International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(18s) 2026 pp. 194-203
https://doi.org/10.70917/ijcisim-2026-4861
Article

A Practical Framework for Monitoring Large-Scale Data Pipelines and Recovering from Failures

Gowtham Reddy Pappula

Lead Data Engineer, gowthamresponds@gmail.com

Abstract: Enterprise data platforms run tens of thousands of scheduled jobs every day to feed operational, financial, and regulatory reporting. When these jobs fail, the failure is usually found late, after a business user notices a missing report. Worse, one upstream failure floods the operations team with hundreds of independent downstream alerts. Most monitoring watches infrastructure status rather than business impact, so teams cannot tell which failure to fix first. This paper presents a practical framework for monitoring large-scale data pipelines and recovering from their failures. The framework has six layers: job metadata collection, data standardization, a rule-based health evaluation engine, dependency-aware business-impact analysis, severity-based alert routing, and tiered recovery that retries transient failures automatically and escalates data and business-rule failures to people. The design is deliberately rule-based rather than model-driven, so that every decision is transparent and an operations team can maintain it. The framework was deployed on a de-identified production estate of more than 12,000 scheduled jobs processing several terabytes per day. In production it cut detection time from about 30 minutes to under 3, reduced mean time to recovery by about 55 percent, and lowered duplicate alerts by about 65 percent while raising service-level-agreement compliance from about 92 percent to over 98 percent. A reproducible reference implementation on a synthetic estate of the same size collapses 1,031 raw job alerts into 299 root-cause incidents, a 71 percent reduction, corroborating the production result. The framework is scheduler-agnostic and transfers to Airflow, Control-M, Azure Data Factory, and AWS Step Functions.

Keywords: Batch processing, data pipelines, dependency analysis, ETL, fault recovery, monitoring, observability, operational reliability, root cause analysis, scheduling

1. INTRODUCTION

A modern enterprise data platform is a large graph of scheduled work. Extract-transform-load jobs, reporting jobs, and data-quality jobs run on fixed schedules that range from every few minutes to once a month, and each job usually depends on several others finishing first. In the de-identified environment this paper is based on, that graph held more than 12,000 jobs organized into over 900 workflow chains, moving several terabytes of data a day into more than 20 downstream business applications used by around 15 teams. Scale figures throughout are approximate and rounded, and no proprietary data is disclosed.

When jobs run at this scale, the hard problem is not writing the jobs. It is knowing, quickly and reliably, when something has gone wrong and what to do about it. For a long time the answer was reactive. Failures were caught by someone scanning the scheduler console in the morning, by an email that no one read in time, or by a business user reporting that a report was missing. By the time anyone looked, the failure was often hours old and the recovery window was already tight.

The deeper issue is triage. The major challenge was not simply detecting failures but identifying which failures required immediate intervention. A single upstream job that fails does not fail alone. Every job downstream of it never runs, and a naive monitor raises a separate alert for each one. One broken job at the top of a chain can turn into hundreds of alerts, and the one alert that matters is buried among them. Industry surveys of observability practice

describe exactly this gap between the volume of telemetry collected and the small amount of it that is actually actionable [10], and recent stock-takes of monitoring practice find it fragmented and short on unified, business-aware signals [15]. Most scheduler monitoring reports infrastructure status, a job ran or it did not, without any sense of what the failure costs the business.

This paper describes a framework built to close that gap in production. Its contributions are:

1. A six-layer framework that unifies detection, dependency-aware root-cause grouping, business-impact prioritization, severity-based routing, and tiered recovery for large-scale scheduled pipelines, rather than treating any one of those as a standalone tool.

2. A dependency-aware business-impact analysis that collapses an alert storm to its single true root cause and orders response by downstream and regulatory impact instead of raw job status.

3. A deliberately rule-based design that stays transparent and maintainable for an operations team, and a discussion of why that choice was made against the current trend toward model-driven operations.

4. An evaluation combining production results from a deployment of more than 12,000 jobs with a reproducible reference implementation that reproduces the direction and magnitude of those results on a synthetic estate of the same size.

The rest of the paper is organized as follows. Section II reviews related work in failure detection, root-cause analysis, and data-quality validation. Section III describes the six layers of the framework. Section IV presents the evaluation, both the production outcomes and the reference implementation. Section V discusses why the approach works, how it compares to model-driven alternatives, and its limitations. Section VI concludes.

2. RELATED WORK

Work on failure management in data and software systems falls into three lines that this framework draws on and, in one respect, departs from.

Failure and anomaly detection. A large body of work detects failures from logs and metrics. Classic empirical studies compared supervised and unsupervised log-anomaly methods and released reusable tooling [2], and deep-learning approaches such as DeepLog model normal log sequences and flag deviations [1]. More recent work fuses logs, metrics, and traces for more robust detection in microservice systems [9]. These methods detect well, but they are built around learned models that need training data and ongoing maintenance, and their outputs are hard for an operations team to interpret at three in the morning. The framework here uses multiple explicit signals, status, runtime against a historical baseline, missed schedules, dependency completion, and data-quality checks, rather than a learned model, so that a flagged job always comes with a plain reason.

Root-cause analysis. A second line localizes the root cause of an incident. Much of it targets online microservice request traces, using trace analysis [6], causal inference [7], and multi-dimensional trace localization [8] to find the service responsible for a performance problem. This work is strong, but its unit of analysis is a service call inside a live request trace, not a scheduled job inside a batch dependency chain. Scheduled estates fail differently: an upstream job fails and its downstream dependents simply never start. The framework here works on the job-dependency graph directly, which is both simpler and a better fit for batch ecosystems. doi: 10.1145/3597503.3639088.

Data-quality validation. A third line treats data itself as something to be tested. Declarative, large-scale data-quality verification with Deequ defines unit tests for data on Spark [4], and continuous data validation in production machine-learning pipelines catches schema drift and bad inputs before they propagate [3], [5]. Regulated settings add auditability and governance requirements to those checks [14]. The health evaluation layer of this framework adopts this idea for its post-load data-quality signals, checking record counts, duplicates, null thresholds, and schema consistency.

Across all three lines, the AIOps literature has moved toward autonomous, increasingly model-driven and language-model-assisted failure management, and recent surveys and evaluation frameworks map that landscape and note its interpretability and data-dependency challenges [11], [12]. Topology-aware alert correlation, which groups alerts using service-dependency graphs, is already established in observability tooling, so the grouping idea itself is not new. What is different here is the setting and the design: dependency-aware grouping applied to scheduled batch job estates rather than online service topologies, with a rule-based, explainable core and prioritization by business and regulatory impact rather than by service topology alone. Site reliability engineering practice, meanwhile, has long argued that teams should alert on symptoms and business impact rather than raw causes and should automate away repetitive recovery toil [13]. The framework in this paper is closer in spirit to the second idea than the first: it is a transparent, deployable, rule-based system that spans the whole detection-to-recovery lifecycle for scheduled pipelines, which no single one of the works above does.

3. FRAMEWORK DESIGN

The framework is organized as six layers. Figure 1 shows the layers and how a job execution flows through them; Figure 2 shows the concrete execution path in the production deployment. The layers are described in order below.

Figure 1 Six-layer monitoring and recovery framework

Figure 1. Six-layer framework architecture: (1) Job Metadata Collection, (2) Data Standardization, (3) Health Evaluation Engine, (4) Business Impact Analysis, (5) Intelligent Alert Routing, (6) Recovery. Arrows show a job execution moving from raw scheduler event to routed, prioritized incident and recovery action.

Figure 2 Production execution flow

Figure 2. Production execution flow: Autosys → scheduler logs → SQL Server → Python processing → business-rules engine → Power BI dashboard → Microsoft Teams / email alerts.

A. Layer 1: Job Metadata Collection

The first layer continuously collects execution metadata from every scheduled job: status, start and end time, runtime, exit code, the job's declared upstream dependencies, and the owning application and team. This metadata, not the data payload, is what the framework reasons over, which keeps it lightweight enough to refresh every minute across the whole estate.

B. Layer 2: Data Standardization

Different tools report execution differently. The second layer normalizes scheduler logs from every source into one common record so the rest of the framework does not care whether a job ran under Autosys, Informatica, or a Python job. This standardized record is the single schema every downstream layer reads.

C. Layer 3: Health Evaluation Engine

The third layer decides whether a job is healthy using explicit rules over the standardized record:

- Status. A non-success exit code is a failure.

- Runtime. A job that runs far longer than its own historical baseline is a runtime anomaly. Judging runtime against each job's baseline, rather than one static ceiling for all jobs, matters: some jobs are naturally long, and a global threshold either alerts on them every night or misses short jobs that have stalled. Historical baselines proved more reliable than static thresholds in practice.

- Schedule. A job that has not started well past its scheduled slot is flagged as missed, which catches the common overnight case where a late source file stops a job from ever kicking off.

- Dependency. A job whose upstream never completed is treated as blocked rather than independently broken.

- Data quality. After a load, the engine checks record counts, duplicates, null thresholds, and schema consistency, following the declarative data-quality line of work [3], [4], [5].

To keep noise down, planned maintenance windows are ignored, and a threshold has to be breached repeatedly before it escalates, so a single transient blip does not page anyone.

D. Layer 4: Business Impact Analysis

The fourth layer is the core of the framework. It answers the triage question the operations team actually cares about: of everything that is unhealthy right now, what matters most, and what is the one thing to fix.

It works on the dependency graph. A job that failed but whose own upstream also failed is a downstream victim, not a root cause. The layer keeps only failures with no failing upstream as roots, then attaches every reachable downstream failure to its root as a suppressed cascade. This is the step that turns hundreds of child alerts into a single incident. Listing 1 shows the grouping function from the reference implementation.

Listing 1. Dependency-graph root-cause grouping (Python).

```
def group_to_root_cause(unhealthy, all_records):
    failing_ids = {r.job_id for r in unhealthy}
    downstream_of = {}
    for r in all_records:
        for up in r.upstream:
            downstream_of.setdefault(up, []).append(r.job_id)
    roots = [r for r in unhealthy if not (set(r.upstream) & failing_ids)]
    incidents = []
    for root in roots:
        cascade, frontier = set(), [root.job_id]
        while frontier:
            current = frontier.pop()
            for child in downstream_of.get(current, []):
                if child in failing_ids and child not in cascade and child != root.job_id:
                    cascade.add(child)
                    frontier.append(child)
        incidents.append(Incident(root_job=root.job_id, cascade_jobs=sorted(cascade)))
    return incidents
```

Each root incident is then scored for business impact. The score combines the criticality of the affected application with how much work the failure blocks downstream, so that a failed regulatory-reporting job with many dependents outranks an isolated low-impact job. Listing 2 shows the scoring parameters. The weights are ordered to match what an operations team escalates first, which is consistent with the site-reliability principle of alerting on business impact rather than raw cause [13]. Failures are prioritized using downstream dependency count, business criticality, SLA deadlines, regulatory impact, affected reports, and historical incident frequency.

Listing 2. Business-impact score parameterization (Python).

```
CRITICALITY_WEIGHT = {
    "regulatory": 100,   # regulatory reporting -- escalate first
    "executive": 60,     # executive / financial dashboards
    "standard": 20,
    "isolated": 5,
}
FAN_OUT_WEIGHT = 3       # each directly-blocked downstream job adds to the score
def impact_score(incident, root):
    blocked = len(incident.cascade_jobs) + root.fan_out
    return CRITICALITY_WEIGHT[incident.criticality] + FAN_OUT_WEIGHT * blocked
```

E. Layer 5: Intelligent Alert Routing

The fifth layer turns scored incidents into notifications. Incidents are placed in four severity tiers, Critical, High, Medium, and Low. Critical incidents, those affecting regulatory reporting or executive dashboards, notify the on-call owner immediately through Microsoft Teams, email, and optionally SMS or an incident-management system. Routing considers ownership, application, SLA, support rotation, and time zone, so people are paged only for what is theirs. Duplicate notifications are suppressed until an incident's status changes.

F. Layer 6: Recovery

The sixth layer handles the failure once it is understood. Recovery is split by failure type. Transient failures, such as a brief database connection loss, a network interruption, cloud-storage latency, or an API timeout, are retried automatically, up to three times at five-minute intervals, and usually clear on their own. Human intervention is reserved for schema changes, corrupted data, and business-rule failures, where an automated retry would be wrong or unsafe. Keeping a person in the loop for anything that touches data correctness was a deliberate choice: for regulated financial data, a wrong automated fix is worse than a two-minute human check.

A real incident shows the layers working together. An Oracle outage caused an ETL failure at 3:05 AM. The framework detected the repeated connection failures within two minutes, retried the job three times at five-minute intervals, resumed execution when the database recovered, restarted the dependent workflows, and completed processing before the business SLA. Before the framework, the same outage would have surfaced as a missing report at 9 AM.

4. EVALUATION

The framework is evaluated two ways: by the operational outcomes measured after it was deployed in production, and by a reproducible reference implementation that exercises the same logic on a synthetic estate of the same scale.

A. Production Deployment

The framework runs on the estate summarized in Table 1. It continuously collects execution metadata from all scheduled jobs and refreshes operational dashboards every minute.

The deployment is de-identified: the organization is not named, and the scale figures in Table 1 and the outcome metrics in Table 3 are approximate and rounded to avoid disclosing proprietary operational detail. The production metrics reported below are operational measurements, not the result of a controlled experiment. Each metric was computed by comparing a multi-month window before the framework was rolled out against a comparable window after it reached steady-state use, using the platform's existing monitoring, incident-management, and SLA-reporting systems. Detection time is the interval between a failure occurring and the operations team being alerted; duplicate-alert counts come from the alerting system; SLA compliance and availability come from the platform's SLA and uptime reports. Because the comparison is before-and-after in a live environment rather than a randomized trial, the numbers should be read as evidence from one deployment.

Table 1. Production environment scale (approximate, de-identified).

Property	**Value**
Scheduled jobs	12,000+
Workflow chains	900+
Data processed daily	several TB
Schedule cadence	every 5 min to monthly
Downstream business applications	>20

Dependent teams	~15
Dashboard refresh interval	1 minute
Schedulers / tools	Autosys, Informatica, Python ETL, SQL Server, Oracle, Snowflake, Hadoop

Table 3 reports the before-and-after operational metrics from the deployment. The two that matter most for the argument of this paper are detection time and duplicate-alert volume: detection fell from about 30 minutes to under 3, and duplicate alerts fell by about 65 percent, because the business-impact layer collapses cascades to their root cause.

Table 3. Production operational metrics, before and after deployment

Metric	**Before**	**After**	**Change**
Mean detection time	~30 min	<3 min	~90% faster
Mean time to recovery (MTTR)	baseline	n/a	~55% lower
Manual monitoring effort	baseline	n/a	~70% lower
Duplicate alerts	baseline	n/a	~65% fewer
SLA compliance	~92%	>98%	+6 pts
Analyst time on monitoring	baseline	n/a	2–4 hrs/day saved
User-reported incidents	baseline	n/a	~45% fewer
Platform availability	baseline	~99.8%	n/a

B. Reference Implementation

To make the central mechanism reproducible, the framework's detection, grouping, scoring, routing, and recovery logic was implemented as a small Python reference and run against a synthetic estate generated at a scale comparable to the deployment (12,500 synthetic jobs across 900 chains), with dependencies wired within each chain, failures split between transient and persistent, runtime anomalies injected, and, critically, blocking propagated so that a failed upstream job leaves its downstream dependents unstarted, exactly the situation that produces alert storms. The implementation uses only the Python standard library and is deterministic under a fixed seed. It is described in the artifact README.

In one representative cycle the synthetic estate produced 203 primary job failures (143 transient, 60 persistent) and 121 runtime anomalies, and the failed upstream jobs left 718 downstream jobs blocked. A naive per-job monitor would raise 1,031 alerts for this cycle, one per unhealthy job. The framework's business-impact layer groups these into 299 root-cause incidents, suppressing 732 alerts, a 71 percent reduction. This is close to the roughly 65 percent duplicate-alert reduction measured in production, and it isolates the mechanism responsible for it. The absolute 71 percent is a property of this synthetic estate's dependency shape and injected failure rate, so it should be read as a demonstration that dependency-aware grouping collapses alert storms, not as an independent measurement of the exact magnitude. Table 2 shows the largest cascades from the run.

Table 2. Largest alert-storm collapses in one synthetic cycle (framework groups each cascade into a single incident). Application labels are illustrative and not from any production system

Root job	**Application**	**Criticality**	**Downstream suppressed**	**Impact score**	**Severity**
job_01043	reinsurance	regulatory	12	154	Critical
job_00379	underwriting	executive	18	129	High

job_02194	policy_admin	regulatory	10	139	Critical
job_03443	underwriting	regulatory	4	118	Critical
job_04011	policy_admin	regulatory	2	112	Critical

Figure 3 Alert volume: raw per-job vs. root-cause grouped

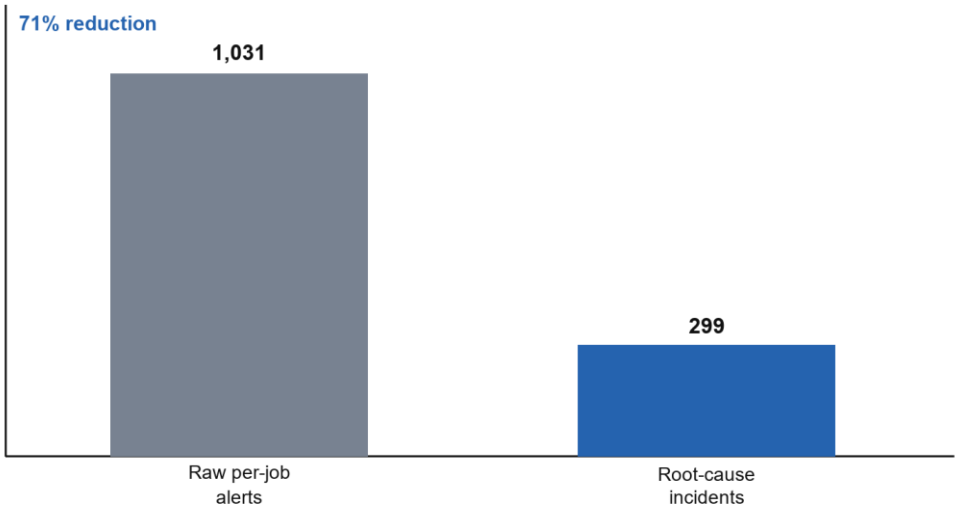

Figure 3. Alert volume in one synthetic cycle: 1,031 raw per-job alerts versus 299 root-cause incidents after dependency-aware grouping (71 percent reduction)

Figure 4 Severity distribution of grouped incidents

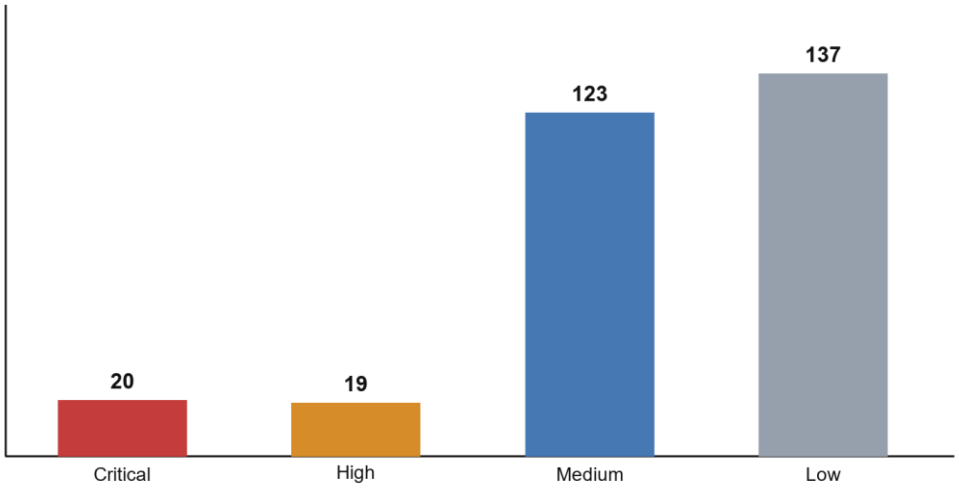

Figure 4. Severity distribution of the 299 grouped incidents: 20 Critical, 19 High, 123 Medium, 137 Low. Prioritization surfaces the 20 Critical incidents ahead of the rest, so the operations team sees the regulatory and executive failures first.

Of the grouped incidents, 134 transient failures were resolved by automatic retry and 57 persistent failures were escalated to their owning team, matching the recovery split the framework is designed to produce.

5. DISCUSSION

Why it works. The gain does not come from detecting more failures. It comes from detecting them sooner, once, and in priority order. Cutting detection to the next one-minute dashboard refresh removes the hours of latency that reactive monitoring carried. Grouping cascades to a root cause removes the noise that made the important alert invisible. Scoring by business impact means the regulatory failure is worked before the isolated one. None of these is individually novel, but no widely reported system combines them for scheduled batch estates, and combining them is what changed the operational numbers.

Comparison to model-driven approaches. The framework is rule-based on purpose. The AIOps literature is moving toward learned and increasingly language-model-assisted failure management [11], [12], and those methods are powerful. But they need training data, they need ongoing model maintenance, and their decisions are hard to explain to the engineer who has to act on them. A rule-based framework is transparent: every flagged job and every priority ordering has a stated reason, and an operations team can read, trust, and change the rules. For a regulated financial-services environment, where an incorrect automated action on sensitive data is costly, that transparency was worth more than the extra detection power a model might add. This mirrors the long-standing site-reliability argument for impact-based alerting and automating only the safe, repetitive parts of recovery [13].

Limitations. The framework has real limits. It remains rule-based, so it does not predict failures before they happen; it reacts quickly rather than forecasting. It assumes the scheduler metadata it reads is reliable, and it inherits any gaps in that metadata. The dependency-aware grouping is only as good as the declared dependencies; undeclared or implicit dependencies are not captured. Maintaining accurate dependency mappings and tuning runtime baselines were the most demanding parts of running it in practice. Finally, the production results come from a single large deployment, so the absolute numbers should be read as evidence from one environment rather than a general benchmark; the reference implementation is included precisely so the mechanism, if not the exact magnitudes, can be reproduced independently.

Threats to validity. The reference implementation uses synthetic metadata, so its numbers demonstrate the grouping and prioritization mechanism rather than an independent field result. The production metrics were measured operationally rather than in a controlled experiment, so they reflect a real but uncontrolled before-and-after comparison.

6. CONCLUSION

This paper presented a practical, rule-based framework for monitoring large-scale scheduled data pipelines and recovering from their failures. By unifying metadata collection, rule-based health evaluation, dependency-aware business-impact analysis, severity-based routing, and tiered recovery, the framework moved a production estate of more than 12,000 jobs from reactive, per-job monitoring to proactive, priority-ordered operations: detection fell from about 30 minutes to under 3, recovery time and manual effort dropped substantially, and duplicate alerts fell by about 65 percent while SLA compliance rose above 98 percent. A reproducible reference implementation collapsed 1,031 raw alerts into 299 root-cause incidents on a synthetic estate of the same size, corroborating the mechanism. The framework's principles, centralized metadata collection, standardized monitoring, dependency-aware analysis, business-impact prioritization, intelligent alerting, automated recovery, and operational visibility, are independent of any single scheduler and transfer to Airflow, Control-M, Azure Data Factory, and AWS Step Functions. Future work will add a predictive layer that uses the historical baselines the framework already maintains to forecast failures before they occur [15], while keeping the transparent, rule-based core that made the system trustworthy to operate.

Author Contributions

G. R. Pappula is the sole author and designed, implemented, and evaluated the framework and the reference implementation, and wrote the manuscript.

Conflict of Interest

The author declares no conflict of interest.

Data Availability

The production metrics reported in Section IV are drawn from a live enterprise deployment and cannot be shared, as the underlying data is proprietary and subject to confidentiality and regulatory constraints. The reference implementation and the synthetic-estate generator that reproduce the mechanism in Section IV-B are self-contained and can be released to reproduce the reported synthetic results.

Funding

This work received no external funding.

Author Biography

Gowtham Reddy Pappula received the M.S. degree in Computer Science from the University of North Texas and the B.Tech. degree in Electronics and Communication Engineering from KL University. He is a Lead Data Engineer with more than four years of experience building and operating large-scale data pipelines across AWS, Azure, and GCP, with a focus on ETL workflows, data governance, and operational reliability in the insurance and banking domains. He holds the AWS Certified Data Engineer, Associate certification.

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 et al., "Enjoy your observability: An industrial survey of microservice tracing and analysis," *Empirical Software Engineering*, vol. 27, no. 1, 2022.
11. L. Zhang et al., "A survey of AIOps for failure management in the era of large language models," *arXiv:2406.11213*, 2024. arXiv:2406.11213.
12. Y. Chen et al., "AIOpsLab: A holistic framework for evaluating AI agents for enabling autonomous clouds," Microsoft Research, Tech. Rep., 2024.
13. 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.
14. "A unified AI system for data quality control and DataOps management in regulated environments," *arXiv:2512.05559*, 2025. arXiv:2512.05559.
15. "Monitoring and observability of machine learning systems: Current practices and gaps," *arXiv:2510.24142*, 2025. arXiv:2510.24142.