National College of Ireland

# Performance Evaluation of Stateful Serverless Computing using Stream Processing

MSc Research Project
Cloud Computing

Abhishek Medhane
Student ID: X21182787

School of Computing
National College of Ireland

Supervisor: Sean Heeney

**National College of Ireland**
**Project Submission Sheet**
**School of Computing**

| | |
|---|---|
| **Student Name:** | Abhishek Medhane |
| **Student ID:** | X21182787 |
| **Programme:** | Cloud Computing |
| **Year:** | 2023 |
| **Module:** | MSc Research Project |
| **Supervisor:** | Sean Heeney |
| **Submission Due Date:** | 14/08/2023 |
| **Project Title:** | Performance Evaluation of Stateful Serverless Computing using Stream Processing |
| **Word Count:** | 6107 |
| **Page Count:** | 19 |

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

**ALL** internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

| | |
|---|---|
| **Signature:** | |
| **Date:** | 15th September 2023 |

**PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:**

| | |
|---|---|
| Attach a completed copy of this sheet to each project (including multiple copies). | ☐ |
| **Attach a Moodle submission receipt of the online project submission**, to each project (including multiple copies). | ☐ |
| **You must ensure that you retain a HARD COPY of the project**, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer. | ☐ |

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

| **Office Use Only** | |
|---|---|
| Signature: | |
| Date: | |
| Penalty Applied (if applicable): | |

# Performance Evaluation of Stateful Serverless Computing using Stream Processing

Abhishek Medhane
X21182787

**Abstract**

In the evolving landscape of cloud-native technologies, serverless Function as a Service (FaaS) platforms are gaining significant traction. In the field of real-time streaming, it is becoming important to process such big data as the use cases are increasing, ranging from IoT, smart devices, and e-commerce, hence a need for storage and query optimization techniques. In this research work, we delve into the advantages of stateless serverless computing that retains state across real-time streaming invocations. Our objective is to gauge its efficacy in executing complex stateful tasks. Through a comprehensive, measurement-driven analysis focused on real-time streaming, we assess various performance, latency, and throughput metrics. This research focuses on the construction and deployment of Kafka streams and the development of a stateful serverless function in Apache Flink. We aim to design modular stateful functions and migrate them to diverse databases on a unified platform, facilitating efficient storage and latency measurements via diverse deployment strategies, as opposed to local storage. Also, significant work has been done to propose a novel approach and compare the stateful operations using Flink's Python Table APIs.

**Keywords– serverless, FaaS, stateful, streaming, latency, Flink, Kakfa**

# 1 Introduction

With the development of cloud computing and the growing explosion of data sources, the previous generation of computing models continually evolved to manage and process vast amounts of information. To manage such a huge and complex volume of data and to handle such a complex computing model, serverless computing became an emerging concept, focusing on the need to build applications without the overhead of infrastructure management and being scalable. Serverless computing allows us to focus solely on writing code and deploying applications without worrying about server provisioning or maintenance. This model offers increased flexibility and cost efficiency as resources are automatically allocated and scaled based on demand.Carbone et al. (2015)

With the help of serverless computing businesses can now eliminate the worry about availability, provision of VM resources, scalability of infrastructure and allow efficient optimization of cloud resources. When there is a high demand, serverless computing allows parallel executions to be started automatically. Furthermore, it may scale up or down as needed to suit demand. The fundamental benefit of serverless computing is that it eliminates server use from developers and runs code on-demand, automatically scaling and billing for only the time the code is executing. This enables elasticity because

developers are not concerned with the minute aspects of server management. Apache Flink which is open-source, supports both event-time and out-of-order event processing, offers managed state consistency with exactly-once promises, and serves both batch and streaming with high throughput and low latency.

## 1.1 Background

Serverless computing allows us to run individual functions in response to events without managing the underlying infrastructure. These functions, when executed, are stateless by default, meaning they do not retain any memory of prior invocations.

However, many real-time applications require some form of state retention to function effectively, leading to the evolution of stateful serverless computing. Stateful serverless computing is the main method used in our research work, where serverless functions can maintain state or context across multiple invocations. This allows for the preservation of data and enables the functions to remember information from previous executions, making them more suitable for complex applications that require a persistent state. By incorporating stateful serverless computing, we can build applications that are not only scalable and cost-effective but also capable of handling dynamic data and providing personalized experiences to users.

Serverless computing and stream processing are complementary technologies that can be used together to build powerful and scalable applications. Serverless computing can be used to run the stateless functions that are needed to process data streams, while stream processing can be used to manage the state of the application and ensure that data is processed in a timely manner.

## 1.2 Importance of performance evaluation in stateful serverless computing

Performance evaluation in stateful serverless computing is pivotal for us to determine economic efficiency, user experience optimisation, and informed architectural decision-making. It guides us in resource allocation, ensures responsiveness and reliability for enhanced user interactions, and aids in making crucial decisions about state management. Additionally, it prepares systems for scalability challenges, supports continuous iterative improvements, and is essential for security and regulatory compliance, giving us optimised applications.

## 1.3 Research Question and Objectives

**Research Question. [How can we leverage using stateful serverless computing to achieve high performance and reduce the impact of low latency using stream processing?]**.

The surge in data volume and speed necessitates efficient processing solutions, with traditional server architectures often faltering due to scalability constraints. While serverless computing offers a scalable, cost-effective alternative, its inherent stateless design poses challenges for real-time stream processing tasks that require rapid, low-latency data handling. This research aims to focuses on how stateful serverless computing could help improve the performance of stream processing. By incorporating stateful serverless computing, stream processing tasks can benefit from the ability to retain and access

data across multiple function invocations. This allows for faster and more efficient data handling, enabling real-time stream processing to meet the demands of high-speed data volume and low-latency requirements. While exploring the potential of stateful serverless computing in this research could result in improvements in resource allocation optimization and overall system performance for stream processing applications. Also, using powerful visualization tools we can achieve more insights of the big data source.

This paper is organized in the following manner,Section 2 discusses related work on various Stream processing frameworks and its processing time and efficiency. **??** specifies the approach we have defined while doing this research and Section 4 describes the architecture and design specification followed by Section 5 containing the brief implementation. Section 6 critically analyse our work with evaluation results, and finally in section 7 we propose our work and future research area where their is scope for improvement.

# 2 Related Work

Serverless computing revolutionises cloud infrastructure by letting cloud providers handle server and resource management, enabling developers to primarily concentrate on their code. Within this framework, applications are transformed into smaller, event-responsive functions, offering both cost efficiency and expedited development. Expanding upon this, stateful serverless computing introduces memory retention between function calls, which is essential for specific applications that need consistent processing. This statefulness is particularly pivotal in real-time streaming applications. Through mechanisms like databases, caches, or message brokers, these stateful serverless systems can process uninterrupted data streams swiftly, ensuring timely data access and management and thus addressing the challenges posed by massive real-time data volumes.

## 2.1 Overview of the current state of serverless computing and stream processing

Stream processing is recognised for its capability to handle data in real-time rather than deferred database storage; therefore, it is widely used for real-time applications. Yet, the inherent stateful nature of these applications clashes with the traditionally stateless architecture of serverless computing. To overcome this, serverless platforms like Azure Durable Functions and AWS Step Functions have integrated stateful extensions.

In this paper, Shahidi et al. (2021) evaluates the effectiveness of two popular stateful serverless offerings, Azure Durable functions and AWS Step functions, for implementing complex stateful workflows. The paper presents a detailed measurement-driven characterization of the performance and cost trade-offs of these platforms for two different applications: machine learning pipelines (training and inference) and video processing. The results of the evaluation reveal several key insights into the factors that determine the suitability of current serverless applications for stateful workloads.

This Singhvi et al. (2021) authors delves into serverless computing, introducing a novel methodology for a tenant-specific, function-tailored shared data plane, aiming to rectify certain shortcomings observed in present-day Function as a Service (FaaS) platforms. This paper shows rigorous experiments that compare the performance of a serverless platform to how it interacts with APIs. When the Shared Data Plane framework is used, the size of the Docker action image and the time it takes to start up are both reduced by a

lot. A practical application of this method is illuminated through a machine learning case study executed on a serverless infrastructure. The main outcomes from this underline the Shared Data Plane's promise of enhancing the efficiency and adaptability of serverless platforms. It emphasises the strategic advantage of distinguishing between code and data entities in a serverless context and posits that the framework can pave the way for broadening the spectrum of applications feasible in serverless settings.

## 2.2 Hadoop based real time data processing and streaming

The Hadoop-based framework, often cited as an initial structure, has been extensively dissected by various researchers who explore its features through diverse methodologies. Vaidya and Kshirsagar (2020) champion Hadoop's legacy power, considering it a reliable and potent platform for analysis. Their discussion delves into SCADA systems, highlighting their MapReduce capabilities that enable scalability across 4000 nodes for 1000 terabytes of data and efficient failure handling. Interestingly, the authors draw a comparison between Hadoop and traditional databases, suggesting Hadoop's superior efficiency, though they haven't considered other platforms like Apache Spark. This comparative stance also resonates in Jabbar et al. (2020) work, where they juxtapose Hadoop with SQL.

Merla and Liang (2017) identify Hadoop as a pivotal component in data analysis, emphasising its prowess in managing vast datasets using distributed mechanisms. They also discuss the latency-accuracy trade-offs, pointing out the complex procedures that might delay operations and reduce result precision, which is in line with Perez et al.'s (2017) sentiment. Furthermore, Grolinger et al. (2014) discuss challenges surrounding MapReduce's fault tolerance, where intermediate results from the Map phase contribute to overheads, thereby inducing processing latency. In addition to the challenges mentioned, Grolinger et al. (2014) also highlight the issue of load balancing in MapReduce. The uneven distribution of tasks across nodes can lead to bottlenecks and decreased overall performance. This further emphasizes the need for efficient resource allocation and task scheduling algorithms in distributed systems utilizing MapReduce.

Fan et al. (2019) shed light on Hadoop's comprehensive data management capabilities, encompassing storage, queries, and resource management. With the integration of algorithms such as XGBoost and decision trees, they demonstrate Hadoop's efficacy, using a case study of 1500 substations from the Hubei power grid. Their findings suggest a commendable accuracy rate of 93.98% and a marked reduction in failure rate by 75%. However, it's noteworthy that their examination remains solely within Hadoop's purview, with no exploration of alternative architectures or real-time streaming data. While the study showcases Hadoop's effectiveness in handling storage, queries, and resource management, it is important to consider other architectural options and the potential impact of real-time streaming data. Exploring alternative architectures and evaluating Hadoop's performance in real-time scenarios could provide a more comprehensive understanding of its capabilities and limitations. Additionally, comparing its results with other technologies would help determine the most suitable solution for specific use cases beyond the scope of this study.

## 2.3 Spark and Flink based real time data processing and streaming

In the rapidly evolving domain of cloud computing, distributed data processing platforms play a pivotal role, in driving large-scale data analytics. One such critical player, Apache Hadoop MapReduce, has emerged as the predominant choice for numerous enterprises, serving as a benchmark in distributed computing. However, its intrinsic programming nature is often regarded as cumbersome, demanding a considerable amount of groundwork for even rudimentary analytical functions. This intricate operational modality has catalyzed the inception of more intuitive, dataflow-centric platforms, most notably Apache Spark and Apache Flink. Both Spark and Flink, in addition to targeting enhanced performance via optimized in- memory processing, prominently emphasize facilitating users with embedded, advanced data processing utilities. These functionalities, ranging from streamlined filtering to join operators, are postulated to render data analysis endeavors more straightforward compared to the traditional Hadoop MapReduce environment. This paper embarks on a mission to decipher whether these platforms genuinely simplify the user experience or if they merely present a facade of simplicity.Gomes et al. (2015)

To gain a holistic understanding, a meticulous usability study was conducted involving a contingent of students. D'silva et al. (2017) familiarized themselves with the intricacies of all three platforms, attempted to resolve diverse use cases anchored in a data science milieu. Preliminary findings unveiled a discernible inclination towards Spark and Flink, relegating MapReduce to a less preferred position. Interestingly, when juxtaposing Spark and Flink, the differential in terms of user preference and development time proved insignificant, indicating that both platforms might be equally proficient for batch-centric big data operations. In essence, this research endeavour furnishes valuable insights into the user-centric efficacy of Big Data platforms. By shedding light on the usability dynamics of these platforms, it sets the stage for further investigations that might pave the way for more intuitive and user- friendly big data processing tools in the future.Toliopoulos and Gounaris (2020)

In the contemporary digital realm, the burgeoning demand for proficient stream analysis has been a catalyst for the emergence of an array of open-source Streaming Data Processing Systems (SDPSs). These systems are characterized by their unique capabilities and performance attributes, offering distinct solutions to the challenges of stream analysis. Despite some preliminary efforts that attempt to juxtapose these systems under rudimentary workloads, there exists an evident paucity of comprehensive insights into their performance dynamics. Addressing this gap, the research paper under scrutiny endeavors to introduce a systematic framework tailored for the benchmarking of distributed stream processing engines. With the primary intent of performance assessment, this research concentrates on three prominent SDPSs, namely, Apache Storm, Apache Spark, and Apache Flink. The study places a heightened emphasis on evaluating the throughput and latency of windowed operations, recognized as the foundational operations in stream analytics. What amplifies the credibility of this benchmark is its meticulous design, which stems from real-world, industrial paradigms, particularly drawing inspiration from the online gaming industry. The Salloum et al. (2016) contribution to the literature can be demarcated into three pivotal facets. Initially, it elucidates the concepts of latency and throughput specifically for stateful operators, providing a standardized definition that aids in comprehensive analysis. The study further accentuates the significance of distinctly segregating the system under examination and the driver. This

segregation is crucial for authentically mirroring the open-world model inherent in typical stream processing deployments, thereby enabling an evaluation of system performance under genuine, realistic conditions. Lastly, the Akil et al. (2017)research is instrumental in pioneering a benchmarking framework dedicated to delineating and examining the sustainable performance of streaming systems. This research piece illuminates the idiosyncratic features and applicable scenarios of each system, making significant strides in the domain of streaming data processing system evaluation.Li et al. (2020) Through its methodological approach and findings, it offers invaluable insights for stakeholders aiming to optimize the performance of their streaming systems.

# 3 Methodology

## 3.1 Data Collection and Data Streaming

In this approach of gathering big data from significant real time applications a large scalable computing is required. This can be achieved by utilising stateful serverless computing system and a stream processing framework. Farrokh et al. (2022) The stateful serverless computing system allows for efficient handling of state management and processing of streams, ensuring smooth operations and high performance. Additionally, the stream processing framework aids in the seamless collection and streaming of data from various real-time applications, enabling comprehensive analysis and evaluation of the gathered big data. In order to conduct our research, we have optimised a system running on a macOS device with an M2 chip and 8 GB of RAM, using Docker to create a seamless configuration centred on the most recent release of Apache Flink. With Kafka topic producing extensive fake data, Apache Flink's prowess in both real-time and batch processing outperforms traditional Lambda architecture. While batch processing deciphers historical data patterns vital for forecasting, real-time data demands instantaneous processing, facilitated by Apache Kafka's high-velocity data handling. Using Apache Flink's integrated Kafka connector ensures efficient data ingestion and processing within this architecture. There are no ethical aspects to use this dataset as the fake data is randomly generated. Ramírez-Gallego et al. (2018)

## 3.2 Static Bounded Table and Unbounded Stream Oriented Tables

After we have processed the data from the real-time streams to process large-scale data analytics, Apache Flink provides two methods for stream processing. In this study, we will use two primary table abstractions: Static Bounded Tables and Unbounded Stream-Oriented Tables. Our research aims to execute data processing tasks, assessing both their runtime performance and inherent characteristics. Static Bounded Tables, encompassing a predetermined data volume with explicit start and end bounds, are evaluated based on batch processing efficiency, data integrity, and query optimization algorithms. On the other hand, the infinite and ever-growing nature of Unbounded Stream-Oriented Tables is assessed through metrics such as event time handling, window operations efficiency, and state backends' performance under varying load conditions.Kune et al. (2016) Employing Apache Kafka as the data source and targeting Elasticsearch sinks, we will evaluate a systematic benchmark for optimizing the table model for specific application

Table 1: Comparison between Static Bounded Tables and Unbounded Stream-Oriented Tables in Apache Flink

| | **Static Bounded Tables** | **Unbounded Stream-Oriented Tables** |
|---|---|---|
| **Definition** | Finite, static datasets | Represent infinite datasets |
| **Size & Nature** | Fixed beginning and end | Can grow indefinitely |
| **Applicability in Stream Processing** | Limited to defined scope | Ideal for real-time insights & continuous processing |
| **Performance & Scalability** | Optimized for finite datasets | Handle limitless data inflow; manage late/out-of-order data |
| **Use Cases & Practical Applications** | Periodic batch processing, reporting on static data | Real-time analytics, event-driven applications, low-latency insights |

requirements. By evaluating the efficiency, data integrity, and query optimization algorithms, we can ensure that the Unbounded Stream-Oriented Tables are able to handle large amounts of data and process it effectively. Additionally, assessing metrics such as event time handling, window operations efficiency, and state backends' performance allows us to optimize the table model to handle varying load conditions. By using Apache Kafka as the data source and Elasticsearch sinks, we can create a systematic benchmark that specifically caters to the application requirements, ensuring optimal performance and seamless integration.Li et al. (2023)

### 3.3 Mental Model of Data Flow in Stream Processing

In stream processing with Apache Flink and Kafka, data is ingested continuously from Kafka topics into Flink. These streams are then transformed within the PyFlink, including operations like filtering or complex windowed aggregations. Lohrmann et al. (2015) PyFlink chunks the continuous data into manageable windows for processing. The Kafka-SQL-Flink connector and flink-sql-elasticsearch connectors which are available to download from Apache Flink official website. Zaharia et al. (2012) This ensures seamless data flow and maintains processing guarantees. Once this data is processed, the data is streamed to elasticsearch sinks or Kafka topics. Essential to this entire workflow is Flink's native support for parallelism, ensuring scalability, and its ability to handle iterative feedback loops for recursive data operations. With the Kafka-SQL-Flink connector and flink-sql-elasticsearch connectors, Apache Flink provides a reliable and efficient solution for data processing. These connectors allow for easy integration with external systems, enabling the seamless flow of data. By streaming the processed data to elasticsearch sinks or Kafka topics, Flink ensures that the output is readily available for further analysis or consumption. Additionally, Flink's native support for parallelism and ability to handle iterative feedback loops make it a powerful tool for handling recursive data operations and achieving scalability.

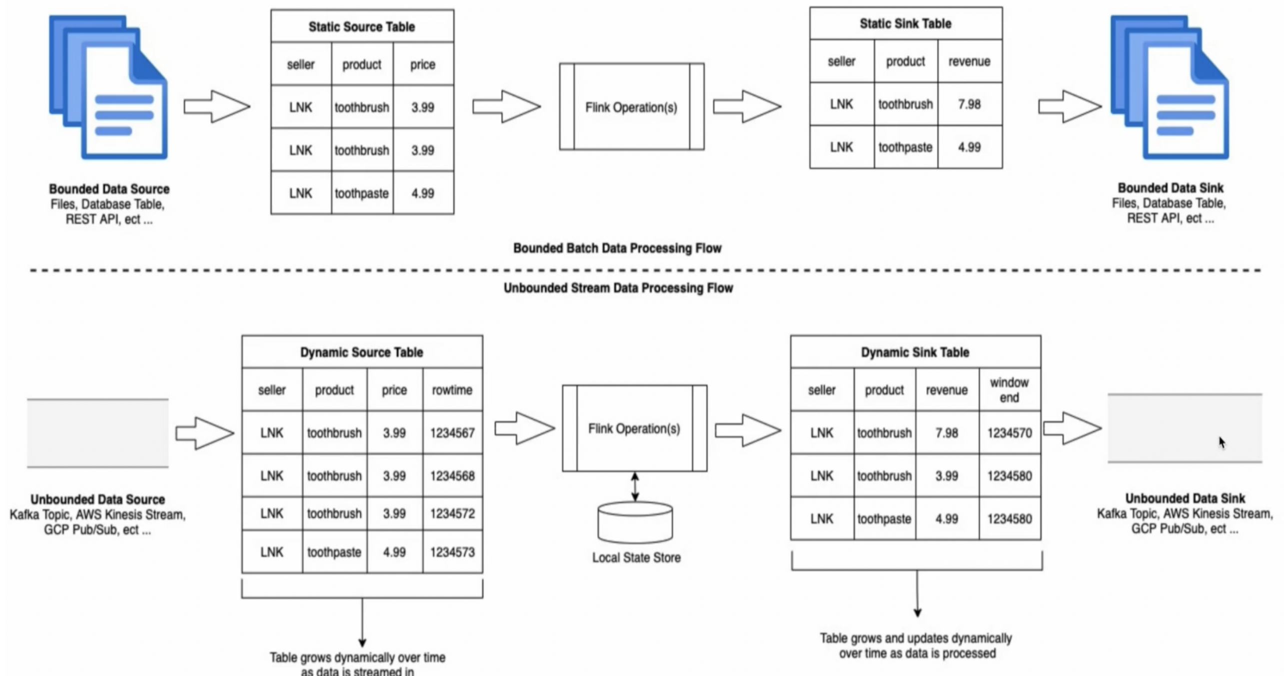


Figure 1: Mental Model of Unified Batch and Stream Processing

### 3.3.1 Data Source and Data Ingestion

It starts with Kafka, a distributed event streaming platform that acts as the source of the data. Kafka allows for the ingestion of large volumes of data from various sources in real time. It provides durability and fault tolerance, ensuring that data is not lost in case of failures. Flink seamlessly integrates with Kafka, allowing for the efficient and reliable consumption of data from Kafka topics. This integration enables Flink to process and analyze the data as it arrives, ensuring that insights are generated and actions are taken in near-real-time. Kafka topics serve as channels where raw data streams are published. Gupta and Agrawal (2019)

### 3.3.2 Stateful Computations

Stream processing applications often need to maintain state (like counting events). Flink provides fault-tolerant state management. This ensures that even in the event of failures or system crashes, Flink can recover and resume processing from where it left off, without losing any data. Flink's stateful computations allow for complex operations on the data streams, such as aggregations, joins, and windowed computations, making it a powerful tool for real-time data processing and analysis. With its seamless integration with Kafka and robust state management capabilities, Flink is an ideal choice for building scalable and reliable stream processing applications.Ramírez-Gallego et al. (2018)

### 3.3.3 Data Flow in Flink

Once data is ingested from Kafka it becomes infinite streams, which Flink processes in real-time. As the data traverses Flink, it undergoes transformations ranging from basic filtering to intricate windowed aggregations. To manage these continuous streams, Flink segments them into windows for batch-like processing. The Kafka-Sql-Flink connector facilitates seamless data exchange, with Flink checkpoints aligning with Kafka offsets to guarantee precise processing. Post-processing, Flink directs the refined data to various endpoints, be it another Kafka topic, a database, or other storage systems, readying it for subsequent analysis or applications. As data flows through Flink, it undergoes various transformations. This could be simple operations like filtering or mapping, or more complex operations like windowed aggregations. Given the continuous nature of streams, Flink uses windows to chunk up unbounded data streams and process chunks as finite batches. These windows can be time-based, where data is grouped into fixed time intervals, or count-based, where data is grouped into fixed-size chunks. Flink allows users to define different windowing strategies based on their specific needs. Once the data is grouped into windows, Flink applies various operations such as aggregations, joins, or calculations on the data within each window. This allows for efficient processing of large amounts of data in a scalable and fault-tolerant manner. After the processing is complete, Flink sends the results to the specified endpoints, ensuring the refined data is ready for further analysis or application consumption.

```
flink-sql-connector-kafka-1.17.1.jar
```

### 3.3.4 Elasticsearch Sink and Data Processing

After processing the data, Flink pushes the results to elasticsearch sinks. The sink is defined in elasticsearch indices which is set up using docker conrainer. This is where the processed data gets its value, as it is will be used for further analysis, visualization, or any downstream application. The elasticsearch-sql-flink connector integration of Flink with kafka sends the data to the designated endpoints to processed index on elasticsearch sinks. The processed data is made readily available for analysis, visualization, and other downstream applications by specifying the sink in elasticsearch indices. The value of the processed data is completely appreciated and may be successfully tapped thanks to this shortened approach.

```
flink-sql-connector-elasticsearch7-3.0.1-1.17.jar
```

# 4 System Architecture and Design Specification

The architecture as shown in 2 underlie the design and the associated requirements for efficient way to reduce latency by usign stream processing of real time big data streaming with an improved and efficient manner. The aim of these architecture is to capture the latency and throughput of the E-commerce Sales data. Evaluation methods and metrics used in previous studies have primarily focused on traditional batch processing methods for analyzing E-commerce Sales data, neglecting the potential benefits of real-time stream processing. This gap in the current literature highlights the need for further research in exploring the effectiveness and efficiency of using stream processing for reducing latency

and improving throughput in analyzing big data streams. By designing a system architecture that leverages stream processing techniques, this study aims to address this gap and provide insights into the potential advantages of this approach.

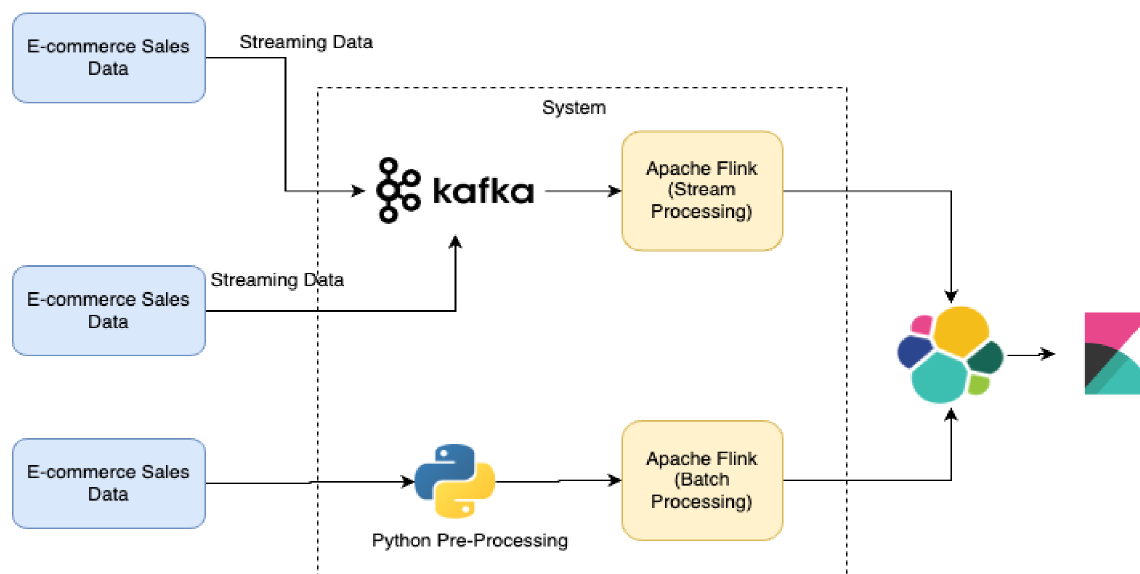


Figure 2: ETL Pipeline for Stream Processing

We are performing operation on Data Stream Table API using two methods which are using string processing and windowing data processing technique. In String processing we are getting data from E-commerce as we can see in 2 and we are processing this data using python pre-processor and the giving to apache flink for post-processing and and sending data to the elasticsearch. We can visiualize our data using kibana from which we can see or create various dashboard based on our data for example. Order count, time-series based order, order types.

String processing will contains the data which does not need to be processed at real time. This data includes historical information, such as past orders and customer details. By utilizing a python pre-processor, we can clean and transform the data before sending it to Apache Flink for further post-processing. Once processed, the data is then sent to Elasticsearch, where it can be easily accessed and analyzed. This approach allows us to efficiently handle large volumes of data and generate insightful visualizations using Kibana.

For Stream processing we will be using windowing technique to enable aggregation and analysis of data within specific time intervals or based on certain conditions. In this research we will be using Tumbling windows and Sliding window technique. In Tumbling windows it divide the stream into non-overlapping, fixed-size time intervals and all events withing a specific windows are grouped together and processed as a batch. In Sliding windows, unlike tumbling windows, it overlap with each other.It consist of both a window size and a slide interval. As the window sliders over the data stream, events are assigned to multiple windows, allowing for continuous analysis of data. In Above 2 we are sending data to Kafka from two source in which one using sliding window technique and another tumbling window technique and to process these data real-time we are kafka and further we are processing it in Apache Flink and it generated output sending to the sink which our Elasticsearch. In Apache Flink, the sliding window technique is applied by defining

a window size and slide interval. This allows for continuous analysis of the data stream as the window slides over it. On the other hand, the tumbling window technique is used by specifying only the window size, which creates non-overlapping windows. Both techniques are utilized to process the data in real-time. The processed output is then sent to Elasticsearch, serving as the sink for the analyzed data.

In 2 to connect Apache Flink we are using Pyflink with two connectors. Connectors are used to connect with Kafka topic and elasticsearch. Connectors are the JAR file.

For these research we are using Docker container to run Apache flink, Kafka and ELK stack without any resource limitation. Our system specifications are a Macbook with an M2 chip, 8GB of RAM, and a 500GB NVME SSD. The Pyflink connector for Kafka allows us to consume data from Kafka topics and process it in Apache Flink. This data is then analyzed and transformed before being sent to Elasticsearch using another connector. These connectors act as intermediaries, providing seamless integration between the various components of our data processing pipeline. With the help of Docker containers, we have been able to run all these components without any resource limitations on our Macbook, ensuring smooth and efficient data processing.

# 5 Implementation

In this section, we specify the required development environment for PyFlink jobs and the various Datastream Table API operators available.

## 5.1 Environment Setup

The Python Flink API architecture, introduced post version 1.9, consists of a user module, a communication module bridging the Python and Java virtual machines (VMs), and a task submission module for the Flink cluster. This architecture transitioned from the older JPython-based DataSet and DataStream APIs, which were not compatible with Python 3.X. In the new design, communication between the Python API and the Java VM occurs through a Python gateway connected to Java's GateWayServer. Adopted for compatibility and unified architecture, the latest Python API is built on the Table API, ensuring seamless operation similar to the Java Table API and offering enhanced advantages in performance and integration. The system should be configured for Python version ¡= 3.8 and the code is recommended to run in a virtual environment setup. Running the code in a virtual environment setup ensures that the dependencies and packages required for the Python API are isolated from the system's global environment. This allows for easier management and avoids conflicts with other Python installations. Additionally, it is important to note that the Python API's compatibility with Python version ¡= 3.8 ensures that the code can be executed without any compatibility issues on newer versions of Python.

```
python --version
Python 3.8.10
pip install apache-flink==1.17.1
pip confluent-kafka==2.2.0
```

## 5.2 Development Environment

To run the PyFlink jobs, first we set up a Confluent Kafka development cluster, loading data into Kafka using SalesProducer.py, and validating the setup. After the launch the Confluent Kafka environment using Docker is launced , we then create a Kafka source and sink topics. Also, it is importnant to download the Kafka Connector Jar, specifying the absolute path and configuring it properly to xecute the sales producer program, and running the provided Flink programs (either the Table API or SQL Interface version). After data processing, users can verify the contents of the 'processedsales' and 'processedsales2' topics using Kafka's console consumer and the elasticsearch sink specified. Additionally, it is recommended to monitor the performance and resource usage of the Docker containers using tools like Prometheus and Grafana. This will help in identifying any bottlenecks or issues that may arise during the data processing. Furthermore, it is critical to regularly backup the processed data stored in Elasticsearch to prevent any data loss in the event of system failures or crashes.

```
docker exec -it broker kafka-topics --create \
    --bootstrap-server localhost:9092 \
    --topic salesitems
    python3 -m flinkenv env
    source/bin/activate
    python salesproducer.py
```

# 6 PyFlink Job Submission

The process of executing a Python Table Job in Apache Flink involves four main steps. Initially, we have to decide on the job mode: batch or streaming. Then we define the data's source, including its schema, source, and data type. The draft of the computational logic for the data and store the results in a specified system, which is termed as defining the sink. Within the Python API, an execution environment is first created, which is essentially a table environment containing a Table Config module for runtime configurations. After setting up the environment, the data source table is defined for which our Kafka topic as a source String data type. Once defined, the data structure is converted into a table in the Table API layer, and fields and types are added using with SCHEMA. After processing the data, a results table is established to store computational outcomes. In our Aggregation job, a table with 'salesitem' and 'count' fields is created. The computational logic is then written; with the Python API, this can be as concise as a one-liner. In the aggregation example, source data is scanned, grouped by words, selected, and then aggregated to calculate total sales, which is then inserted into the results table.

```
docker exec -it jobmanager bash

./opt/bin/flink run -py /opt/flink/examples/python/datastream/aggragation.py

Job has been submitted with JobID c1e2f6767934121c07c7af6849fb1f24
```

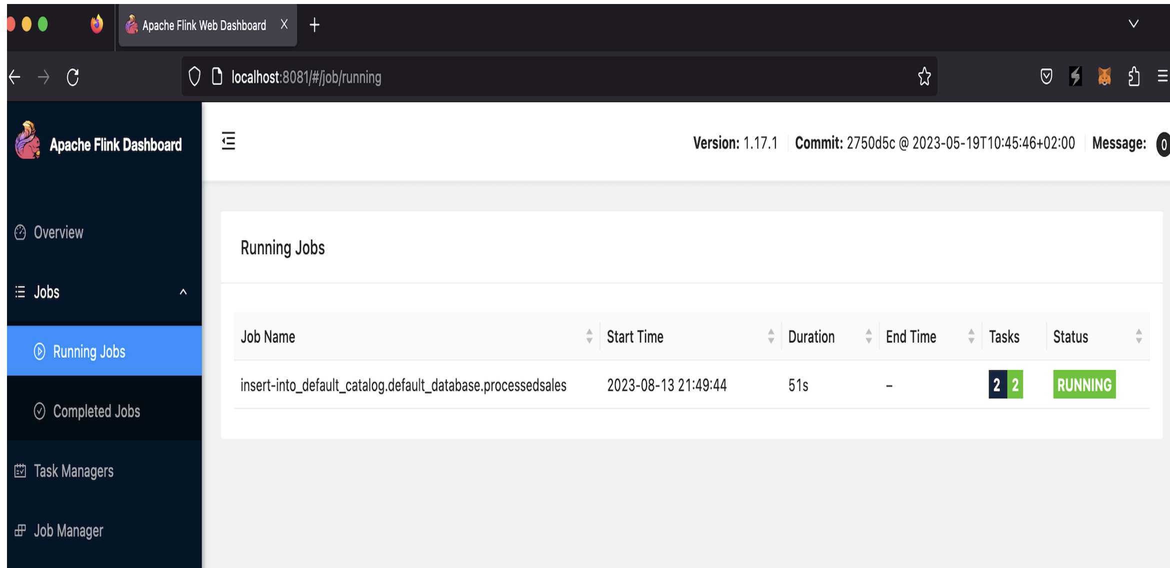


Figure 3: Flink Cluster running on docker container

# 7 Evaluation and Critical Analysis

For the evaluation of the proposed architecture the following case studies have been performed. The expirimental setup uses the fake dataset produced by the kafka topic as SalesProducer.

## 7.1 Case Study 1: Aggregation Functions using Kakfa Streaming

Apache Flink's Table API and SQL interfaces in Python (PyFlink) are used to perform aggregation operations on batch data. The data, sourced from a CSV file named csv-input, is ingested in a table called product sales with specific field names and types.

Using the Table API, it calculates the average price of unique products and the total revenue for each seller. These computations are done by selecting distinct product prices and then computing their average, and by multiplying quantities by product prices to derive sales, followed by aggregating these sales per seller. In parallel, similar aggregations are demonstrated using Flink's SQL interface. In the Table API, the average price of unique products and the total revenue for each seller are calculated by selecting distinct product prices and computing their average. The sales are derived by multiplying quantities by product prices, and then these sales are aggregated per seller. Similarly, Flink's SQL interface also demonstrates similar aggregations by using parallel processing. The AVG function in SQL computes the average price, while the SUM function, combined with a GROUP BY clause, calculates the total revenue per seller.

```
docker exec -it jobmanager bash

./opt/bin/flink run -py /opt/flink/examples/python/datastream/aggragation.py

Job has been submitted with JobID c1e2f6767934121c07c7af6849fb1f24
```

## 7.2 Case Study 2: Tumbling Windowing using Kafka Streaming

In the context of stream processing, windows are a way to group a continuous stream of data into finite chunks for processing. In Apache Flink, there are different types of windows, and among them, tumbling windows is one type. This event defines a tumbling window with a duration of 30 seconds. So, the data stream will be divided into non-overlapping 30-second chunks. The processing time (proctime) is the windows created based on when events which arrive in the Flink cluster. The start and end of the window calculates the total sales (by multiplying quantity and product price and then summing it up) for each seller within that window. The tumbling window allows for easy and efficient data processing as it ensures that each event is assigned to only one window. This means that no event will be counted multiple times or left out of any window. The arrival time of events determines the window's start and end times, ensuring that the calculation of total sales for each seller within that window takes into account all pertinent information. Overall, tumbling windows provide a reliable and accurate way to analyze and summarize data in real-time.

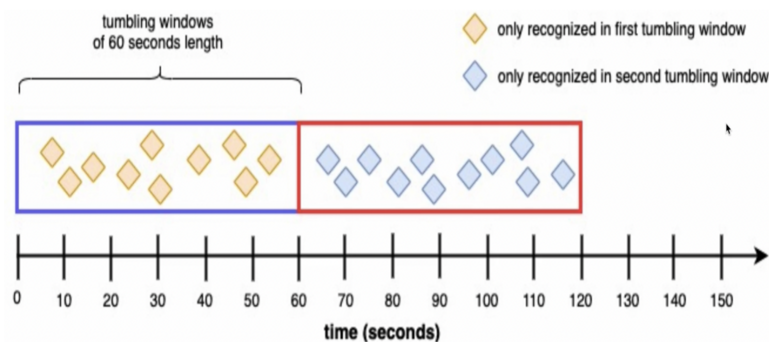


Figure 4: Tumbling Windowing in Apache Flink

Figure 5: Tumbling Windowing Index Latency on Elasticsearch

## 7.3 Case Study 3: Sliding Windowing using Kafka Streaming

In stream processing, a sliding window moves over the incoming data stream in predefined increments, and each window captures a subset of the data stream. Unlike tumbling windows, which are non-overlapping, sliding windows often have overlapping data. Sliding windows move over a data stream in specific increments and capture overlapping data subsets. In this, a 30-second window slides every 10 seconds, analyzing sales data. For each window, we calculate the total sales for each seller within that duration. Using sliding windows provides us with more frequent analysis into streaming data, capturing sales metrics for every overlapping 30-second window that starts every 10 seconds. This allows us to track the performance of each seller in real-time and identify any sudden changes or trends in their sales. By analyzing the sales data in non-overlapping, sliding windows, we can also compare the performance of different sellers within the same time frame. This approach provides a more granular view of the sales data and helps us make timely decisions to optimize our sales strategies.

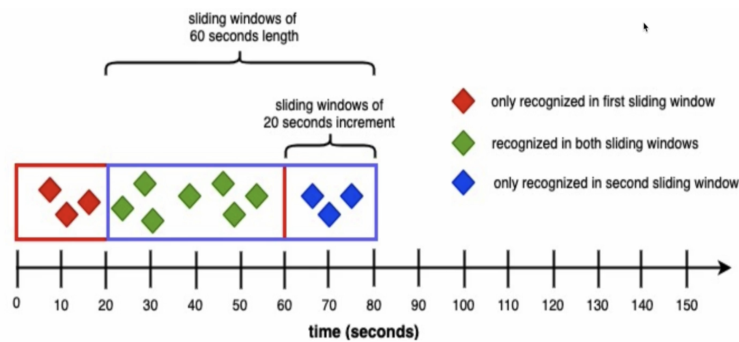


Figure 6: Sliding Windowing in Apache Flink

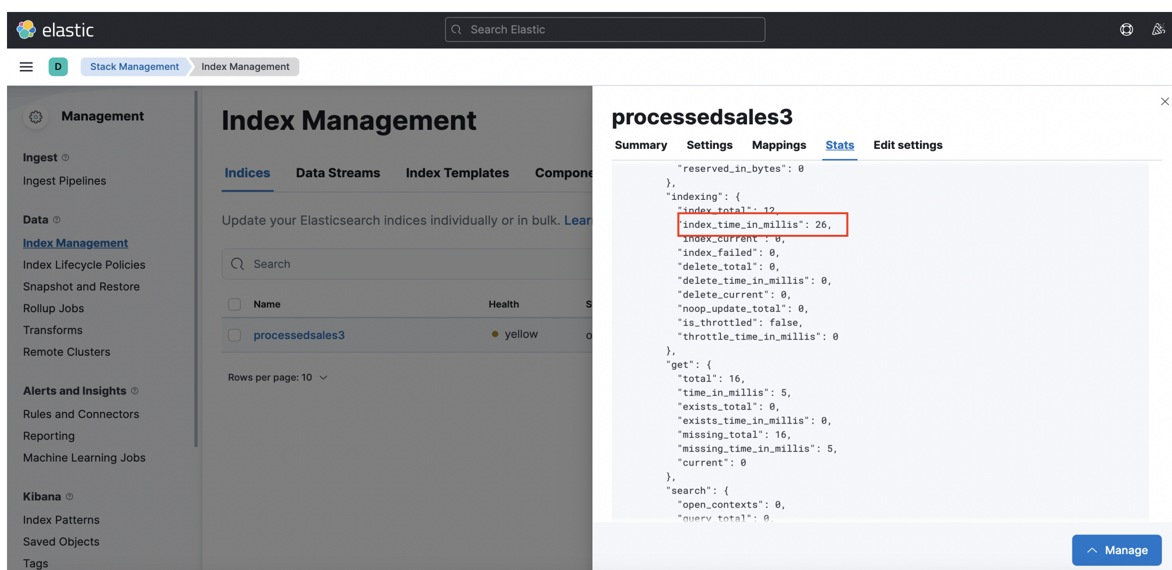


Figure 7: Sliding Windowing Index Latency on Elasticsearch

| Function | Apache Spark | | PyFlink | |
|---|---|---|---|---|
| | Latency | Throughput | Latency | Throughput |
| Count in Batch Processing | 189873.08765(ms) | 7754.8993(events/sec) | 1890(ms) | 45.11(events/ms) |
| Aggregation Function using Stream Processing | 445663(ms) | 21397(events/s) | 16704(ms) | 16553.87(events/s) |
| Tumbling Windowing using Stream Processing | 21 seconds | NA | 1 min 9 seconds | 1205.4(Mb/sec) |
| Sliding Windowing using Stream Processing | 32 seconds | NA | 1 min 3 seconds | 1154.5(Mb/sec) |
| Event Time Processing using Stream Processing | 28 seconds | NA | 1 min 32 seconds | 7877.8999(events/s) |

Figure 8: Observartions between Spark and PyFlink

## 7.4 Case Study 4: Event Time Processing

In the event-time processing in PyFlink, the streaming data is processed based on event timestamps, rather than the time it arrives into the streaming. Specifically, the data contains a sales timestamp from which the event time field is derived, and a 5-second allowance is provided for late-arriving data using watermarks. This setup facilitates the aggregation of sales per seller in distinct 30-second, non-overlapping windows based on this event-time. This approach allows for accurate analysis of sales data, as it takes into account the actual time at which events occurred rather than when the data arrived. By using event timestamps and watermarks, PyFlink ensures that late-arriving data is still considered within a certain time frame, allowing for a comprehensive view of sales per seller in specific windows. The resulting aggregated sales metrics, delineated by these 30-second intervals, are then stored in an Elasticsearch indexes the processedsales data for real-time analysis of sales data.

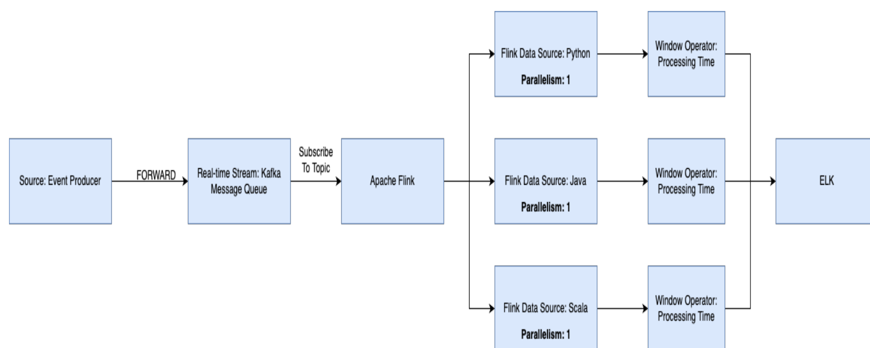


Figure 9: Timely Stream Processing

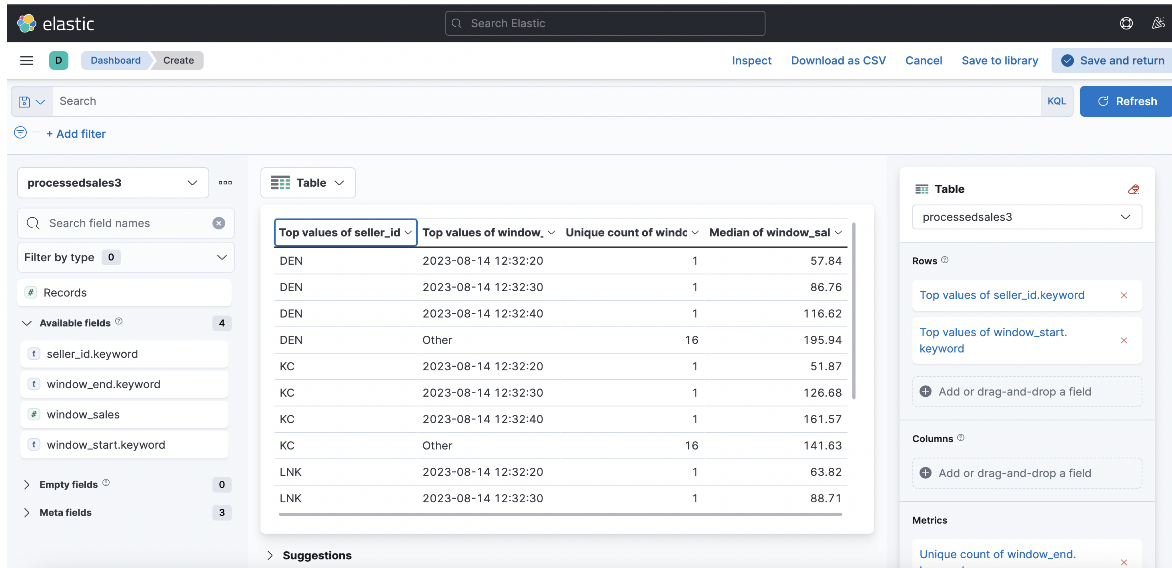


Figure 10: Data Visualization on Kibana Dashboard

# 8 Conclusion and Future Work

Our research on serverless computing in the context of real-time streaming revealed the enormous potential of stateless serverless platforms that can preserve state across invocations. Through careful analysis based on measurements, we found the key trade-offs between performance, latency, and cost. This showed how reliable and flexible Kafka streams and Apache Flink's stateful serverless functions are. We managed to process and analyse the data stream in real time, using pyFlink for processing, Kafka for data extraction, Elasticsearch for storing the processed data, and Kibana for visualisation, stream processing, and indexing the data into Elasticsearch. The combination of pyFlink, Kafka, Elasticsearch, and Kibana allowed us to achieve efficient and scalable data processing and analysis. Additionally, the stateful serverless functions provided by Apache Flink ensured reliable and flexible stream processing capabilities. Overall, this architecture enabled us to make real-time decisions based on the analysed data stream while maintaining optimal performance, low latency, and cost-effectiveness. The modular approach we adopted for designing stateful functions, especially with the capability to integrate with diverse databases on a shared platform, showed notable advantages over traditional local storage methods. This study substantiates the idea that cloud-native technologies, specifically serverless FaaS platforms, can be optimized and fine-tuned to handle complex real-time streaming applications with efficiency and scalability.

Further research work could delve into the scalability of our system, assessing its behaviour under heightened workloads and exploring its interoperability with diverse serverless platforms hosted on cloud platforms and databases. We evaluated the elasticsearch for advanced optimization techniques, further evaluation and comparative analysis can be done using different databases. There's also an avenue to enhance security measures, ensuring data integrity and robust access control in shared platforms. Further, machine learning algorithms could be harnessed to the full potential of big data. This would enable more accurate predictions and insights, leading to improved decision-making processes. Additionally, integrating natural language processing capabilities could enhance

the system's ability to understand and analyze unstructured data, further expanding its potential applications.

# References


Akil, B., Zhou, Y. and Röhm, U. (2017). On the usability of hadoop mapreduce, apache spark  apache flink for data science, *2017 IEEE International Conference on Big Data (Big Data)*, pp. 303–310.

Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S. and Tzoumas, K. (2015). Apache flink: Stream and batch processing in a single engine, *The Bulletin of the Technical Committee on Data Engineering* **38**(4).

D'silva, G. M., Khan, A., Gaurav and Bari, S. (2017). Real-time processing of iot events with historic data using apache kafka and apache spark with dashing framework, *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*, pp. 1804–1809.

Fan, L., Li, J., Pan, Y., Wang, S., Yan, C. and Yao, D. (2019). Research and application of smart grid early warning decision platform based on big data analysis, *2019 4th International Conference on Intelligent Green Building and Smart Grid (IGBSG)*, pp. 645–648.

Farrokh, M., Hadian, H., Sharifi, M. and Jafari, A. (2022). Sp-ant: An ant colony optimization based operator scheduler for high performance distributed stream processing on heterogeneous clusters, *Expert Systems with Applications* **191**: 116322.
**URL:** *https://www.sciencedirect.com/science/article/pii/S0957417421016237*

Gomes, D. G., Calheiros, R. N. and Tolosana-Calasanz, R. (2015). Introduction to the special issue on cloud computing: Recent developments and challenging issues, *Computers & Electrical Engineering* **42**: 31–32.

Grolinger, K., Hayes, M., Higashino, W. A., L'Heureux, A., Allison, D. S. and Capretz, M. A. (2014). Challenges for mapreduce in big data, *2014 IEEE world congress on services*, IEEE, pp. 182–189.

Gupta, B. and Agrawal, D. P. (2019). *Handbook of research on cloud computing and big data applications in IoT*, IGI Global.

Jabbar, A., Akhtar, P. and Dani, S. (2020). Real-time big data processing for instantaneous marketing decisions: A problematization approach, *Industrial Marketing Management* **90**: 558–569.

Kune, R., Konugurthi, P., Agarwal, A., Rao, C. R. and Buyya, R. (2016). The anatomy of big data computing, *Softw., Pract. Exper.* **46**(1): 79–105.

Li, H., Xia, J., Luo, W. and Fang, H. (2023). Cost-efficient scheduling of streaming applications in apache flink on cloud, *IEEE Transactions on Big Data* **9**(4): 1086–1101.

Li, Z., Yu, J., Bian, C., Pu, Y., Wang, Y., Zhang, Y. and Guo, B. (2020). Flink-er: An elastic resource-scheduling strategy for processing fluctuating mobile stream data on flink, *Mobile Information Systems* **2020**: 5351824.
**URL:** *https://doi.org/10.1155/2020/5351824*

Lohrmann, B., Janacik, P. and Kao, O. (2015). Elastic stream processing with latency guarantees, *2015 IEEE 35th International Conference on Distributed Computing Systems*, IEEE, pp. 399–410.

Merla, P. and Liang, Y. (2017). Data analysis using hadoop mapreduce environment, *2017 IEEE International Conference on Big Data (Big Data)*, IEEE, pp. 4783–4785.

Ramírez-Gallego, S., Mouriño-Talín, H., Martínez-Rego, D., Bolón-Canedo, V., Benítez, J. M., Alonso-Betanzos, A. and Herrera, F. (2018). An information theory-based feature selection framework for big data under apache spark, *IEEE Transactions on Systems, Man, and Cybernetics: Systems* **48**(9): 1441–1453.

Salloum, S., Dautov, R., Chen, X., Peng, P. X. and Huang, J. Z. (2016). Big data analytics on apache spark, *International Journal of Data Science and Analytics* **1**: 145–164.

Shahidi, N., Gunasekaran, J. R. and Kandemir, M. T. (2021). Cross-platform performance evaluation of stateful serverless workflows, *2021 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, pp. 63–73.

Singhvi, A., Balasubramanian, A., Houck, K., Shaikh, M. D., Venkataraman, S. and Akella, A. (2021). Atoll: A scalable low-latency serverless platform, *Proceedings of the ACM Symposium on Cloud Computing*, pp. 138–152.

Toliopoulos, T. and Gounaris, A. (2020). Adaptive distributed partitioning in apache flink, *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*, pp. 127–132.

Vaidya, G. M. and Kshirsagar, M. M. (2020). A survey of algorithms, technologies and issues in big data analytics and applications, *2020 4th International Conference on Intelligent Computing and Control Systems (ICICCS)*, pp. 347–350.

Zaharia, M., Das, T., Li, H., Shenker, S. and Stoica, I. (2012). Discretized streams: An efficient and {Fault-Tolerant} model for stream processing on large clusters, *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*.