Optimizing Real-Time E-Commerce Data Streams: A Comparative Analysis of Serialization and Topic Design Patterns in Apache Kafka

Mengdi Wei, Yifan Li, Yiyi Wang, Zijian Cao, Michal Aibin
Khoury College of Computer Sciences, Northeastern University, Vancouver, BC, Canada
vancouver@northeastern.edu

***Abstract*—As online shopping grows in popularity, electronic commerce has become an essential aspect of people's daily lives. The volume of data generated by customers' interactions with e-commerce platforms has significantly increased. Consequently, analyzing real-time data has become increasingly crucial. In this paper, we propose a set of performance metrics for analyzing real-time data streaming from e-commerce using Apache Kafka pipelines, considering several design considerations such as serialization selection and topic design patterns.**

***Index Terms*—kafka, e-commerce, data streams**

I. Introduction

Apache Kafka is a widely used distributed event streaming platform that stores, processes, and analyzes large amounts of real-time data. It is a good choice for developing real-time data pipelines and streaming applications, which can be made even more flexible through dynamic topic design. The platform supports different serialization frameworks, such as Avro and Protocol Buffers, which can affect performance depending on the data schema. Plenty of previous research papers have presented frameworks for real-time data processing using Kafka and its architecture. For example, in [1], the author explains the model of using Kafka in data streaming. Also, article [2] discusses the partition design in Kafka and how it improves real-time data processing. However, choosing the best serialization strategy and topic design pattern can be challenging, especially for companies that handle large data streams and build customer data platforms. Serialization and design patterns are crucial for Kafka's design, but their combination for optimal results has yet to receive much research attention [3], [4]. Moreover, ensuring the security of data streams is crucial, as highlighted in the overview of security challenges in communication networks [5]. We aim to fill the gap by evaluating various serialization formats and design patterns for performance. We thoroughly reviewed articles in Google Scholar that contained the keywords 'dynamic', 'performance' 'static' and were published after Kafka's initial release in 2011 and found no prior studies on this topic. Identifying the best combinations is essential for optimizing Kafka's use, as the right choice can significantly impact the scalability and performance of Kafka-based applications, thus improving real-time data processing and analysis.

This research aims to address the current research gap by better understanding how to select a serialization strategy and topic design patterns in the context of Apache Kafka pipelines. We compare serialization strategies such as Avro and Protobuf and different topic design patterns such as dynamic and single topic design patterns. Also, we use CPU, memory, storage, and data throughput metrics as performance metrics to determine the best Kafka configurations in an e-commerce application scenario for chain restaurants. In this case, the findings of this research can have important implications for various industries, including the retail industry, freight and marine shipping, finance and other industries that require large-scale live data processing [6].

TABLE I
Pipeline Configuration Matrix

	Single Topic Pattern	**Dynamic Topic Pattern**
Avro	Scenario 1 (Benchmark)	Scenario 3
Protobuf	Scenario 2	Scenario 4

II. Problem Statement

We analyze the cost and performance trade-off between two topic design patterns and serialization methods. We compare two topic design patterns: Dynamic and Single topic patterns. Also, we indicate the difference between Avro and Protobuffor serialization in Kafka. We create four combinations of comparisons as seen in Table I and test these different scenarios on Kafka.

A. Initial Pipeline Design

We start the project by building a pipeline using the single topic design pattern (Figure 1). First, we have various Points of Sale (POS) to provide live purchase data. In this design, we configure the Kafka Cluster using only one topic, order, to process streaming data. An internal kSQL component processes those data to generate the evolving table, which can be used later for further data analysis.

After building up the single topic design pattern, we can change it into a dynamic topic design (Figure 2) in different ways. First, we can change the configuration of the Kafka Cluster so that it would use multiple topics (Topic A, B, and C). On the other hand, if we choose to keep the same settings for the Kafka Cluster, we can keep processing the evolving table (Topic X) filtered by the internal kSQL initially by

979-8-3315-2096-0/25/$31.00 ©2025 IEEE

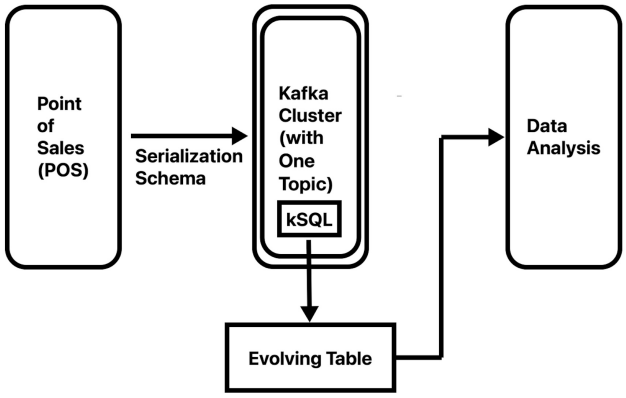

Fig. 1. Single Topic Design Pattern

adding more kSQL components to generate new tables (Topic Y) as many as desired. In this paper, we will use the first method for dynamic topic design.

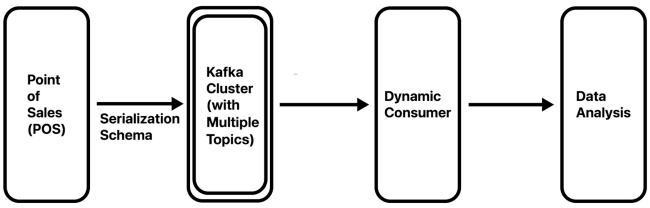

Fig. 2. Dynamic Topic Design Pattern

B. Sample Data Record

This section shows the sample data record of Kafka and how Kafka serialized the message.

In the Kafka Producer API, a ProducerRecord object with a key-value pair format is created when data is passed to the Producer. The constructor of ProducerRecord is as follows.

```
ProducerRecord(String topic, K key, V value)
```

For example, the data passed to Kafka is as follows:

```
{
    "user_id": 001,
    "user_name": "John"
}
```

The Kafka producer will create a ProducerRecord object as follows:

```
User user = record.value();
ProducerRecord<String, User> =
        new ProducerRecord<>("user",
        record.key(), user);
```

After the Producer creates the ProducerRecord, it is sent to the serializer, where it is converted to serialized bytes for further processing. As mentioned above, this paper will implement two different serialization methods and compare their performance.

C. Performance Metrics

In this article, we compare the efficiency of different strategies concerning computing resources, based on [7], [8]. We set up various metrics to measure the performance of different strategies on Kafka.

1) CPU Usage: it measures what percentage of the total server capacity is being utilized, and it is a critical metric to measure the pipelines' overall performance and resource utilization.

$$CPU\ Usage = \frac{Kafka\ \ CPU\ \ Time}{Total\ \ CPU\ \ Time} * 100 \quad (1)$$

2) Memory Percentage: it reports the memory stats of the server by showing the percentage of server utilization in memory.

$$Memory\ Percentage = \frac{Kafka\ Memory\ usage}{Total\ Memory\ Limit} * 100 \quad (2)$$

3) Disk Usage: different topic design patterns and serialization approaches will affect the storage usage in the server. This factor is important for testing the two topic design patterns in this project since the dynamic topic design pattern will create more topics and perhaps use more storage on the machine than kSQL to query data on a single topic.

$$Disk\ Usage = \sum_{1}^{n} disk\ usage \quad (n = Total\ number\ of\ brokers) \quad (3)$$

4) Data Throughput: We focus on two data throughput rates: the data throughput rate on Kafka producer and the throughput rate on Kafka consumer. The throughput rate is defined by the amount of data processed in kilobits(Kbs) per second on the server. We use the average throughput rate in a minute to indicate the speed at which data is transmitted or processed.

$$Data\ Throughput = \frac{Data\ Processed\ in\ Kbs}{Second} \quad (4)$$

5) Latency: This measures the time the system takes to respond to the request. Similarly, we consider the latency of Kafka producers and consumers. This metric is measured in milliseconds (ms) and uses the average over a minute to indicate the delay or responsiveness of the system.

III. Framework Design

Let us discuss the pipeline design:

1) Prepare the Data: The first step is defining what data will be transmitted through the pipeline. In this project, we use the order information as the data passed from the Point of Sales. Each order sample will include the

ID, credit card, order details with an array of items, timestamp and store location information.

We have created a separate Java project, POSClient, to randomly generate this order data and store the data in a local file. It randomly generates various numbers of order information with five different stores and ten different order items, and details will be demonstrated in the simulation section. Reading from the file will simulate collecting data from POS machines. This dataset is used for all pipelines to test their performance and computing cost with the same input.

2) Set expected data analytic result: All four pipelines will process and analyze the same input data and get the same expected result using kSQL (in single topic design) or consumer layer (in dynamic topic design). In this project, we want to know which store has the highest sales with the order information from POS.
3) Set up the topic: A static topic is created for processing data in a single topic pattern. In a dynamic topic pattern, topics are created automatically during the processing of a message. The appropriate configuration settings are defined in this step.
4) Create a producer: The producer will encode the data into the binary format and send it to the appropriate topic. Different serialization formats are applied in this step to test the performance.
5) Create a consumer: The consumer will read data from the pipeline, decode the binary format, and process the data for real-time data analysis.
6) Use kSQL to generate evolving tables (in Single Topic Pattern only): The kSQL could help generate the real-time analytic result based on different requirements, such as finding the customer who has the highest purchase amount or which store has the highest sales. We will use consumers to process messages directly and pass data for dynamic topic patterns for further analysis.
7) Maintain the pipeline: The pipeline must be maintained to ensure it functions correctly over time. This includes tasks such as upgrading software, scaling the pipeline to handle increased volume, and troubleshooting issues that arise.
8) Measure the performance of the pipeline: To ensure optimal performance of the Apache Kafka pipeline, it is essential to monitor several key performance metrics such as CPU usage, memory percentage, disk usage, data throughput, and latency. These metrics can be monitored using various system monitoring tools, Kafka's built-in and other specialized monitoring tools. By measuring these metrics regularly, we could identify any performance bottlenecks or issues affecting the pipeline and take corrective action to improve performance and reliability.

IV. Algorithms Design

In the two single topic design pipelines, we use "orders" as the Kafka topic and send data to this topic. Please see Figure 3 for the single topic pipeline.

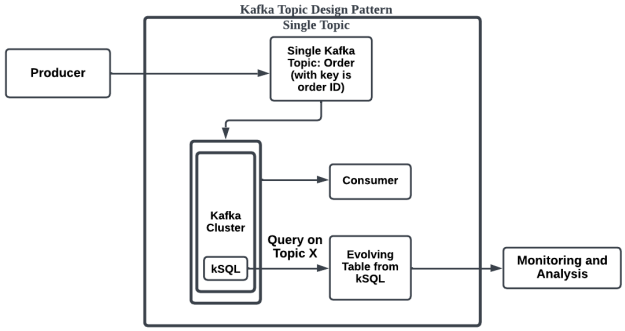

Fig. 3. Pipeline for Single Topic Design Pattern with different serialization

The Kafka producers in this pipeline are responsible for sending data to Kafka topics. They also communicate with Schema Registry to register the schema for different serialization formats. The data is serialized using the serialization schema and sent to the corresponding topic.

The Kafka consumers in this pipeline are responsible for reading data from Kafka topics. The data is deserialized using the serialization schema and processed by the consumer.

In this pipeline, we analyze the streaming data using kSQL. The following two sections discuss the two serialization methods with the Single Topic Pipeline.

A. Single Topic Design with Avro

Avro offers several advantages over other serialization methods. One major advantage is its speed and ability to produce smaller serialized data compared to commonly used methods like JSON. Additionally, compared with Protobuf, Avro supports real-time compilation without the need to define a data format file before compiling and using it. This makes Avro a more efficient and user-friendly option for serialization.

The serialization process of Avro in Kafka involves encoding data into a compact binary format to transmit it between Kafka producers and consumers. This process starts with defining a schema that specifies the structure of the data being serialized. The schema is then used to generate code representing the data in a language-agnostic format, enabling interoperability between different programming languages. When data is serialized, it is converted into a compact binary format based on the schema, which allows for efficient storage and transmission. When a consumer receives the data, it uses the schema to deserialize the binary format back into its original form.

The consumer must have access to the same schema used for serialization, typically stored in a schema registry. When the consumer receives the binary format, it uses the schema to convert the data to its original form. The deserialization process is performed by first reading the header of the serialized data to determine the schema ID. The consumer then retrieves the corresponding schema from the schema registry and uses it

to deserialize the data. The deserialized data is then available for further processing by the consumer.

B. Single Topic Design with Protobuf

Using Protobuf as a serialization format is very similar to using Avro in the pipeline. In this project, we are using proto3 and the following protobuf schema for the order information from the POS client.

Each field has a unique number from the schema, which will be added to the binary format to represent the fields and match byte sequences to fields. However, when we use the Avro serialization format with schema registry, no schema or fields are added to the binary message. Therefore, using Protobuf serialization creates a larger message than using Avro.

Encoding messages to Protobuf in this pipeline is also handled by the schema registry module. In the producer, it connects with the schema registry and sends the local Protobuf schema for registering. If a schema exists for the current topic "orders", it updates the schema with auto-increment id. After that, the schema registry converts the message to Protobuf format using the current schema and sends the encoded message with the schema ID to the Kafka broker. When the consumer poll message from the topic, it first looks for schema in the schema registry module with the schema ID. After getting the schema, the consumer will map the field name with the unique number and create the Java Object based on that.

C. Dynamic Topic Design with Avro

In this section, we discuss the algorithm design for the Kafka pipeline with Avro serialization with a dynamic topic design pattern. Instead of using "orders" in single topic design, we create topics dynamically based on attributes selected for analysis in point of interest. Kafka Producer will redirect messages to corresponding topics with Avro serialization. Below is the algorithm in steps, which differs in topic selection from the Single Topic design. When there is a message with an attribute from a new topic, we utilize the AutoTopicCreating feature in the Confluent Kafka platform, and a new topic will be created corresponding to the attribute. Figure 4 demonstrates this design's workflow.

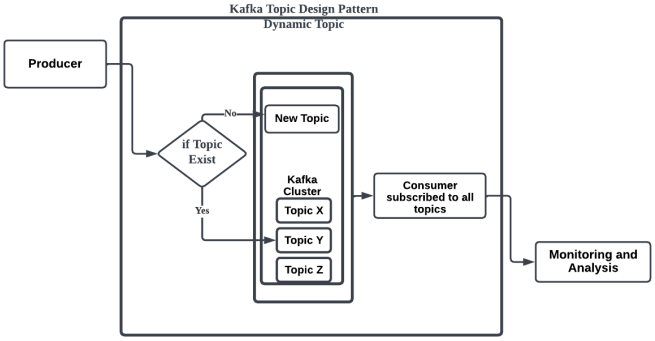

Fig. 4. Pipeline for Dynamic Topic Design Pattern with different serialization

The consumer will subscribe to all topics related to the point of interest, process the messages, and then provide the data to further layers for visualization and demonstrations. Since the topics will be created dynamically, the consumers will also be responsible for subscribing to new topics if they are detected. The dynamic consumer will, therefore, check all topics from Kafka and filter them to include topics only desired by the point of interest. Then, the messages will be processed and passed to further layers for analysis and visualization.

D. Dynamic Topic Design with Protobuf

Last, we discuss the algorithm design for the Kafka pipeline with Protobuf serialization and dynamic topic design pattern. Similar to the dynamic topic design with Avro above, we also create topics dynamically based on attributes selected for analysis, but with Protobuf serialization. The dynamic Kafka Producer will redirect messages to corresponding topics (different location IDs in this pipeline), and different consumers would consume the messages from each topic (store location), accordingly.

We check the terminal output of different consumers after the producer begins sending messages to the Kafka server. Each consumer deserializes the messages with Protobuf and then prints out the desired fields. A sample output row from the terminal will include the store ID, key values, orderID, and credit card information.

In the following sections, we will analyze the throughput, Latency, CPU usage, and Disk usage for all four pipeline settings. Additionally, we will visualize the result from a sample point of interest, the store location, with SuperSet. One thing to notice is that the pipelines no longer use SQL to query the output data for the dynamic topic designs as they did in the static topic design pattern.

V. Simulation Plan

1) Experiment Setup: We setup the same environment for the experiments for all four pipelines. The same environment includes the following:

1) Run the pipelines on one device: MacBook Air with Apple M2 chip, 16GB memory and 1TB SSD is selected for running Docker containers, and the virtual machines on Docker with 4 CPUs, 8 GB memory and 64 GB disk limits. This setup ensures the computing resources for all pipelines are the same.
2) Same Kafka cluster setup: Since the number of brokers and the topic partition setup will affect the performance of Kafka, we use the same Kafka setup for all scenarios. All clusters have 1 broker with 1 partition for the topic. Also, to eliminate the potential effects of cloud service latency, the integrated pipelines are deployed locally via Docker instead of running on Confluent Cloud.
3) Same structure of data source: Even though the data size is variable for the experiment, the data structure is identical for all datasets. This will mitigate the risk of inaccuracy caused by using various data structure.

In the experiments, the parameters are the data size and four different pipelines. We plan to use three different data sizes, 100,000, 250,000 and 500,000 messages, to test the performance and analyze the result. A general rule of thumb for choosing a test dataset size is to ensure the size is large enough to make a difference from the background but not too large compared to the calculating power. Here are the specific reasons why we chose this dataset size in this paper:

1) Testing scalability: The chosen data sizes allow for testing how the system's performance scales with increasing amounts of data, providing insights into how the system will perform as the data volume grows over time.
2) Covering a range of use cases: Different applications have different data processing requirements. Testing with different data sizes allows for understanding the performance across various use cases, which is particularly relevant for a fast-food chain [9] like Dairy Queen with many locations and a wide range of products.
3) Comparison with industry benchmarks: The chosen data sizes are commonly used in the industry [10]–[12], and using them allows for easy comparison with other systems that have been tested using the same data sizes, providing insights into how the system compares to industry benchmarks.
4) Striking a balance between accuracy and feasibility: Choosing data sizes of 100,000, 250,000, and 500,000 messages strikes a balance between providing enough data to accurately test the system's performance [13] while also being feasible within the project's constraints.

All scenarios are summarized in the Table II.

TABLE II
PIPELINE CONFIGURATION MATRIX

	Single Topic	Dynamic Topic
Avro (w/ 100,000 messages)	Scenario 1	Scenario 2
Protobuf (w/ 100,000 messages)	Scenario 3	Scenario 4
Avro (w/ 250,000 messages)	Scenario 5	Scenario 6
Protobuf (w/ 250,000 messages)	Scenario 7	Scenario 8
Avro (w/ 500,000 messages)	Scenario 9	Scenario 10
Protobuf (w/ 500,000 messages)	Scenario 11	Scenario 12

VI. RESULTS AND DISCUSSION

In this section, we discuss the results from the experiment of all 12 pipelines. There are 3 different sizes of datasets and the results are as follows. Table III shows the computing resource usage for the 4 pipelines with 100,000 messages. The following tables (Table IV and V) show pipeline performance with the metrics of producer and consumer with throughput rate and latency.

TABLE III
PIPELINE RESULTS WITH 100,000 MESSAGES

	CPU Usage (%)	Disk Usage (MB)
Avro w/ Single	32.5	9.19
Avro w/ Dynamic	44.5	9.71
Protobuf w/ Single	46.25	13.63
Protobuf w/ Dynamic	30	7.9

TABLE IV
PRODUCER RESULTS WITH 100,000 MESSAGES

	Throughput (KB/s)	Latency (ms)
Avro w/ Single	299.72	16
Avro w/ Dynamic	294.42	43
Protobuf w/ Single	314.45	14
Protobuf w/ Dynamic	309.35	15

TABLE V
CONSUMER RESULTS WITH 100,000 MESSAGES

	Throughput (KB/s)	Latency (ms)
Avro w/ Single	300.33	580
Avro w/ Dynamic	292.31	530
Protobuf w/ Single	312.88	520
Protobuf w/ Dynamic	307.61	510

Table VI shows the computing resource usage for the 4 pipelines with 250,000 messages. The following tables (Table VII and VIII) show pipeline performance with the metrics of producer and consumer with throughput rate and latency.

TABLE VI
PIPELINE RESULTS WITH 250,000 MESSAGES

	CPU Usage (%)	Disk Usage (MB)
Avro w/ Single	42.5	28.89
Avro w/ Dynamic	23.75	31.21
Protobuf w/ Single	36.25	12.39
Protobuf w/ Dynamic	23	51.8

TABLE VII
PRODUCER RESULTS WITH 250,000 MESSAGES

	Throughput (KB/s)	Latency (ms)
Avro w/ Single	726.79	51
Avro w/ Dynamic	721.32	27
Protobuf w/ Single	765.37	46
Protobuf w/ Dynamic	761.65	35

TABLE VIII
CONSUMER RESULTS WITH 250,000 MESSAGES

	Throughput (KB/s)	Latency (ms)
Avro w/ Single	727.68	540
Avro w/ Dynamic	721.96	630
Protobuf w/ Single	765.04	515
Protobuf w/ Dynamic	765.43	580

Table IX shows the computing resource usage for the 4 pipelines with 500,000 messages. Finally, Tables X and XI show pipeline performance with the metrics of producer and consumer with throughput rate and latency.

TABLE IX
PIPELINE RESULTS WITH 500,000 MESSAGES

	CPU Usage (%)	Disk Usage (MB)
Avro w/ Single	28.75	117.12
Avro w/ Dynamic	40	64.83
Protobuf w/ Single	39.5	110.54
Protobuf w/ Dynamic	43.75	102.76

TABLE X
PRODUCER RESULTS WITH 500,000 MESSAGES

	Throughput (KB/s)	Latency (ms)
Avro w/ Single	1400	10
Avro w/ Dynamic	1410	15
Protobuf w/ Single	1480	10
Protobuf w/ Dynamic	1470	9

TABLE XI
CONSUMER RESULTS WITH 500,000 MESSAGES

	Throughput (KB/s)	Latency (ms)
Avro w/ Single	1410	530
Avro w/ Dynamic	1410	530
Protobuf w/ Single	1470	510
Protobuf w/ Dynamic	1480	660

A. Discussion

From the data the authors collected above, it is reasonable to believe that different configurations of the pipelines have noticeable impacts on data throughput for producers and consumers without significant drawbacks from other aspects, such as CPU usage and latency. Such throughput differences may exist due to serialization schemas (i.e. Avro vs. Protobuf) and topic design patterns (i.e. Single vs. Dynamic). Therefore, we filter the results by calculating the relative differences. The average throughput from the pipelines using the Protobuf serialization schema is higher than those pipelines using the Avro serialization methods by around five percent. On the other hand, the average throughput from the pipelines using the dynamic topic design pattern is not necessarily better than that using the static topic design pattern. However, increasing the number of messages could potentially increase the pattern.

VII. CONCLUSIONS

We successfully implemented four pipeline scenarios and conducted simulation samples for testing. Additionally, our simulation showed a valuable result in Apache Kafka performance analysis. To conclude, we observed a throughput improvement of approximately 5% from the Protobuf serialization format over Avro. This improvement is relatively stable for different test sample data sizes. We also discovered there is a positive correlation between the throughput increase of dynamic topic design patterns as the test sample data size increases for both serialization formats. Additionally, we confirmed the positive relation of throughput over sample data size for all simulation scenarios.

Future research could explore traffic prediction, leveraging machine learning techniques to further enhance performance. This approach has been effectively utilized in various networking contexts, as demonstrated in [14].

REFERENCES

[1] B. R. Hiraman, C. Viresh M., and K. Abhijeet C., "A study of apache kafka in big data stream processing," in *2018 International Conference on Information , Communication, Engineering and Technology (ICICET)*, 2018, pp. 1–3.

[2] B. Leang, S. Ean, G.-A. Ryu, and K.-H. Yoo, "Improvement of kafka streaming using partition and multi-threading in big data environment," *Sensors*, vol. 19, no. 1, 2019. [Online]. Available: https://www.mdpi.com/1424-8220/19/1/134

[3] M. Miloradović and A. Milovanović, "Data streaming architecture based on apache kafka and github for tracking students' activity in higher education software development courses," *E-business technologies conference proceedings*, vol. 2, no. 1, p. 132–136, Jun. 2022.

[4] A. Akanbi, "Estemd: A distributed processing framework for environmental monitoring based on apache kafka streaming engine," in *Proceedings of the 4th International Conference on Big Data Research*, ser. ICBDR '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 18–25. [Online]. Available: https://doi.org/10.1145/3445945.3445949

[5] M. Furdek, L. Wosinska, R. Goścień, K. Manousakis, M. Aibin, K. Walkowiak, S. Ristov, M. Gushev, and J. L. Marzo, "An overview of security challenges in communication networks," in *2016 8th International Workshop on Resilient Networks Design and Modeling (RNDM)*, 2016, pp. 43–50.

[6] N. K. K. Kit and M. Aibin, "Study on high availability and fault tolerance," in *2023 International Conference on Computing, Networking and Communications (ICNC)*, 2023, pp. 77–82.

[7] M. Aibin and K. Walkowiak, "Resource requirements in fixed-grid and flex-grid networks for dynamic provisioning of data center traffic," in *2016 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2016, pp. 1–4.

[8] M. Aibin, M. Kantor, P. Boryło, H. Niedermayer, P. Chołda, and T. Braun, "Resilient sdn, cdn and icn technology and solutions," in *Guide to Disaster-Resilient Communication Networks*, ser. Computer Communications and Networks, J. Rak and D. Hutchison, Eds. Springer, Cham, 2020, pp. 621–650.

[9] P. C. Patel, E. M. Struckell, and D. Ojha, "Calorie labeling law and fast food chain performance: The value of capital responsiveness under sales volatility," *Journal of Business Research*, vol. 117, pp. 346–356, 2020.

[10] B. Leang, S. Ean, G.-A. Ryu, and K.-H. Yoo, "Improvement of kafka streaming using partition and multi-threading in big data environment," *Sensors*, vol. 19, no. 1, p. 134, 2019.

[11] A. Akanbi and M. Masinde, "A distributed stream processing middleware framework for real-time analysis of heterogeneous data on big data platform: Case of environmental monitoring," *Sensors*, vol. 20, no. 11, p. 3166, 2020.

[12] H. Lv, T. Zhang, Z. Zhao, J. Xu, and T. He, "The development of real-time large data processing platform based on reactive micro-service architecture," in *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, vol. 1. IEEE, 2020, pp. 2003–2006.

[13] Y. Fu and C. Soman, "Real-time data infrastructure at uber," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2503–2516.

[14] M. Aibin, K. Walkowiak, S. Haeri, and L. Trajković, "Traffic prediction for inter-data center cross-stratum optimization problems," in *2018 International Conference on Computing, Networking and Communications (ICNC)*, 2018, pp. 393–398.