Engineering (ISSRE), 2016, pp. 207-218.

[3] N. Polyzotis, M. Zinkevich, S. Roy, E. Breck, and S. Whang, "Data validation for machine learning," in Proc. Machine Learning and Systems (MLSys), 2019.

[4] S. Schelter, D. Lange, P. Schmidt, M. Celikel, F. Biessmann, and A. Grafberger, "Automating large-scale data quality verification," Proc. VLDB Endowment, vol. 11, no. 12, pp. 1781-1794, 2018, doi: 10.14778/3229863.3229867.

[5] E. Caveness, P. Suganthan G. C., Z. Peng, N. Polyzotis, S. Roy, and M. Zinkevich, "TensorFlow Data Validation: Data analysis and validation in continuous ML pipelines," in Proc. ACM SIGMOD Int. Conf. Management of Data, 2020, pp. 2793-2796, doi: 10.1145/3318464.3384707.

[6] Z. Li et al., "Practical root cause localization for microservice systems via trace analysis," in Proc. IEEE/ACM Int. Symp. Quality of Service (IWQoS), 2021.

[7] R. Xin, P. Chen, and Z. Zhao, "CausalRCA: Causal inference based precise fine-grained root cause localization for microservice applications," J. Systems and Software, vol. 203, 2023, Art. no. 111724, doi: 10.1016/j.jss.2023.111724.

[8] C. Zhang et al., "Trace-based multi-dimensional root cause localization of performance issues in microservice systems," in Proc. IEEE/ACM 46th Int. Conf. Software Engineering (ICSE), 2024.

[9] C. Zhao et al., "Robust multimodal failure detection for microservice systems," in Proc. 29th ACM SIGKDD Conf. Knowledge Discovery and Data Mining (KDD), 2023.

[10] B. Li, X. Peng, Q. Xiang, H. Wang, T. Xie, J. Sun, and X. Liu, "Enjoy your observability: An industrial survey of microservice tracing and analysis," Empirical Software Engineering, vol. 27, no. 1, 2022, doi: 10.1007/s10664-021-10063-9.

[11] L. Zhang et al., "A survey of AIOps for failure management in the era of large language models," arXiv:2406.11213, 2024.

[12] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, Eds., Site Reliability Engineering: How Google Runs Production Systems. Sebastopol, CA, USA: O'Reilly Media, 2016.

[13] "Monitoring and observability of machine learning systems: Current practices and gaps," arXiv:2510.24142, 2025.

[14] R. Y. Wang and D. M. Strong, "Beyond accuracy: What data quality means to data consumers," J. Management Information Systems, vol. 12, no. 4, pp. 5-33, 1996, doi: 10.1080/07421222.1996.11518099.

[15] C. Batini, C. Cappiello, C. Francalanci, and A. Maurino, "Methodologies for data quality assessment and improvement," ACM Computing Surveys, vol. 41, no. 3, pp. 1-52, 2009, doi: 10.1145/1541880.1541883.

[16] L. Ehrlinger and W. Wöß, "A survey of data quality measurement and monitoring tools," Frontiers in Big Data, vol. 5, 2022, doi: 10.3389/fdata.2022.850611.

[17] Z. Abedjan, X. Chu, D. Deng, R. C. Fernandez, I. F. Ilyas, M. Ouzzani, P. Papotti, M. Stonebraker, and N. Tang, "Detecting data errors: Where are we and what needs to be done?," Proc. VLDB Endowment, vol. 9, no. 12, pp. 993-1004, 2016, doi: 10.14778/2994509.2994518.
[18] M. Herschel, R. Diestelkämper, and H. Ben Lahmar, "A survey on provenance: What for? What form? What from?," The VLDB Journal, vol. 26, no. 6, pp. 881-906, 2017, doi: 10.1007/s00778-017-0486-1.
[19] S. Khan, "AI-driven data quality monitoring for enterprise cloud data platforms," in Proc. 2026 IEEE Cloud Summit, 2026, pp. 252-257, doi: 10.1109/cloudsummit68932.2026.00050.
[20] G. R. Pappula, "A practical framework for monitoring large-scale data pipelines and recovering from failures," 2026, manuscript.