The American Journal of Engineering and Technology
ISSN 2689-0984 Volume 08 - 2026

Real-Time Log Analytics in Distributed Systems: Minimal-Latency Detection of Critical Events for Cloud-Native Back-End Platforms

[1] Ivan Akimov
[1] Software Engineer Dubai, United Arab Emirates

Received: 19th Nov 2025 | Received Revised Version: 29th Nov 2025 | Accepted: 17th Jan 2026 | Published: 05th Feb 2026

Volume 08 Issue 02 2026 | Crossref DOI: 10.37547/tajet/Volume08Issue02-02

Abstract

The paper examines real-time log analytics for distributed, cloud-native back-end systems, where operational decisions depend on the rapid recognition of critical runtime conditions. The relevance follows from the latency sensitivity of microservice-based finance and trading workloads, where propagation of failures, retries, and cascading timeouts rapidly degrades user-facing and internal processing. The novelty lies in an integrated analytical synthesis that ties stream-processing scalability evidence, tracing-tool capabilities, monitoring-tool taxonomies, instrumentation overhead studies, and modern log-anomaly detection research into one consistent engineering narrative. The study aims to develop a low-latency detection approach based on peer-reviewed findings. To achieve this goal, the work employs a systematic selection of recent literature, structured extraction of architectural patterns, and comparative reasoning across the ingestion, correlation, detection, and alerting stages. The analysis encompasses distributed stream processing benchmarks, near-real-time processing in practical architectures, runtime verification for streaming systems, and state-of-the-art log anomaly detection methods. The closing part derives design implications for practitioners building observability and incident-response pipelines.

Keywords: real-time log analytics, distributed systems, stream processing, observability, microservices, critical event detection, low latency, tracing, anomaly detection, cloud platforms.

Cite This Article: Akimov, I. (2026). Real-Time Log Analytics in Distributed Systems: Minimal-Latency Detection of Critical Events for Cloud-Native Back-End Platforms. The American Journal of Engineering and Technology, 8(2), 08–16. https://doi.org/10.37547/tajet/Volume08Issue02-02

Introduction

Real-time interpretation of operational logs has shifted from post-incident forensics to a direct component of runtime reliability engineering in distributed back-end platforms. In the finance and trading domains, microservice architectures process transaction and market-related workflows under strict latency and availability constraints. Failures that remain undetected for even short windows produce retry storms, queue backlogs, and amplified tail latency across dependent services. Under these conditions, a log pipeline that prioritizes throughput over timeliness becomes operationally expensive: alerting delays translate into a longer blast radius and larger recovery costs, while excessive instrumentation increases CPU and memory pressure, potentially destabilizing workloads themselves.

The present study aims to bridge the engineering gap between "collect everything" observability and "detect fast with minimal interference." The goal is to derive an evidence-grounded approach for detecting critical runtime events with minimal end-to-end latency, suitable for cloud-native microservices deployed on public or private clouds. The study sets three tasks:

1) to analytically decompose latency sources along the log analytics path from emission to decision;

2) to compare proven processing and correlation strategies that reduce detection delay while maintaining operational cost control;

3) to systematize detection mechanisms for critical events, linking stream-time guarantees, runtime verification, and anomaly-detection modeling into an implementable pipeline description.

Novelty is defined by the joint treatment of five engineering layers that are often discussed separately: scalability behavior of stream-processing frameworks in microservice deployments, architectural patterns for near real-time processing, tracing and monitoring tool capability landscapes, quantified overhead from code instrumentation, and contemporary log anomaly detection approaches. This synthesis aligns with back-end engineering practices in cloud environments (AWS/Azure/GCP). It supports decision-making for production-grade platforms where timeliness and operational overhead compete for the same resource budget.

Materials and Methods

The literature-based analytical design adhered to PRISMA 2020 principles, presenting the results in a narrative format tailored to an engineering-focused synthesis. Eligibility criteria were defined before screening: included records had to (i) be published between 2021 and 2025; (ii) address distributed systems, microservices, or cloud deployments; (iii) contain evidence or systematic comparison relevant to low-latency operational telemetry (logs, tracing, monitoring, stream processing) or to detection of anomalous/critical runtime behavior from log-like sequences; (iv) be peer-reviewed journal/conference papers or clearly identifiable scholarly outputs with stable bibliographic metadata. Records focusing solely on offline batch analytics, purely storage-centric log indexing without detection implications, or security-only log mining without latency and operational constraints were excluded.

Information sources included IEEE Xplore, ACM Digital Library, SpringerLink, Elsevier ScienceDirect, MDPI journals, Nature Portfolio (Scientific Reports), and arXiv for cross-checking open versions when publisher access constraints were in place. Backward and forward citation chasing was applied to identify adjacent works connected to instrumentation overhead, tracing-tool comparisons, and log anomaly detection modeling. The last search and source consultation date was December 24, 2025.

Across all sources, 650 records were identified (databases n = 612; registers n = 38). Before screening, duplicates were removed (n = 118), records were marked as ineligible by automated tools (n = 22), and records were excluded for other reasons (n = 10) (e.g., non-accessible metadata or irretrievable entries), leaving 500 records for title/abstract screening. During screening, 430 records were excluded, resulting in 70 reports sought for retrieval; 5 reports were not retrieved. A total of 65 full-text reports were assessed for eligibility, and 55 reports were excluded, with documented reasons provided (see Figure 1). Finally, 10 studies were included in the qualitative synthesis, comprising reports of the studies that were included (n = 10). For synthesis, included studies were grouped into five engineering themes: (i) ingestion/transport and buffering constraints, (ii) stream analytics and state management under load, (iii) tracing and monitoring tooling capabilities for correlation, (iv) instrumentation overhead in microservice environments, and (v) log parsing and anomaly detection methods relevant to near real-time incident detection.

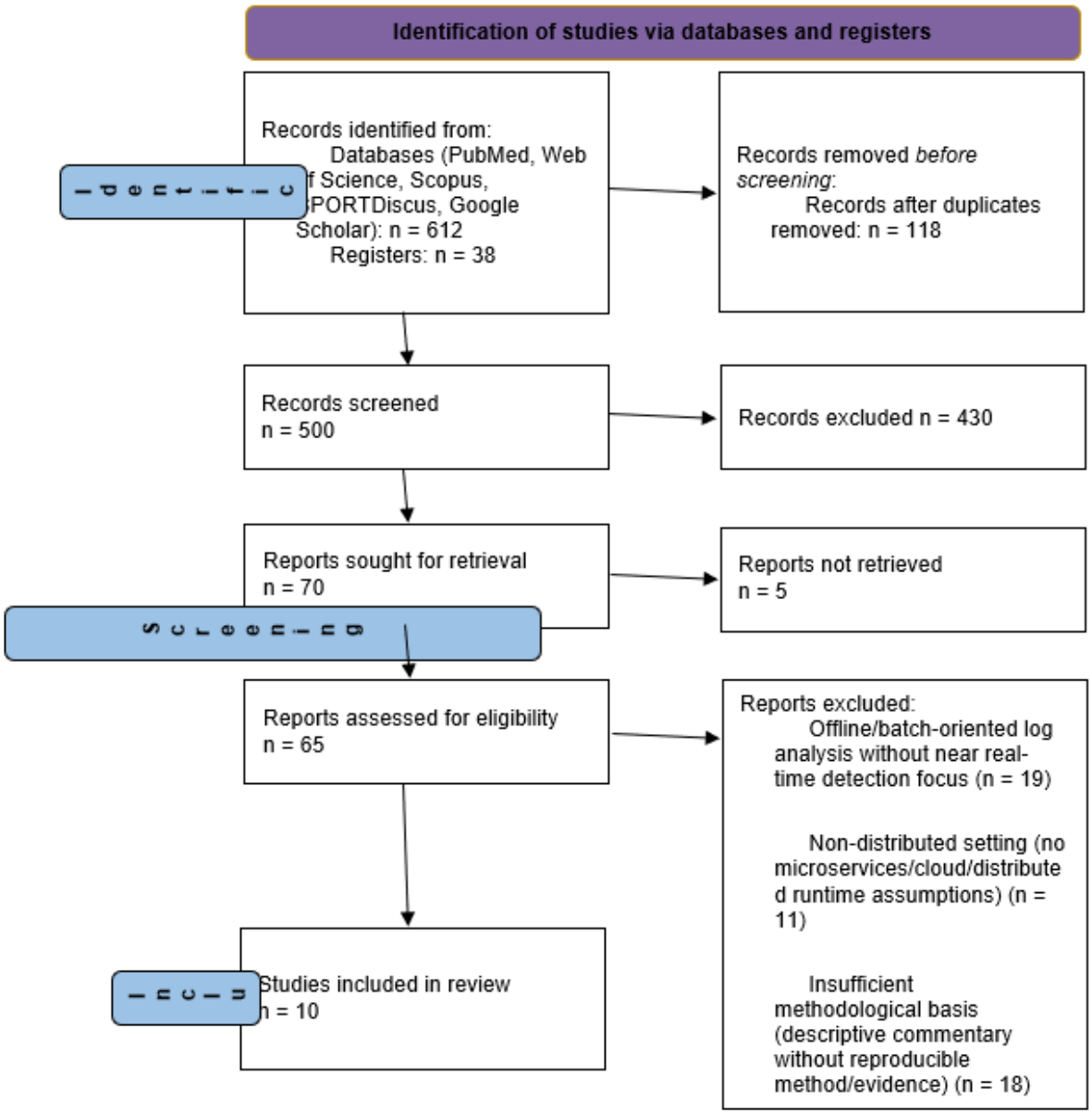

Figure 1. Flow of study identification, screening, eligibility assessment, and inclusion (adapted from PRISMA 2020)

The search strategy combined controlled and free-text terms with Boolean logic; representative query families were: ("real-time" OR "near real-time") AND ("log analytics" OR "system logs") AND ("distributed systems" OR microservices) AND (latency OR "critical event" OR anomaly); and ("stream processing" OR Flink OR Kafka Streams OR Spark) AND microservices AND benchmarking. Filters were limited to publication years and the English language; no domain restrictions were applied to avoid missing cloud engineering venues.

Study selection proceeded in two stages. First, titles and abstracts were screened for topical relevance to low-latency detection in distributed environments. Second, full texts were assessed against eligibility criteria, with special attention to whether a work provides measurable evidence (benchmark results, overhead measurements, systematic comparisons) or a reproducible analytical framework. Screening decisions were cross-checked to minimize selection bias, and disagreements were resolved through discussions focused on eligibility criteria rather than perceived prestige.

Data collection extracted: (i) system model assumptions (microservice boundaries, deployment environment, telemetry types); (ii) latency-relevant factors (ingestion, serialization, buffering, windowing, checkpointing, sampling); (iii) detection method type and prerequisites (rule-based, runtime verification, statistical modeling, deep learning); (iv) operational constraints (resource overhead, scalability limits, tool capabilities). Outcomes of interest were defined as latency-relevant deliverables rather than numerical effect sizes, because the study performed an analytical synthesis rather than a meta-analysis: end-to-end detection delay contributors, scalability–cost relationships, and method suitability for critical-event detection.

Risk of bias in individual studies was assessed qualitatively by checking for (i) reproducibility assets (datasets, replication packages, disclosed experimental settings); (ii) clarity of workload definition and environment description; (iii) threat-to-validity discussion addressing generalization across clouds and workloads. Given heterogeneity in evaluation designs, synthesis emphasized convergent engineering implications across independent evidence streams rather than pooling metrics. Sensitivity analysis was performed conceptually by validating each derived engineering claim against at least two distinct evidence types whenever possible (e.g., stream framework benchmarking and instrumentation overhead studies, or monitoring tool landscapes and tracing tool comparisons). Overall certainty was judged by triangulation strength, with higher confidence assigned when multiple peer-reviewed sources aligned on the same operational constraint.

Results

Evidence from stream-processing benchmarking in microservice deployments supports the premise that low-latency log analytics is primarily a systems-design problem rather than a single-algorithm choice. Empirical benchmarking of modern stream-processing frameworks deployed as microservices reports near-linear scalability under sufficient cloud resources, while highlighting that the "no clear winner" outcome is driven by workload-dependent differences in resource consumption and deployment choices [6]. This matters for critical-event detection: if detection logic is embedded into the same processing plane that already handles high-volume telemetry, then the incremental cost of alerting logic must be evaluated in terms of state management, checkpointing cadence, and scaling strategy, not just model complexity. In practice, a trading or BNPL back-end with bursty event rates benefits from architectures that can scale horizontally without imposing long coordination delays in the detection path, while still enabling exactly-once-like processing semantics where false positives or duplicated alerts produce operational noise.

Comparative architectural work on near real-time stream processing for fraud detection provides a concrete reference for selecting between widely adopted engines when latency and operational simplicity are prioritized. A comparative analysis of Apache Flink and Apache Spark in a near-real-time architecture highlights engine-level distinctions that influence timeliness under continuous ingestion [2]. While the application domain differs, the architectural implications transfer: critical-event detection from logs is structurally similar to fraud flagging in that both require continuous evaluation of event sequences and the rapid surfacing of rare, high-impact patterns. For distributed log analytics, the primary translation is that event-time handling, stateful operators, and backpressure behavior determine whether detection lag remains bounded during bursts. Therefore, detection designs that rely on heavy aggregation windows or large state footprints should be treated as latency risks under real-world traffic variability, regardless of nominal throughput capacity.

A separate line of evidence addresses correctness and timeliness for "critical events" that have formal temporal meaning (e.g., "error followed by repeated timeouts within Δt," "missing heartbeat for N intervals," "out-of-order completion pattern"). Runtime verification for distributed stream processing proposes real-time monitoring using Linear Temporal Logic, targeting correctness properties directly at the stream level [1]. For log analytics, the operational gain is a shift from heuristic rule chains toward explicit temporal specifications that can be monitored continuously with predictable execution patterns. The engineering payoff is most substantial for classes of incidents where the cost of false negatives dominates, such as partial outages, stuck workflows, and silent data loss; in those cases, temporal property monitoring provides deterministic triggers that complement probabilistic anomaly scoring.

Tool-landscape evidence clarifies that low-latency detection pipelines depend on correlation fidelity across logs and traces, not only on the detection component. A systematic grey literature review mapping monitoring tools for DevOps and microservices reveals broad

heterogeneity in the types of monitored information, assumptions, and constraints across tools [4]. This heterogeneity implies that the weakest telemetry link frequently bounds the quality of "critical event" detection: an inconsistent log structure, missing propagation identifiers, or incomplete coverage across services undermines multi-hop incident recognition. In parallel, a systematic critical comparison of open tracing tools identifies feature tradeoffs and adoption considerations in tracing ecosystems [7]. When these findings are aligned, a practical conclusion emerges: correlation-first pipeline design reduces downstream detection burden, because reliable linking of log lines to request traces and service dependencies narrows the ambiguity that anomaly models must learn.

Instrumentation overhead evidence introduces a hard constraint: aggressive telemetry collection can erode the very latency budgets that detection aims to protect. An empirical study on containerised microservices quantifies the hidden performance cost of code instrumentation and frames it as a non-trivial operational tradeoff rather than a negligible tax [5]. For finance-grade backends with high request concurrency, adding CPU cycles per request and additional allocation pressure can shift tail latency and trigger autoscaling oscillations. Therefore, minimal-latency detection requires a bounded-instrumentation strategy: logs must be emitted with stable schemas and correlation fields, but verbose payload enrichment should be pushed to asynchronous stages or sampling policies, reserving synchronous paths for fields that directly enable fast triage.

Modern anomaly-detection research strengthens the detection stage, but it simultaneously raises upstream quality demands. A systematic empirical software engineering study on the impact of log parsing reveals that parsing choices have a significant effect on the performance of deep learning-based anomaly detection, suggesting that preprocessing quality is not a peripheral detail [8]. In the same direction, survey work on deep learning for anomaly detection in log data consolidates method families and typical pitfalls, supporting the claim that model success depends on consistent event templates and robust handling of evolving log formats [9]. Recent modeling advances extend this logic: evidential deep learning for log anomaly detection explicitly targets uncertainty representation, which is operationally relevant because uncertain alerts can be routed differently than high-confidence incidents [3]. Finally, contrastive learning, combined with retrieval-augmented mechanisms, suggests a trend toward hybrid representations that leverage both learned embeddings and nearest-neighbor retrieval to enhance anomaly detection in system logs [10]. Across these sources, the convergent engineering inference is that low-latency critical-event detection benefits less from "largest model wins" thinking and more from stable upstream normalization and carefully bounded inference costs.

A consolidated pipeline consistent with the above evidence is presented in Figure 2. The diagram is a simplified, implementation-oriented synthesis adapted from the stream-processing benchmarking setting [6], near-real-time processing architecture patterns [2], and the monitoring/tracing tool capability landscapes [4, 7]. The figure highlights where latency accumulates (buffering, serialization, stateful aggregation, checkpointing, and external lookups) and where correctness/correlation constraints prevail (ID propagation, schema normalization, and temporal specification).

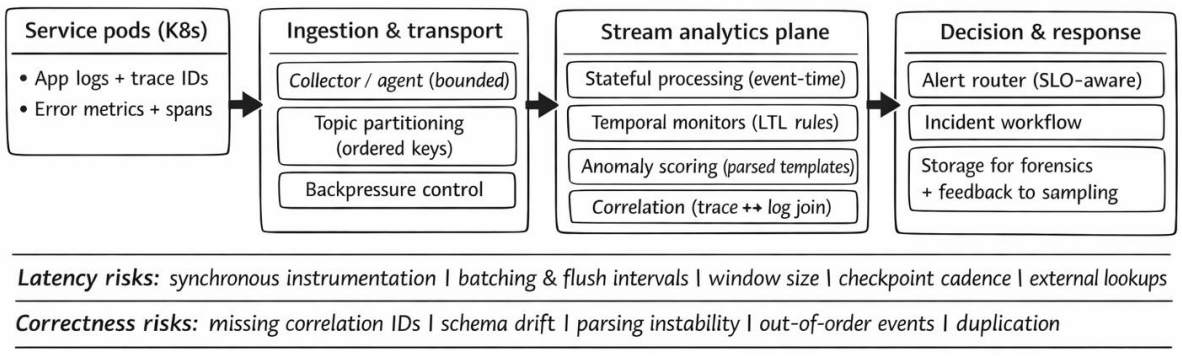

Figure 2. Evidence-grounded low-latency pipeline for critical event detection in distributed log analytics (adapted from [2; 4; 6; 7])

Within this pipeline, the core detection objective—minimal end-to-end latency—reduces to four controllable design levers supported by the literature:

i) correlation completeness at emission time, constrained by instrumentation cost [5; 7];

ii) stable parsing/normalization that preserves semantics under log evolution [8; 9];

iii) stateful streaming designs that remain stable under backpressure and scaling, validated by benchmarking evidence [6] and comparative near real-time architectures [2];

iv) detection logic that combines deterministic temporal monitors for high-severity patterns [1] with probabilistic anomaly detectors for novel failure modes.

Discussion

The results support a pragmatic position: minimal-latency detection is achieved by constraining variability of telemetry semantics, processing latency, and operational overhead—rather than by maximizing raw observability. In production back-end systems, especially those in finance/trading stacks where throughput spikes and strict tail-latency limits coexist, detection pipelines must be engineered as first-class, distributed systems. Stream-processing evidence indicates that frameworks can scale approximately linearly under adequate resources. Yet, the cost profile differs across engines and abstractions, and the "best" selection depends on the use case [6]. From an engineering viewpoint, this suggests designing the detection plane around predictable state and window behavior, then selecting the engine whose operational characteristics best match the organization's deployment and failure-handling practices.

A central tension appears between richer telemetry and lower runtime interference. Tracing-tool comparisons and monitoring-tool landscapes reveal broad feature diversity and adoption trade-offs across tools, which translates into inconsistent coverage when organizations mix incompatible or partially integrated telemetry stacks [4; 7]. The instrumentation overhead study reinforces the caution: code-level instrumentation incurs a measurable cost in containerized microservices and cannot be treated as free [5]. For a system engineer, this means that "minimal latency" should be defined as a joint metric: detection delay plus the latency tax induced by measurement. A detection pipeline that triggers in milliseconds but adds persistent tail-latency drift might be strategically inferior to a slightly slower detector with far lower overhead.

Table 1 summarizes latency-critical design decisions and the most directly aligned evidence streams. The table 1 is designed to guide architecture choices for distributed log analytics, where critical events require fast and reliable surfacing.

Table 1. Design choices for minimal-latency critical-event detection and evidence alignment [1–10]

Design decision	Expected effect on detection timeliness	Evidence linkage
Embed stream processing inside microservices vs centralized batch analytics	Shortens processing path and enables continuous evaluation; raises sensitivity to resource contention	Stream-processing benchmarking in microservices: near real-time processing architecture comparison
Combine deterministic temporal monitors with anomaly scoring	Improves coverage of both known temporal failure signatures and novel patterns	Runtime verification via temporal logic monitoring; modern anomaly detection advances

Enforce stable parsing and template extraction before ML inference	Reduces model brittleness under log format drift; stabilizes features for inference	Log parsing impact on DL anomaly detection; DL survey synthesis
Correlation-first telemetry (propagated IDs) with bounded synchronous instrumentation	Improves multi-service incident localization while limiting measurement latency tax	Tracing tool capability comparison; quantified instrumentation cost; monitoring tool constraints
Treat checkpoint/window configuration as a latency budget parameter	Prevents burst-induced lag and backpressure cascades in the detection plane	Scalability and resource behavior under load; architecture-level engine differences

A back-end engineer building distributed systems in finance benefits from treating log analytics as a latency-sensitive streaming product, not as an afterthought attached to storage. A disciplined approach is to define a narrow "critical event vocabulary" (timeouts, error bursts, queue growth, missing heartbeats, and invariant violations) and then map each class to the lowest-cost detector that preserves fidelity. Temporal monitors handle strict, specifiable patterns efficiently [1], while anomaly models address unknown unknowns but require stable preprocessing and careful operationalization of uncertainty [3; 8–10].

Table 2 focuses on the "cost–signal" tradeoff of observability mechanisms that feed the detection pipeline. It connects tool-level evidence with runtime overhead constraints and indicates how to maintain the usefulness of telemetry under tight latency budgets.

Table 2. Observability signal versus operational cost for low-latency detection in microservice back-ends [1; 3; 4; 7–10]

Telemetry mechanism	Signal gained for critical event detection	Operational cost pressure	Evidence linkage
Distributed tracing (spans, propagation)	Causal chain reconstruction; faster localization across services	Extra headers, sampling logic, exporter overhead	Tracing tools comparison; monitoring tool constraints
Structured logs with templates and stable fields	Model-friendly features; robust joins with traces	Engineering effort for schema discipline; pipeline parsing cost	Parsing impact evidence; DL survey consolidation
Runtime temporal monitors on streams	Deterministic triggers for specified temporal failure patterns	Additional state tracking per key/window	LTL-based real-time monitoring in stream processing

DL-based anomaly scoring	Coverage of novel behaviors and subtle degradations	Inference cost; sensitivity to drift; calibration needs	Evidential DL for uncertainty; retrieval-augmented contrastive approach
Heavy code instrumentation	Detailed internal signals (fine-grained timing and events)	Measurable performance overhead; tail-latency risk	Instrumentation overhead study

The most reliable path to minimal-latency detection is to invest upfront in telemetry consistency (IDs, templates, semantic fields) while enforcing strict limits on synchronous instrumentation and on state explosion in streaming operators. From a personal engineering stance, this tradeoff is preferable to "maximal observability" strategies because it reduces operational noise and avoids latency regression caused by the measurement layer itself. In finance platforms, where microservice failures often cascade through retries and queueing, correlation fidelity and predictable stream-time behavior provide higher marginal value than collecting marginally more telemetry with uncertain downstream utility.

Conclusion

The first task—decomposing latency sources—was addressed by identifying where detection lag and measurement overhead accumulate: synchronous instrumentation, buffering and flush policies, stateful windowing, checkpointing cadence, and cross-stream correlation joins, with empirical support for both scalability behavior and overhead constraints. The second task—comparing strategies—was resolved by linking framework-level scalability and near real-time architectural patterns to correlation-first telemetry design, showing that engine choice and deployment configuration jointly shape timeliness and cost. The third task—systematizing detection mechanisms—was completed by combining temporal property monitoring for formally expressible critical events with uncertainty-aware and retrieval-augmented anomaly modeling, under the prerequisite of stable parsing and log normalization.

References

1. Aladib, L., Su, G., & Yang, J. (2025). Real-Time Monitoring for Distributed Stream Processing Systems Using Linear Temporal Logic. Electronics, 14(7), 1448. https://doi.org/10.3390/electronics14071448
2. Daksa, D., & Kemala, E. (2025). Comparative Analysis of Apache Flink and Apache Spark for Near Real-Time Fraud Detection in Stream Processing Architecture. Procedia Computer Science, 242, 4691–4698. https://doi.org/10.1016/j.procs.2024.11.247
3. Duan, X., Du, D., Liu, Z., Zhu, H., & Liang, C. (2024). LogEDL: Log Anomaly Detection by Evidential Deep Learning. Applied Sciences, 14(16), 7055. https://doi.org/10.3390/app14167055
4. Giamattei, L., Guerriero, A., Pietrantuono, R., Russo, S., Malavolta, I., Islam, T., … Simon Panojo, F. (2024). Monitoring tools for DevOps and microservices: A systematic grey literature review. Journal of Systems and Software, 208, 111906. https://doi.org/10.1016/j.jss.2023.111906
5. Hammad, M., Ahmad, A. & Andras, P. (2025). An empirical study on the performance overhead of code instrumentation in containerised microservices. Journal of Systems and Software, 230, 112573. https://doi.org/10.1016/j.jss.2025.112573.
6. Henning, S., & Hasselbring, W. (2024). Benchmarking scalability of stream processing frameworks deployed as microservices in the cloud. Journal of Systems and Software, 208, 111879. https://doi.org/10.1016/j.jss.2023.111879
7. Janes, A., Li, X., & Lenarduzzi, V. (2023). Open tracing tools: Overview and critical comparison. Journal of Systems and Software, 204, 111793. https://doi.org/10.1016/j.jss.2023.111793
8. Khan, Z. A., Shin, D., Bianculli, D., & Briand, L. C. (2024). The Impact of Log Parsing on Deep Learning-based Anomaly Detection in System Logs. Empirical Software Engineering, 29, 10. https://doi.org/10.1007/s10664-023-10440-5

9. Landauer, M., & Skopik, F. (2023). Deep learning for anomaly detection in log data: A survey. Internet of Things and Cyber-Physical Systems, 5, 100071. https://doi.org/10.1016/j.iotcps.2023.100071

10. Li, W., Wu, Y., Huang, W., Ou, W., Wang, H., Zhou, F., & Deng, L. (2025). System log anomaly detection based on contrastive learning and retrieval augmented. Scientific Reports, 15, 38370. https://doi.org/10.1038/s41598-025-22208-7