ISSN-L 2664-9969; E-ISSN 2706-5448

UDC 004.75
DOI: 10.15587/2706-5448.2026.362294

Roman Moravskyi,
Yevheniya Levus

DEVELOPMENT OF AN ADAPTIVE DATA PROVENANCE CONTROL METHOD FOR RESOURCE-AWARE REAL-TIME STREAM PROCESSING

The object of research is the process of managing data provenance in real-time stream processing systems. The subject of research is an adaptive data provenance method for resource-aware stream processing in Apache Flink, which changes provenance granularity at runtime without affecting business results.

The problem addressed is how to maintain explainable provenance for anomaly detection or auditing while avoiding system resource overload from always-on, fine-grained data provenance.

The significance of the results obtained is a provenance level controller driven by live CPU and heap memory usage metrics. Its scientific novelty lies in a three-level logic (detailed, summary, none) that dynamically adjusts the provenance level without interrupting the pipeline. Performance was evaluated on 5000 simulated IoT sensor workloads with 60-second processing windows and a target of 15000 records per second under 3-, 15-, and 30-minute runs.

These results indicate that the suggested method maintains throughput close to the no-provenance baseline while significantly reducing CPU cost compared to full provenance. For equal workloads, adaptive runs consume roughly 3–4 times the baseline CPU, whereas full provenance requires about 6–9 times the baseline CPU, because the controller demotes provenance under pressure and restores detailed mode only when resources are recovered. Heap usage is a weaker control signal because it reflects only part of the total memory and can change when the runtime periodically frees unused memory.

The proposed method is most useful during anomalies, incidents, or audits where selective, on-demand provenance is needed. This adaptability makes it highly suitable for IoT gateways, edge devices, and regulated industries operating under strict resource constraints.

Keywords: *stream processing, data provenance, Apache Flink, distributed systems.*

Received: 14.01.2026
Received in revised form: 12.05.2026
Accepted: 22.05.2026
Published: 29.05.2026

How to cite
Moravskyi, R., Levus, Y. (2026). Development of an adaptive data provenance control method for resource-aware real-time stream processing. Technology Audit and Production Reserves, 3 (2 (89)), 60–65. https://doi.org/10.15587/2706-5448.2026.362294

1. Introduction

Real-time stream processing systems (e. g. Apache Flink, Spark Streaming, Kafka Streams) are widely used to handle telemetry, financial events, and other high-volume data flows. In many applications, the pipeline must later explain how it produced a result. For instance, which sensor readings triggered an alarm or which transactions contributed to a fraud score. This requirement is addressed by data provenance [1], which is additional metadata that records the provenance of every tuple as it moves through a pipeline.

Modern stream-processing systems balance two conflicting needs. Operations teams and auditors require a reliable record of input events producing results for debugging, compliance, and explaining anomalies. At the same time, high-throughput pipelines must limit latency and resource usage to meet service-level objectives (SLOs) and service-level agreements (SLAs) [2]. Capturing detailed provenance for each stream record increases memory and CPU usage. During traffic spikes, these costs cause backpressure, potentially leading to job restarts or data loss.

The research [3] proposes a provenance method for deterministic stream processing that links outputs to source tuples, minimizing overhead for edge and cyber-physical systems. It replaces variable-length provenance with fixed metadata per tuple using standard operators for modular, distribution-independent provenance. This allows analysts to trace each sink tuple to its sources without storing all raw inputs. Evaluation on vehicular and smart-grid cases shows effective, low-cost backward provenance on constrained devices.

In paper [4], a live forward provenance framework for streaming is introduced. It extends the backwards-provenance tool to emit a live mapping that connects each sink's continuous results to the source tuples that produced them. This framework also marks when sources expire, meaning they can no longer influence future results. While the method in [3] builds backward provenance only when requested using small metadata records per tuple, the approach in [4] continuously produces forward provenance, providing ordered links and explicit source expiration. As a result, monitoring and control can be driven in real time rather than through post-hoc analysis.

The research [5] addresses the provenance overhead on edge devices by giving operators a simple switch between full and none. This framework creates a duplicate query graph with provenance disabled when the user-defined condition is met, for example, during a nightly schedule. It routes the stream through a duplicated graph and then tears down the original graph. This approach does reduce overhead, but during the switch, two graphs run in parallel, memory usage doubles for a short time, and when provenance is disabled, the system retains no trace at all.

The approach in paper [6] is completely different from previous frameworks. Its intrusion-detection rules can adapt to the workload by

using gradient descent while continuously recording full provenance. However, this approach configures gradients offline, so it cannot respond to sudden resource pressure during runtime.

Recent researches have improved resource-adaptive stream processing through memory management and dynamic scheduling [7]. However, the overhead of internal metadata, such as continuous provenance tracking, remains largely unaddressed.

Existing solutions provide provenance as either always-on or always-off, forcing a choice between traceability and stable performance. No widely adopted method allows the system to adapt provenance level at run-time in response to current resource conditions while preserving a minimum level of diagnostic information.

In a review of real-time stream processing for metering and IoT [8], it was identified that while streaming provenance is highly relevant, it remains under-engineered. Researches [9, 10] show that while provenance helps auditing and fault isolation, it also creates significant CPU, latency, and storage overheads.

The goal is to control provenance collection to prevent system overload while keeping enough data for later investigation. This work introduces a method for runtime control of provenance granularity in Flink-based streams. It treats provenance detail as a dynamic parameter of any Flink job. At runtime, the job can remain in detailed mode, fall back to a summary mode that stores only window statistics, or, if resources are exhausted, turn off provenance entirely. A lightweight control stream broadcasts the current level to every operator, so the change is instantaneous and does not require a second directed acyclic graph (DAG). Because the control stream is pluggable, the trigger can be CPU/RAM metrics (as shown in the prototype), an SLA alarm, or a manual flag. In short, paper [5] demonstrates that turning provenance off can save resources, and paper [6] shows that adaptive logic can enhance provenance-based analytics. However, neither offers a live, multi-level throttle. The approach proposed in this paper combines both ideas. It lowers overhead when necessary, restores full traceability when affordable, and does so without graph duplication or pre-training.

The object of research is the process of managing data provenance in real-time stream processing systems.

The subject of research is a framework that extends existing provenance methods [3, 4] with a lightweight control layer that selects the provenance level without altering job semantics.

The aim of this research is to develop an adaptive data provenance control method for resource-aware real-time stream processing. This will allow the system to reduce CPU overhead and minimize overload risks during data spikes while maintaining baseline throughput.

The distinctive feature of the proposed method is an adaptive, resource-aware controller that sets the provenance depth at runtime based on live CPU and heap metrics, aiming to keep throughput near the baseline and reduce overload risk.

The following objectives are set in the scope of this research:

1) to design a controller that adjusts provenance levels based on live CPU and memory usage. Apache Flink implementation should use the broadcast state pattern to synchronize updates across all parallel subtasks;

2) to evaluate the suggested method's performance and resource efficiency against baselines using simulated IoT sensor workloads.

2. Material and Method

2.1. Cost model

The method's costs depend on the level and the number of inputs per window. Computational cost per window $C_{comp}(L, N)$ is defined as

$$C_{comp}(L,N)=C_{ba}+\begin{cases} c_l \times N & \text{if } L=\text{Detailed},\\ c_s & \text{if } L=\text{Summary},\\ 0 & \text{otherwise}, \end{cases} \quad (1)$$

where N – inputs per window, C_{ba} – the cost of business aggregation, c_l – the per-tuple provenance overhead in detailed mode, and c_s – the constant overhead of writing a summary object.

Definition of storage footprint per window $S_{prov}(L, N)$ for provenance payload

$$S_{prov}(L,N)=\begin{cases} s_h + s_t \times N & \text{if } L=\text{Detailed},\\ s_h + s_m & \text{if } L=\text{Summary},\\ 0 & \text{otherwise}, \end{cases} \quad (2)$$

where N – inputs per window, s_h – a small header, s_t – bytes per stored input reference, and s_m – bytes for the summary object.

Control-plane traffic rate

$$B_{ctrl}=M\times\frac{60}{T_{poll}}\times S_{msg}, \quad (3)$$

where M – the number of subtasks that receive the broadcast, T_{poll} – the poll interval in seconds, and S_{msg} – the encoded size of one control message.

These formulas are sufficient for rough estimation. Equation (1) explains the CPU gradient across levels because only the detailed mode scales with N. Equation (2) explains storage growth under the detailed mode and the constant footprint of the summary mode. Equation (3) shows why the control plane remains negligible at the minute scale.

2.2. Setup

All experiments were executed on a single workstation (Intel Core i7-11700, NVIDIA GeForce RTX 3080, 16 GB RAM, 1 TB SSD). The streaming job does not use the GPU, but the specification is reported for completeness. In higher-load scenarios, the machine is intentionally driven to near full CPU utilization to expose the system to realistic resource pressure.

The software environment is containerized with Docker Compose. The stack combines Apache Flink 1.19.1 (Scala 2.12, Java 17), Apache Kafka 7.6.1 with ZooKeeper, OpenSearch 2.15.0 with OpenSearch Dashboards, and Prometheus with Grafana. Flink configuration consists of a single job manager and a single task manager configured with 8 task slots. The Kafka input topic is provisioned with 16 partitions, which are sufficient to keep the source busy and evenly distribute keys during keyed aggregation. For reproducibility and to keep local runs lightweight, the Kafka broker and the OpenSearch node are deployed in single-node mode. Flink exposed metrics to Prometheus, which is scraped every 5 seconds. Grafana showed averages over the selected time range for Flink Task Manager CPU load, heap usage, and source throughput, plus time-series traces for CPU and heap.

2.3. Anomaly detection, workload, and experimental procedures

A simple per-window anomaly detector is used, which maintains a running mean and flags an anomaly when the absolute deviation of a reading from that mean exceeds a fixed threshold. This detector serves only to exercise the provenance levels and controller behavior, not as a contribution to anomaly-detection research.

Sensor readings are simulated with the following fields: sensorId, event timestamp, and value. The mapper adds a processing timestamp. Each run uses 5000 sensors with 30% marked as anomalous. Anomalies start at one-third of the run. At minute 1 for 3-minute runs, at minute 5 for 15-minute runs, and at minute 10 for 30-minute runs. The sensor readings generator does not use a random seed. Instead, it produces fixed constant values for normal readings to ensure reproducibility. The runner sets selected values to 2147483647 after the start offset to force clear anomalies. 3, 15, and 30-minute durations are selected to test different system phases. The 3-minute runs show reactions to sudden spikes. The 15-minute runs show system stabilization. The 30-minute runs show long-term memory management.

ISSN-L 2664-9969; E-ISSN 2706-5448

Traffic is configured to 15000 records per second (rec/s) by emitting one record per sensor per tick, with the tick interval calculated as the sensor count divided by the target rec/s, rounded down to 1 millisecond. Records are Avro-encoded and keyed by sensorId to evenly distribute across 16 Kafka topic partitions. The anomaly detection job read sensor data from Kafka and sent out alerts to a separate Kafka topic. Job parallelism is set to 8, the window size to 60 seconds, and the controller thresholds and sink batching are kept constant across modes. IDs follow a stable pattern within a run to maintain partition balance, and the runner rotates the starting index each tick to prevent hot keys.

The same workload is repeated for no-provenance, adaptive, and full-provenance jobs to obtain comparable results. Each configuration was executed five times to account for system background noise.

The job stored aggregated results and provenance logs in OpenSearch. All runs used 60-second processing-time tumbling windows. For the adaptive-provenance job, the following parameters were used:

1) 10-second control poll;

2) 2 hot polls to escalate issues, and 3 cool polls to relax;

3) CPU thresholds were set at 0.40 for demotion to none and 0.30 for demotion to summary;

4) heap thresholds were 0.70 and 0.60 for none and summary, respectively.

3. Results and Discussion

3.1. Design and implementation of the adaptive provenance controller

The proposed method is implemented as URD, a framework for runtime control of provenance granularity in Flink-based streams. Named after the Norn, which tends the past at Yggdrasil, the URD framework governs how much of the stream's recorded history is retained or compressed as conditions change.

Adaptive method extends a stream-processing job (Fig. 1) with a runtime controller that selects the granularity of captured provenance during execution. The functional operators remain unchanged. The job uses processing-time tumbling windows over a Kafka source and emits both aggregated results and optional provenance logs. Provenance semantics are forward-oriented. Each window result is linked to either the exact contributing inputs, a compact summary, or no metadata, depending on the current level. Timing is based on processing time, so window closing and control actions depend on system time. Ordering is per key within a window as provided by the stream processor. Failures are recovered with checkpoints and RocksDB state. Recovery reconstructs the operator state and the last broadcast control message. Sinks provide at-least-once delivery in this configuration.

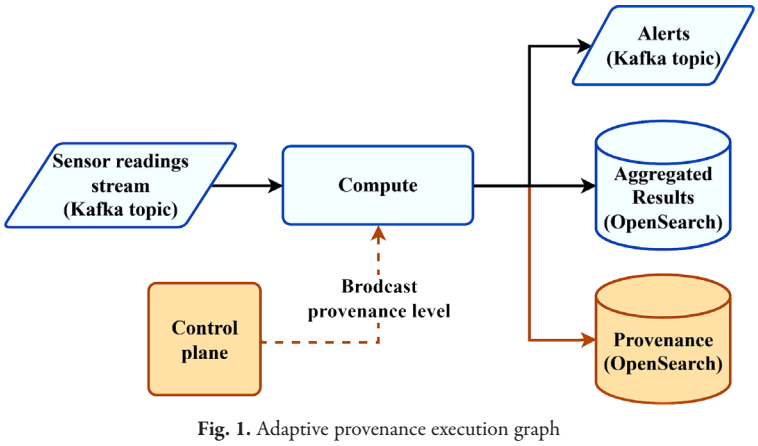

Fig. 1. Adaptive provenance execution graph

The method exposes three levels that do not change business results:

1. "Detailed" records the list of all input tuples that contribute to a window result. The result tuple carries a forward reference that allows iteration over this list.
2. "Summary" records a small summary object for the window such as the number of contributing tuples. No per-tuple links are stored.
3. "None" records nothing. Result tuples carry no provenance payload.

The mapping from inputs to outputs stays fixed across levels. Only the attached metadata changes. Aggregation logic and alerts are identical for all levels. Iteration over contributors is only possible in detailed mode.

The controller samples task-local CPU load and heap usage at a fixed poll interval and evaluates simple thresholds. If CPU or heap exceed the "summary" threshold on consecutive polls, the level is changed from detailed to summary. If the CPU or heap exceeds the "none" threshold on consecutive polls, the controller changes the level to none. If both signals remain below the summary threshold for a few polls, the level is changed back to "detailed". Control messages include the chosen level. The controller broadcasts messages to all subtasks via a broadcast state.

The policy maintains "hot" and "cool" counters. A "hot" counter tracks how many consecutive polls show that the system is overloaded. A "cool" counter counts the number of consecutive polls that show stable resource usage. In the experiments, the poll was set to 10 seconds. Escalation required two hot polls, while cooldown required three cool polls. The CPU thresholds were 30% for summary and 40% for none, and the heap thresholds were 60% for summary and 70% for none.

The method integrates through a strategy-based accumulator and a broadcast control stream. The Kafka source of raw readings is mapped to an internal tuple type with source and processing timestamps. The control source produces control messages at the set cadence and the stream is broadcast. A broadcast process function keeps the latest control message in broadcast state. The keyed stream uses 60-second tumbling windows. The window aggregator uses an adaptive accumulator that always calls the current provenance strategy before and after business aggregation. The adaptive strategy delegates to one of three implementations.

The method preserves the functional output for all levels. Aggregated values and alerts do not depend on the provenance level. In detailed mode, each window result provides an exact, ordered iterator over the contributing inputs. In summary mode, only a summary object is present, so the exact set of inputs cannot be reconstructed. In "none" mode, no provenance is present. Control actions do not reorder inputs inside a window. Control messages are broadcast and applied consistently within a task. On recovery, the processor restores the operator state and the last control state, so subsequent windows continue with a well-defined level. Late data handling is not used in this research processing-time windows. Backpressure does not change semantics but can delay window close and, therefore, the timing of control evaluation.

3.2. Evaluation of performance and resource efficiency

Results obtained in this research are presented in Table 1 as detailed per-run records and in Table 2 as aggregated summaries by job mode and run duration. Table 1 lists the configuration and observed metrics for each run, while Table 2 reports the corresponding averages used for interpretation. Throughput is shown for completeness but is not central to the analysis, so the discussion focuses on CPU load and heap usage.

Table 1

Detailed results per run

Exp. ID	Job type	Total duration (min)	Average CPU load (%)	Average heap memory used (%)	Average throughput (tuples/s)
1	Adaptive provenance (URD)	3	41.30	57.10	13100
2		3	10.70	17.90	14800
3		3	8.83	56.60	14700
4		3	19.80	31.70	14800
5		3	10.60	13.90	14700
6		15	18.00	39.10	14600
7		15	7.09	24.00	14700
8		15	32.00	57.10	14700
9		30	8.75	23.10	14700
10		30	7.08	47.30	14700
11	Full provenance (Ananke/ GeneaLog)	3	18.20	53.30	14700
12		3	18.40	42.80	14700
13		3	60.60	34.00	14700
14		3	61.80	60.80	14600
15		3	18.90	18.40	14700
16		15	32.90	32.10	14700
17		15	63.50	75.50	14700
18		15	18.40	47.10	14700
19		30	31.60	24.00	14800
20		30	51.20	24.10	14700
21	No provenance	3	6.90	45.10	14700
22		3	4.25	30.20	14500
23		3	6.73	74.40	14700
24		3	4.65	55.00	14600
25		3	4.90	59.20	14700
26		15	5.35	27.50	14700
27		15	4.50	28.80	14700
28		15	4.59	30.90	14600
29		30	4.64	53.30	14700
30		30	4.55	43.50	14600

Table 2

Summary of mean CPU load and heap memory usage by job type and run duration

Job Type	Total duration (min)	Average CPU load (%)	Average heap memory used (%)
Adaptive provenance (URD)	3	18.25	35.44
Full provenance (Ananke/GeneaLog)	3	35.58	41.86
No provenance	3	5.49	52.78
Adaptive provenance (URD)	15	19.03	40.07
Full provenance (Ananke/GeneaLog)	15	38.27	51.57
No provenance	15	4.81	29.07
Adaptive provenance (URD)	30	7.92	35.20
Full provenance (Ananke/GeneaLog)	30	41.40	24.05
No provenance	30	4.60	48.40

Experiment ID is the sequential index of the run in the batch. Job type indicates the job configuration: No provenance (baseline), Adaptive provenance (URD), or Full provenance (Ananke/GeneaLog). The number of sensors is the count of distinct sensor keys produced each tick (5000). Total run duration (minutes) is the planned length of the run (3, 15, or 30). Target input rate (records per second, rec/s) is the requested production rate in the test runner (15000), which is a target rather than a guarantee because Kafka, scheduling, and the network can throttle the producer. Producer tick interval (milliseconds) is the inter-tick spacing chosen to reach the target rate (≈333 ms for 5000 sensors at 15000 rec/s). The number of anomalous sensors is the count of sensors that emit anomalous values during the anomaly phase (30% of sensors in these runs, i. e., 1500). Anomaly start (minutes) is the offset from run start when anomalies begin (1 minute). Average CPU load (%) is the arithmetic mean of the Task Manager JVM CPU load over the analysis window. Average heap used over max (%) is the arithmetic mean of the JVM heap used metric and heap max metric ratio. It excludes off-heap and RocksDB-managed memory. Average throughput (records per second) is the arithmetic mean of the operator output rate recorded by Flink.

Across all durations, CPU load shows a consistent pattern: full provenance has the highest, no provenance the lowest, and adaptive in between. In adaptive mode, the controller adjusts provenance level based on load, switching from detailed to summary, then to none when needed. This allows jobs to spend less on costly provenance during stress and revert to detailed provenance when resources recover, as shown in Table 2 and Fig. 2. At 3 minutes, means are about 5.49% (no provenance), 18.25% (adaptive), and 35.58% (full). At 15 minutes, 4.81%, 19.03%, 38.27%. At 30 minutes, 4.60%, 7.92%, 41.40%. The 30-minute data suggest that longer, steady periods reduce oscillations and stabilize the controller by spending more time in cheaper modes.

Table 1 shows an outlier in Experiment 1, with an average CPU of about 41.3% and a throughput of about 13.1 krec/s. Short three-minute windows are sensitive to warm-up and the start of anomalies at one minute. In this run, the producer seemed to lag behind the 15,000 rec/s target, probably due to scheduling jitter or Kafka backpressure. Meanwhile, the controller might not have had enough polls to demote aggressively, since it needs two high-load polls in a row every 10 seconds to escalate, and three polls to cool down. These factors can keep the job in a more expensive state for longer during the short window, explaining the higher CPU usage.

Fig. 3 displays a typical adaptive run with an average CPU usage of about 7%, an average heap near 47%, and a steady throughput of approximately 14.7 krec/s. The throughput curve rises and then remains flat, demonstrating that the producer maintained the target rate and the pipeline did not experience backpressure.

The CPU trace shows repeating spike-and-decay shapes. There are three reasons for this pattern. First, the 60-second tumbling windows group work at the end of each window. This causes aggregation, anomaly checks, and sink writes to come in short bursts. Second, periodic checkpoints and RocksDB snapshot operations cause short CPU spikes. Sometimes, flushes or compaction also produce these peaks. Third, the adaptive controller checks CPU and heap levels every ten seconds. It then adjusts provenance details accordingly. When it promotes to detailed provenance, per-tuple instrumentation kicks in and CPU use increases. When thresholds are lowered, it demotes to summary, or no provenance, and CPU use drops. Over longer periods, the controller spends more time in cheaper modes. This explains the low average CPU shown in the top panel and the ratios in Table 2.

The average heap memory used metric in Table 1 does not exhibit a stable ordering across modes or durations, and Table 2 confirms that this instability persists when values are averaged by job type and window length. At first glance, it is counterintuitive that the no-provenance runs sometimes show the highest mean heap ratio, while adaptive or full provenance show lower ratios. This is an artefact of how memory is organized in this configuration. With RocksDB and

ISSN-L 2664-9969; E-ISSN 2706-5448

managed memory enabled, much of the state and network buffers are outside the JVM heap in Flink's off-heap memory and RocksDB's native memory, like memtables and cache. The heap ratio only shows Java heap use, not the main memory consumers during steady state. The JVM G1 collector on Java 17 manages young and old regions with mixed collections, collecting more often at high allocation rates, which lowers the heap ratio despite stable or increasing total memory. When allocation slows, G1 defers collections, raising the heap ratio until a mixed collection occurs. Short analysis windows intensify these effects by starting anomaly phases at one minute and shifting patterns during execution.

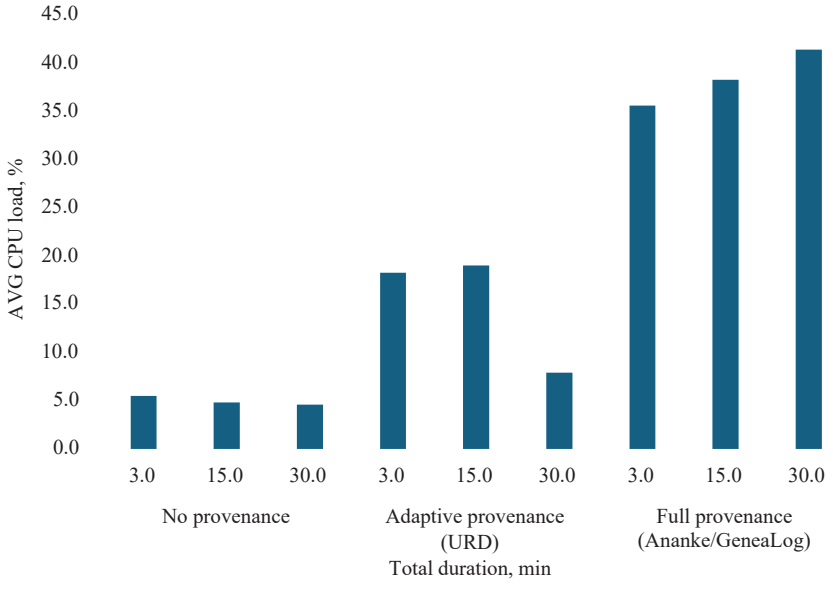

Fig. 2. Average CPU load (%) per total duration and job type

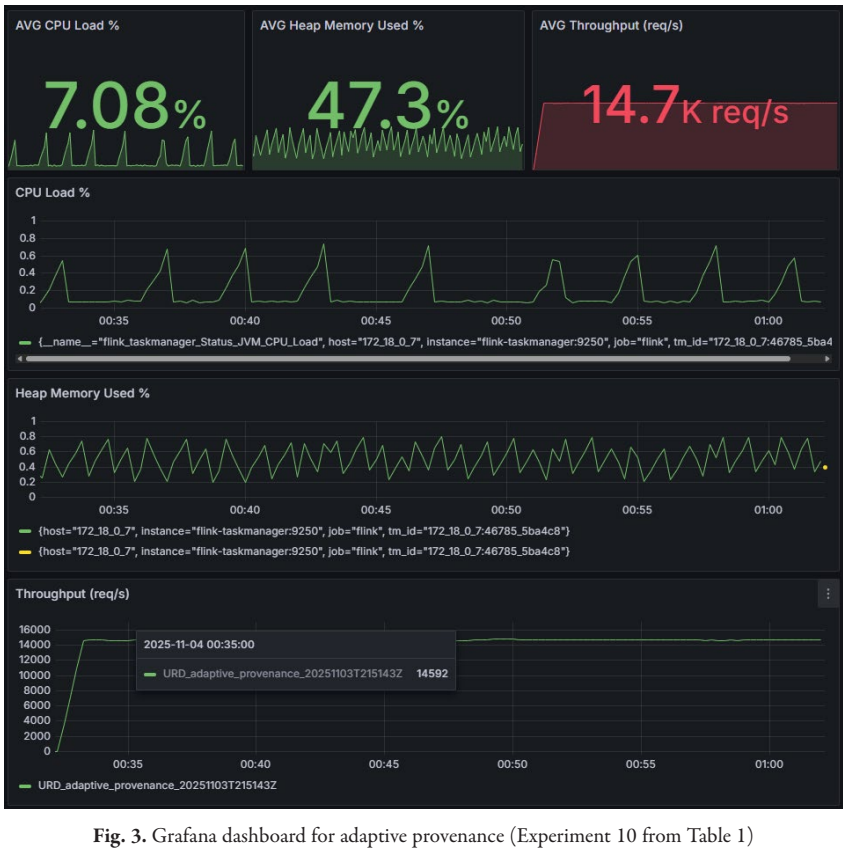

Fig. 3. Grafana dashboard for adaptive provenance (Experiment 10 from Table 1)

Differences by job type explain the mixed heap results. Full provenance uses more provenance objects and metadata, leading to more frequent young-generation collections and a lower average heap-used ratio, even though the job does more work and uses more CPU. No-provenance uses the least, with fewer collections and larger heap growth, making its average ratio appear higher. Adaptive switching between detailed, summary, and no provenance depends on CPU and heap activity. Short runs (~3 minutes) may show mid-transition states with few controller polls, making heap ratio look lower, especially if recent GCs and mode changes dominate. Longer runs (~30 minutes) allow the JVM and RocksDB caches to stabilize, with a regular GC cycle, reducing the link between heap ratio and actual memory use. Overall, heap usage relative to the max heap reflects only Java heap occupancy, not total process memory, and is a practical control signal rather than evidence of memory savings. In Experiment 1 (Table 1), an outlier shows high CPU (41.30%) and lower throughput, as the three-minute window was too short for the controller to reduce the provenance level. The job remained at detailed provenance, disrupting GC and misleading heap readings.

For each job type at 3, 15, and 30 minutes, the number of runs is the same, but throughput does not explain the provenance question. While the test runner aimed for 15000 rec/s, it did not ensure precise pacing, so the consumer also needed to keep up. Apart from one outlier in the adaptive provenance run (Experiment 1 at about 13000 rec/s), all tests clustered near the "no-provenance" baseline. This suggests that the producer was close to the target rate and the pipeline experienced no backpressure. Consequently, throughput is considered sufficient, but not useful for distinguishing between provenance modes in this batch.

The results show that the pattern aligns with the design. Full provenance builds and ships detailed provenance on every path, which is intensive. No-provenance has no provenance work in the hot path. Adaptive monitors CPU and heap, demoting from detailed to summary and none during pressure, then promoting back when pressure falls. Across all windows, the adaptive CPU reflects a time-weighted mix of these modes. Longer windows allow the controller to converge, which explains the drop from about 4 baselines at 15 minutes to 1.7 times at 30 minutes.

Research [5] treats provenance at a fixed level during deployment. Systems that support detailed provenance [3, 4] accept steady overhead in exchange for always-on traceability. Approach [6] uses a retrospective method that doesn't take much time to run, but it takes longer to get answers. Research [11] splits provenance into an online stream and an offline store for anomaly analysis and reports about 13–32% end-to-end overhead and 11–24% throughput loss. Time-travel debugging [12] adds a provenance inspector to replay pipeline state, but it does not control resource use at runtime. Research [13] streamlines threat detection on provenance graphs, lowering detection lag and saving memory, but assumes a fixed provenance setting.

Recent researches have shown that provenance is useful, but it adds runtime costs. This research demonstrates that a dynamic approach can maintain regular operations near baseline while providing detailed provenance on demand.

3.3. Research limitations and future work

Practical significance. This research informs stream-processing systems on maintaining traceability efficiently. An adaptive provenance method balances routine analytics and detailed provenance for anomalies or audits. It reduces storage and computational stress on constrained nodes and speeds investigations. This approach suits edge, single-node, cost-sensitive clusters and regulated domains.

Research limitations. The simulated IoT sensors data producer targeted a rate but didn't guarantee it, so short runs are sensitive to pacing and warm-up. The analysis spans multiple time windows, limiting the controller's ability to escalate or cool down. The heap metric reflects only Java heap, not off-heap memory or RocksDB native memory, making it unreliable across modes. Using a single Task Manager limits concurrency and may not mimic a multi-node setup. Long-term costs are also hard to assess as latency, backpressure, or storage growth were not measured.

Impact of martial law in Ukraine. No impact on the research plan, execution, or interpretation of the results. All experiments ran on local infrastructure with Docker.

Future work. To address these limitations, future research will build on these results by exploring backpressure-driven control signals for multi-node clusters and latency optimization, as well as developing graph-aware, tiered storage solutions to support long-term audits.

4. Conclusions

1. The adaptive framework was developed in Apache Flink that replaces the fixed "always-on" or "always-off" approach. This controller automatically shifts between detailed, summary, and zero-provenance modes by tracking live CPU and memory metrics. An important qualitative feature of this approach is that it can change the provenance level without restarting the pipeline. It was achieved by using a broadcast state pattern that synchronizes a new provenance level across all parallel subtasks.

2. Tests with simulated IoT sensors showed the adaptive approach keeps the system stable during heavy data spikes without dropping throughput. Over 30-minute runs, the method used only 1.7 times the baseline CPU, while full provenance used 9.0 times. Shorter tests confirmed the adaptive controller was more CPU-efficient (3.3x to 4.0x) than full provenance (6.5x to 8.0x). Heap usage was unstable due to Flink's off-heap RocksDB memory and JVM garbage collection. These results demonstrate a cost-effective, reliable strategy for real-world stream processing, enabling systems to run near baseline costs by activating full data tracking only when resources permit. This improves cost sensitivity, especially for IoT gateways, edge devices, and regulated industries, while maintaining accountability without increasing crash risk.

Conflict of interest

The authors declare that they have no conflicts of interest in relation to this research, including financial, personal, authorship, or other, that could affect the research or its results presented in this article.

Financing

The research was performed without financial support.

Data availability

Data will be made available on reasonable request.

Use of artificial intelligence

The authors declare that Grammarly Edu was used throughout the article except for the reference section. It was used to fix grammar, spelling, and word-choice errors. Each suggestion was carefully reviewed and either accepted or rejected. Grammarly Edu usage had no impact on the article results.

Authors' contributions

Roman Moravskyi: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Visualization, Writing – original draft; ***Yevheniya Levus***: Supervision, Methodology, Validation, Writing – review and editing.

References

1. Herschel, M., Diestelkämper, R., Ben Lahmar, H. (2017). A survey on provenance: What for? What form? What from? *The VLDB Journal, 26 (6),* 881–906. https://doi.org/10.1007/s00778-017-0486-1
2. Bashtovyi, A., Fechan, A. (2025). Development of a standardized approach for evaluating business insights in stream processing systems based on technical metrics. *Technology Audit and Production Reserves, 2 (2 (82)),* 15–20. https://doi.org/10.15587/2706-5448.2025.325717
3. Palyvos-Giannas, D., Gulisano, V., Papatriantafilou, M. (2018). GeneaLog: Fine-Grained Data Streaming Provenance at the Edge. *Proceedings of the 19th International Middleware Conference,* 227–238. https://doi.org/10.1145/3274808.3274826
4. Palyvos-Giannas, D., Havers, B., Papatriantafilou, M., Gulisano, V. (2020). Ananke: a streaming framework for live forward provenance. *Proceedings of the VLDB Endowment, 14 (3),* 391–403. https://doi.org/10.14778/3430915.3430928
5. Gordani Shahri, M., Erlandsson, A., Palyvos-Giannas, D., Gulisano, V. (2021). Poster: Twins, a Middleware for Adaptive Streaming Provenance at the Edge. *Proceedings of the 22nd International Conference on Distributed Computing and Networking.* New York, 235–236. https://doi.org/10.1145/3427796.3433931
6. Wang, L., Shen, X., Li, W., LI, Z., Sekar, R., Liu, H., Chen, Y. (2025). Incorporating Gradients to Rules: Towards Lightweight, Adaptive Provenance-based Intrusion Detection. *Proceedings 2025 Network and Distributed System Security Symposium.* https://doi.org/10.14722/ndss.2025.230822
7. Fernando, L., Kim, T., Daudjee, K., Rabl, T. (2025). Enjima: A Resource-Adaptive Stream Processing System. *Proceedings of the ACM on Management of Data,* 3 (6), 1–27. https://doi.org/10.1145/3769790
8. Moravskyi, R., Levus, Y. (2024). Analysis of Real-time Processing Approaches for Large Data Volumes in Metering Infrastructure. *Journal of Lviv Polytechnic National University: Information Systems and Networks, 15,* 169–183. https://doi.org/10.23939/sisn2024.15.169
9. Mohamed, Z. (2023). *Data streaming provenance in advanced metering infrastructures.* [Master's thesis; Chalmers University of Technology]. Available at: https://hdl.handle.net/2077/79292
10. Taube, J., Johnsson, W. (2022). *Streaming analytics with provenance in the advanced metering infrastructure.* [Master's thesis; Chalmers University of Technology]. Available at: https://odr.chalmers.se/handle/20.500.12380/305852
11. Ye, Q., Lu, M. (2021). s2p: Provenance Research for Stream Processing System. *Applied Sciences, 11 (12),* 5523. https://doi.org/10.3390/app11125523
12. Räth, T., Schlegel, M., Sattler, K.-U. (2024). Everything Everyway All at Once – Time Traveling Debugging for Stream Processing Applications. *2024 IEEE 40th International Conference on Data Engineering (ICDE),* 1606–1618. https://doi.org/10.1109/icde60146.2024.00131
13. Goyal, A., Liu, J., Bates, A., Wang, G. (2024). *ORCHID: Streaming threat detection over versioned provenance graphs.* arXiv:2408.13347v1. https://doi.org/10.48550/arXiv.2408.13347

✉***Roman Moravskyi,*** *PhD Student, Assistant, Department of Software, Lviv Polytechnic National University, Lviv, Ukraine, e-mail: roman.o.moravskyi@lpnu.ua, ORCID: https://orcid.org/0000-0002-6695-1920*

Yevheniya Levus, *Candidate of Technical Sciences, Associate Professor, Department of Software, Lviv Polytechnic National University, Lviv, Ukraine, ORCID: https://orcid.org/0000-0001-5109-7533*

✉*Corresponding author*