UDC 004.75, 004.42
DOI: 10.15587/2706-5448.2025.344185

Artem Bashtovyi,
Andrii Fechan

DEVELOPMENT OF ADAPTIVE RECONFIGURATION METHOD FOR STREAM DATA PROCESSING SYSTEMS USING SYSTEM METRICS

The object of research is the process of adaptive configuration changes for stream processing applications which is focused on improving specific performance properties. The absence of the generalized automated approach for dynamic reconfiguration of state-store in limited hardware environment is the research problem addressed in this paper. The proposed solution helps to avoid a need for manual application reconfiguration from engineers. The implementation is based on Kafka Streams but designed to be portable across other frameworks that use RocksDB as a state store. Static configuration of modern stream processing systems limits efficiency under variable workloads. In this study, an adaptive module is proposed that monitors system metrics in real-time and automatically updates state-store configurations. The module performs deterministic check to derive new configuration based on predefined thresholds or utilizes a fine-tuned Large Language Model (LLM) to select new configuration values when decisions are vague. The method dynamically applies updates to the affected instance. High-load experimental results reveal the fact that adaptive executions eliminated write stalls, increased memtable hit ratio from 2% to 40% and block-cache hit ratio from 15% to 80%, reduced disk I/O by approximately 50%, and improved throughput by around 5%, at the cost of higher memory consumption. To avoid redundant adaptive updates and outlier-based bias a 10-minute observation frequency was selected. The approach is suitable for systems with fixed resources, state-intensive workloads with high key cardinality. Additionally, if covers the need for safe configuration change under operational constraints. The architecture is framework agnostic for the RocksDB-based based stream processing with state stores.

Keywords: *distributed systems, stream processing, Kafka Streams, adaptivity, adaptive, dynamic, RocksDB.*

Received: 16.08.2025
Received in revised form: 27.10.2025
Accepted: 18.11.2025
Published: 29.12.2025

How to cite
Bashtovyi, A., Fechan, A. (2025). Development of adaptive reconfiguration method for stream data processing systems using system metrics. Technology Audit and Production Reserves, 6 (2 (86)), 15–22. https://doi.org/10.15587/2706-5448.2025.344185

1. Introduction

Stream processing is state-of-the art technology for the real-time applications that should be both resilient and efficient under varying workloads. Stream processing frameworks such as Flink, Spark, and Kafka Streams are popular across communities, specifically because they provide low-latency data handling and stateful operations across large-scale infrastructures [1]. Such systems rely on different architecture principles and methods to guarantee fault tolerance. Fault-tolerance itself can be achieved in various ways for different stream processing frameworks. Mainly, majority of frameworks utilize storage in order to achieve join, grouping, counting and other state-related operations. Furthermore, storage is used for restore actions if the application fails and needs to be restarted again. For instance, Apache Flink framework, uses checkpointing strategy based on storing checkpoints on a disk to restore data in case of failure [2]. Other lightweight frameworks are popular across the community as well. For example, Kafka Streams is used for variety of tasks, including event a migration task [3]. To implement fault tolerance, it applies a different mechanism that integrates with Apache Kafka's internal architecture and storage [4]. Kafka Streams leverages local disk state stores backed by changelog topics, in contrast to external checkpoints approach. These changelog topics capture every modification to the state store, allowing the application to reconstruct its state after a failure by replaying the changes from Kafka itself. The state store is essentially a local disk or memory storage that saves aggregation, latest values and meta data. This design is particularly useful applications where lightweight and embedded processing is desirable. Specifically, it enables fine-grained fast state recovery. Like most modern stream processing frameworks, for state store Kafka Streams utilizes a key-value local database, RocksDB [5], which is designed for high-performance and low-latency operations. State store is used for writing and reading stateful records or operations like groupBy, count, etc. Meaning when a stream processing app requires getting an element from a state store, it performs a read operations from the state store source. By default, stream processing frameworks such as Apache Flink, Spark Structured Streaming, Kafka Streams, etc. rely on static configuration properties. In other words, they are set by default and can be changed only when necessary. The properties are related to state stores as well, including block cache size, write buffer size, and compaction settings etc. While these defaults simplify deployment and avoid human involvement, a stream processing application's performance may be affected by adapting the state store configurations for runtime variability, such as changing data rates, highly diversified access patterns or fluctuating resource availability. Authors in the previous research [6] have shown that dynamically changed configurations based on the system's needs at the moment can significantly improve throughput and increase latency or recovery time of stream processing applications. Other research [7] has proposed a comprehensive review of automatic performance tuning techniques, including ML (Machine Learning)-based prediction, experiment-driven search, and adaptive runtime controllers that improve specific aspects of stream processing. However, the presented solutions mainly change specific configuration parameters, i. e. checkpointing intervals, operator scheduling or tied to a framework. To the best of our knowledge, there are no dynamic adaptive methods that focus on cross framework solutions. Based on that, it is difficult to adapt the approach for the various stream processing engines. Specifically, because of the lack of portability and high engineering efforts. This highlights a key gap in current research: the lack of defined framework-agnostic layer for real-time configuration change, which can cover performance gains and fault-tolerance across DSPS (Distributed stream processing systems) platforms. Configuration of state store is one of the ways achieving cross-platform solution, because it is used in almost all the stream processing frameworks. To address this gap, a series of experiments had to be performed to define the correlation of application performance and state store parameters. Based on the results, this paper presents a lightweight adaptive was developed that monitors key performance metrics and determines and adjusts state store configuration if needed. The module helps to perform dynamic reconfiguration and is designed to be integrated across stream engines that support state store.

The object of research is the process of adaptive configuration changes for stream processing applications which is focused on improving specific performance properties.

The aim of this research is to develop an adaptive configuration method for state stores in stream processing applications based on system observability, enabling dynamic performance optimization without additional hardware resources. Based on that said that, several research gaps were identified in these areas:

1. To investigate how changes in state store configuration parameters correlate to the performance metrics (throughput, latency, disk I/O) of stream processing applications under various workloads.
2. To design, validate and evaluate a dynamically managed, resource-efficient state store adaptation method that operates without additional hardware requirements and can be applied across multiple stream processing frameworks.

2. Materials and Methods

2.1. Adaptive method for stream processing

The popularity of stream processing frameworks has been increasing dramatically over last years. Particularly, Apache Flink, Apache Spark Streaming, Hazelcast Jet, and Kafka Streams popular tools for real-time analytics with low latency and high throughput needs. In spite of high popularity, the frameworks configuration is rarely the subject of custom tuning by the engineers, occasionally because it is time-consuming. There are papers presenting evidence of the fact that streaming systems benefit from adaptive configuration. The authors of Drizzle [8] present custom tuning for reducing the latency, by creation of custom stream processing framework that decouples processing from coordination intervals to keep millisecond-level latency while adapting less frequently to failures and load shifts. Other authors present a Khaos method [9] that automatically optimizes checkpoint intervals at runtime. It is done by observing the system metrics and training analytical models to keep recovery time and end-to-end latency within defined QoS constraints. Based on their results, static default configuration is a not the best fit for stream processing applications. Authors in paper Ca-Stream [10] present adaptive approach for system state that dynamically configures operator placement and resources, particularly state store in real-time. It utilizes a model to guide dynamic elasticity decisions based on monitoring and adaptive assignment of operators to optimize latency and resource use. The work focuses on adaptivity at the level of resource scaling and operator placement rather than dynamic configuration tuning. Moreover, it is focused on the custom architecture which is difficult to port across different frameworks. Other authors [11] suggest a reinforcement learning (RL)-based approach to compress in-memory stream aggregates dynamically. This allows to reduce memory consumption. They present an adaptive manager that monitors the size and characteristics of stream aggregates in real time and decides when and how apply changes in real-time without making a trade-offs in terms performance or latency constraints. The general solution is to find a perfect balance between accuracy and memory usage in order to run the system efficiently. The authors present a unique way of application reinforcement learning stream processing changes. The approach performs dynamic changes to aggregates, which in turn, affect memory compression. However, it is the only space where adaptive changes happen. Additionally, the solution is oriented on a particular stream processing framework optimization problem. Authors of PA-SPS [12] propose an automated tuning system for DSPS across different objectives. The idea lays in using evolutionary algorithms for multiple configurations optimization, including throughput, latency, and general resource consumption. A huge advantage of the approach is a hybrid configuration tuning combining based on model training and online refinement. The framework performs well in dynamic load environment. Nevertheless, it focuses on tuning the defined set of parameters which is framework-specific, leaving an idea for contributions in future. Based on the research, it is clear that adaptive tuning approach is relevant for different frameworks and it is used by variety of projects in order to optimize or boost performance of stream processing. Generally, the existing methods focus on a specific parameter space and specific stream processing solution. Many approaches assume either static or stable load on a stream processing system, which is not necessarily matches with production environment and specifically dynamic variable load. Most critically, the state store configuration is not covered in the existing research, which is a huge area to investigate and optimize for variety of frameworks. It is not completely evident how state store changes affect system performance under different conditions. Having said that, were identified several research gaps in these areas:

1. How specific changes to the state store configuration affect stream processing application performance?
2. How can state store adaptation be dynamically managed without relying on additional hardware resources, such that the approach remains lightweight and portable across stream processing frameworks?

In order to address these questions, this paper suggests an adaptive method that monitors stream processing system metrics and applies reconfiguration in real time based on the metrics. The adaptive algorithm includes a deterministic layer with defined thresholds for metrics and configuration values which is inspired by attribute-based quality forecasting [13]. Specifically, the modules collect metrics, aggregates them over a fixed window, creates a normalized feature vector and performs a smart analysis of metrics. The prototype was implemented and tested on single stream processing framework that supports state store, however the approach is designed to be modular enough for use across different frameworks that support state store.

The research method included. Generating of synthetic events with dynamic unpredictable load patterns. This setup is meant to simulate the production environment. The method was allowing reproducibility and defining precise correlation between framework configuration parameters and resulting performance metrics. Analytical modeling solely would not completely cover the behavior of distributed state management. Therefore, metrics-driven empirical method is required to identify how the constrained adaptive changes affect observable system.

2.2. Experiment setup

There were conducted a series of experiments for simulating close to production environment that matches load of medium-size companies. The selected instruments and their versions were selected based on practical experience, popularity across the community and being open-source tools for the experiment use-cases.

The experimental setup is an emulation of a production use stream processing environment using a Docker-based virtual machine configuration orchestrated with Docker Compose (v2.32.4). Which is built on the following hardware: Apple M2 Pro chip, 10-core CPU with 6 performance cores and 4 efficiency cores with 32 GB of RAM to power Docker-compose setup. Docker-based virtual machine is a great candidate for software which can be run across different platforms, which in turn a good choice for our experiment. The modern hardware guarantees that observed improvements result from configuration adaptivity rather than raw hardware extension. Each container was provisioned with static resources, specifically 512 MB of RAM with extra 256 MB buffer, 40 GB of disk space, and a 2-core CPU. Kafka Streams (v3.8.1) was used because it integrates closely with Apache Kafka (vcp-kafka:7.1.0-1-ubi8), supports embedded state stores, and is lightweight for adaptive testing. Additionally, it does not require the setup of the separate infrastructure for stream processing, like other frameworks. It can be integrated as a library to an application. RocksDB is a default option for Kafka Strems. It was selected as the state store due to its great performance, configurable memory and disk layers that directly affect latency and throughput. Spring Boot was used for the application based on Kafka Streams via build-in library. This allows to run the simulation is a short period of time. Prometheus and Grafana were used for metrics scraping and visualization as they are the de facto standard in production observability according to common industry practice and specifically the simplicity of integration. Docker Compose provided deterministic, isolated and adjustable execution environments.

Fig. 1 shows an experimental evaluation involving Kafka Streams client and an adaptation module that reads the data. To ensure consistent and comparable input patterns, a synthetic event generator was developed. It emits messages at predefined rates of 1000 and 1400 events per second. Varied data were simulated with random event key ranging from 0 to 1 million. Kafka Streams instances consume and process data embedded using processors that utilize state stores. The adaptation module monitors metrics and performs an update of configurations when needed. The decision of whether to update the configuration is performed by an LLM component, which is separately hosted by OpenAI. Specifically, the GPT-4.1-2025-04-14 model was selected. The OpenAI was selected because it provides the easy and production-ready features of fine-tuning without extra costs except the subscription expenses. In addition, it provides easy configurable GUI (Graphic User Interface) which allows speed up the testing phase of fine-tuning significantly.

The model was fine-tuned via OpenAI interface for specific stream processing metric analysis tasks, which outputs specified configuration properties values as a structured response.

2.3. Adaptive configuration architecture

As it was mentioned above, the adaptation module is a separate service that is responsible for monitoring metrics and acting accordingly. The algorithm presented in Fig. 2 run by the adaptive module, is intended to replace human involvement by automatically checking the metrics regularly. Eventually, the algorithm decides if there is a need to change the configuration of a specific application based on the processing decisions.

During the initialization phase, the module scans the current configuration of all Kafka Streams instances individually and starts to scan the application metrics by the regular monitoring interval. The monitoring interval is a fixed value $\Delta t = 10$ minutes, which is executed a configurable variable for the algorithm. Nevertheless, based on our observations, 10 minutes is enough to make sure that it is possible to avoid false positive or false negative decisions for adaptive changes during application irregular behavior. Unusually behavior is possible when the application restarts, catches up on ongoing events, and there is a short-term spike or a trough in metrics. The system and state-store metrics are aggregated into a vector and represent metrics variable on the algorithm

$$M_t = \left(m^1, m^2, \ldots, m_i\right), \tag{1}$$

where m_i includes metrics described in Section 3.1.

There are two major scenarios for the algorithm: deterministic check and ML based check. Deterministic check step is performed as a sequence of straightforward and clear rules that are based on the correlation properties in Section 3.1. In scenario, where the metrics and configuration correlation is clearly evident or metrics are above or below certain thresholds the systems configurations are changed in the following

$$\begin{aligned} & deterministic\_check\left(M_t, C_t, I\right) = \{ \\ & deterministic\_update, \;\; if \;\; patterns \;\; detected \\ & ml\_update, \;\; otherwise \\ & \}. \end{aligned} \tag{2}$$

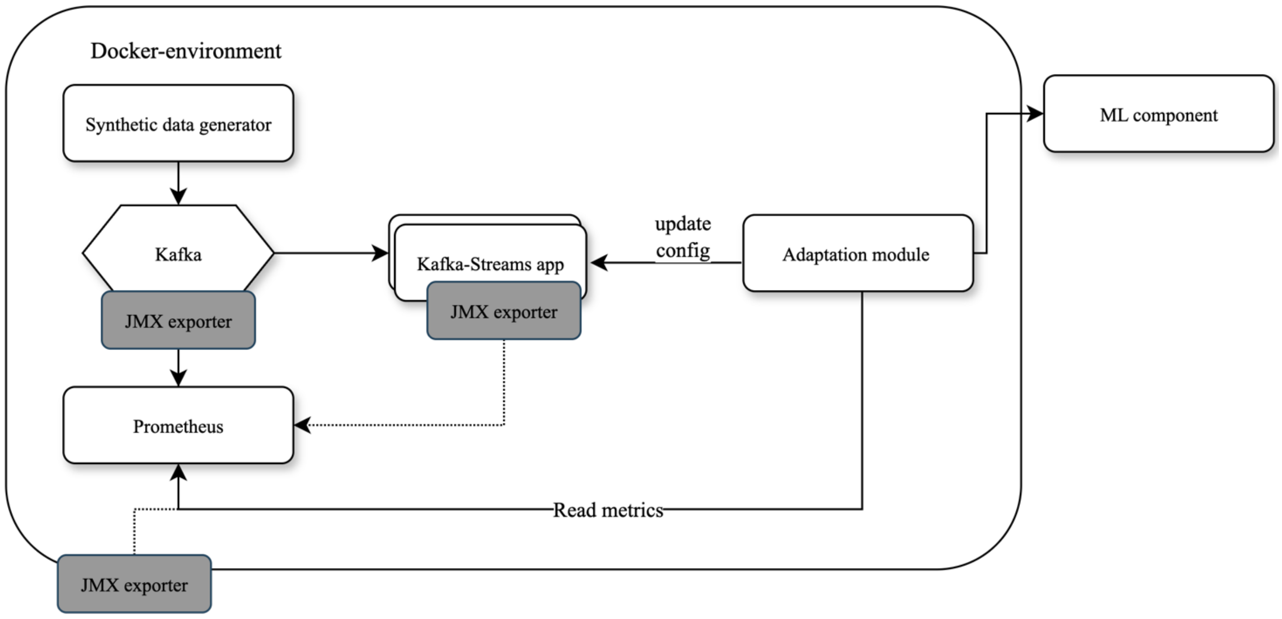

Fig. 1. Architecture for the experiment

```
Algorithm 1: Adaptive Configuration Algorithm for Kafka Streams State Store
Data: $metrics$: collected from Prometheus (memory, disk I/O, latency, throughput, Kafka Streams state
      store stats)
$current\_configs$: current Kafka Streams state store configurations
Result: Adaptive tuning of Kafka Streams state store configurations
initialization of $current\_configs$;
set $monitoring\_interval \leftarrow$ 10 minutes;
while application is running do
    wait($monitoring\_interval$);
    while all instances checked do
        metrics ← collect_from_prometheus(last 10 minutes, instance);
        decision ← deterministic_check(metrics, current_configs, instance, context);
        init $response$;
        if decision = deterministic_update then
            $response \leftarrow$ deterministic_update(metrics, current_configs, instance, context);
        else
            // Pass to ML module
            $response \leftarrow$ ML_module_update(metrics, current_configs, instance, context);
        end
        valid ← = sanity_check($response$);
        if valid == true then
            apply_configs(response.new_configs, instance);
        end
        else
            ignore_update();
        end
        save_decision(decision, response, valid);
    end
end
```

Fig. 2. Adaptive configuration algorithm

This expression illustrates the general principle of deterministic evaluation the system checks for the specific patterns. For the simplicity it shows the main idea rather than the details of algorithm itself. On practice, the actual deterministic logic includes more sophisticated checks that consider memory limits, CPU load, JVM memory and other compound conditions defined in the adaptive module. For the special scenarios the memory and CPU thresholds were checked if it is more than 80% from allowed maximum. In this scenario the changes should be applied that will correct the resources usage in order to avoid possible resources overflow. For instance, let's apply lower and upper boundaries for our configurations to avoid out of memory. To make sure it is impossible to keep updating configs forever. This also plays the role of a sanity layer mentioned later. Second scenario is executed, when deterministic decision cannot be clearly identified. The action involves a machine learning check based on the metrics data, current configs, and specific instance metadata. ML module performs AI-based analysis of metrics and outputs a decision about the analysis. The decision is based on additional historical context as well, and previous metrics are included for the decision process. The ML module is based on the LLM model that is fine-tuned with a stream processing context based on the Kafka Streams documentation, state store configurations and our experiments decision records. The decision records were composed based on investigation defined in Section 3.2.

Both *deterministic_check* and *ML_module_update* represent a function *f* which returns the result of calculation in the format of new configuration values

$$C_{t+1} = f\left(M_t, C_t, H_t\right), \tag{3}$$

where C_t – the current configuration and H_t – the previous metrics and configurations result.

Resulting configuration is essentially a set of new configuration parameters values

$$C_{t+1} = \left\{c_1, c_2, \ldots, c_i\right\}. \tag{4}$$

Eventually, the sanity check is performed based on the result from a previous step and current metrics. Sanity rules $R(C_{t+1})$ are applied to prevent unsafe reconfiguration, the result is one decision

$$\begin{aligned} &R\left(C_{t+1}\right) = \{ \\ &\quad update, \;\; if \;\; (c_i > low_{c_i} \wedge c_i < high_{c_i} \;\; for\ all\ i) \\ &\quad postpone, \;\; otherwise \\ &\}, \end{aligned} \tag{5}$$

where low_{c_i} – the lowest and $high_{c_i}$ – the highest acceptable values for a metric based on the current system setup. Sanity check additionally takes into account the current metric M_t state in a final operation for determining the result.

3. Results and Discussion

3.1. Framework state store metrics and configuration correlation

Stream processing frameworks have a lot of configuration properties that can be set based on the specific application requirements and environment. The properties are categorized and play role in different parts of the application: application metrics, state store metrics, and even custom user-defined metrics. As it was mentioned previously, state store configuration properties were selected as core adaptation subject, specifically RocksDB-related settings.

In order to understand how to tune configurations of applications based on the specific scenario, it was necessary to identify metrics and configuration correlation. A couple of Kafka Streams engineers shared a theoretical discussion on how to configure the state store based on the metrics [14]. However, the article describes hypothetical reasons for relationships. As a standard, framework contributors leave default values that suit the majority of scenarios. Nevertheless, the default values are not necessarily the most optimal in all scenarios. For the actual understanding of Kafka Streams' state store impact on configuration, a series of experiments was conducted to understand precise patterns and define strict rules. The experiments included multiple experimental runs and tracked the metrics and application behavior based on changes to the three configurations mentioned above. Every experimental run was isolated from the other executions to avoid benchmarking bias.

After the experiments the large set of properties was reduced to the main three: *write_buffer_size*, *block_cache_size*, and *max_write_buffers*. That were selected by multiple experiments and proved to have a significant performance impact on applications that utilize state store. A larger *write_buffer_size* or *max_write_buffers* values reduce the frequency of disk writes by batching updates. This allows Kafka Streams to write the updates faster and reduce frequency of Disk I/O, which in turn, positively affects throughput. Property block cache size controls the amount of cached state store values stored in memory, which in turn, reduces disk reads by serving more lookups from memory. Since memory access is much faster than disk access, this lowers latency and improves overall processing performance. Table 1 describes the impact of 3 configuration properties: *write_buffer_size*, *block_cache_size*, *max_write_buffers* on different metric.

The experiment narrowed down the list of metrics and configurations that should be used for adaptive configuration.

Table 1

General configurations for the experiment

Metric	Impact	Comment
jvm_memory_used_bytes	Low	State store uses the memory of the container mostly
container_memory_usage_bytes	High	The higher the values the more memory is used for container
kafka_stream_state_write_stall_duration_avg	High	If the buffer size is low or if not enough buffers are set, then writing to disk happens more slowly than writing to the buffers
kafka_stream_processor_node_process_rate	Medium	The more and the bigger the buffers, the better the throughput of the application
kafka_stream_processor_node_record_e2e_latency_avg	Medium	Generally, latency decreases when more memory is allocated for the state store

3.2. Adaptive method execution and evaluation

Actual experiments were executed in a controlled and reproducible environment (Section 2.2), using artificially created workloads that simulate dynamic real-world load patterns. Every experiment was repeated several times under identical conditions and respective averages of the collected metrics are reported. The initial run used the default static configurations. These values remained unchanged throughout the runtime of the experiment. As illustrated in Fig. 3, the observed metrics stayed relatively stable in the early phases. However, as the application running time increased, performance gradually decreased. In particular, write stalls and disk I/O load accumulated over time, indicating that the static setup could not effectively adapt to increasing workload demands.

Fig. 4 present the results obtained with our adaptive method during the second experiment:

– *Iteration 1*: With the minimal configuration, the block cache hit ratio averaged only 15% (0.15) and the memtable hit ratio was as low as 2% (0.02). Write stalls lasted for about 9 seconds, clearly indicating high disk I/O pressure. Based on these results, the adaptation module triggered a configuration update logic and forwarded a request to ML module, then it passed sanity check. The ML decision increased both write_buffer_size and block_cache_size by approximately 10 times.

– *Iteration 2*: After the 1st iteration adjustment, state store metrics improved dramatically. The memtable hit ratio rose to 40% (0.40), which is 20× higher than in the baseline. The block cache hit ratio reached 80%, compared to the baseline average of 15%. Write stalls were eliminated (reduced to 0), and disk I/O load decreased significantly. Throughput did not increase dramatically, but the reduction in stalls provided a more stable processing rate and an improvement of 5% on average. A short-term trade-off was observed: end-to-end latency temporarily increased by ~14%, and container memory consumption grew due to the enlarged cache and buffer sizes, meaning that it is possible to save more state in memory.

– *Iteration 3*: Ten minutes later, the adaptation module opted not to apply further changes. Decision logs confirmed that the LLM component chose to wait, since metrics were stable: block cache hit ratio remained high, disk I/O was low, and memory use was within acceptable limits.

– *Iteration 4*: On the next cycle, the algorithm increased both *write_buffer_size* and *block_cache_size* by an additional 32%. This raised memtable hit ratios from 40% to 60% on average. Throughput improved by 1–2% immediately after the update, but later returned to the same level as in Iteration 2. End-to-end latency decreased by ~10% after the update, but gradually trended back toward previous values. No significant improvement was observed in block cache hit ratio, because throughput was static across experiments. As expected, container memory usage increased because of the higher size of memory in state store.

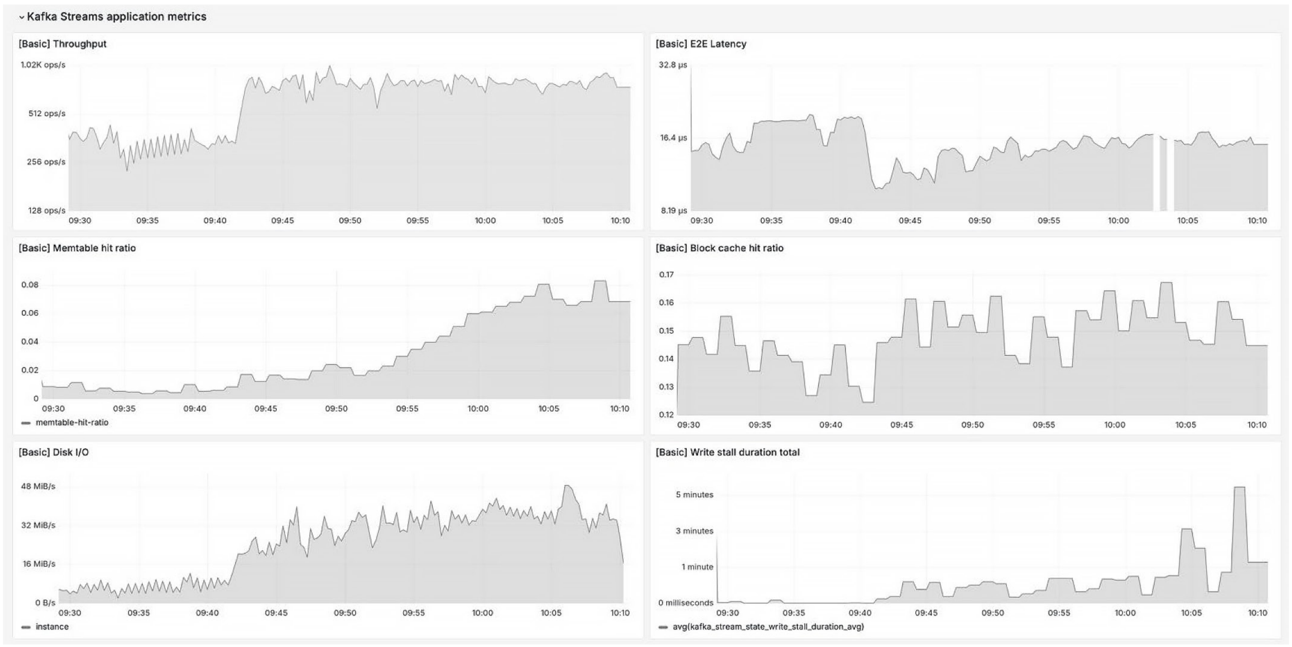

Fig. 3. Results of metrics for the basic experiment with static configuration

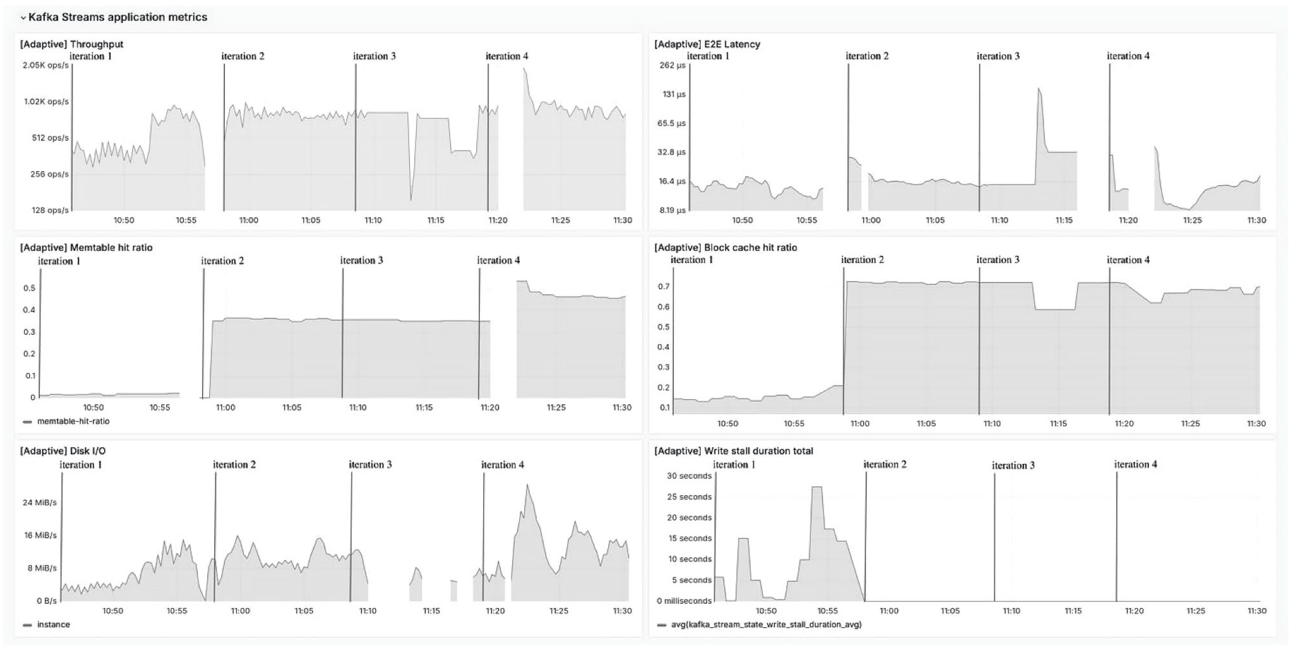

Fig. 4. Results of metrics for adaptive method

The algorithm execution corresponds to the functions and steps defined in Section 2.3. Retrospectively decision log was examined and for every iteration the detailed execution steps of the algorithm were identified. Iteration 1 represents initialization and start of monitoring algorithm procedure no changes happen. Iteration 2 represents metrics of the application after the application of adaptive changes. The ML-based configuration update included the execution of whole adaptive configuration algorithm, including *deterministic_check* and *ML_module_update* steps based on the values defined above. Iteration 3 represents stabilization stage where the configuration remained unchanged because of *deterministic_check* function responded with ignoring update decision. The decision was based on the fact that the delta throughput, e2e latency and Disk I/O between current step and previous step was not larger than 5% specifically. The behavior observed during Iteration 4 corresponds to the decision of *deterministic_check* to perform *ML_module_update*, the algorithm proceeded to the *ML-based configuration update* stage, since no resource limits were violated. The LLM-driven decision function recommended incremental scaling of both *write_buffer_size* and *block_cache_size* and finally perform configuration update. Thus, Iteration 4 actually represents a complete adaptive cycle of the adaptation algorithm for application a new configuration as it was done in previous iterations as well.

Disk I/O was thoroughly monitored specifically. It is clear from the results that the rate for write and read operations is twice as low for the adaptive method as for the static configuration run. The more memory used for storing state, the less frequently the application needs to perform a search on a disk. The behavior can be explained by the intrinsic trade-off between memory and I/O in RocksDB: enlarging in-memory structures increases hit ratios and defers flush operations, thereby lowering disk access latency and smoothing throughput fluctuations. It was clearly identified that the latency temporarily rises strictly after reconfiguration, which is explained by memory adaptation and cache change.

The main idea of providing the details below in the section below is to identify unknowns of community-level hypotheses and actual Kafka Streams dynamic configuration. The core idea of the adaptive configuration approach is to ensure that a Kafka Streams application dynamically operates with optimal state store properties for the given hardware environment. In other words, the method is intended to use the available resources to their better performance than static configurations. Since the key aspect of our results is that the method updates configurations on its own, using both fixed rules and LLM-based inference. This hybrid approach resolves the core problem: eliminating manual tuning while keeping throughput and latency within acceptable limits. While existing papers [15] in adaptive ML-based tuning for stream processing have shown promising results, they primarily target dynamic resource changes connected to specific technology. Another paper [16] highlights system-wide optimization across sources, engines, and sinks using ML. Both works highlight key limitations of adaptive optimization in stream processing, especially the difficulty of choosing how frequent adaptative actions should occur during execution. To address this, the authors propose refining the ML models so they can better handle edge cases related to decision frequency and accuracy, though this would require additional time and resources. This conclusion is relevant for our adaptive approach as well. In contrast, our method focuses on a common solution for tuning the state store, which can be used across different stream processing frameworks [17]. Generally, any stream processing configurations and not necessarily RocksDB can be tuned in a way it is suggested in the paper. By narrowing the scope to fine-grained, metrics-driven state store adaptation, the presented method in this paper remains lightweight and suitable for production usage without requiring a complete redesign of the streaming architecture.

In the experiments, the base static configuration shows performance degradation over time as runtime increased, particularly when write stalls and disk I/O load were suffered a lot. Some trade-offs were observed, mainly higher memory usage and temporary dips in core metrics. Nevertheless, the adaptive method consistently showed better results than the static baseline under dynamic load. The expectations formed during manual testing were confirmed: decisions produced by the deterministic layer, combined with refined LLM outputs led to better throughput during iterative configuration updates. The experiments revealed a clear dependency between block cache size, memory, and throughput. Larger cache and buffer configurations generally increased throughput because events were retrieved from memory more often than from disk. This effect became especially visible at higher input rates, when the system processed thousands of records per second. Increasing memory allocation also reduced disk I/O significantly. As expected by the experiment in Section 2, this introduced a trade-off: larger buffers and caches eased disk pressure but raised overall memory consumption. Latency tended to be lower with smaller block cache and write buffer sizes. Latency between the adaptive and static setups remained similar, though it is possible to anticipate more substantial end-to-end latency improvements under much heavier workloads. In scenarios with several thousand events per second in the Kafka Streams input topic, the adaptive method is expected to yield more pronounced latency reductions. Across other metrics, no major differences were observed. While the performance gains were not large on every metric, the results show that the algorithm responds effectively to runtime variation, advancing beyond static, one-time tuning.

This illustrates that adaptive configuration not only improves efficiency but also reduces the need of manual supervision, which is both time-consuming and involves a human error risk. Based on the experiments and observations, there are several conclusions about the applicability of adaptive configuration change in stream processing:

– *Hardware constraints:* adaptive configuration changes are particularly valuable when hardware resources are static and cannot be scaled easily. Instead of adding computing or storage power, adaptive tuning offers an alternative route to mitigate performance degradation. However, eventually the resources will be drained, and vertical or horizontal scale is inevitable.

– *Workload diversity:* the method is most effective when data is highly diversified and involves many unique events, since these scenarios require intensive use of state stores for lookups and updates.

– *Thresholds and safeguards:* defined thresholds are mandatory. A sanity-check layer within the adaptive module was introduced in our implementation. While some parameter adjustments reduce disk utilization, they may increase memory usage, which can destabilize the application. By combining deterministic checks with ML-driven decisions, it is possible to ensure that LLM-generated updates remain valid and avoid excessive or hallucinated reconfigurations.

– *Dynamic application:* at present, new configurations require a restart for changes to take effect, which introduces short unavailability windows. The reliability of the LLM-based decision layer also depends on the representativeness of its training data; unbalanced or incomplete datasets may yield suboptimal parameter proposals. Moreover, adaptive gains are bounded by physical resource ceilings – once CPU or memory saturation occurs, further tuning cannot improve performance. Additionally, the reliability of the LLM-based decision layer depends strongly on the representativeness and quality of its training data. Because the model was fine-tuned on a limited corpus of historical metric-configuration pairs and documentation examples, its ability to generalize to unseen workload patterns or extreme conditions may be constrained. For instance, when encountering nonregular metric combinations, the LLM may propose suboptimal configuration values or defer decisions unnecessarily. To mitigate this, periodic retraining and inclusion of more diverse operational traces are required. Furthermore, adaptive improvements are inherently bounded by the available hardware resources. Even the most optimal configuration of RocksDB parameters cannot overcome physical memory or CPU saturation. Once the container reaches its memory ceiling or I/O throughput limit, the marginal gain from reconfiguration diminishes to near zero. In such cases, horizontal scaling or infrastructure upgrades become the only viable means of sustaining performance. The adaptive method makes effective use of available capacity, it cannot overcome fundamental resource limits. Notably, configuration updates are applied at the instance level, allowing the remaining application instances to continue operating without interruption during restarts.

– *Operational costs:* running adaptive and ML modules incurs extra computational overhead, as they operate as separate monitoring and decision-making services.

– *Conditions of Application and Reproducibility:* for researches who aim to reproduce or apply the proposed adaptive configuration method several setup conditions must considered. Specifically, to ensure consistency of results and effective operation under production-like circumstances. Firstly, a complete observability stack such as Prometheus and Grafana is required to enable real-time visualization of state-store and system metrics, since the adaptive module depends on continuous metric ingestion and human-verifiable dashboards for validation of changes. Secondly, synthetic data should be generated using a pseudo-random number generator for event identifiers to guarantee realistic key distribution and avoid deterministic key collisions that could bias state-store access patterns. Thirdly, the stream-processing topology should include stateful operators that intensively utilize RocksDB and emulate real state pressure, topology [18] was used. Finally, the number of Kafka Streams application instances must match the number of Kafka topic partitions, ensuring balanced workload distribution and consistent metric comparability across runs.

Future research related to the topic can include investigation and analysis of other metrics and different parameters for Kafka Streams to identify the correlation. Once correlation of different properties and configuration is identified it can contribute to the new versions to adaptive deterministic module, which covers the combined framework and state store configurations. The reliability of the LLM-based decision layer depends on the quality of its training data. Because it was fine-tuned on a small set of historical metrics and examples, it may struggle with unseen workloads or extreme conditions. For instance, when encountering atypical metric combinations, the LLM may propose not optimal configuration values or defer decisions unnecessarily. To mitigate this, periodic retraining and inclusion of more diverse operational traces are required. Furthermore, adaptive improvements are inherently bounded by the available hardware resources. Even the most optimal configuration of RocksDB parameters cannot overcome physical memory or CPU overhead. Once the container reaches its memory threshold or the operating-system I/O throughput limit, the marginal gain from further reconfiguration drops to nearly zero. In such cases, horizontal scaling or infrastructure upgrades become the only viable means of sustaining performance. While the adaptive method efficiently exploits existing capacity, it does not eliminate fundamental resource constraints.

Impact of martial law in Ukraine: the ongoing martial law in Ukraine has not influenced the research process significantly. Unstable internet connectivity and air alarms created minor challenges in conducting continuous benchmarking tests, occasionally disrupting the process.

4. Conclusions

1. The conducted investigation confirmed that variations in state-store configuration have a measurable effect on the performance metrics of stream processing applications. Under production-like synthetic loads (about 1000 events per second), parameter adjustments led to increased memtable and block-cache hit ratios (from 2% to 40% and from 15% to 80%, respectively), elimination of write stalls, and an approximately twofold reduction in disk I/O. Throughput increased by ~5%, which is still expected to be improved under the higher loads. A short-lived latency uptick (~14%) and higher memory use were observed. A later iteration (32% increase to buffers and cache) raised the memtable hit ratio to ~60%, produced a slight throughput gain and ~10% latency drop that later changed toward normal levels. Across runs, it was observed that read/write rates on disk were roughly halved under the adaptive method versus a static baseline. These effects are consistent with the mechanics of RocksDB: larger write buffers batch writes and reduce stalls, while a larger block cache serves more lookups from memory, reducing disk reads. These results describe how state-store configurations influence throughput, latency, and stability under experiment conditions.

2. Experimental evaluation of the adaptive configuration method demonstrated that it maintains system stability and improves performance efficiency under variable workloads without requiring hardware expansion. The adaptive module, operating with a 10-minute observation window, ensured metric convergence and prevented oscillations. Across repeated runs, adaptive configurations consistently halved read/write disk rates and maintained higher throughput compared to the static approach. The method proved most effective in scenarios with fixed resources, workloads that are state-intensive with high key diversity, and operators who seek safe automation with deterministic thresholds. The use of an LLM extended adaptability and enabled more accurate metric-based tuning, which consistently stabilized performance compared to static baselines and reduced manual configuration efforts. Based on the adaptive module decisions memory utilization increased by 20–25%, reflecting expected cache growth while processor utilization remained below 70%. These outcomes confirm the feasibility and practical applicability of the proposed adaptive approach across stream processing frameworks with RocksDB-like backends, at the cost of available hardware resources.

Conflict of interest

The authors declare that they have no conflict of interest in relation to this research, including financial, personal, authorship or other, which could affect the research and 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

During the preparation of this work, the authors used Grammarly in order to: grammar and spelling correction. Additionally, authors used GPT-5 to perform analysis, summarizing, formatting of some references. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

Authors' contributions

Artem Bashtovyi: Methodology, Software, Investigation, Formal analysis, Writing – original draft; ***Andrii Fechan***: Terminology, Conceptualization, Project administration, Supervision.

References

1. Fragkoulis, M., Carbone, P., Kalavri, V., Katsifodimos, A. (2023). A survey on the evolution of stream processing systems. *The VLDB Journal, 33 (2),* 507–541. https://doi.org/10.1007/s00778-023-00819-8
2. Checkpointing. *Apache Flink*. Available at: https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/fault-tolerance/checkpointing/ Last accessed: 27.10.2025
3. Bashtovyi, A., Fechan, A. (2023). Change Data capture for migration to event-driven microservices Case Study. *2023 IEEE 18th International Conference on Computer Science and Information Technologies (CSIT).* IEEE, 1–4. https://doi.org/10.1109/csit61576.2023.10324262
4. Vyas, S., Tyagi, R. K., Sahu, S. (2023). Fault Tolerance and Error Handling Techniques in Apache Kafka. *Proceedings of the 5th International Conference on Information Management & Machine Intelligence.* Association for Computing Machinery, 1–5. https://doi.org/10.1145/3647444.3647844
5. A persistent key-value store for fast storage environments. *RocksDB*. Available at: https://rocksdb.org/ Last accessed: 27.10.2025
6. Cardellini, V., Lo Presti, F., Nardelli, M., Russo, G. R. (2022). Runtime Adaptation of Data Stream Processing Systems: The State of the Art. *ACM Computing Surveys, 54 (11s),* 1–36. https://doi.org/10.1145/3514496
7. Herodotou, H., Odysseos, L., Chen, Y., Lu, J. (2022). Automatic Performance Tuning for Distributed Data Stream Processing Systems. *2022 IEEE 38th International Conference on Data Engineering (ICDE).* IEEE, 3194–3197. https://doi.org/10.1109/icde53745.2022.00296
8. Venkataraman, S., Panda, A., Ousterhout, K., Armbrust, M., Ghodsi, A., Franklin, M. J. et al. (2017). Drizzle. *Proceedings of the 26th Symposium on Operating Systems Principles.* Association for Computing Machinery, 374–389. https://doi.org/10.1145/3132747.3132750
9. Geldenhuys, M., Pfister, B., Scheinert, D., Thamsen, L., Kao, O. (2022). Khaos: Dynamically Optimizing Checkpointing for Dependable Distributed Stream Processing. *Proceedings of the 17th Conference on Computer Science and Intelligence Systems, 30,* 553–561. https://doi.org/10.15439/2022f225
10. Sun, D., Peng, J., Zhu, T., Kua, J., Gao, S., Buyya, R. (2025). Toward High-Availability Distributed Stream Computing Systems via Checkpoint Adaptation. *Concurrency and Computation: Practice and Experience, 37 (15-17).* https://doi.org/10.1002/cpe.70171
11. Liu, J., Gulisano, V. (2025). On-demand Memory Compression of Stream Aggregates through Reinforcement Learning. *Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering.* Association for Computing Machinery, 240–252. https://doi.org/10.1145/3676151.3719369
12. Wladdimiro, D., Arantes, L., Sens, P., Hidalgo, N. (2024). PA-SPS: A predictive adaptive approach for an elastic stream processing system. *Journal of Parallel and Distributed Computing, 192,* 104940. https://doi.org/10.1016/j.jpdc.2024.104940
13. Hovorushchenko, T., Medzatyi, D., Voichur, Y., Lebiga, M. (2023). Method for forecasting the level of software quality based on quality attributes. *Journal of Intelligent & Fuzzy Systems, 44 (3),* 3891–3905. https://doi.org/10.3233/jifs-222394
14. How to Tune RocksDB for Your Kafka Streams Application (2021). *Confluent*. Available at: https://www.confluent.io/blog/how-to-tune-rocksdb-kafka-streams-state-stores-performance/ Last accessed: 27.10.2025
15. Oh, S., Moon, G. E., Park, S. (2024). ML-Based Dynamic Operator-Level Query Mapping for Stream Processing Systems in Heterogeneous Computing Environments. *2024 IEEE International Conference on Cluster Computing (CLUSTER).* IEEE, 226–237. https://doi.org/10.1109/cluster59578.2024.00027
16. Vysotska, V., Kyrychenko, I., Demchuk, V., Gruzdo, I. (2024). Holistic Adaptive Optimization Techniques for Distributed Data Streaming Systems. *Proceedings of the 8th International Conference on Computational Linguistics and Intelligent Systems. Volume II: Modeling, Optimization, and Controlling in Information and Technology Systems Workshop (MOCITSW-CoLInS 2024).* https://doi.org/10.31110/colins/2024-2/009
17. Dong, S., Kryczka, A., Jin, Y., Stumm, M. (2021). RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Transactions on Storage, 17 (4),* 1–32. https://doi.org/10.1145/3483840
18. Bashtovyi, A. V., Fechan, A. V. (2025). Evaluating fault recovery in distributed applications for stream processing applications: business insights based on metrics. *Radio Electronics, Computer Science, Control, 3,* 17–27. https://doi.org/10.15588/1607-3274-2025-3-2

✉***Artem Bashtovyi,*** *PhD, Assistant, Department of Software, Lviv Polytechnic National University, Lviv, Ukraine, e-mail: artem.v.bashtovyi@lpnu.ua, ORCID: https://orcid.org/0000-0003-4304-8605*

Andrii Fechan, *Doctor of Technical Sciences, Professor, Department of Software, Lviv Polytechnic National University, Lviv, Ukraine, ORCID: https://orcid.org/0000-0001-9970-5497*

✉*Corresponding author*