iKafka: Intelligent Storage Management for Adaptive Event Streaming in Kafka

Yangyang Wang*, Alaa Saleh†, Praveen Kumar Donta‡, Naser Hossein Motlagh*, Lauri Lovén†, Sasu Tarkoma* and Schahram Dustdar§

*Department of Computer Science, University of Helsinki, Finland

†Center for Ubiquitous Computing, University of Oulu, Oulu 90014, Finland

‡Department of Computer Systems and Sciences, Stockholm University, Stockholm 164 25, Sweden

§Distributed Systems Group, TU Wien, Vienna 1040, Austria; and ICREA, UPF Barcelona, Barcelona 08002, Spain

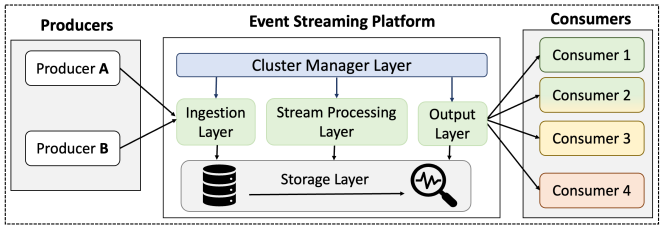
{wang.yangyang,naser.motlagh,sasu.tarkoma}@helsinki.fi,
{alaa.saleh,Lauri.Loven}@oulu.fi, praveen@dsv.su.se, dustdar@dsg.tuwien.ac.at

***Abstract*—Continuous evaluation of Distributed Computing Continuum Systems heavily relies on efficient communication models, with event streaming Publish-Subscribe (Pub-Sub) systems playing a key role in ensuring fault tolerance, scalability, and real-time analytics across heterogeneous tiers. On event streaming platforms like Apache Kafka, consumer applications often exhibit periodic or event-driven patterns when revisiting historical events, which requires storing these events over time. However, accumulating all events increases storage demands. To address this challenge, Kafka's default log retention policies, governed by static time or size thresholds, may prematurely delete data that will be revisited in future cycles. Unfortunately, static time- or size-based retention policies are insufficient, as they fail to maintain equilibrium between resource utilization, cost, and quality of service (QoS). As a result, intelligent and adaptive storage management strategies are required to minimize storage requirements while maintaining enhanced QoS. In this context, we propose a Light Gradient Boosting Machine (LightGBM)-based adaptive storage optimization in event streaming Kafka broker (namely, `iKafka`) to identify periodic consumer patterns and determine near-optimal retention times for events. `iKafka` also considers adversarial attacks by monitoring prediction accuracy to determine whether to use the predicted retention times or revert to default retention times. We evaluate the proposed `iKafka` system with an air quality use case, and our results demonstrate approximately 5.5× less memory resources over traditional Kafka under ideal conditions.**

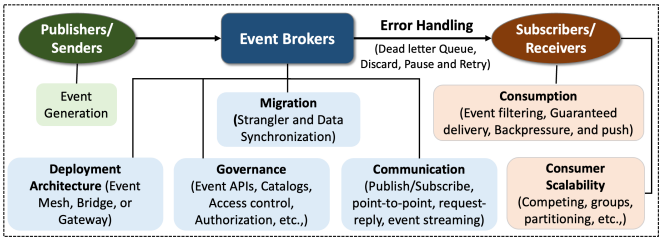
***Index Terms*—Publish/subscribe systems; Kafka broker; event-driven architecture; storage management; light gradient boosting machine; and quality of service**

I. Introduction

DISTRIBUTED Computing Continuum Systems (DCCS) integrate multiple tiers, including wearable embedded devices, Internet of Things (IoT), Edge, Fog, and Cloud infrastructures, working concurrently [1]–[5]. Each of these tiers offers distinct advantages and challenges, contributing to the overall performance and flexibility of the system. Due to its flexibility and superior performance, this technology is widely adopted across various applications, leading to an increase in data and event communications. The Publish-Subscribe (Pub-Sub) model [6], [7] is a widely used messaging paradigm in internet technologies, and it blends seamlessly into DCCS [8],



(a) Streaming platform.



(b) Key patterns of event-driven architecture.

Fig. 1: General Publish/Subscribe communication patterns in the event streaming systems.

enabling efficient, asynchronous communication [9] between different tiers and enhancing real-time data exchange. In this model, publishers generate messages (events) without direct knowledge of their consumers, while subscribers register interest in specific event categories, receiving only relevant data. A broker or middleware (e.g., Kafka, MQTT, AMQP, RabbitMQ) manages message flow, effectively decoupling producers from consumers [10], [11]. The Pub-Sub model offers several key benefits, including scalability, seamless decoupling of producers and consumers, and efficient event-driven communication, making it a preferred choice for applications such as real-time notifications, the IoT, stock market data feeds, and distributed logging systems [12]–[14].

Despite its advantages, the traditional Pub-Sub model has limitations, such as message reliability, message expiry, complex subscription management, and high availability costs [15], [16]. Event streaming (shown in Fig. 1) addresses these

limitations by extending the Pub-Sub model with persistent, replayable event storage and real-time processing capabilities [17], [18]. Unlike traditional Pub-Sub systems, where messages are transient and delivered in real time, event streaming platforms store events in distributed logs, allowing subscribers to access historical data as needed. This ensures reliable message processing, even if a subscriber temporarily disconnects, eliminating data loss [19]. In the literature, there are several event streaming platforms, such as Apache Kafka[1] [20], Apache Pulsar[2], and Amazon Kinesis[3], enable fault tolerance, scalability, and real-time analytics, making them essential for large-scale, real-time applications. While event streaming pub/sub systems offer significant advantages, it's important to note that they may introduce their own complexities, such as the need for proper configuration and maintenance.

For example, storing each event in an event-streaming architecture provides significant benefits, including flexibility, the ability to backtrack events for analysis, and a more reliable data processing solution [21], [22]. However, it also introduces multiple challenges, such as increased storage requirements, retrieval latency, maintenance complexity, etc., [6], [17]. As the volume of stored events grows over time, the system must allocate and manage substantial storage resources, which can lead to higher operational costs. Additionally, most event-streaming architectures employ replication strategies to enhance fault tolerance and data availability, further amplifying storage demands. Additionally, retrieving specific events from large data streams can introduce latency, particularly when querying historical data across distributed nodes in a computing continuum. Moreover, maintaining an event-streaming system requires robust mechanisms for data retention policies, log compaction, and efficient indexing to balance performance and storage efficiency. Furthermore, ensuring system scalability while managing event persistence can be complex, especially in high-throughput environments where millions of events are processed per second. In the literature, Chaves et al. [23] introduced federated learning for collaborative model training, enabling multiple devices or clients to work together while keeping their data decentralized and secure. Calderon et al. [24] evaluated in real use cases, highlighting performance factors in Edge Nodes, Data Streaming, Cloud Servers, and Search Engines. However, these works does not address resource limitations or retention policies within Kafka.

Considering the challenges mentioned above, particularly the increasing memory requirements for storing streaming events, we introduce **iKafka**, an intelligent storage optimization strategy for adaptive event streaming in the Kafka architecture. The primary goal of **iKafka** is to achieve equilibrium between resource usage, cost, and Quality of Service (QoS). To accomplish this, we employ a widely-used machine learning approach, the Light Gradient Boosting Machine (LightGBM) model [25], [26], to analyze and predict Kafka consumers' consumption patterns, focusing on their periodic revisiting behavior to estimate message retention times. Recognizing the potential risks associated with real-time training accuracy, we implement a robust strategy to minimize any impact on **iKafka**'s overall performance. For example, if LightGBM prediction accuracy exceeds 90%, messages are assigned to topics based on the predicted retention categories, with retention policies adjusted to the near-optimal predicted retention period. However, if accuracy falls below 90%, the system defaults to the standard retention policy to maintain reliability and consistency.

In this context, the key contributions of **iKafka** are summarized as follows:

- Initially, we monitor and analyze the periodic behaviors of Kafka consumers by examining their logs. This analysis helps identify event reuse patterns, providing valuable insights into the frequency with which Kafka stores and retrieves events.
- Next, we predict consumer access patterns based on historical data using the LightGBM algorithm. The primary features of LightGBM–scalability, efficiency with limited memory, and speed in handling large datasets—make it ideal for **iKafka**.
- Further, we extend our framework to estimate near-optimal event retention times, aiming to optimize memory usage in Kafka by controlling resource allocation. This ensures more efficient storage management.
- Finally, we evaluated **iKafka**'s performance using a real-time air-quality monitoring use case, comparing it against traditional Kafka retention policies. We also accounted for adversarial attacks, monitoring prediction accuracy to determine whether to apply the predicted retention times or revert to the default Kafka settings.

Our results shows that **iKafka** reducing memory consumption by up to 5.51× compared to traditional Kafka retention policies under ideal conditions without compromising QoS.

II. Research gaps and Problem definition

Consumer applications using event streaming platforms (e.g., Apache Kafka) may exhibit periodic or event-driven patterns in revisiting historical logs. For example, a financial institution may periodically reprocess transaction logs for a weekly fraud detection cycle, while an e-commerce platform may revisit user activity data during a monthly sales audit. These use cases generate recurring demands for data in specific time windows, often aligned with business rhythms such as financial quarters or regulatory timelines.

Kafka's default log retention policies, which are governed by static *time* or *size* thresholds, may prematurely delete data that will be revisited in future cycles. Conversely, overly conservative retention settings can unnecessarily increase storage requirements, cascading into multiple interconnected issues. For example, as the volume of events increases, the system must allocate and manage substantial storage resources. This growing storage demand requires additional infrastructure to support the expanding data. As a result, operational costs

[1]https://kafka.apache.org/
[2]https://pulsar.apache.org/
[3]https://aws.amazon.com/kinesis/

rise, affecting hardware (servers, storage devices), energy consumption, and system management overhead. Event retrieval, particularly historical data across distributed nodes, introduces latency. This latency can degrade system performance, causing delays in processing data requests. To mitigate this, a more powerful computing infrastructure or advanced data retrieval strategies, such as caching, may be necessary, which will drive up operational costs. Increasing resource utilization, such as allocating more storage, improving indexing, or enhancing computational power, can improve QoS by reducing retrieval latency and ensuring real-time event availability. However, this improvement comes at a higher cost, necessitating significant infrastructure investment, which may not always be justifiable.

Conversely, cost reduction through limited storage or computational resources can degrade QoS, leading to increased latency, potential data loss, or decreased system reliability. Focusing on QoS improvements, such as implementing redundant data replication and high-speed retrieval mechanisms, further amplifies cost and resource consumption. For example, higher replication factors ensure fault tolerance and data availability but demand additional storage and processing power, increasing operational expenses. Similarly, strategies aimed at optimizing low-latency event retrieval, such as indexing and caching, improve QoS but require additional resources, further raising costs. The inter-dependencies between *cost*, *resources*, and *quality* can be presented as a three dimensional Cartesian Space [2]. To intelligently navigate the competing factors of cost, resource utilization, and quality, adaptive retention and resource management strategies are essential.

The proposed **iKafka** system addresses these challenges by striking a *balance between the right to drop and the need to keep*, allowing the discarding of unnecessary events while retaining critical insights. It excels at determining optimal retention policies for event-driven systems, where data must often be retained for specific periods aligned with business cycles. However, retaining excessive data or over-provisioning resources can quickly escalate operational costs. In this context, **iKafka** addresses the following research questions:

- **RQ1**: How can machine learning-based prediction models, like LightGBM, be incorporated to predict event revisiting patterns, and how can these predictions influence dynamic retention policies to optimize storage while minimizing costs?
- **RQ2**: What are the trade-offs between retaining events for future access and the cost implications of resource consumption, and how can a system balance these trade-offs without compromising QoS?
- **RQ3**: How can adaptive data retrieval strategies to be integrated into event streaming platforms to reduce latency and improve system reliability without significantly increasing infrastructure costs?

III. Methodology

In this section, we discuss our proposed **iKafka** and its integration of the LightGBM model to uncover and analyze periodic revisiting behaviors among consumers. By aggregating metadata, such as consumer offsets, access timestamps, and frequently consumed events/data, LightGBM enable to identify recurring patterns. These patterns may include weekly or daily batch jobs that reprocess historical events or data for compliance checks. In general, when using Kafka, certain scenarios may arise where historical backtracking becomes necessary. In such cases, consumers may need to revisit past events that were not originally retained under default policies.

- If a Kafka *consumer crashes or restarts* due to issues such as sensor malfunctions, hardware failures, or network jitter, some messages may remain unprocessed. In such cases, the consumer must reprocess these messages.
- If critical records, such as transaction details or user profiles, are *lost from a downstream database* while Kafka retains the historical data, it becomes necessary to retrieve and replay messages from Kafka to restore the missing information.
- In various real-time streaming applications, data analysts often need to *retrieve historical data over a specific period for analysis*, model training, or reprocessing. For instance, in financial markets, maintaining unaltered and chronologically accurate data is crucial for training predictive models. Similarly, in IoT streaming environments, analytical algorithms are continuously refined, requiring access to original data for recalibration and validation.

From the observations above, storing events becomes essential; however, determining how long each event or data should be retained remains a critical question. As previously discussed, retaining all events or messages can lead to several issues, making the calculation of optimal retention times necessary. To address this challenge, **iKafka** provides a solution for managing retention times to optimize memory usage in real-world applications. The framework operates is predicated as shown in Fig. 2. First, consumer logs for a given period are collected, and the messages are analyzed by examining the frequency of their offsets within each Kafka partition. Next, the framework checks if these consumption patterns exhibit periodic characteristics, such as recurring consumption at specific intervals (e.g., weekly, daily). If periodic patterns are identified, the retention time for each message is calculated. This is done by subtracting the timestamp of the producer's initial message production from the timestamp of the consumer's last message consumption.

We illustrate the process by plotting and calculating the change in retention time over time to identify any periodic fluctuations in message consumption patterns. To better understand this, we present an idealized periodic data flow, which serves to illustrate the characteristics of a periodic retention-time pattern. As shown in Fig. 3, we analyze the consumption behavior of four distinct consumers, each displaying unique periodic retention-time patterns.

Next, we apply the Fast Fourier Transform (FFT) to analyze the time windows of each consumer's consumption pattern, followed by peak detection to identify any underlying periodic

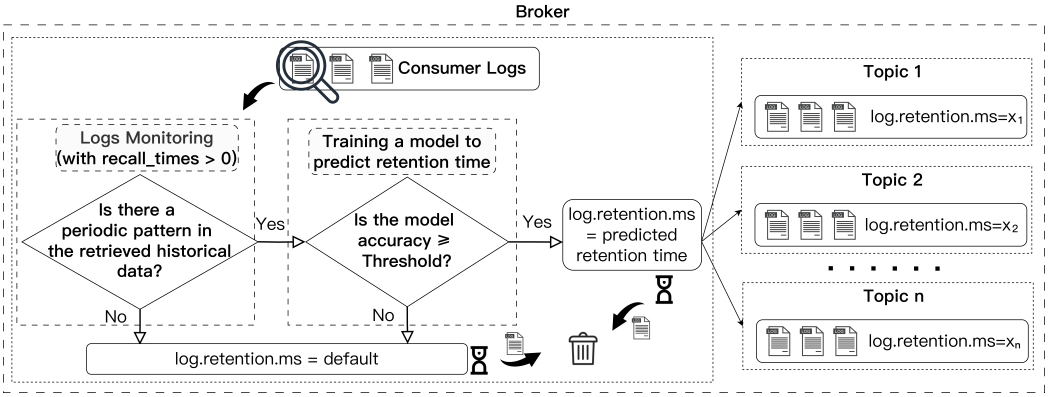


Fig. 2: **iKafka** system event log retention process.

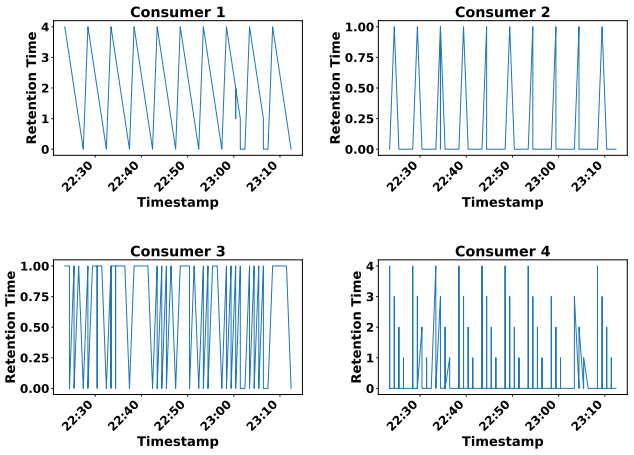


Fig. 3: Periodic retention time for each of the four **iKafka** consumers, with retention time measured in minutes.

TABLE I: Period estimation results for retention time.

Consumer	FFT (s)	Peak detection (s)
Consumer 1	294.14	300.00
Consumer 2	294.14	300.00
Consumer 3	294.14	60.00
Consumer 4	294.14	300.00

patterns. The quantitative results of the FFT analysis and peak detection are presented in Table I. It reveal distinct periodic patterns in consumer consumption behavior. For instance, Consumer 3 exhibits a one-minute peak in retention time, as it reviews data from the previous minute every minute, resulting in a consistent one-minute retention pattern for the first four minutes of each cycle. In contrast, Consumers 1, 2, and 4 demonstrate a clear five-minute periodicity. These cyclic consumption patterns across the consumers allow us to predict the retention time for each message.

Given the cyclical nature of the `iKafka` consumer's historical backtracking behavior, we employ LightGBM techniques to predict the retention time for each message and assess the model's prediction accuracy. The primary reason for selecting LightGBM is its suitability for predicting message retention times and estimating the likelihood of messages adhering to similar patterns in subsequent cycles. Since this is a regression problem, LightGBM is an ideal choice due to its ability to efficiently handle large-scale data. Its faster training speed, lower memory consumption, and excellent scalability in processing big data streams make it particularly well-suited for deployment in large-scale applications [27].

To train the model, we split the dataset into a training set (70% of the data) and a test set (30% of the data). The retention time for each topic is determined by the longest predicted retention time among all messages within that topic. To account for potential delays, processing variations, or unexpected consumption patterns, the retention time is generally set slightly higher than the maximum predicted value. For instance, if messages with predicted retention times of 20 and 50 minutes are grouped under one topic, a retention time of one hour may be assigned to that topic, and messages will be deleted once this retention period expires. Before applying the newly predicted retention times, the model's prediction accuracy is validated. If the accuracy exceeds a predefined `threshold` (in our case, 90%), the new retention times are applied to each Kafka topic and partition. It is worth noting that the threshold can be adjusted based on use case requirements. A 90% threshold is commonly regarded as a high-confidence level in industrial and ML practices. In more sensitive domains such as finance or healthcare, this threshold may be set to 95% or higher, while use cases with greater tolerance for error may accept thresholds as low as 80–85%.

If the accuracy falls below this `threshold`, the framework defaults to Kafka's original retention policies to ensure system reliability. When the buffer is full, messages with the highest prediction accuracy are prioritized for deletion, if necessary. This strategy allows Kafka to manage its retention buffer more effectively, reducing unnecessary storage usage while ensuring that essential data remains available for future retrieval.

IV. Experiments and Results

A. Experiments

1) Consistent periodic consumer patterns across all rounds: For the evaluation of **iKafka** system, we abstracted four

typical scenarios for Kafka's historical backtracking to conduct the experiment setup. For the scenarios requiring periodic and historical data retrieval, we designed and tested **iKafka** system consisting of one real-time data consumer alongside four specialized consumers employing distinct consumption strategies. The details of these scenarios, including the actions for each consumer, are summarized in Table II.

We created seven producers, each producing one type of air quality variable such as CO or $PM_{2.5}$. Each variable type contains 10^4 random measurements. These seven producers run in parallel whereas each handled by a separate process. For each consumer, we defined a five-minute cycle per round to represent a specific time scope and repeated this pattern for ten rounds to mimic cyclical and repetitive consumption behaviors. While we use air quality data as a case study, our system and retention prediction method are domain-agnostic and apply to any setting with repetitive or temporally correlated access patterns, such as finance, consumer behavior, or high-frequency trading.

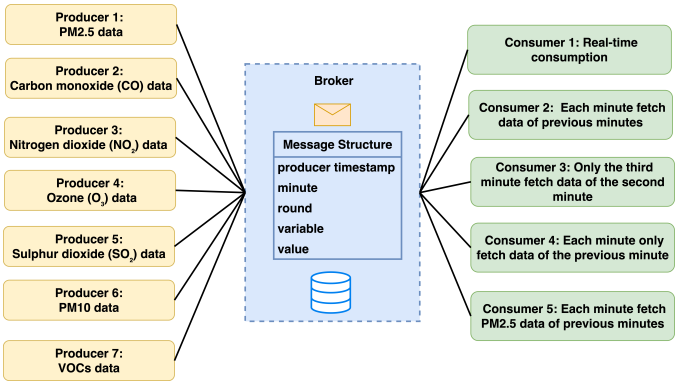


Fig. 4: **iKafka**-based air quality data streaming and backtracking system.

Fig. 4 illustrates the overall process of air quality data production by seven producers and consumption by five consumers. To process these consumers concurrently, we utilize five separate processes, each assigned to a unique consumer group with a distinct group ID. Since all five consumers read from the same partition, the message offset remains identical across consumers. This setup enables us to track which messages are consumed in real-time and later revisited by backtracking consumers. While the real-time consumer is processing data, it records both the beginning and ending offsets for each minute, as well as all offsets for the air quality data (e.g., $PM_{2.5}$). These recorded offsets are then used by the other four consumers to retrieve messages from the same partition. In this setup, the consumers record the message content, which includes the production timestamp, the round, the time sequence (at the minute level), the air quality variable, and the measured value of the air quality variables. Additionally, the consumers fetch the offset and record the corresponding timestamp (received time) on the consumer side. After running the data production and consumption process for 10 rounds, we gather the records for each consumer.

We track the recall frequency of each message by monitoring its offset. When a message's offset matches that of the real-time consumer, it signifies that the message has been accessed by one or more of the four historical backtracking consumers. In this way, we determine the retention time for each message based on records collected from each round.

Fig. 5 depicts the results for retention time for each consumer, obtained from the scenarios presented in Table II. Notably, some retention times in Fig. 5 appear as fractional values (e.g., 0.8 or 0.6). This occurs because retention time is calculated as the difference between the exact final backtracking consumption time (e.g., 2025/3/3 11:24:02) and the original production time (e.g., 2025/3/3 11:22:44). As a result, data that arrives later within this time window will have a shorter retention time. To address this, we compute the mean retention time for each time sequence. Additionally, we observe that the maximum retention time for each consumer category aligns with the predefined consumer actions. For instance, Consumer 1 backtracks the complete history for all previous minutes every minute. From Consumer 1's heatmap, we can see that if we round up all retention times, the longest retention time in the first minute is 4 minutes, gradually decreasing by one minute per round.

Afterward, we use the time sequence (in minutes) and variable type as features to train a LightGBM model that predicts the retention time for each category, as depicted in Fig. 6. From the results, we observe that the predicted retention times align with our predefined values. For instance, for Consumer 1, the maximum retention time in the first minute is four minutes and decreases by one minute per time sequence, matching the expected consumption pattern. For Consumer 2, only data from the second minute is retained for one minute, and the model correctly captures this behavior. For Consumer 3, data from all four initial minutes has a retention time of one minute, which is accurately reflected in the model's predictions. As for Consumer 4, only $PM_{2.5}$ data is retained. However, the model's accuracy for Consumer 4 is relatively low, leading to less precise predicted retention times, as shown in the predicted heatmap.

Fig. 7 illustrates the accuracy of the predicted retention times for consumers. The results show that when following an ideal historical data consumption pattern, retention times can be accurately predicted for Consumers 1, 2, and 3. However, for Consumer 4, the accuracy of the predicted retention time for certain variables falls below 90%, which may result in approximately 13% of the data being retrieved again after the predicted retention time expires. Therefore, in this case, deleting messages based on the predicted retention time is not advisable. The lower accuracy for Consumer 4 is primarily due to the simultaneous generation of different variables by all seven producers every minute. Consequently, the producer timestamp and the received timestamp for PM2.5 data within the same minute can vary significantly. For example, at the beginning of a minute, a consumer might process 5,000 PM2.5 messages, and by the end of that minute, another 5,000 $PM_{2.5}$ messages might be consumed. These two groups may have

TABLE II: Comparison of different data retention scenarios. Each scenario presents actions for each consumer.

Scenario (Consumer)	**Description**	**Trigger Condition**	**Data Scope**	**Example**
Full Retention (Consumer 1)	Continuously revisits all historical logs every minute	Every minute	The logs of previous minutes from start of the round to the current minute	At `00:01:00` fetch the data from `00:00:00` to `00:01:00`; at `00:02:00` fetch the data from `00:00:00` to `00:02:00`
Certain Part Retention (Consumer 2)	Revisit logs only when errors or a task occur in a time window	When an error or task is detected	Logs are revisited only during the error or task period	There were errors or tasks at `00:03:00`, data is always revisited from `00:02:00` to `00:03:00`
Regular Retention (Consumer 3)	Revisit previous minute's logs every minute	Every minute	Always revisits logs from the most recent minute	At `00:01:00` fetch the data from `00:00:00` to `00:01:00`; at `00:02:00` fetch the data from `00:01:00` to `00:02:00`
Certain Key Retention (Consumer 4)	Revisit only specified data (e.g.,$PM_{2.5}$) every minute.	Every minute	Only previous minutes $PM_{2.5}$ data	At `00:01:00` fetch the $PM_{2.5}$ data from `00:00:00` to `00:01:00`; at `00:02:00` fetch the $PM_{2.5}$ data from `00:00:00` to `00:02:00`

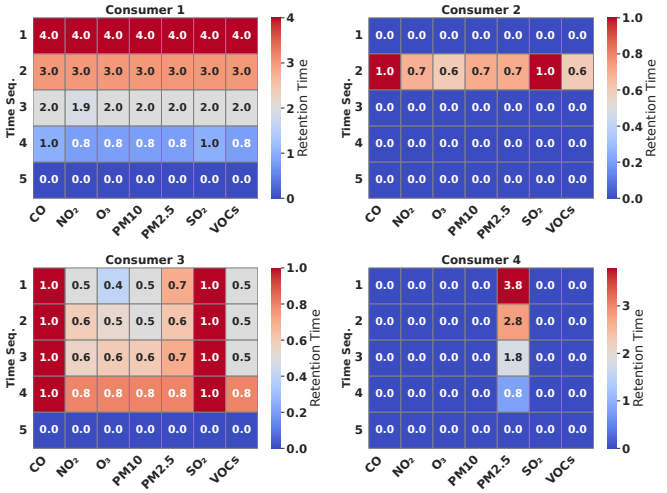


Fig. 5: Heatmap of message retention time across consumers. "Time Seq." denotes the time sequence (in minutes), and "Retention Time" is defined as the interval between when the producer creates the data and when the consumer performs its last write operation (in minutes).

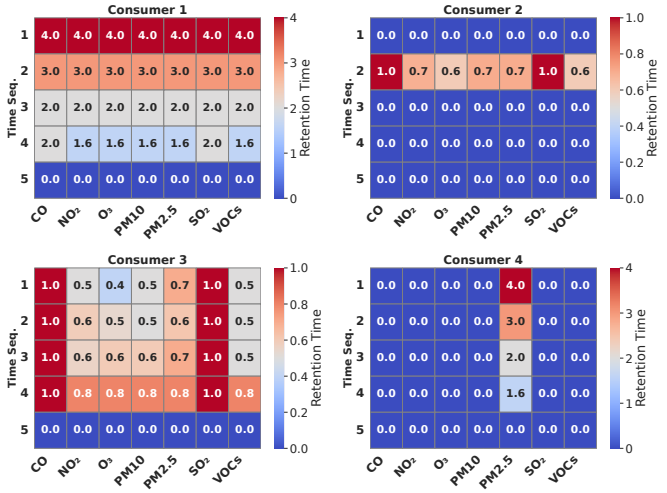


Fig. 6: Predicted heatmap of message retention time across consumers, follow the rules for Fig. 5.

different retention times, introducing variations that negatively impact prediction accuracy.

2) Intermittent periodic consumer patterns in select rounds: If all rounds of retention times follow a consistent cyclical revisit pattern, we can easily use the predicted retention time to determine which data should be deleted when memory is full. However, real-world scenarios are often unpredictable. In some cases, users do not revisit data in every round; instead, they may retrieve data only in specific rounds, often randomly, due to errors or urgent data needs. To address this, we introduce four additional consumers that retrieve data at random intervals and apply the same model to predict retention times, even when data is not revisited in every round. The actions of these four consumers are as follows:

- **Consumer 1:** backtracks the complete history for all previous minutes every minute at round 1, 3, 5 and 7.
- **Consumer 2:** at minute 3, backtracks the historical data for minute 2 at round 1, 3, 4, 5 and 7.
- **Consumer 3:** backtracks the previous minute's data at rounds 1, 2, 3, 4, 7, 8, and 9.
- **Consumer 4:** backtracks the historical $PM_{2.5}$ data for the preceding minute every minute at round 1 to round 9.

Next, we estimate R^2 scores to evaluate how well our trained model fits the intermittent periodic consumer patterns (i.e., the newly introduced consumers) shown in Fig. 8. Our observations found that the model cannot fully capture the

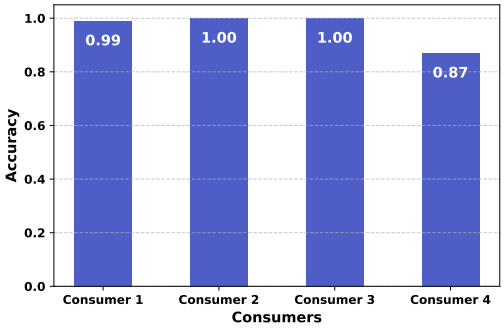


Fig. 7: The accuracy of the retention time prediction.

consumption patterns across different consumer models, as Consumers 1, 2, and 3 exhibit low R^2 scores. However, Consumer 4, which continues to follow its previous behavior over nine rounds, till maintains moderate accuracy in predicting retention time.

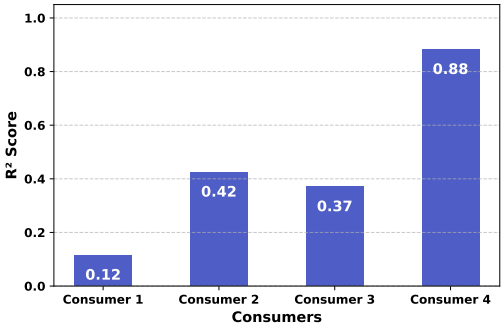


Fig. 8: R^2 scores for bootstrap configuration retention times.

We further generate the actual retention time heatmap (shown in Fig. 9) for the intermittent periodic consumer patterns. In this, we observe that the mean retention time across all rounds is reduced compared to the results obtained for consistent periodic consumer patterns across all rounds (refer to Fig. 5). In the consistent periodic consumer patterns, Consumer 1 exhibited retention times of approximately four, three, two, and one minute(s) for each time sequence (in minutes). However, in the original retention time heatmap for intermittent periodic consumer patterns (Fig. 9), when only four rounds follow the same revisit pattern, the mean retention time is approximately 40% of what was observed when running for 10 rounds. Similarly, for Consumer 2, the mean retention time is about 50%; for Consumer 3, it is approximately 70%; and for Consumer 4, it is around 90%.

Next, we generate a heat map of the predicted retention times (shown in Fig. 10) for intermittent periodic consumer patterns. The prediction results illustrate that the model struggles to accurately predict retention times for rounds without revisiting behaviors. As a result, the predicted mean retention time is higher than the actual mean retention time. In the absence of a clear periodic pattern, retention times that should

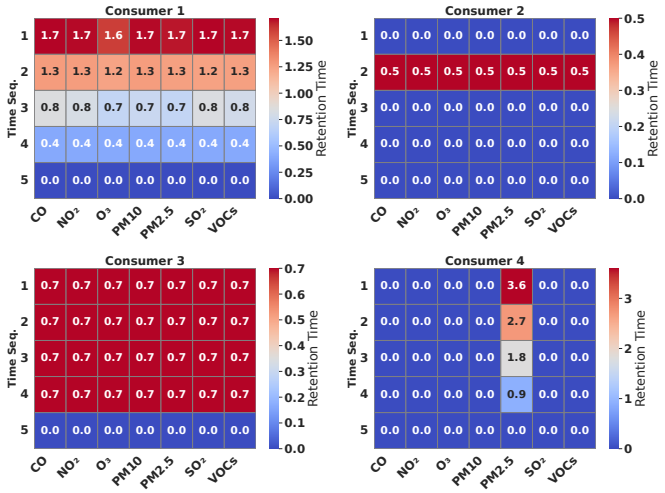


Fig. 9: Heatmap of message random round configuration retention time across consumers, (follow the rules for Fig. 5).

be zero are incorrectly predicted as non-zero. For example, in the case of Consumer 2, where only 50% of the rounds (1, 3, 4, 5, and 7) follow the revisit pattern, the model mistakenly predicts a retention time of one minute for all rounds. Moreover, the model occasionally predicts a retention time of zero when a non-zero value is expected, potentially resulting in premature data deletion.

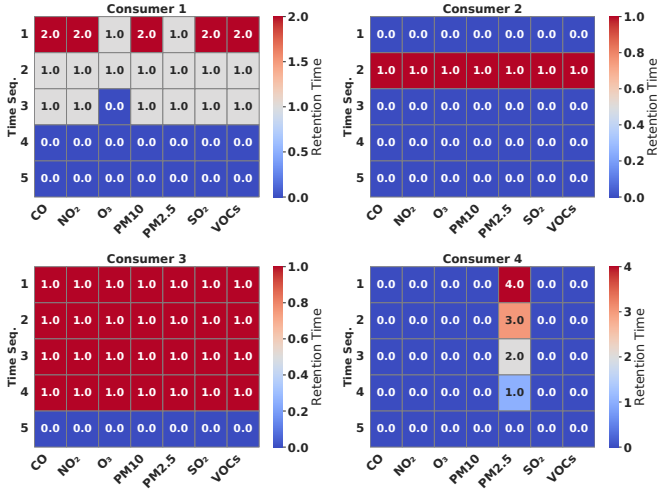


Fig. 10: Heatmap of message random round configuration retention time across consumers.

In summary, for the intermittent periodic consumer patterns observed in select rounds, our **`iKafka`** system designed to handle message retention in real-world scenarios, showed that the model's accuracy did not reach 90%. As a result, these messages will be placed in a topic with retention based on Kafka's default retention policies. In the next subsection, we will test memory usage under our predicted retention time policy using the retention times predicted for Consumer 1,

Consumer 2, and Consumer 3.

3) Memory usage evaluation based on predicted retention time policies: After predicting the retention time for each message, we categorized the messages into five groups, each with its own retention time, the producer will put these messages into five topics according to the retention time policy of messages considering the time sequences (in minutes). To prevent messages from being deleted before consumers process them, in our implementation design, we added one extra minute to each retention time. Furthermore, in the Kafka broker's settings, we set the `log.retention.check.interval.ms` parameter to one minute. This parameter defines the interval (in milliseconds) at which Kafka scans logs to identify and remove messages that have exceeded their retention period. This means that the system checks every minute whether any messages have reached their retention time and need to be deleted. However, if a segment is active (i.e., data is still being written into the segment), the deletion of the data in the segment will be delayed even if the retention time has been reached. To ensure that messages in each topic are deleted when approaching to the messages assigned retention time, in the broker we configured the topic-level segment settings as follows: `log.segment.ms` = 3×10^4 and `log.segment.bytes` = 2^{23}. Our predicted retention times of each topic from 1–5 are 1 minute, 2 minutes, 3 minutes, 4 minutes and 5 minutes, respectively.

This configuration ensures that segments roll over quickly and become inactive once the specified time or maximum storage size is reached, allowing historical data to be cleared according to the retention policy. Once these settings are configured, we conduct experiments to evaluate memory usage. In this scenario, there are four consumers on the consumer side, while the producer side consists of seven producers that send messages to different topics based on the predicted retention time of each message. In the followings, we explain the behavior of each producer and consumer:

Consuming pattern of Consumer 1: The maximum predicted retention time for messages in the first minute of each round is 4 minutes. As explained earlier, all retention times need to be increased by one minute. Hence, we send these messages to the topic with a retention time of five minutes. Similarly, the messages from the second minute are sent to the topic with a retention time of 4 minutes, and the messages from the 5^{th} minute are sent to the topic with one minute retention time. Consumer 1 backtracks messages from five topics according to the retrieval behavior (explained in Table II), while the real-time consumer consumes messages from all five topics simultaneously.

Consuming pattern of Consumer 2: The maximum predicted retention time for messages in the second minute of each round is one minute. Thus, these messages are sent to the topic with a retention time of two minutes. All other messages are sent to the topic with a retention time of one minute. Consumer 2 backtracks messages from these two topics following the retrieval behavior (explained in Table II), while the real-time consumer consumes messages from five topics simultaneously.

Consuming pattern of Consumer 3: The maximum predicted retention time for messages from the first to the fourth minute of each round is one minute. Therefore, these messages are sent to the topic with a retention time of two minutes. All other messages are sent to the topic with a retention time of one minute. Consumer 3 backtracks messages from these two topics following the retrieval behavior (explained in Table II), while the real-time consumer consumes messages from five topics simultaneously.

Default retention policy: To get the comparison memory, we also created one real-time consumer following the default retention policy the retention time is 24 hours and producers just send messages to the default retention time topic.

After each round, the producers sleep for two minutes to ensure that all messages from the final minute are deleted. Then, the process continues with the next round of experiments, repeating for a total of 10 rounds.

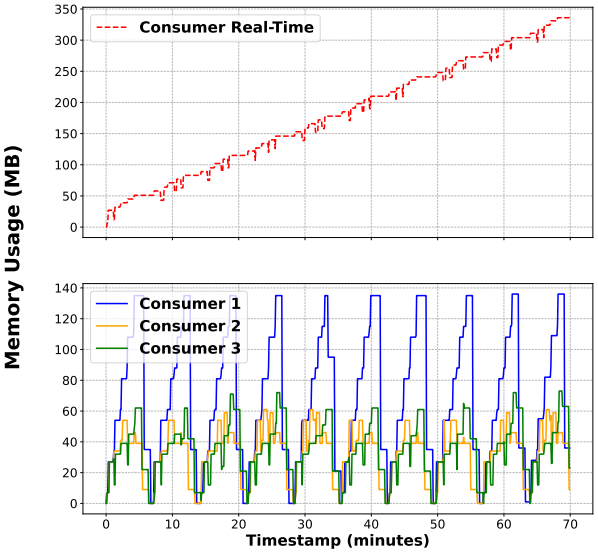


Fig. 11: Memory usage comparison: model-retention vs. default policy (default policy: 2.47×, 5.51×, 4.60× vs. Consumers 1, 2, 3).

B. Performance Analysis of Memory Usage

In the consistent periodic consumer patterns across all rounds, we measured the memory usage for the broker. For Consumers 1, 2, and 3, which achieve up to 99% accuracy, we calculate their memory usage based on the predicted retention time policy. This means that all messages are deleted within one minute after their predicted retention time expires. In addition, we compare this with the idealized memory usage for a real-time consumer following the default retention time policy. Fig. 11 depicts how memory usage compares between these consumers.

From the figure, we observe that memory usage under the default retention policy is approximately 2.47× higher than

that of Consumer 1, 5.51× higher than that of Consumer 2, and 4.60× higher than that of Consumer 3 at peak usage. In memory usage experiment (presented in previously), we assume that all messages are deleted within one minute after their retention time expires. However, in practical scenarios, retention times should be categorized according to the number of topics, with each topic storing messages that have different maximum retention times. For instance, a message with a retention time of 50 minutes might be placed in the same topic as a message with a retention time of 30 minutes, with both set to a retention time of one hour. As a result, messages with a predicted retention time of 30 minutes will be stored for an additional 30 minutes, leading to higher memory usage.

Our experiment involved seven producers, each sending 10,000 messages per minute, repeated over 10 rounds, with each round lasting 5 minutes. Under these conditions, we observed that memory usage under the default retention policy was up to 5.51× higher than using our model-predicted retention times, which is also the maximum times (×) among three consumers. In real-world big data scenarios, this difference in memory usage could be even greater. Therefore, we can conclude that, for long-term, large-scale data management, our approach offers an effective solution for adjusting **iKafka**'s retention time for periodically retrieved data streams, significantly reducing memory usage.

V. Discussion

Based on the performance of our model, we have developed a method to address the data preservation challenge in Kafka data streams. This approach enables the retention time to be more effectively set, ultimately optimizing Kafka's memory usage across various scenarios. In our idealized experiment, this method resulted in a memory savings of approximately 5.51×, which is particularly advantageous in high-concurrency big data environments. Furthermore, we proposed a general strategy for handling random data that does not exhibit periodic patterns, broadening the applicability of our approach.

A. Limitations, Open challenges and Future works

While **iKafka** offers benefits for efficient data transmission and storage, there are also some associated limitations. First, due to the lack of mechanisms to handle adversarial attacks, we cannot fully guarantee accuracy under such conditions. As a result, we rely on the default settings to maintain system stability. In the future, we aim to enhance the system so that it becomes fully adaptable and no longer dependent on default settings. Secondly, Kafka topics are designed to organize data based on themes or categories rather than retention times. Therefore, if we were to divide topics based on both categories and retention times, it could lead to inefficient memory usage if not managed properly. There are a few more possibilities to improve adaptive storage management in **iKafka** as outlined below:

1) Sub-topics: In general, Kafka is efficient in topic-based partitioning, but not necessarily for sub-topics [28]. Since Kafka topics are primarily defined by business logic, if different data lifecycles are required within the same context, it is advisable to split the topics into smaller sub-topics. This approach allows for applying tailored policies to each sub-topic while maintaining overall business logic. AI/ML can assist by analyzing event context to automatically identify patterns, determine the optimal number of sub-topics, predict when data becomes less relevant, and enable dynamic adjustments [29].

2) Compression: Kafka's built-in support for compression formats like gzip, snappy, and LZ4 helps reduce storage requirements and memory usage by compressing log data, but they are computationally intensive [30]. Integrating AI/ML into Kafka can enhance efficiency by enabling context-aware compression, where the system intelligently adjusts compression based on factors such as resource availability (e.g., network bandwidth). AI can also monitor system performance in real time and dynamically adjust compression settings to maintain optimal throughput and latency [29].

3) Scalability under Large-Scale Producers and Consumers: In general, **iKafka** inherits Kafka's strong scalability properties, as different brokers and partitions can be distributed across separate physical servers. Similarly, **iKafka** enable to scale horizontally with minimal risk of resource bottlenecks at the messaging layer. However, the adaptive retention mechanism introduces a prediction component. If not carefully designed, the centralized execution of prediction tasks (e.g., metadata aggregation, model inference) could become a scalability bottleneck. We extend this feature in **iKafka** in future while considering offloading prediction to distributed edge nodes.

4) Large Language Models (LLMs) or Generative AI (GenAI): can play a key role in processing and understanding events in a more intelligent way, reducing the need to store all information [6]. Instead of retaining vast amounts of raw event data, LLMs can be used to analyze and summarize key insights, patterns (based on understanding the context of events), and trends in real time. This enables Kafka or similar event streaming systems to focus on storing essential data while discarding redundant or less relevant information, ultimately reducing storage requirements and improving system efficiency. In addition, LLMs can enhance prediction analytics by providing real-time recommendations and insights based on historical event data.

5) Noisy and Adversarial Attack Handling: To mitigate potential adversarial attacks that might degrade prediction accuracy (e.g., due to poisoned metadata or anomalous consumer behaviors), we will implement a mechanism to track the deviation between predicted retention durations and actual message access patterns over time. To further enhance security, **iKafka** plans to restrict access to retained messages, ensuring only authorized consumers can retrieve them.

VI. Conclusions

This paper presents **iKafka**, an intelligent and adaptive storage management system designed to optimize memory

usage in Apache Kafka by predicting consumer access patterns. **iKafka** employs a Light Gradient Boosting Machine model to analyze consumer logs, identify recurring data access patterns, and enable dynamic retention policies. This brings into focus the delicate balance between *the right to be forgotten and the need to be remembered*, where unnecessary events can be discarded, while retaining essential insights. Further, it achieve equilibrium in memory usage–minimizing resource requirements while maximizing efficiency–alongside cost optimization and maintaining QoS by dynamically adjusting retention times based on predicted usage. Experimental results using air quality data demonstrate significant memory savings—up to 5.51× under ideal conditions—compared to Kafka's default retention policies. Looking ahead, further advancements can enhance **iKafka** system, including integrating novel compression techniques, adopting LLMs for intelligent event summarization and retrieval, and enhancing real-time learning capabilities for even more efficient storage management. We will also evaluate our enhanced version by simultaneously connecting multiple applications, handling a large number of publishers and subscribers under resource-constrained environments.

ACKNOWLEDGMENT

This project is partially funded by the Research Council of Finland (Grant Numbers 345008 and 362594) and 6G Flagship (Grant Number 369116), and by Business Finland through the Neural Pub/Sub research project (Diary Number 8754/31/2022). Also, partially supported by the *Svenska Institutet* under the 'SI Baltic Sea Neighbourhood Programme 2024' (Project No. 31005669). We would also like to express our gratitude to Mr. `Alexander Knoll` from the Distributed Systems Group, TU Wien, Austria, for his invaluable support and for promptly providing the computing resources essential for our experiments.

REFERENCES

[1] D. Katare, E. Marin, N. Kourtellis, M. Janssen, and A. Y. Ding, "ARASEC: Adaptive resource allocation and model training for serverless edge–cloud computing," *IEEE Internet Computing*, vol. 28, no. 6, pp. 17–27, 2024.

[2] S. Dustdar, V. C. Pujol, and P. K. Donta, "On distributed computing continuum systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 4, pp. 4092–4105, 2022.

[3] T. Meuser, L. Lovén, M. Bhuyan, S. G. Patil, S. Dustdar, A. Aral, S. Bayhan, C. Becker, E. De Lara, A. Y. Ding *et al.*, "Revisiting edge ai: Opportunities and challenges," *IEEE Internet Computing*, vol. 28, no. 4, pp. 49–59, 2024.

[4] P. K. Donta, B. Sedlak, I. Murturi, V. Casamayor-Pujol, and S. Dustdar, "Human-based distributed intelligence in computing continuum systems," *IEEE Internet Computing*, vol. 29, no. 2, 09 2025.

[5] P. K. Donta, I. Murturi, V. Casamayor Pujol, B. Sedlak, and S. Dustdar, "Exploring the potential of distributed computing continuum systems," *Computers*, vol. 12, no. 10, p. 198, 2023.

[6] A. Saleh, S. Tarkoma, S. Pirttikangas, and L. Lovén, "Publish/subscribe for edge intelligence: Systematic review and future prospects," *Available at SSRN 4872730*, 2024.

[7] P. K. Donta and S. Dustdar, "Towards intelligent data protocols for the edge," in *2023 IEEE International Conference on Edge Computing and Communications (EDGE)*. IEEE, 2023, pp. 372–380.

[8] L. Lovén, M. Bordallo López, R. Morabito, J. Sauvola, S. Tarkoma *et al.*, "Large language models in the 6G-enabled computing continuum: a white paper," 2025.

[9] J. Lee, A.-H. Lee, V. Leung, F. Laiwalla, M. A. Lopez-Gordo, L. Larson, and A. Nurmikko, "An asynchronous wireless network for capturing event-driven data from large populations of autonomous sensors," *Nature Electronics*, vol. 7, no. 4, pp. 313–324, 2024.

[10] D. Kreković, P. Krivić, I. Podnar Žarko, M. Kušek, and D. Le-Phuoc, "Reducing communication overhead in the IoT-edge-cloud continuum: A survey on protocols and data reduction strategies," *Internet of Things*, p. 101553, 2025.

[11] P. K. Donta, S. N. Srirama, T. Amgoth, and C. S. R. Annavarapu, "Survey on recent advances in IoT application layer protocols and machine learning scope for research directions," *Digital Communications and Networks*, vol. 8, no. 5, pp. 727–744, 2022.

[12] N. Pavlopoulou and E. Curry, "PoSSUM: An entity-centric publish/subscribe system for diverse summarization in internet of things," *ACM Trans. Internet Technol.*, vol. 22, no. 3, Mar. 2022. [Online]. Available: https://doi.org/10.1145/3507911

[13] S. Tarkoma, *Publish/subscribe systems: design and principles*. John Wiley & Sons, 2012.

[14] P. Bellavista, A. Corradi, and A. Reale, "Quality of service in wide scale publish—subscribe systems," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 3, pp. 1591–1616, 2014.

[15] C. Esposito, D. Cotroneo, and S. Russo, "On reliability in publish/subscribe services," *Computer Networks*, vol. 57, no. 5, pp. 1318–1343, 2013.

[16] L. Lovén, R. Morabito, A. Kumar, S. Pirttikangas, J. Riekki, and S. Tarkoma, "How can ai be distributed in the computing continuum? introducing the neural pub/sub paradigm," *arXiv preprint arXiv:2309.02058*, 2023.

[17] A. Saleh, R. Morabito, S. Tarkoma, S. Pirttikangas, and L. Lovén, "Towards message brokers for generative ai: Survey, challenges, and opportunities," *arXiv preprint arXiv:2312.14647*, 2023.

[18] J. Vestin, A. Kassler, S. Laki, and G. Pongrácz, "Toward in-network event detection and filtering for publish/subscribe communication using programmable data planes," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 415–428, 2020.

[19] C. Esposito, M. Platania, and R. Beraldi, "Reliable and timely event notification for publish/subscribe services over the internet," *IEEE/ACM Transactions on Networking*, vol. 22, no. 1, pp. 230–243, 2013.

[20] T. P. Raptis and A. Passarella, "A survey on networked data streaming with apache kafka," *IEEE access*, 2023.

[21] J. Phuttharak and S. W. Loke, "An event-driven architectural model for integrating heterogeneous data and developing smart city applications," *Journal of Sensor and Actuator Networks*, vol. 12, no. 1, p. 12, 2023.

[22] L. Lazzari and K. Farias, "Uncovering the hidden potential of event-driven architecture: A research agenda," *arXiv preprint arXiv:2308.05270*, 2023.

[23] A. J. Chaves, C. Martín, and M. Díaz, "Towards flexible data stream collaboration: Federated learning in Kafka-ML," *Internet of Things*, vol. 25, p. 101036, 2024.

[24] G. Calderon, G. del Campo, E. Saavedra, and A. Santamaría, "Monitoring framework for the performance evaluation of an IoT platform with elasticsearch and apache kafka," *Inf. Syst. Front.*, pp. 1–17, 2023.

[25] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "LightGBM: A highly efficient gradient boosting decision tree," *Advances in neural information processing systems*, vol. 30, 2017.

[26] B. Quinto, *Next-generation machine learning with spark: Covers XGBoost, LightGBM, Spark NLP, distributed deep learning with keras, and more*. Apress, 2020.

[27] L. Developers, "Experiments: LightGBM vs XGBoost," LightGBM Documentation, n.d., accessed: 2025-03-02. [Online]. Available: https://lightgbm.readthedocs.io/en/latest/Experiments.html#baseline

[28] T. P. Raptis, C. Cicconetti, and A. Passarella, "Efficient topic partitioning of Apache Kafka for high-reliability real-time data streaming applications," *Futur. Gener. Comp. Syst.*, vol. 154, pp. 173–188, 2024.

[29] H. Kokkonen, L. Lovén, N. H. Motlagh, A. Kumar, J. Partala, T. Nguyen, V. C. Pujol, P. Kostakos, T. Leppänen, A. González-Gil *et al.*, "Autonomy and intelligence in the computing continuum: Challenges, enablers, and future directions for orchestration," *arXiv preprint arXiv:2205.01423*, 2022.

[30] S. Deshmukh, "Message compression in Apache Kafka," IBM, 2021, accessed:2025-03-07. [Online]. Available: https://developer.ibm.com/articles/benefits-compression-kafka-messaging/