An exploratory study of trade-offs in traditional vs. serverless stream processing

Master's thesis in Computer science and engineering

NILS TRUBKIN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

An exploratory study of trade-offs in traditional vs. serverless stream processing

NILS TRUBKIN

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

An exploratory study of trade-offs in traditional vs. serverless stream processing

NILS TRUBKIN

© NILS TRUBKIN, 2023.

Supervisor: Vincenzo Gulisano (Computer and Network Systems), Pedro Petersen Moura Trancoso (Computer and Network Systems)
Examiner: Vincenzo Gulisano (Computer and Network Systems)

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

An exploratory study of trade-offs in traditional vs. serverless stream processing

NILS TRUBKIN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Stream is the natural form of data that is in a perpetual process of being generated. Stream processing is a way to draw valuable insights from a data stream. With the rapid increase in data volumes primarily driven by IoT devices, stream processing has emerged as a practical approach for data processing. Some characteristics, such as volumes of data and their distribution, can vary over time, leading to changes in the computational requirements of such streaming applications. To be able to adjust frameworks used to the changing requirements, elasticity is needed. As traditional frameworks commonly used to run streaming processing applications, known as Stream Processing Engines (SPE) are not flexible enough, there is often some degree of over-provisioning. It means that the allocated resources are greater than required and remain unutilized. Alternative approaches, such as serverless, can ease scalability, but there are both pros and cons to the approach that this work delves into. This work has implemented a SPE-like API for serverless framework and with its help explores the differences between traditional and serverless models of stream processing engines using Apache Flink and Apache OpenWhisk.

The study shows that OpenWhisk can be used for implementing and executing streaming applications similar to those run by Flink. By correctly implementing the logic and code, a behavior similar to Flink's can be achieved in OpenWhisk. The serverless nature of OpenWhisk, with its pay-per-use pricing model, allows for reduced costs when the framework remains idle. Performance evaluation was performed using a stateless application type (does not require the state of the application to be preserved across multiple executions) utilizing *map()* API. Also, a stateful type of application (requires the state of the application to be preserved across multiple executions) was evaluated using *windowAll()* API with sum aggregate. The findings indicate a latency increase of 300-400% in the most intensive test cases and lowered throughput to 50% for OpenWhisk compared to Flink.

Conclusions that can be drawn reveal that Flink exhibits greater capacity and performance compared to OpenWhisk for comparable workloads. Flink's extensive resource base, including APIs and support resources, makes it easier to develop applications and positions it as a robust and well-established solution. On the other hand, OpenWhisk is best suited for projects that do not require rich stream processing libraries or explicit state management. Its high-level scalability abstraction, utilizing Kubernetes, simplifies scaling operations. Both frameworks can be configured to act similarly, with various benefits and tradeoffs depending on an individual use case.

Keywords: Stream, data, serverless, Flink, OpenWhisk, latency, throughput

Acknowledgements

I'm grateful to Vincenzo Gulisano for their invaluable guidance and feedback that improved this work.

Nils Trubkin, Gothenburg, 2023-09-15

Contents

List of Figures **xi**

1 Introduction **1**
1.1 Problem definition . . . 5
1.2 Scope of the work . . . 7
1.3 Thesis Organization . . . 8

2 Background **9**
2.1 Stream processing . . . 9
2.2 Windows . . . 10
2.3 Use Case Example . . . 10
2.4 Aggregation of data streams . . . 11
2.5 Watermarks in data streams . . . 12
2.6 Streaming frameworks . . . 12
2.7 Serverless frameworks . . . 13
2.8 Apache Flink . . . 13
2.9 Apache OpenWhisk . . . 14
2.10 Infrastructure for stream processing . . . 14
2.11 Scaling of stream processing applications and infrastructure . . . 15

3 Related Work **17**

4 Use Cases **19**
4.1 Examples of use cases for Flink . . . 19
4.2 Examples of use cases for OpenWhisk . . . 20
4.3 Picking the right tool for the job . . . 21

5 Features and Capabilities **23**
5.1 Description of key features and capabilities of Flink . . . 23
5.2 Description of key features and capabilities of OpenWhisk . . . 24
5.3 Comparison of the features and capabilities . . . 24

6 Ease of Use and Deployment **27**
6.1 Ease of use and deployment for Flink . . . 27
6.2 Ease of use and deployment for OpenWhisk . . . 27
6.3 Comparison of the ease of use and deployment . . . 28

7 System Overview and Architecture **31**
7.1 Explanation of Flink and OpenWhisk 31
7.2 Testing environment 32
7.2.1 Dashboard 33
7.2.2 Test batching 34
7.3 Purpose of the comparison 35
7.4 Implementation of "KNN" application 35
7.5 Implementation of "Twitter" application 35

8 Evaluation **39**
8.1 System Overview 39
8.2 'KNN'-test 40
8.3 'Twitter'-test 41
8.4 General observations of OpenWhisk 41
8.5 General observations of Flink 42
8.6 Results 42

9 Performance and Scalability **49**
9.1 Discussion of Flink's performance and scalability 49
9.2 Discussion of OpenWhisk's performance and scalability 49
9.3 Comparison of the performance and scalability 49

10 Conclusion **51**

Bibliography **53**

List of Figures

1.1 Data volumes growth and prediction for years 2010-2025[3] 2
1.2 Non-IoT vs. IoT devices for years 2010-2025[4] 3
1.3 Increases in spending within IT for years 2020/2021[5] 4
1.4 Batch processing timeline 4
1.5 Stream processing timeline 5
1.6 Serverless computing 6
1.7 Traditional and serverless architecture comparison 7

2.1 Data streams compared to data batches 10
2.2 Use case example input and output schema and the parameters of the aggregate 11

7.1 Flink pipeline 32
7.2 OpenWhisk pipeline 32
7.3 Dashboard used in the testing environment. 1) Current framework selection indicator 2) Framework selection buttons 3) Test selection buttons 4) Input table 5) Output table 6) Interval and batch size configuration 7) Debug information displaying timestamps 8) Download button for exporting the statistics 34
7.4 "KNN" and "Twitter" applications graphs 38

8.1 Flink KNN Test (2000 ms interval) 44
8.2 Flink KNN Test (1100 ms interval) 44
8.3 Flink Twitter Test (2000 ms interval) 45
8.4 Flink Twitter Test (1100 ms interval) 45
8.5 OpenWhisk KNN Test (2000 ms interval) 46
8.6 OpenWhisk KNN Test (1100 ms interval) 46
8.7 OpenWhisk Twitter Test (2000 ms interval) 47
8.8 OpenWhisk Twitter Test (1100 ms interval) 47

1

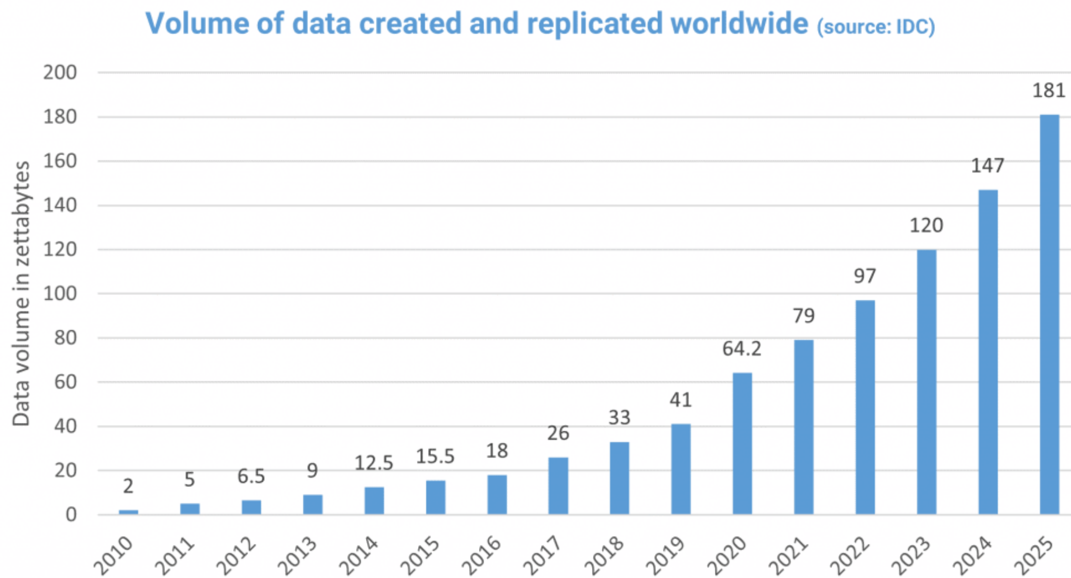
Introduction

The digitization of society and the emergence of the Internet of Things (IoT) has led to an explosion of data volumes and requirements for real-time processing. Companies and organizations need to draw valuable insights from the data and be able to analyze, alert and make decisions based on the data being generated and processed. The computational devices can now be found outside data centers, in small sensor and monitoring devices. New requirements on data processing techniques have led to new paradigms, each with its trade-offs. Some approaches revolve around moving the processing down to individual devices, while others have concentrated on moving the data up. While each approach has specific trade-offs in efficiency and performance, stream data processing has gained much traction due to its ability to handle data quickly in real-time and provide valuable insights[1][2].

Fig. 1.1 illustrates the volumes of data and the prediction for the years 2010-2025. In the last three years (2020-2023), the volumes have doubled, and the trend seems to continue for years to come[3]. As shown in Fig. 1.2, the increased volumes come from the IoT devices, as the amount of non-IoT devices stays at the same levels[4]. As IoT devices are usually low-power and offer limited computing power, the data must be sent somewhere else to be processed meaningfully, rapidly increasing data volumes. As a result, spending in all sectors utilizing IT in the form of cloud applications and data analytics is increasing quickly. The increase for 2020/2021 is presented in Fig. 1.3. Cloud application and cloud infrastructure have increased their cost by 83% and 71%, respectively, while data analytics has increased by 63%[5]. Such a significant increase over many years will compound and lead to magnitudes higher costs just a few years later if there is no change to the approaches used to process large data volumes.

Before discussing the streaming applications, it is essential to mention what solutions have existed before and what differences they have in comparison. An alternative to stream processing that has existed for a long time is known as batch processing, also called DB-based[6][7][8]. Batch processing is an approach for working with bounded data sets. The data set typically has to be processed in a data warehouse (data center), where it must first be uploaded, then processed one batch at a time. This approach separates the stage where data is being collected and stored and the stage where the collected data is processed to extract valuable insights. These two stages do not have to be performed close to each other in terms of time or location. Therefore, compared to the streaming approach, its main drawback is the high latency from

Figure 1.1: Data volumes growth and prediction for years 2010-2025[3]

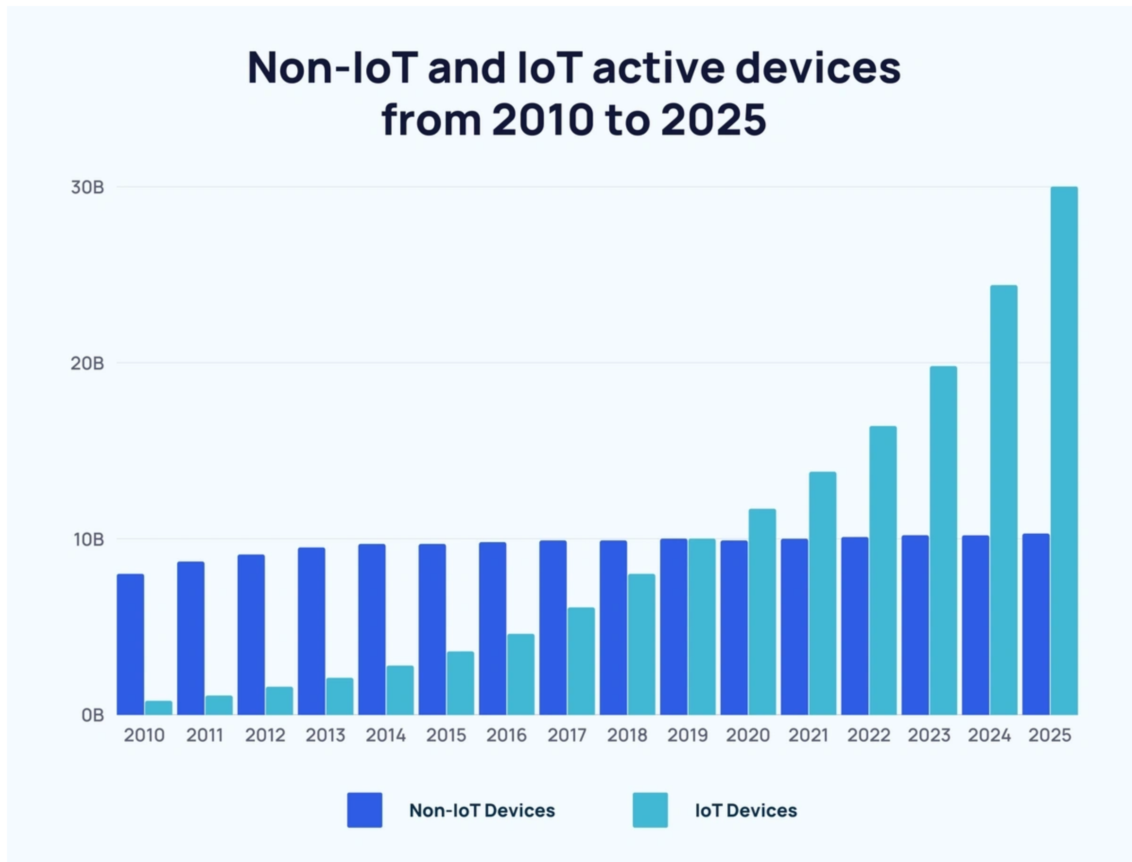


when a data point is created to the final results. All the data has to arrive before processing begins. Not only does the batch collection have to be finalized to begin the processing, but it also has to be uploaded to the data warehouse and processed from start to finish, which takes time.

Furthermore, this process typically does not involve reusing calculations for overlapping data parts in subsequent runs. In certain instances, like with a standard map-reduce operation, it may rely more heavily on disk than memory. Fig. 1.4 shows a typical batch-processing timeline. In that timeline, data generation is immediately followed by storage in the database. The process is repeated for all data generated until a batch of a predefined size or interval is collected. The collection of one or multiple batches is then transferred to a data warehouse where the processing and associated calculations take place. Once the batch processing is completed, the output result is produced and can be retrieved and used for the intended purpose.

On the other hand, streaming applications are used to process and analyze data as it is being generated or received. A streaming application runs on a stream processing engine (SPE), a framework that can be configured for one or multiple stream jobs. While reasonably new, this approach is very well-suited for the task. Almost all data is created as a continuous stream of events. It is hard to give an example where a data set is generated simultaneously and has no continuity[9]. Some data that can be processed as a stream are stock price, temperature, traffic flow, or manufacturing process. This data type is in modern cyber-physical systems such as smart grids, intelligent vehicular networks, and agricultural or ecological projects[10]. The data can be processed and analyzed to help draw conclusions and make meaningful insights on how the system could be improved or help detect anomalies quickly. This type of data processing is known as stateful stream processing. While stateless streaming

Figure 1.2: Non-IoT vs. IoT devices for years 2010-2025[4]



operators exist, it is common for streaming operators to maintain a state. A state can be preserved for an indefinite amount of time, thus avoiding the need for recalculating it. Stateful stream processing is, therefore, more efficient and offers lower latency from when a new data point is ingested to the result being produced. Fig. 1.5 shows a typical stream-processing timeline. In the timeline, the generated data goes straight into SPE for processing, after which the output result is produced. The delay of such a system can be very low, depending on the application.

When it comes to a batch-based approach, it's easy to identify the beginning and the end of finite input data. However, streaming presents a challenge because streams, as described in detail in Sec. 2.1 and Fig. 2.1, are unbounded. Despite this, the state maintained and updated while processing streams cannot grow indefinitely. To handle this, the focus typically lies within the most recent portion of data, consisting of finite data segments to be processed in smaller portions, using windows, as explained in Sec. 2.2.

On the other hand, the batch-based approach poses no such challenge since the data sets always have a defined beginning and end. If any state is maintained, it will only be kept for the current batch processing.

The following works and examples will explore different ways streaming technology

Figure 1.3: Increases in spending within IT for years 2020/2021[5]

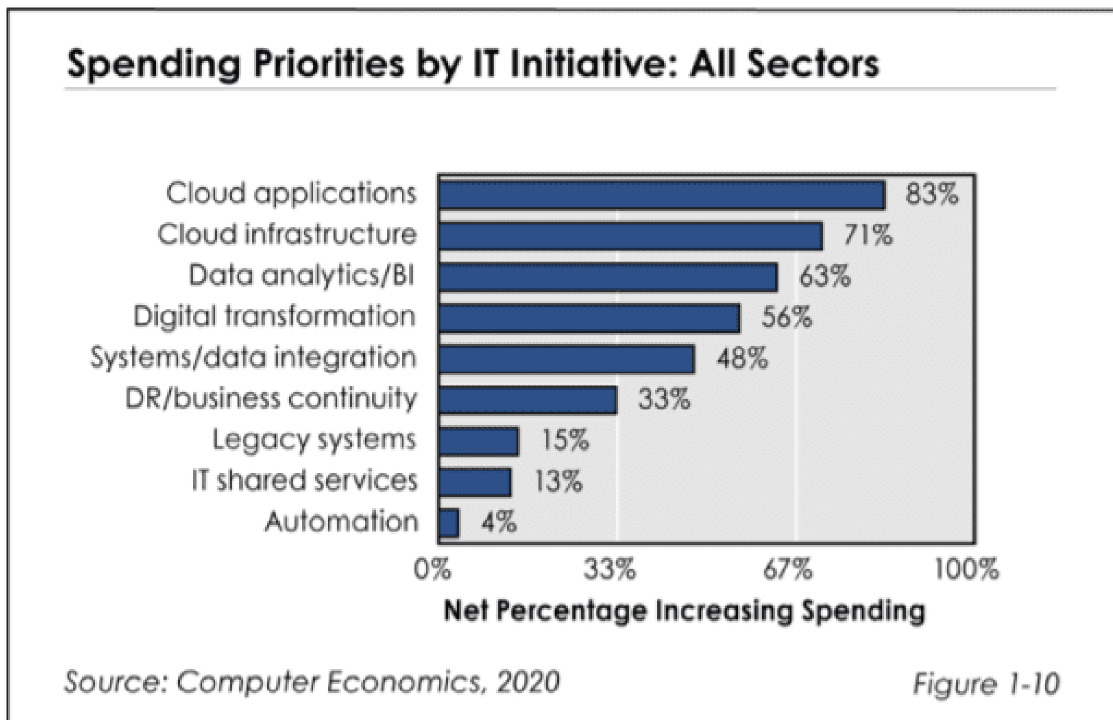
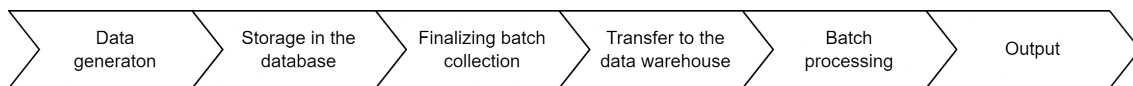


Figure 1.4: Batch processing timeline

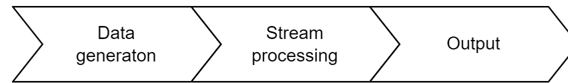


is used in today's digital systems. It will also explain how it can benefit various domains, such as data processing and industrial manufacturing.

A work by Najdataei et al. describes a system that can optimize stream processing and related clustering by reusing the calculated data across multiple iterations of the algorithm[11]. As the processing of one batch is finished and the next is started, all the calculated variables are usually reset, and a new batch is calculated from scratch. In that case, the calculated data is not preserved in memory across the batches, which may negatively affect latency if it has to be recalculated. The latency of a couple of hours to multiple days for the data point to appear in the results are typical[9].

A stream processing system that can help save material, energy, and associated costs in 3D manufacturing has been described in work by Gulisano et al. By analyzing the data generated during the process, the system can improve the final product's quality and detect defects early on, allowing for timely adjustments to the process[12]. A system like this can be used in many different types of manufacturing where the process is lengthy, and materials are costly. Moreover, this tool can assist the

Figure 1.5: Stream processing timeline



operator in enhancing their control over the entire manufacturing process.

1.1 Problem definition

The serverless computing model originated in the early 2010s and gained some traction with the rise of cloud computing[13][14]. The traditional application deployment model manages servers and the underlying infrastructure to adjust to the application's workload. This process required application developers to consider server capacity, scalability, and availability. The drawbacks of such an approach are that it can be complex, time-consuming, and often inefficient, as resources are often underutilized during periods of low workload.

Serverless frameworks aim to provide developers with tools that allow them to execute code in response to events without the need to manage servers. The actions can be integrated within rules and triggers that dictate their behavior and produce results. More details on serverless frameworks can be found in Sec. 2.7. An example of a serverless framework can be a registration form that collects user information such as name and email address and saves it to a database. While a traditional approach would require a server to be up and running at all times, a serverless approach can achieve the same functionality but allow for much idle time, during which the cloud provider does not incur costs for provided services.

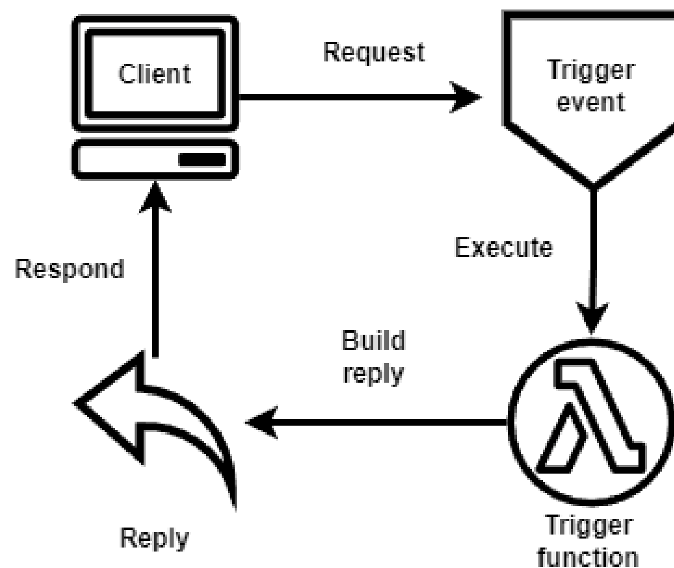
The concept of virtualization is another factor that has helped push the computation "toward edge" analysis. The scalability and ease of deployment have made virtualization a technique that found its use in many areas. One such area is enabling more efficient streaming, as described in work by Gulisano[15]. Virtualization can exist in many different forms. One such form is known as Docker containers. A self-contained environment often contains an operating system, libraries, application packages, and all the necessary configurations for a given task. Orchestrating many such containers working together has quickly become the norm, and tools for handling it were developed. One such tool is Kubernetes, which allows for quick deployment, configuration, and control of multiple Docker containers succinctly.

The way traditional SPEs have been designed is a framework executing on a machine that receives data and processes it according to a graph of operators. However, serverless computing is a new broader, more general variation of stream processing. While capable of performing stream processing, it still needs to be performant and studied well enough to replace the traditional SPEs and is, therefore, still an active research area. Serverless has advantages and disadvantages over a traditional SPE, as discussed in Sec. 2.10 and 6.2.

Fig. 1.6 shows an overview of a serverless computing pipeline. The interaction

begins with the client that creates a request and sends it to the serverless application endpoint. Once received by the endpoint, the request triggers an event trigger that, according to the configured rules of the application, triggers one or more functions to process the request. The function returns a result and builds a reply to respond to the client.

Figure 1.6: Serverless computing



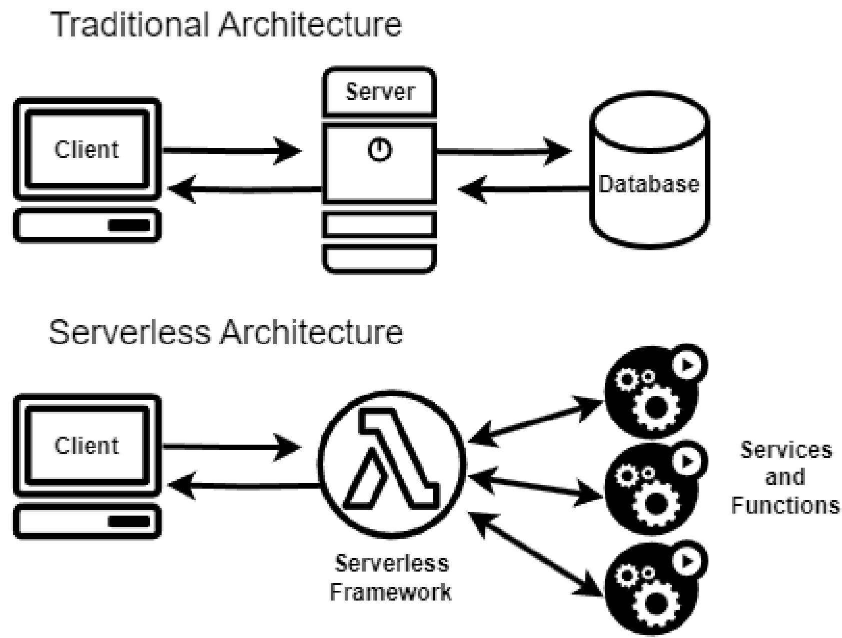
This work compares a traditional SPE and a serverless framework in the context of stream processing applications. The comparison aims to convey the differences between frameworks in running streaming applications and discuss the tradeoffs of each approach. The motivation behind comparing a traditional SPE and a serverless framework in stream processing applications is to understand how a serverless framework can be configured to perform the tasks of a traditional SPE.

Due to the increasing costs in the IT sector related to stream processing, an option where a serverless framework handles the task would tackle the problem of overprovision of resources[5]. It is interesting to find out what tradeoffs and challenges are present in the area of serverless computing and how they can be approached. On a broader plane, it is interesting to understand if and how a serverless framework can replace traditional SPE and implement an API similar to that of traditional SPEs'. The discussion describes the use cases most appropriate for each framework, features and capabilities, performance and scalability, ease of use and deployment, observations, and references to related research in the field. It aims to provide insights and a better understanding of the strengths and weaknesses of each approach.

Fig. 1.7 illustrates a comparative overview of traditional and serverless architectures in a broad context. The top half of the figure shows a traditional architecture where a client connects to a server's front end that handles the requests and computes the response, often based on a database in the back end. The serverless architecture shown in the lower half eliminates the need for a dedicated server. Instead, it delegates the tasks of managing the requests to a serverless framework, which executes functions

and actions to fulfill the request. Once there are no requests to handle, the serverless framework can go into an idle state, effectively pausing the cloud service billing for the application. Details on billing for serverless frameworks are discussed in Sec. 7.1 and 6.2.

Figure 1.7: Traditional and serverless architecture comparison



The experimental evaluation assesses the performance metrics for applications implemented for each framework. The results translate well to real-world scenarios as they simulate a load comparable to simple streaming applications. The work's primary focus is on each framework's differences, requirements, and similarities regarding the achievable goals with both approaches.

1.2 Scope of the work

Given the broad scope of the streaming processing topic, the work is limited to creating and testing applications to explore the differences between traditional and serverless stream processing mentioned in Sec.1.1. For traditional stream processing, the Apache Flink framework was used. For serverless processing, Apache OpenWhisk was used.

The metrics collected in performance evaluation (see Sec. 8) are limited to latency and throughput. The latency metric is collected from the input request to the system to the acknowledgment of that request. The throughput is collected as the number of tuples the system processes per given time interval. Latency and throughput metrics together indicate the application's performance, which can be used and compared between the frameworks.

Discussion of performance and scalability will be based on these results. Use cases discussed will be limited to the related work on the subject. Features and capabilities

discussion will mention and briefly summarize the differences without detailed discussion on, for example, redundancy, security, and reliability. Observations and ease of use will be covered to the extent required during the implementation and evaluation phases. The deployment will be discussed with different use cases in mind based on the experience from this work and information available on the topic.

A good understanding of the Docker system and the higher-level Kubernetes framework is necessary to deploy OpenWhisk using Kubernetes in its production deployment mode, which is outside this paper's scope. Therefore, it was deployed in its standalone mode.

1.3 Thesis Organization

Sec. 2 gives the necessary background with an overview of the key concepts and technologies relevant to the research. It starts by explaining stream processing basics, aggregation, and watermarks. Furthermore, it introduces a running example to be referred to throughout the paper. Lastly, it introduces streaming and serverless frameworks, briefly describing Apache Flink and OpenWhisk.

Related work is presented in Sec. 3. It lists and discusses the work that has been done on the topic of stream processing and serverless computing. Examples of use cases are presented in Sec. 4. This includes the use cases that stream processing, and serverless computing is aimed at and perform well in.

Features and capabilities of the frameworks are described in Sec. 5. Sec. 7 describes system overview and architecture. It includes an explanation of the testing environment used in the evaluation. Ease of use and deployment are discussed in Sec. 6. Both Flink and OpenWhisk are described in these sections.

The evaluation and results are presented in Sec. 8, while Sec. 9 delves into the performance and scalability. Lastly, Sec. 10 wraps up and concludes the work.

2

Background

This section introduces concepts and technologies relevant to understanding the following sections and gives the background relevant to the research.

Stream processing and serverless frameworks are relatively new but have shown their usefulness in high-volume data environments where low latency is preferred. In addition, stream processing and serverless architecture become more relevant and essential for systems that require manageable scalability and high reliability. Examples can be found in many areas, such as online shopping, weather forecast, intelligent vehicle networks, and various safety systems.

2.1 Stream processing

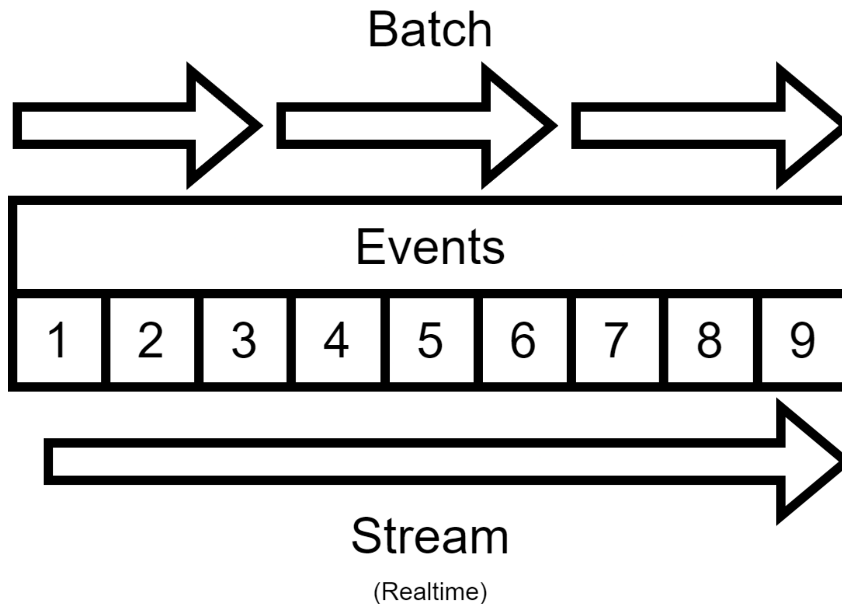
Stream processing is a type of data processing where one or multiple sources, such as sensors, social media feeds, and financial transactions, can be processed in real time and produce an output with a relatively small time delay that can be used to make a decision. Both input and output data are defined as streams. Streams are boundless, as they do not have an end; that is, their characteristical difference from other data types, such as sets or batches. Fig. 2.1 illustrates the unbounded nature of streams compared to the batch processing model. Streams do not have a defined start or end, unlike batches.

Boundlessness of data produces a challenge in producing a result, as the end of the data stream does not exist, and new data can arrive at any point in time—no assumptions on when the data stream ends can therefore be made and presents a challenge. The challenge can be solved with the help of so-called windows that segment the stream into finite portions. Windows are presented in Sec. 2.2

Usually, streaming applications are modeled as graphs of operators. They work in a chain, with one operator's output connected to the following operator's input. These connections can be one-to-one as well as one-to-many or many-to-one. Depending on the goal of the application, multiple operators can be used to solve load-balancing challenges and ensure high availability as long as operators are being executed on separate machines.

In an example of an e-commerce platform, data processing can be beneficial to keep track of what items are currently being sold the most and monitor the trends and purchasing behavior of the users. Recommendations can be generated, and stock

Figure 2.1: Data streams compared to data batches



replenishment can be requested based on the stream processing result with short notice. Stream processing implies that the data is moved up to a processing unit that accumulates the steams from various sources and processes those according to the implemented logic.

2.2 Windows

As streams do not have a bound and can be arbitrarily large, they create a challenge of maintaining and updating a state within finite memory. One common way of approaching this challenge is the division of the stream into finite segments known as windows. Windows exist in many different types and sizes, and various implementations exist for each type. This approach focuses on a finite data portion, which can be seen as a smaller batch that includes the most recent data.

Windows are often used with aggregate functions, reducing the window's contents into a single value based on how the function is designed. Typical aggregate functions include maximum, minimum, and average, which display the largest, smallest and average values in given windows. More information on the aggregation of data streams is presented in Sec. 2.4.

2.3 Use Case Example

To have a reference point for future discussion, we introduce an analysis system as an example. It is a stream processing application that receives and window aggregates data to inform the traffic throughput situation on a particular road, updated hourly.

To keep it simple, the system makes a detection each time a car passes. Once per

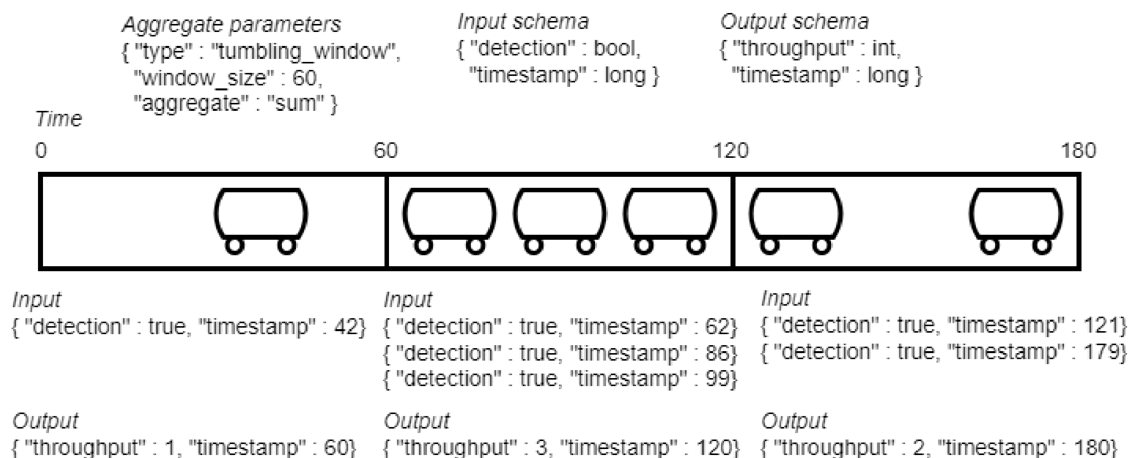
hour, it sums the detections to output a throughput metric, just the number of cars in the past hour.

Fig. 2.2 illustrates such a system, with input and output schema and the aggregate parameters. In the figure, three tumbling windows are defined, with a window size of 60 minutes (one hour). The first window contains one car, the second window contains three cars, and the third window contains two cars.

The input tuple schema defines a boolean for detection that is set to true and the timestamp of the event, in this case, in minutes. Usually, the timestamp is expressed in milliseconds since the UNIX epoch[16], but for this particular example, we express it in minutes and begin at zero for simplicity. What is essential is that the timestamp is granulated enough for the task at hand, which in our case, is set to one minute.

The output schema contains the throughput result (the sum of all the vehicles in the past hour) and a timestamp to indicate when the result was produced. The result is produced at the end of each hour, and a new window is created that begins counting from zero.

Figure 2.2: Use case example input and output schema and the parameters of the aggregate



2.4 Aggregation of data streams

While aggregation with the help of windows solves the previously presented challenge of data boundlessness, they introduce another challenge related to defining a point in time at which the window can be aggregated. When a window is aggregated, it produces a result that comes with an implied assumption that all of the tuples related to the given window will arrive in time. In other words, when windows are aggregated, we need to be confident that no delayed tuples may come late after the result is already produced. Such a challenge is usually solved with the help of watermarks. They allow for a configurable tradeoff between the produced result's latency and confidence in its correctness. Watermarks are described in more detail in Sec. 2.5.

Streaming aggregation is receiving and accumulating a data stream over a specific time, for example, a rolling window. An arbitrary function can be performed on the aggregated windows, such as sum, mean, and median, to name a few. Many different variations and implementations exist, but what is common for all of them is their reliance on even time and watermarks for correct execution. When aggregating a data stream, choosing the suitable algorithm for the application's requirements is essential. The requirements refer to latency, out-of-order, and sharing.

Windows' behavior comes down to its implementation and how a window is created and maintained. While some implementations deal with one window only, others might have multiple windows with addition only, panes, and other features. Hirzel et al. write about various algorithms that can implement a sliding window aggregation (SWAG), such as DABA, B-Int, and others[17].

Regarding the use case example introduced in Sec. 2.3, it would be using a sum aggregation with a tumbling window if the goal is to gain knowledge of the throughput for a given hour. On the other hand, a sliding window could be used to collect statistics for the past hour for a more frequently updated output, for example, every minute.

2.5 Watermarks in data streams

Watermarks are specific types of messages injected into the stream input to signal the passage of time to the framework. Watermarks exist in different types and can be divided into periodical and punctual. Watermarks segment the stream into finite sections and help synchronize the system. They are generated and sent through the stream independently, whereas the time event is usually part of metadata[18]. It is also possible to mark the stream data itself to be interpreted as a watermark.

The tuple schema must include a common timestamp element for watermarks to work. If all the tuples received contain a timestamp of the event time, it can be used to assign a tuple to the correct window and compared to the watermarks' timestamp. The event time is essential for the watermark to indicate progress, as it must rely on it for the correct result to be produced. If event time is unavailable, only the time when the tuple is received can be used as an indicator, and the correctness of the produced result can not be guaranteed if the tuple is late.

In the example application introduced in Sec. 2.3, the watermark would be of a periodical type and be sent as a separate message once per hour to aggregate the last hour window and produce a result of the total throughput of cars in the past hour.

2.6 Streaming frameworks

Some software frameworks are specifically designed to handle data stream processing as their primary purpose. Hence, they offer APIs specialized for data streams and include the tools and instruments to manage and modify them and organize the flow

between different system parts, be it one or multiple machines. Some examples of such systems are Apache Flink, Amazon Kinesis, and Google Cloud Dataflow.

A developer can configure and use these systems with data streams to create applications that process and output a data stream containing valuable insights. APIs that target steam processing define many operators that developers can leverage to create complex systems without implementing the low-level details of the algorithms themselves.

2.7 Serverless frameworks

Another type of framework allows users to express how to react to stream events. These frameworks are known as serverless and have much broader use in applications. Despite being generally aimed at processing a stream of events, these frameworks have minimal data manipulation functionality compared to the API of typical SPEs. The logic has to be implemented from scratch for the application to manipulate and transform the stream data as an SPE. The correct code and logic can achieve the same result as the stream processing frameworks.

While a serverless framework can react to streams and new data, it lacks the tools and libraries to handle specialized stream operations. It does not implement the logic for state maintenance. Serverless frameworks are generally more scalable and require less scale-up configuration. Given enough time and knowledge, any application aimed at data stream processing can be converted to a serverless application.

In the example application introduced in Sec. 2.3, the serverless framework would implement a function to receive the detection tuple and store it in the database. There would be no need for explicit windowing semantics. The aggregation would be performed hourly, using a timed trigger. The system would search the database for entries timestamped within the previous hour and return the count every hour. The count could also be stored back in the same database in a different collection and be accessed later either directly or via a front-end system, a webpage, for example.

2.8 Apache Flink

Apache Flink is an open-source software project that can process both batches and streams. It allows for filtering, mapping, aggregation, sorting, grouping, merging, joining, and splitting manipulations. It offers high performance and is made for efficiently handling data streams, resulting in low latency and high throughput. Flink also offers functions that may be highly interesting for high-availability services and mission-critical applications. These functions include recovery after failure, flow control, and task chaining. Recovery after failure means the application can be recovered to a state before failure and continue the operation as usual. Flow control regulates data flow from the sender to the receiver, ensuring data is sent at a manageable rate. Flow control prevents the sender from overloading the receiver, increasing latency, slow throughput, and other issues. Finally, task chaining

is an optimization that reduces the overhead between the different tasks of the streaming application, increasing the processing speed. These tasks are individual transformations, be that map or an aggregation function, and can be chained. It is achieved by coupling one function's output to another's input via a function call, avoiding serialization and deserialization, resulting in higher efficiency and faster processing.

Other features Flink offers are checkpoints. Those can be configured to be created at a specific interval and be used as a backup solution for the current Flink system state. In case of a system failure, the checkpoint can restore the system to the previous state. Various configuration options exist depending on how critical it is for the tuples to be processed only once, at least once, or neither.

Flink can be configured in separate clusters of machines within the same project to offer parallelization, load balancing, and redundancy. However, while scaling can be achieved, it requires manual configuration. Such a configuration would require the developer to identify the requirements, configure cluster size, and tune Flink parameters such as checkpointing intervals, network buffers, task parallelism, and data serialization.

2.9 Apache OpenWhisk

Apache OpenWhisk is also an open-source software project that is a serverless event-driven platform. It can execute arbitrary code in response to various types of events. For example, this platform can issue an HTTP request, send messages, and execute tasks due to some event. It can also process data streams but does not provide an API that targets streaming applications like Flink. Therefore, it requires custom implementation stream handling and specific operations on it.

Out of the box, OpenWhisk does not provide any state management or redundancy features. State management implementation is up to the developer, who can use a library or database adapter to store the state. Redundancy features are usually handled on a higher level of abstraction and are not directly related to the task being performed.

Flink and OpenWhisk can be used together or separately to build various real-time data processing pipelines and event-driven applications.

2.10 Infrastructure for stream processing

Infrastructure in this context can be the hardware itself but is mainly related to the software layer between the actual application and the hardware it is run on, such as the operating system itself. For example, Flink generally aims to be run as a program on top of an operating system that developers must maintain. Maintenance in this context involves keeping the system updated, applying security patches, performing configurational changes, and migrating in the eventual end-of-life of a particular operating system. However, instances of stream processing applications run in the

cloud as a service, such as Amazon Kinesis, do not require manual maintenance and can be automatically updated and maintained by the hosting provider.

OpenWhisk, on the other hand, aims to be a cloud service first and can be used on a pay-per-use basis. That means that the developer is only billed for the time their actions are actively running and not for when they are idle. As a result, the hosting provider automatically handles all of the infrastructure, allowing the developer to deploy code to an already configured system. This setup differs from when a developer chooses to set up their cluster of Kubernetes with the OpenWhisk framework running, as it requires considerably more configuration and knowledge to be done correctly.

The described issue is a case of overprovisioning. When a developer rents a software/hardware setup, the cost is set at a fixed rate and does not depend on the amount of load or resource utilization. This billing model frequently results in charges that exceed the actual usage of the system. Because even during periods of low utilization, the same rate is charged, even if only a tiny amount of work is performed.

The benefits of selecting a cloud instance of OpenWhisk come from the fact that no billing is incurred when no actions are executed. It is also a much simpler process, as developers must only submit their code to deploy it on their instance immediately. In addition, if the developer wishes to scale up or down, the OpenWhisk provider will take the configuration task upon them with minimal developer interaction required. On the other hand, running a self-hosted instance of OpenWhisk in a cloud or locally requires the developer to configure the underlying Docker/Kubernetes environment to deploy the OpenWhisk instance. However, once configured, both cases offer similar features, as the OpenWhisk does not have to be restarted when a new code is submitted. Other factors like downtime, redundancy, and security should also be considered when picking the deployment variant, as data centers usually have better security routines for confidentiality, integrity, and availability.

In the system described in Sec. 2.3, the car distribution will not be uniform throughout the day. Instead, there will be a distinct peak and valley. We can assume that the throughput is low, 1000 cars per day, and that adding each entry and aggregating the values takes a fraction of a second. Recording and processing the data for each hour will, over 24 hours, accumulate just a couple of seconds of execution time. Therefore, an OpenWhisk instance billed per use will only bill a small amount based on action execution time. However, an identical application deployed on a generic computing cloud server running an OpenWhisk instance or a traditional SPE will bill for the entire 24-hour period regardless of the system being idle most of the time.

2.11 Scaling of stream processing applications and infrastructure

Serverless frameworks are created with scaling in mind. The code is executed on the cloud, and the infrastructure is automatically assigned and adjusted depending on the application's requirements. This immediate deployment strategy decreases

the time from the idea to a live application. It removes the need to understand and configure the servers and up or down-scaling them when needed. In addition, this approach puts the logic first and frees up resources that would otherwise be needed to configure the cluster of data streaming applications.

Compared to a streaming application running on a traditional SPE, that query, once deployed, can not be changed. If an application has to be modified, the current job hosting the application has to be terminated, and a new job containing the updated logic has to be deployed. Depending on the complexity of the application and the consequences of downtime, this can introduce additional complexity related to the deployment of updates to the application. In some cases, the current state of the operators might need to be backed up and restored after re-deployment. Availability of the service during the re-deployment process can also introduce additional downsides and require careful planning of the procedure.

The use case example from Sec. 2.3 could be scaled in terms of computational requirement and terms of results produced and insights gained. Computational requirements could be increased by more traffic or additional roads added to the streaming application. Additional features could include data processing and functionality, such as multiple results produced, for example, for the last hour, day, and week with the mean, maximum, and sum of each metric.

3

Related Work

Work within the area of stream processing surrounds a wide range of topics. This section will mention some work that focuses mainly on the use cases and the usefulness of the technology in other systems. It will also discuss other types of work that concentrate more on the performance and optimization aspect of the stream processing itself. Furthermore, the section will touch upon some work in serverless processing. Most of it aims to understand how to use it most effectively and what patterns make for good architecture and scalability of the code base.

Works related most to this paper include a work by Taibi et al., a multivocal literature review of patterns for serverless functions, bringing forward the point that while data processing can be expensive in a serverless setting, it allows for many defined patterns to be put together and form a very scalable and fault-tolerant system[19].

The literature review work differs from the current work mainly in the scope of covered use cases. While the current work focuses on streaming and compares the differences between the two ways of processing them, the literature review focuses on serverless only. It gives a much broader perspective of possible scenarios. The literature review discusses patterns for authorization, availability, communication, and aggregation. These patterns are not discussed in depth in the current work, although some are introductorily mentioned in Sec. 2.

The system's performance is a topic not covered by the literature review and is listed as future work in the conclusion. The current work performs a performance measurement to draw meaningful conclusions via experimental evaluation and compare the expected latency and throughput for the tested frameworks.

Another work related to the comparison of various stream processing methods is a work comparing window types. Verwiebe et al. detail the types of aggregation that windows can support and methods for calculating such. The different approaches can yield different performances, depending on, amongst other things, whether all the tuples have to be stored or not, as well as the ability to perform the approximate calculations of windows if the precision is not essential. Different frameworks for processing windows were compared, and Flink was found to have the most extensive support for various types and allows for a high configuration level[20].

The survey is related to the current work comparing different approaches to stream processing. Similar to the previous related work, this work focuses not on performance metrics but on each window type's capabilities, benefits, and drawbacks. The

similarities include the comparison of various SPE to highlight what types of windows are supported by each of them.

Window aggregation is a widespread type of stream processing. It is a suitable candidate for a use case when discussing and comparing different frameworks and the stream processing capabilities of each framework. Both survey and the current work frame the discussion around general-purpose stream processing systems. However, the survey provides a more detailed examination of the specific window types and their resulting performance consequences.

Stream processing in serverless environments is new, and less work is focused on it today than on stream processing with the help of traditional SPEs. Works available today often compare different setups and implementations within traditional SPEs and only one system, such as Flink. Most stream processing work aims to understand what bottlenecks exist and how to tackle them. This includes works such as [11], [18], [21] and [22].

Performing comparable measurements between two different approaches has its challenges. The challenges include minimization of the differences in the testing environment. It can affect the evaluation results if the frameworks have different ways of executing logic, such as the programming language used, libraries, and available features.

For the evaluation results to be comparable, differences in the testing environment should be minimized, and one approach is to develop equivalent implementations of the logic in each framework using comparable code. Minor implementation differences regarding buffer sizes, memory allocation, and even debug printouts can drastically shift the performance evaluation metrics, making it look like the difference between the frameworks is much larger than it is. Such differences can be challenging to uncover and address without deep knowledge of the languages and algorithms used in the implementation.

Evaluation will usually include testing the same function over a set of data with slight changes to the configuration. The frameworks used and the tests performed will have much in common, making the results consistent with minor variance. However, evaluating different frameworks with each implementation can affect the results as the degrees of freedom in such evaluation is much larger. The environments and tests can often have more differences than commonalities, leading to increased difficulty in performing a fair and non-biased evaluation. These challenges could partially explain why most works focus on a more restrictive scope rather than a wide one. The smaller scope entails less variation and more reliable data.

4

Use Cases

Stream processing can be valuable in many different areas revolving around cyber-physical systems. Some use cases, such as those used in manufacturing, have been presented in Sec1.

Botev et al. have described a system for stream processing that uses aggregation to identify instances of fraud and irregularities concerning non-technical energy losses in the field of electricity infrastructure[23]. The solution mainly applies to developing countries, where energy losses can be as high as 50% due to theft and power siphoning.

Duvignau et al. define a system to analyze vehicular network data using stream aggregation in the context of querying data from individual vehicles as data sets[24]. A system of this type can help in both the areas of traffic control as well as energy balancing when it comes to the charging of electric vehicles.

4.1 Examples of use cases for Flink

Regarding Flink's financial aspect, its deployment must always stay active and consume resources no matter the streaming application's actual utilization. Over-provision can increase costs associated with Flink being deployed and active during low utilization. Use cases better suited for this framework include applications with consistent demand throughout the day, week, month, and beyond.

Flink is a framework that can leverage state management and is closer to hardware than a serverless counterpart, offering an advantage for use cases that can benefit from it. An application implementation that can use optimization techniques related to the hardware is best suited for Flink and other traditional stream processing frameworks. Use cases with strict requirements for performance are better suited for this framework.

Some techniques, such as PiLisco, have been presented in Sec. 1. Other examples of this approach could be found in the work of Walulya et al., which describes a system that shows better streaming aggregation performance via modifications to the underlying infrastructure and better hardware utilization, achieving higher efficiency and throughput[25].

Logic running for extended periods (longer than a few minutes) will also be best

suited for Flink. It is well suited due to the state being preserved and quickly available at a later point in time. Use cases running complex logic requiring long computational times are better suited for this type of framework. An example of utilizing state management to optimize the application is described by Van Rooij and his team, which developed a system to reduce delays and improve responsiveness when tuples arrive late. The system uses predictive analysis based on the received data to achieve this[26].

Flink is a well-established SPE and, as such, has a great deal of documentation and community resources that can aid a developer in building streaming applications. The availability of support resources, on its own, can be a considerable argument in favor of Flink in developing applications requiring large amounts of documentation and examples available during the development process, as it can decrease the time features take to implement.

Lastly, other requirements, such as determinism and correctness guarantees, might be present and must be considered. Gulisano et al. talk about the role of event-time order in data streaming analysis. Due to the nature of multi-threaded applications and their ability to produce results out of order, it is essential to produce deterministic results no matter in which order the data arrives. Use cases with strict correctness requirements require less implementation from scratch in this type of framework. While the same result is possible within the serverless setting, it requires the implementation of such logic from scratch. The watermarks have a crucial role in achieving this. Notifying the data streaming application that all the late tuples should have arrived by some point in time can, depending on the correctness guarantees, offer a reasonable trade-off between latency and the correctness of the produced result[27].

4.2 Examples of use cases for OpenWhisk

OpenWhisk's advantages in stream processing come primarily from its versatility. Unlike Flink, which can execute code in Java and Scala, OpenWhisk can execute code within a container. Possible executables involve any arbitrary code meeting the computational requirements and being supported by the given architecture and container environment. Use cases within established systems written in various languages and might require many adapters make OpenWhisk a good choice due to its versatility.

A smaller amount of work has been done in the field of serverless systems, as it is a relatively new concept—one such work by Taibi et al., presented in Sec.3. The paper talks about how serverless applications get progressively more complex as more self-sustained pieces are added together and can be challenging to grasp, as well as present difficulties in debugging such systems.

Yu et al. studied the performance and latencies incurred by a serverless application regarding data storage. They have concluded that serverless applications do not offer a single one-fits-all solution for passing data between different application parts. While some methods worked well for smaller data, they performed worse with larger volumes. The team has developed their way of passing the data using

"data buckets" and performed performance evaluations for data in memory and dedicated S3 storage. The findings indicate that an application must be made more non-serverless for optimal outcomes since data must be stored in a specific location. So while resource provisioning, autoscaling, logging, and fault-tolerance are all the benefits of serverless applications, the additional latency incurred might make them a poor fit for latency-sensitive applications[28].

OpenWhisk is relatively new and, as such, has a limited amount of documentation and examples available. It can lead to increased times during implementation and testing as the support resources can be scarce and hard to come by. Use cases that use this type of framework should therefore be ready to implement some logic from scratch and not rely heavily on well-written guides about best practices and boilerplate code that usually exists freely available on the internet. Serverless computing is still very much in its infancy and is an open area of research, which can lead to situations with minimal support available.

Furthermore, a point can be made about the serverless stream processing financial model. Use cases with a low or uneven distribution of resource utilization can be financed with lesser funds for the same amount of computational power. This type of financial model makes the framework a good choice for applications looking to be deployed with the lowest incurred costs.

Sec. 2.10 and 6.2 discusses some other advantages and disadvantages of serverless systems.

4.3 Picking the right tool for the job

The choice between a traditional SPE and a serverless stream processing falls to the individual use case. A traditional SPE should be the primary choice if the application is used frequently and has high through and low latency requirements. If the application is part of a more extensive system consisting of various programming languages and solutions, OpenWhisk can offer seamless integration.

Applications that run infrequently or for long periods will make more sense financially deployed on an OpenWhisk cloud instance. If an application implementation wishes to leverage optimization techniques relying on the hardware, traditional SPE is often a better fit. Small, stateless applications are a good fit for serverless applications.

Adding more features, state management, and complexity can quickly outgrow and incur costly maintenance and future development costs that are difficult to measure. Complexity should be distinct from scalability, as serverless platforms offer superb scalability options for applications and instant deployment paradigms.

It is also essential to consider what types of operators are required in the system and if they need to be implemented or are already present in the library. Hirzel et al. compare different types of sliding window aggregation algorithms and compare the pros and cons of each[17]. As shown, the choice of the framework will define what types of windows are available for use and should therefore be considered at the application design stage.

The example presented in Sec. 2.3 could be implemented in either of the frameworks due to the simplicity of the task. From the point of cost-effectiveness, OpenWhisk, with a pay-per-use model, offers good efficiency for low-medium rate streams that have no activity a lot of the time. As low delay and high throughput are not essential to produce a result in this application, the choice is dictated by other factors, such as the specifics of the implementation for a particular system that detects cars.

5

Features and Capabilities

5.1 Description of key features and capabilities of Flink

Flink supports both processing of streams as well as batch processing. Aimed at businesses and large-volume data operations, it has many features and configuration options across many use cases.

One of its compelling features is the ability to use many different streams as input; it supports various interfaces, such as file-based sources, message queues, socket streams, and custom sources. This ability is essential to incorporate Flink into an already existing system.

However, Flink requires the data to be converted to one of its internal objects, such as DataStream, to be used with its stream processing libraries. This requirement comes from the fact that the data must be serialized between different parts of the Flink to perform efficiently. DataStream implements Serializable and can transform arbitrary data into a native type that Flink understands[29].

Flink's limitations include primarily supporting only two languages, Java and Scala. However, it does offer experimental support for Python and SQL as well. As the primary choice, Java allows parts of the code to be unit tested and is generally debug-friendly with the ability to pause the execution of selected parts of the code. On the other hand, getting useful debug info from the Flink runtime is more complicated, as it will often produce errors that do not convey the precise root of the exception.

One of Flink's strengths is its close reliance on the hardware compared to OpenWhisk. Tangwongsan et al. explored various general incremental sliding-window aggregation improvements and optimization. However, while the performance benefits are measurable, they mostly rely on hardware-side factors such as memory allocation and cache miss minimization. Moreover, these improvements can be hard to use in the serverless setting, where small functions are run in the local scope, and the memory stored variables are deinitialized on return[21].

Flink's primary use cases it performs well in include real-time data processing, data pipelines, batch processing, and complex event processing.

5.2 Description of key features and capabilities of OpenWhisk

OpenWhisk is mainly offered as a service on the 'IBM Cloud' but can also be deployed to any supported hardware using Kubernetes. Its primary function is the ability to create actions self-contained, lightweight pieces of code that are being executed in response to events.

The only two ways of accessing the OpenWhisk API are the URLs that actions expose or call the API from within the code. The second way can only be used on the server side, leaving any events with only one point on entry, the generated URLs. These can take the form of an arbitrary HTTP request to post, get, or, in other ways, transfer any data necessary.

The actions can be executed via triggers and rules. In general, an event will request the OpenWhisk API endpoint. For example, the event can be a completed purchase in e-commerce, a sensor reading of an IoT device, or the creation of a new social media post.

OpenWhisk offers many options to create and configure actions using virtually any programming language. Furthermore, it will be run almost immediately if the language is interpretable, like Python or JavaScript. In the case of compiled languages like C and Java, a designated Docker container will be initialized for that action, as it was run on a newly installed OS.

Due to the nature of OpenWhisk, all actions are executed in separate containers and do not keep a state once execution is finished. Therefore, any variables and results should be passed to the following action or stored in a persistent database (usually using an action that stores data in a database).

OpenWhisk offers nearly automatic scalability options. As it is deployed in a Kubernetes environment, more containers can be initiated and connected to increase the computational capabilities of the system as a whole.

Drawbacks of the OpenWhisk system include the minimal ability to debug the deployed actions. In addition, due to the aim of instant code deployment, there is no easy access to the environment where the code is being executed. Logs are the only available way to look into the execution flow, and the logic is tested once actions are deployed.

OpenWhisk prominent use cases include event-driven processing, microservices, and web backends. In addition, it can handle data processing, albeit with higher overhead than dedicated streaming frameworks.

5.3 Comparison of the features and capabilities

In general, Flink and OpenWhisk are different systems designed for different purposes. While there are some similarities between the two, they are primarily different.

Flink is meant to be efficient, pipelined, and able to process large volumes of data with minimal overhead. It offers an API specifically designed to process data streams, with many operators for mapping, filtering, and aggregating the data. A work by Gulisano et al. discusses parallelization overheads within stream processing and proposes a way to avoid those, improving the scalability and performance of streaming systems[18].

OpenWhisk aims to be a "one size fits all." framework that can, with minimal effort, be retrofitted into an existing system and connect various parts of the application. When looking into different ways to connect actions, OpenWhisk offers nearly limitless possibilities with rules, triggers, and action chains. The encapsulation paradigm works well for this approach but can also present increased complexity problems for larger systems. In addition, the serverless approach is still an active field of study, and best practices still need to be clarified as some research attempts to formulate.

Work has been performed in the context of optimizing operations often performed by streaming applications. For example, Gulisano et al. have shown some methods for implementing specifically stream "join" operations to increase performance in shared memory and determinism. These implementations can benefit use cases where the system is closely integrated with hardware and unsuitable for serverless solutions[22].

6

Ease of Use and Deployment

6.1 Ease of use and deployment for Flink

Flink's support for various windows aggregation algorithms is one of the largest compared to alternative stream processing applications[17]. Considering the importance of high performance in stream processing, Flink offers a solid foundation for any developer looking into implementing Flink's libraries in their work. Given well-written documentation and community support, developers can be sure that methods used in the Flink library are written with the best practices in mind and should include fewer bugs than a custom-made solution.

On the other hand, Flink only offers limited functionality other than directly related to streams. For example, it will not be able to run a custom web server for stream input if a stream is to be fed to Flink via HTTP requests. Instead, a separate software, such as Flask, could handle the reception of requests and forward them to Flink via one of Flink's input adapters, such as Apache Kafka. It will also not be able to execute code written in other languages that are supported natively.

The deployment procedure of Flink is very straightforward and can be done without many changes to the default configuration. However, to take advantage of the various extra features that Flink offers, it should be configured manually and tested, which requires knowledge and understanding of the system. These features include checkpoint functionality, backups, and custom storage for the stateful elements of the framework.

Deployment of the example application presented in Sec. 2.3 using Flink would likely result in over-provision of resources but provide good tools for analysis. Libraries included with the framework would require comparably low coding effort to implement the system. Other tools would need to be used with Flink to forward the data from the sensors to the SPE.

6.2 Ease of use and deployment for OpenWhisk

OpenWhisk is a framework mainly aimed at serverless environments, and its primary goal is to run actions as a reaction to events. Actions, triggers, and rules allow for configuration with a high degree of customization and re-configuration on the fly.

However, it lacks any library that would make it easy to configure stream processing, and any code related to it should be created as a custom implementation. So while custom implementation is possible, it may result in bugs that are hard to trace and detect, given OpenWhisk's limited logging and debugging abilities. As a result, bugs in the stream processing implementation could significantly decrease performance and even affect correctness in some cases.

The deployment of the OpenWhisk in its standalone is as simple as the deployment of Flink. However, the standalone mode is generally used for testing logic and has many limitations relating to the rate of action execution and, as a result, the maximal performance. A Kubernetes cluster of isolated Docker containers must be configured to deploy OpenWhisk in its full-feature variant. The configuration of Kubernetes is relatively complex and requires knowledge of the cloud computing domain to be done correctly. It also puts high requirements on the hardware and is meant to be deployed as a cloud instance. If the developer wishes to leverage the serverless paradigm, they have a choice of renting a set-up instance of OpenWhisk from IBM and financing it using a pay-per-use model.

Deployment of the example application presented in Sec. 2.3 using OpenWhisk would not result in over-provision as the framework's resources would scale down and idle at times when no new tuples are received. Coding effort would be comparably higher, as the logic would need to be implemented without native libraries. However, no other tools would need to be used to forward the data from the sensors to the SPE via HTTP requests, as OpenWhisk natively supports them and can receive and process data this way.

6.3 Comparison of the ease of use and deployment

The choice between Flink and OpenWhisk should generally involve an evaluation of the requirements for the application that will be running. If it is a complex system that uses many different stream operators or it has requirements for periodic backups, then Flink might be a good fit.

On the other hand, if the streaming operators in use are relatively simple in their design and logic, or even better, stateless, OpenWhisk has compelling features it can offer for this type of application. For example, suppose the application has a low rate of requests for action execution. In that case, it can be a more viable option financially, as it does not cost the developer anything when it is not actively executing actions.

Deployment of the SPE for the example presented in Sec. 2.3 would depend mainly on the system doing the computation and its connection to the system that detects cars. If it is a local machine, Flink could be an excellent alternative to avoid OpenWhisk complexity related to deployment. On the other hand, if the machine can exist in the cloud and the detection system has internet access, renting OpenWhisk as a service is the easiest and quickest way to deploy the application.

One alternative solution worth considering is to opt for a cloud-based Flink deploy-

ment. As the data volumes change over time, it may be necessary to overprovision Flink to handle peak loads, but this would provide more extensive analysis capabilities. On the other hand, OpenWhisk is better suited for scaling up/down based on demand but may require more coding effort. Both cases have benefits and drawbacks and require consideration depending on the intended use.

7

System Overview and Architecture

This section explains the systems being compared and evaluated. It also defines the testing environment in which experimental evaluation is performed.

7.1 Explanation of Flink and OpenWhisk

Flink and OpenWhisk are frameworks that execute applications and, as such, need to be deployed and configured to execute operations on streams and be evaluated.

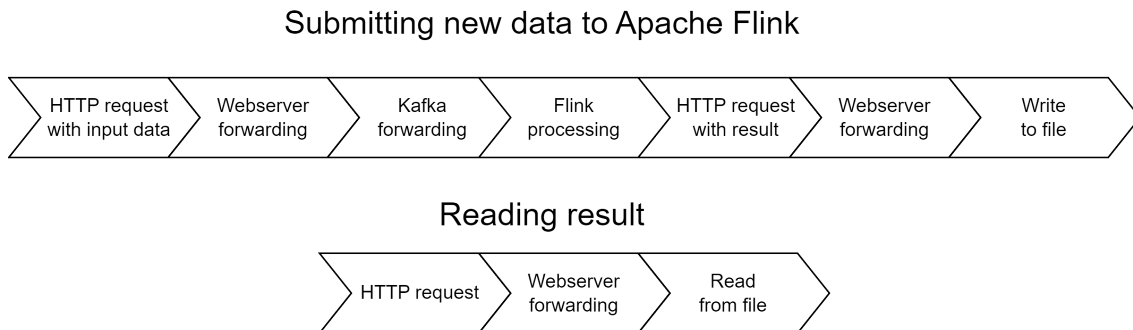
The deployment and configuration of Flink is a manageable and well-documented task that does not present a challenge when executed on a system that meets all of Flink's requirements. In addition, it has a comparably lower overhead and requirements than OpenWhisk. Running the Flink framework is done via a service that handles the execution, querying, and managing of the job tasks with the actual applications. It exposes a web dashboard related exclusively to Flink, visually presenting information about streams, load, tuples, and other helpful debugging info. It is distinguished from the Sec. 7.2 dashboard. The jobs are executed via the command line, after which the dispatcher handles the newly created job and waits for the new data to arrive at the input point.

On the other hand, OpenWhisk, in its standard deployment mode, has a more complex deployment procedure than Flink and allows for a very high degree of configuration. In addition, it is designed to be run in the cloud on a cluster of servers using Kubernetes. Therefore, a good understanding of cloud architecture is required to configure it correctly for cloud operations. Alternatively, IBM has the service all configured and ready to be rented out on a pay-per-use basis, which moves the entire server setup and maintenance away from the developer's area of responsibility. There is an option to run it in a standalone mode that will execute a Java-based version of the OpenWhisk in a single instance and allow for actions to be deployed and executed. However, it has significant drawbacks, such as the inability to configure limitations. The most prominent is a limitation of 120 actions per minute, meaning that the input should be batched if more than 120 tuples per minute must be supplied to the application. It is also worth noting that if the result is to be fetched at the same rate as the input, the rate is halved, leaving room for at most 60 combined write-read operations per minute.

Feeding new data and fetching results data is usually done via a pipeline of multiple

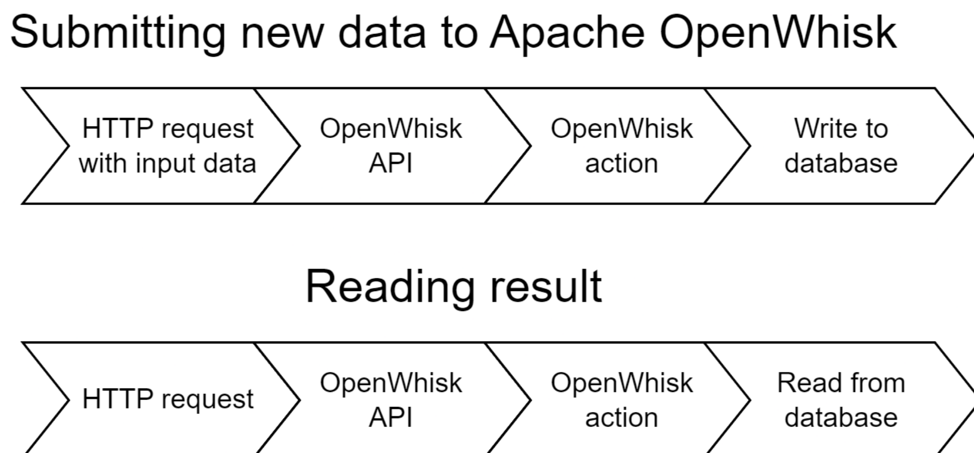
programs. This work selected Apache Kafka as an input to Flink, which serves as a message and queue system connected directly to the Flink input. In addition, a Flask web server receives requests and stores result data locally in a file. It acts as a Flink output sink, to which Flink sends the processed data at the end of the operators' graph. The reason for feeding new input via HTTP requests was selected to make the evaluation of Flink and OpenWhisk more comparable, as OpenWhisk only accepts HTTP requests as input. See Fig. 7.1 for the Flink Pipeline.

Figure 7.1: Flink pipeline



OpenWhisk pipeline is more straightforward than Flink's because it is natively HTTP only. It involves a direct connection to the OpenWhisk API and subsequent execution of actions. As a serverless framework that does not keep a state between actions, all the stateful actions should write the result to the database, which is different from Flink, which can keep a state internally. See Fig. 7.2 for the OpenWhisk Pipeline.

Figure 7.2: OpenWhisk pipeline



7.2 Testing environment

A web dashboard (a dynamic webpage) was created to automate the creation and visualization of the stream's input and output. The dashboard consists of two tables,

one for input tuples sent to the streaming application and one for output to get the results from the streaming application. It is shown in Fig.7.3. It can switch between two modes of framework used (Flink or OpenWhisk) and multiple benchmark modes with different computational load types.

Furthermore, it can be configured with an interval for repeating the HTTP requests to the tested service and payload size. Size can mean different things depending on the test being executed, but the main goal is to create batches of a specific size and send these all at once with each request. Most tests create the tuples using random values for the scheme.

Various inputs to the Flink framework, such as sockets, files, Kafka, and storage in memory, were evaluated. While some methods allow for high performance and processing throughput, they lack a comparable counterpart in OpenWhisk. When using OpenWhisk, the input must be provided through HTTP requests due to the platform's design. It is necessary because OpenWhisk action or trigger can only be executed and triggered via an HTTP request, and the comparison should have as few differences as possible between the systems. One such input that Flink supports is Kafka, and it is relatively easy to allow Kafka to emit the messages received from a simple HTTP web server.

The dashboard serves as a client. It generates tuples according to a specification and sends them to a local application, OpenWhisk or Flink. It was configured to automatically perform many instances of the same test with different variables for interval and size. Interval defines the time between requests, while size controls the benchmark-specific variable, for example, the size of a tuple. After the tests, each test's data is recorded and exported.

Each test generates two files, 'latency' and 'throughput.' The first file includes the latencies of each request which can be used to conclude the average latency, for example. The second file includes the data on throughput for each test; it can be used to understand how many packets were processed versus how many were dropped due to the high receiver load.

Two different scripts were created to generate visual graphs illustrating the data for throughput and latency. The graphs' x-axis is the size of the test, and the y-axis is either the percentage of the successfully sent data tuples or the latency of the response.

7.2.1 Dashboard

The dashboard described in Sec. 7.2 is illustrated in Fig. 7.3. It includes features for a selection of current frameworks and tests indicated with labels one, two, and three. The tables for input and output are indicated with labels four and five. Input fields indicated by label six are configuring the interval and batch size. Label seven points to timestamp information used in debugging. A download button for the export of statistics is indicated by label eight.

Other features not visible on the dashboard front page yet configurable directly in

the code include automatic benchmark execution that loops multiple tests iterations. After each test, a delay is inserted to keep test results separate and not interfere. The results are automatically labeled and exported into files.

Figure 7.3: Dashboard used in the testing environment. 1) Current framework selection indicator 2) Framework selection buttons 3) Test selection buttons 4) Input table 5) Output table 6) Interval and batch size configuration 7) Debug information displaying timestamps 8) Download button for exporting the statistics

7.2.2 Test batching

As mentioned in Sec. 7.2, tests with different configurations can be batched and run automatically, producing and exporting the results that can later be assessed and used to generate visual plots.

When the test batch is started, it is run until even and stable latency responses can be observed. This warmup period usually takes up to 30 seconds. At that point, the frameworks are initiated, the memory is allocated, and the caches are filled. As the test is run, the following latencies will have predictable, stable values with minor variations. Once such equilibrium is achieved, the test is restarted from the beginning and left to be run on its own until the completion. Restart ensures that the outlier values from the first couple of requests before everything is up and running are not included in the results.

The evaluation phase includes delays between configurations to ensure that no tuples are in transit when the data is recorded in a file and that frameworks are not

processing old data when new tests are started. All periods are configurable but have been set to static values throughout the evaluation. The phase in which evaluation is performed, and results are recorded was set to five minutes. The cooldown period was set to ten seconds. It is the time delay after the evaluation during which no new requests are generated but replies to the old requests can still be received and included in the result data. Once five minutes and ten seconds have passed, the data is automatically exported and downloaded as a file. No new data points can be added to the set. Lastly, a grace period, a delay set to one full minute, is taking place to ensure that frameworks load have minimized, no more processing of old data is occurring, and new tests can begin after the delay.

7.3 Purpose of the comparison

The comparison performed with this setup and the help of the dashboard described in sec 7.2.1 only collected the performance metrics of latency and throughput used in plotting a visual graph. Two applications are tested, a stateless and a stateful one. The stateless application is named "KNN," and its implementation details are described in Sec. 7.4, while the evaluation results are described in Sec. 8.2. The stateful application is named "Twitter," and its implementation details are described in Sec. 7.5, while the evaluation results are described in Sec. 8.3.

The dashboard runs applications as benchmarks with different sizes and intervals for collecting multiple data points. The different types of load can then be compared between the systems.

7.4 Implementation of "KNN" application

As a stateless application does not depend on a persistent state, the input of a new tuple immediately generates a response with the result. Pseudo-code is presented in algorithm 1. The code is based on Kotlin language, closely related to Java. In line 2, the incoming JSON object is converted to an appropriate object. Line 8 instantiates the iris dataset and KNN object, which implements the k-closest neighbors algorithm. 'For loop' begins at line 10, goes through each incoming tuple, and calculates a prediction assigned to the list of predictions initiated in line 9. Lastly, a response with the current timestamp is created and returned at lines 15 and 16.

The graph in Figure 7.4 provides an overview of how the "KNN" application operates.

7.5 Implementation of "Twitter" application

As a stateful application that depends on a persistent state, the application has two main parts. The first part acts as input and stores the tuples in a database. The second part acts like a watermark (see Sec. 2.5) and produces a result calculated from the tuples stored in the database.

Algorithm 1 addKNN

```
1: function addKNN(args)
2:     dataArray ← args?.getAsJsonArray("data")
3:     if dataArray is null then
4:         return { "body" : { "success" : false, "error" : "data is null" } }
5:     end if
6:     data ← Gson().fromJson(dataArray, Array<KNNArray> :: class.java)
7:     print "add: data"
8:     knn ← KNN()
9:     predictions ← mutableListOf()
10:    for tuple in data.mapNotNull{it.values} do
11:        prediction ← knn.predict(tuple)
12:        predictions.add(prediction)
13:    end for
14:    print "predictions: predictions"
15:    time ← System.currentTimeMillis()
16:    return { "body" : { "timestamp" : time, "result" : predictions } }
17: end function
```

The first part is presented as a pseudo-code in the algorithm 2. In line 2, the incoming JSON object is converted to an appropriate object. Lines 5 to 10 are related to establishing contact with the database and reading the database contents. Lines 11 to 15 contain a 'for loop' that writes the new tuple data and updates the related field for the current count. Lines 16 to 18 write the changes back to the database. Finally, line 21 returns the result of the database operation; it is not related to the tuple result itself.

The second part is presented as a pseudo-code in the algorithm 3. Lines 2 to 7 are related to establishing contact with the database and reading the database contents. Lines 8 to 13 contain a 'for loop' that, for each user, sorts the entries according to the associated count and returns the top three most used words for each user. Finally, line 15 returns the result as a JSON object.

In Figure 7.4, the graph illustrates the operational process of the "Twitter" application.

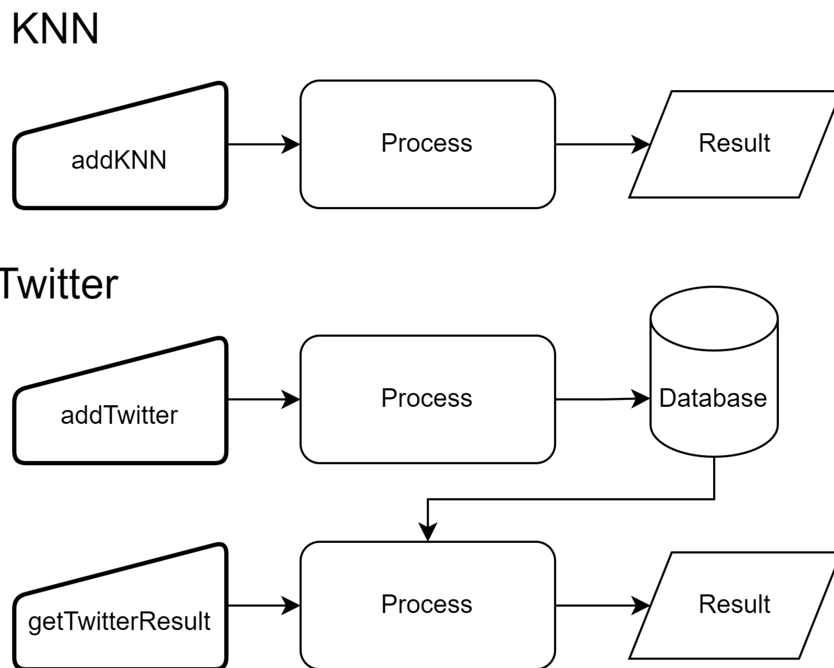
Algorithm 2 addTwitter

```
1: function addTwitter(args)
2:     tuple ← arg?.getAsJsonObject(”data”)
3:     print "add: tuple"
4:     result ← null
5:     MongoClients.create(uri).use { mongoClient in
6:         database ← mongoClient.getDatabase(”streaming”)
7:         collection ← database.getCollection(”twitter”, TwitterData ::
   class.java)
8:         query ← Document(”user”, tuple.user)
9:         existingDoc ← collection.find(query).first()
10:        wordsDoc ← existingDoc.values.toMutableMap()
11:    for word in tuple.values do
12:            currentCount ← wordsDoc.get(word)?:0
13:            wordsDoc.set(word, currentCount + 1)
14:            print "word: word, currentCount: currentCount"
15:    end for
16:        update ← Document(”$set”, Document(”values”, wordsDoc))
17:        options ← UpdateOptions().upsert(true)
18:        result ← collection.updateOne(query, update, options)
19:        print "result: result"
20:    }
21:    return Gson().toJsonTree({ "success" : (result.wasAcknowledged()
   == true), "result" : result }).asJsonObject
22: end function
```

Algorithm 3 getTwitterResult

```
1: function getResult
2:     const client ← new MongoClient(’mongodb://172.17.0.6:27017’)
3:     await client.connect()
4:     const database ← client.db(′streaming′)
5:     const collection ← database.collection(′twitter′)
6:     const result ← {}
7:     const users ← awaitcollection.find().toArray()
8:     users.forEach(user → {
9:         const values ← user.values
10:        const sortedValues ← Object.entries(values).sort((a, b) → b[1] −
   a[1])
11:        const top3 ← sortedValues.slice(0, 3)
12:        result[user.user] ← top3
13:    })
14:    client.close()
15:    return { "body" : { "result" : result } }
16: end function
```

Figure 7.4: "KNN" and "Twitter" applications graphs



8

Evaluation

This section will cover the evaluation part of the work, the nature of the experiments, and discuss their results. The experimental evaluation covers both cases for stateful and stateless applications. The details of the system that the evaluation is performed on will be explained. The methodology of the tests performed is also described in this section. Furthermore, details observed and found relevant to the evaluation will be highlighted and discussed.

8.1 System Overview

A virtual machine (Oracle VM VirtualBox) running Ubuntu 22.10, Flink version 1.16, and OpenWhisk 1.0.0 with 'openwhisk-cli' version 1.2.0 was used. The virtual machine was configured as an 8-core Intel i9-13900K with 32GB of RAM.

Flink was running in its normal operational mode, while OpenWhisk was running in the standalone mode to avoid the additional complexity of clustering and Kubernetes, which is part of its original design and main use case.

The dashboard described in Sec. 7.2.1 was implemented in JavaScript and deployed with the help of a local Python-based 'Flask' web server. It uses libraries such as Bulma for the style and cosmetic parts and Axios to perform the HTTP requests. The rest of the dashboard is written using the standard JavaScript library. Data was exported from the dashboard in CSV format. Scripts for the creation of plots were written in Python with the use of the 'pandas' library.

Flink has many configurable ways to input data via a source. It does, however, not have a native way to input the data with HTTP requests, as it only runs a web server within the Flink dashboard for overview and diagnostics of jobs that are running. In other words, Flink cannot listen on a port and accept HTTP requests. This limitation is an expected design choice, as SPEs primarily aim to process streams and are not expected to handle communications outside the local machine, as that is usually handled by other packages appropriate for communication with the network. Keeping the framework's functionality limited is an excellent way to avoid unnecessary complexity within the framework and outsource the specific functionality to other packages.

Kafka is a suitable connector between the two, as Flink natively supports it and can be easily accessed with HTTP requests, as it listens on a local port, where the

new data tuples can be sent. Kafka is one of the packages that can be used for this purpose, and while it can do much more than accept HTTP requests, it is used exclusively as a communicator between the network and Flink. Furthermore, the network is limited purely to the local host, and the Flask web server handles the actual communication with the network outside the local host. Kafka acts as a buffer that temporarily stores messages from the web server and emits those at a rate appropriate for the subscriber, Flink.

Some implementation is necessary to store the data related to stateful operations in a serverless environment. Such an implementation can take many different forms. The traditional and standard way of doing it is using a database. A database does not have to be used, however. It can also be implemented as a file, variable, or persistent state. The primary consideration is that it can not be implemented inside the OpenWhisk framework, as the deployed containers are not persistent and can be shut down or reset anytime, as described in the documentation. While OpenWhisk has persistent settings and rules that can be written and read, they are not meant for application-specific variables and should not be used for this purpose.

A local or remote database is an implementation a streaming application will likely use, as it is the preferred way in many tutorials and examples[30]. Therefore, a choice was made to go with MongoDB. The database is deployed using Docker as an isolated container. The database can be reached locally by the OpenWhisk actions to write and read the data in JSON format or documents, as internally referred to within MongoDB.

8.2 'KNN'-test

To perform a test with a stateless application, it should receive a tuple, process it in some way and return the result, all without storing any persistent variables.

Such a test is suited well for serverless applications. It does not use any database since the operation is stateless. It can be performed over a large set of tuples to increase the workload. The result returned is a list of predictions for the tuples. Computations are similar to window aggregation, although on a much shorter time scale. Test of this type evaluates the system's ability to perform computations unaffected by any overhead that comes with storing persistent variables. In this case, no overhead is associated with contacting a database on the local host.

The KNN test is an implementation of the K-nearest-neighbors algorithm. An Iris flower data set, also known as Fisher's Iris data set, consisting of 4-dimensional tuples with float values[31] is used. When a request is sent to the stream processing framework, it is done in batches of a predefined size. Then, all the tuples are processed, a prediction of 3 possible choices is made, and one for each tuple is returned in the response. In this manner, the application can receive and classify tuples containing information about objects. In this case, the objects classified are flowers, classified according to type. Prediction is based on the four properties of each flower.

8.3 'Twitter'-test

To perform a test with a stateful application, it should receive a tuple, process it in some way and return a reply while storing the tuple as a persistent variable. In this case, the operation is a window aggregation, and the tuples belonging to a window are the persistent variables.

Such a test requires some extra steps to be implemented as a serverless application. Since the operation is stateful, some form of persistent variables are required. For that, a database is used. It can be performed over tuples of larger size to increase the workload. The result returned is a window aggregation. Test of this type evaluates the system's ability to perform computations that rely on state management. The overhead that comes with storing and reading persistent variables is present in this evaluation.

The Twitter test aims to aggregate strings and find out what words are the most used by a specific user. The request contains a username alongside a set of configurable sizes containing randomly selected words from a defined list. Once the stream processing receives the request, it is aggregated every 5 seconds, and the result produced returns the three most used words of all time for a specified use, as well as the total count for these three words. The operator is stateful and acts as a sum window with infinite size. It also sorts the results to return the top three most repeated values, an extra but comparably light operation.

8.4 General observations of OpenWhisk

The OpenWhisk has some inherent limitations when run in the standalone mode, unlike a Kubernetes deployment. These limitations have been discussed in Sec. 7.1.

When the limit is exceeded, the responses to action requests will return the error "429 TOO MANY REQUESTS". This limitation makes it impossible to run the actions more than one time per second (one action for sending data and one more action to fetch the result). To ensure no overlapping occurs and triggers the error over time, an interval of 1100ms was found to be suitable as the "fastest" possible rate.

A way of circumventing such a limitation is partial batching so that each request contains more than one dataset to be processed, which was a good solution for this limitation. However, suppose the request interval is short and the size is large. In that case, the system will inevitably get loaded to the maximum and might return various errors (such as ERR_NETWORK_CHANGED) for some requests while processing others once the load decreases somewhat.

If the processing algorithm does not require long computation times, there is yet another obstacle when sending data to OpenWhisk. If the HTTP request exceeds a specific size, OpenWhisk will deny the request claiming that the request is too long. This limitation can only be solved by splitting the request in two and sending one

after the other, at which point the previous issues regarding the maximum rate of requests come into play.

The performance and function of OpenWhisk in standalone mode can be affected by the configured Java heap size, as the standalone mode is essentially a Java application under the hood. In many long tests with large data sets, a Java out-of-memory error was observed numerous times when the size was set at 4GB. Changing the Java heap size via an environment variable to 8GB has solved the issues for the tests run within this paper's scope.

Evaluating the OpenWhisk accurately in its standalone mode presets some challenges related to computational overhead, as it exceeds the computational overhead of Flink and imposes limitations discussed in Sec. 7.1 and 6.2. Therefore, OpenWhisk must be appropriately deployed using Kubernetes and the related configuration file for use cases requiring a very high data submission rate and result polling.

OpenWhisk has also shown a significant deviation in latency times at the begging of some tests, which required some adjustments to the testing methodology. Tests and measurements were to be performed and recorded after an initial warm-up period once the latencies stabilized. This deviation is caused by the design of OpenWhisk, which starts and allocates resources for the containers lazily, or in other words, only when required and not earlier.

8.5 General observations of Flink

In the evaluation process, it was observed that Flink is sensitive to unhandled exceptions. These exceptions primarily stem from Kafka, which sends out outdated data from previous tests. This results in an endless loop of restarts. As soon as the job was restarted, Kafka began to send out the same data that had caused it to crash and forced Flink to restart the job. Other than explicitly handling such an exception in the code, the only solution would be to flush the Kafka and clear the Kafka topic of any old data. This issue should be considered when designing the application, as incorrectly formatted data sets can easily throw an exception in multiple parts of the code. The input should therefore be sanitized and checked, and any exceptions handled accordingly.

8.6 Results

Tests performed with the help of the dashboard described in Sec. 7.2.1 produced two types of results; throughput and latency. The throughput is collected as the total size of the input packages sent during the test and responses (acknowledgment of successful reception) received. Since the length of each test is statically set to be five minutes, the throughput is expressed in units over time, that is, tuples processed per five minutes.

The second metric collected is latency. The latency is calculated at the input request. Therefore, it does not include the entire path of all the intermediary systems. Latency

shows how quickly the pipeline can consume new data and sustain a configured throughput rate without dropping requests.

On the other hand, if we consider the time it takes for the input request to be handled and the response to be received, we end up with an accurate metric for the system load and the time it takes to process a request for a specific size. This latter metric has shown a stable linear relation between the request size, the system load, and the observed latency value. Moreover, it shows a linear increase with a bigger input size or shorter request interval, which is expected and much more interesting than a metric limited by the window calculation interval.

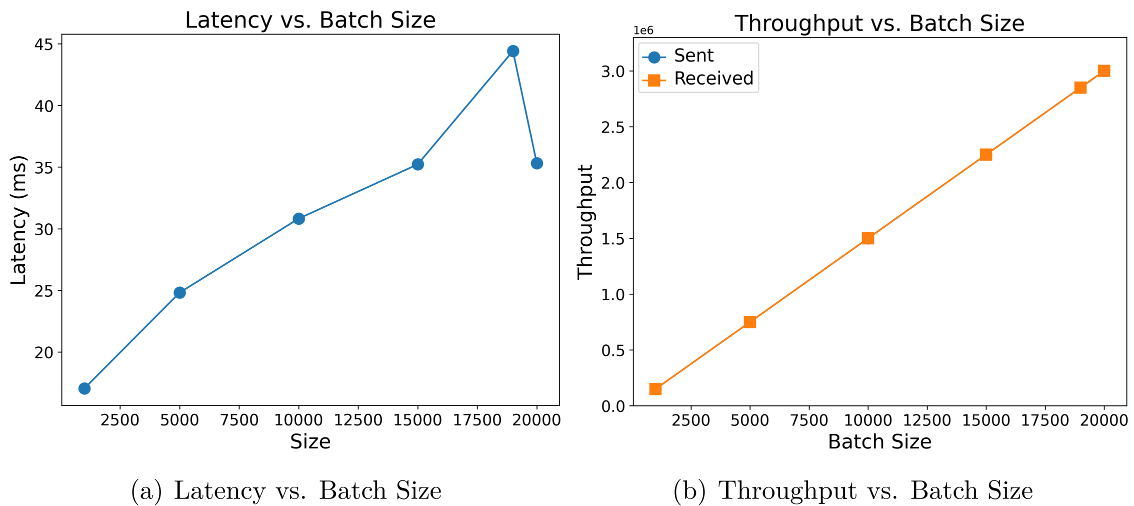
The results presented below have been produced by running the tests for 5 minutes for each size configuration. Each size has been tested with two intervals, 2000 ms and 1100 ms. The second option creates nearly twice the workload and approaches the standalone OpenWhisk's maximum capacity without risking the limit error's triggering.

Care has been taken to ensure that the CPU was not busy performing background tasks, and an initial grace period before starting the benchmark was allowed, during which the results were not recorded. This period ensured that all the systems were running, all containers had started up, and all the one-time calculations at the start were executed and would not influence the result.

Test points resulting in no package throughput for an OpenWhisk test mean that at that size, the request entity is too large to be processed and is denied by the OpenWhisk entirely, which can be seen in Figs. 8.5, 8.6, 8.7 and 8.8.

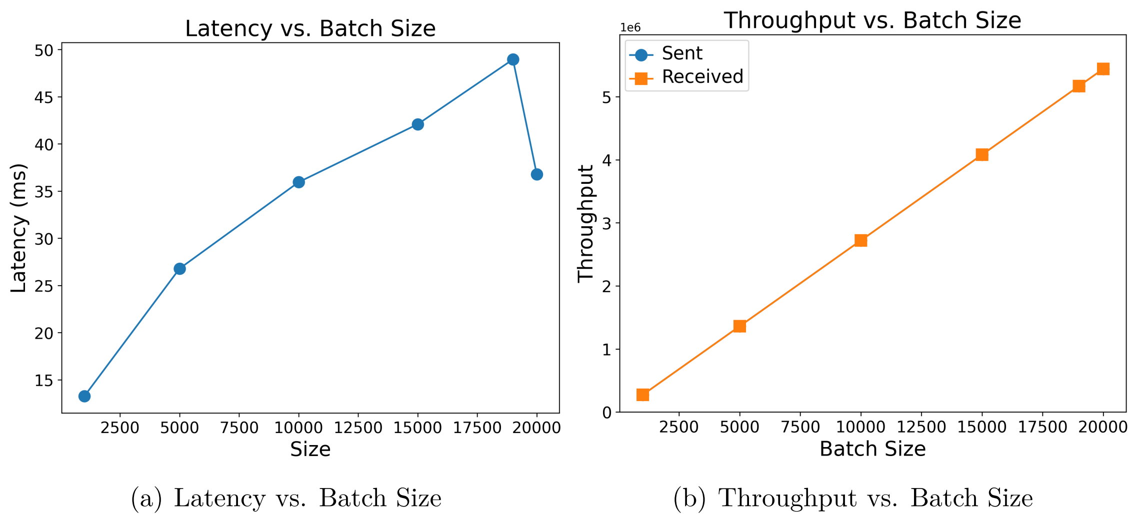
Figures 8.1 and 8.2 show metrics of latency and throughput for a Flink KNN test described in Sec. 8.2 with an interval of 1100ms and 2000ms respectively. Comparable performance can be observed, indicating that stateless applications scale very well with Flink. Throughput can be doubled with negligible effect on latency and no dropped packets in this test.

Figure 8.1: Flink KNN Test (2000 ms interval)



(a) Latency vs. Batch Size (b) Throughput vs. Batch Size

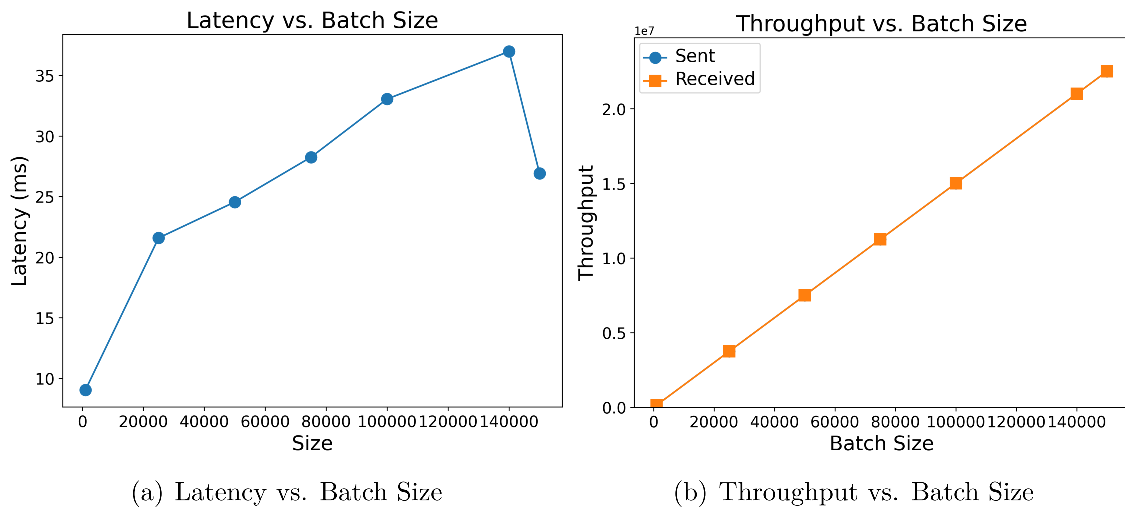
Figure 8.2: Flink KNN Test (1100 ms interval)



(a) Latency vs. Batch Size (b) Throughput vs. Batch Size

Figures 8.3 and 8.4 show metrics of latency and throughput for a Flink Twitter test described in Sec. 8.3 with an interval of 1100ms and 2000ms respectively. Comparable performance can be observed, indicating that stateful applications also scale very well with Flink. Throughput can be doubled with some effect on latency, and no dropped packets in this test. The biggest increase in latency is about 10ms, which is acceptable given double throughput.

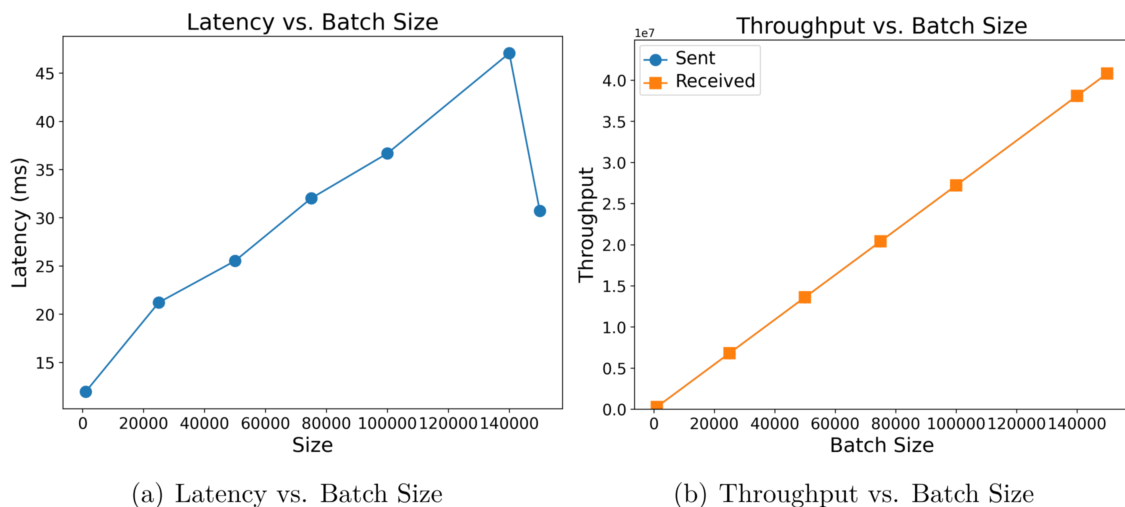
Figure 8.3: Flink Twitter Test (2000 ms interval)



(a) Latency vs. Batch Size

(b) Throughput vs. Batch Size

Figure 8.4: Flink Twitter Test (1100 ms interval)

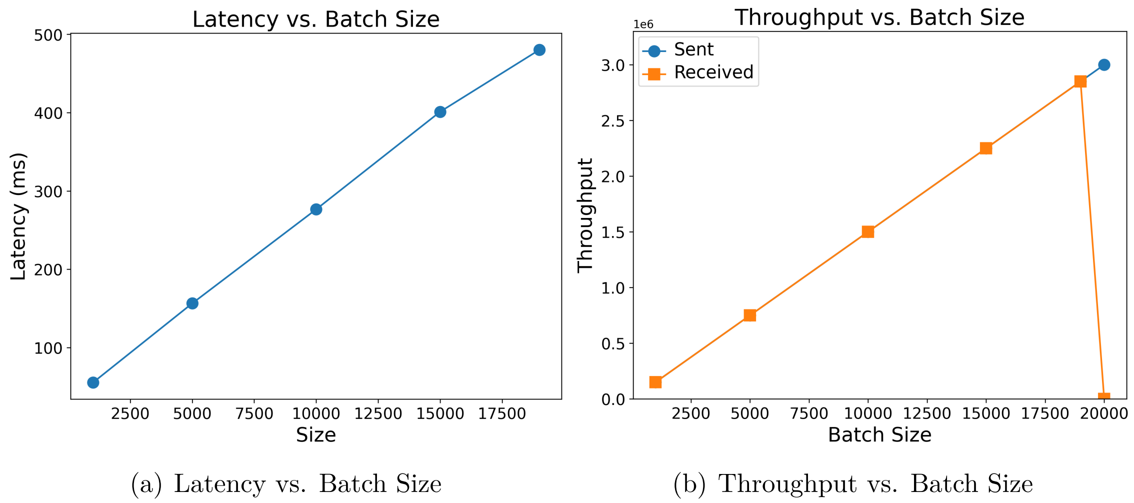


(a) Latency vs. Batch Size

(b) Throughput vs. Batch Size

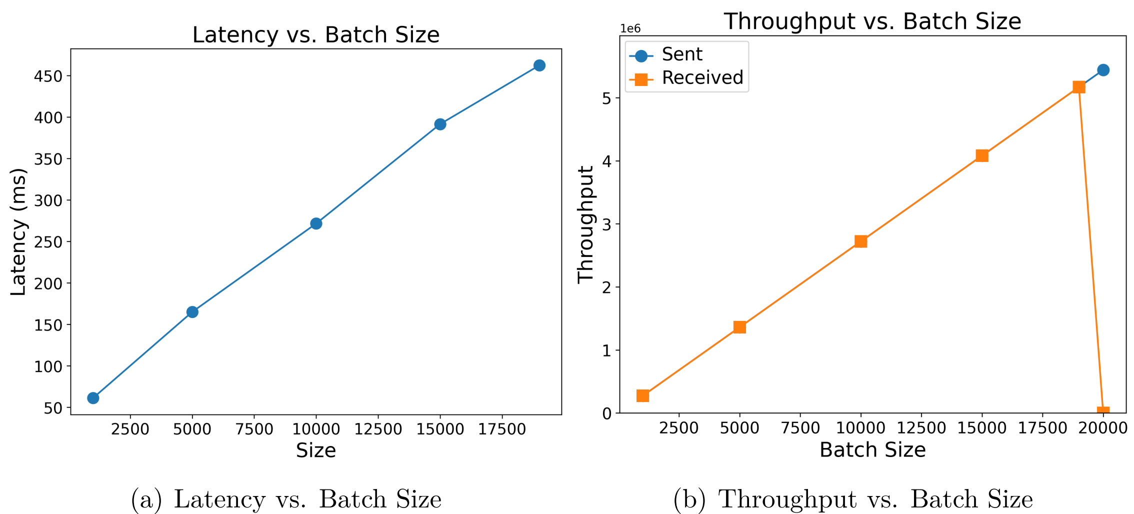
Figures 8.5 and 8.6 show metrics of latency and throughput for an OpenWhisk KNN test described in Sec. 8.2 with an interval of 1100ms and 2000ms respectively. Comparable performance can be observed, indicating that stateless applications scale very well with OpenWhisk. Throughput can be doubled with little effect on latency. Dropped packets can be observed at large batch sizes, related to the request size being too big; it has no connection to the increased throughput. The biggest latency increase is around 50ms which is acceptable given the latency magnitude of 500ms.

Figure 8.5: OpenWhisk KNN Test (2000 ms interval)



(a) Latency vs. Batch Size (b) Throughput vs. Batch Size

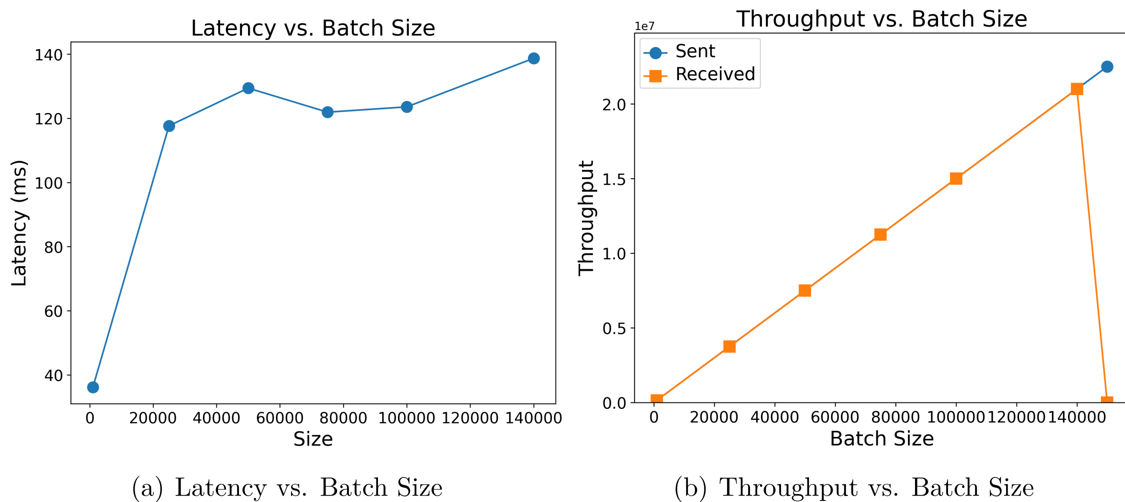
Figure 8.6: OpenWhisk KNN Test (1100 ms interval)



(a) Latency vs. Batch Size (b) Throughput vs. Batch Size

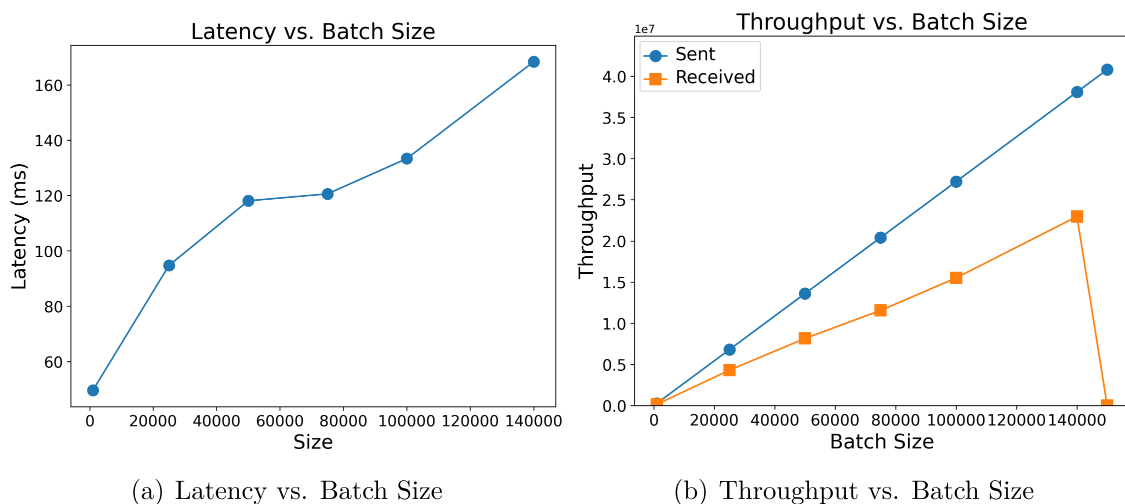
Figures 8.7 and 8.8 show metrics of latency and throughput for an OpenWhisk Twitter test described in Sec. 8.3 with an interval of 1100ms and 2000ms respectively. Worsen performance indicates that stateful applications are the most difficult to scale well with OpenWhisk. Throughput can be doubled with some effect on latency. Dropped packets can be observed throughout the second part of the test, related to the increased throughput. The biggest latency increase is around 20ms which is acceptable given the latency magnitude of 140ms. It is, however, clear that it is increasing fast with additional throughput, and the framework cannot keep up at a certain point. Dropped packets can be observed at large batch sizes, related to the request size being too big.

Figure 8.7: OpenWhisk Twitter Test (2000 ms interval)



(a) Latency vs. Batch Size (b) Throughput vs. Batch Size

Figure 8.8: OpenWhisk Twitter Test (1100 ms interval)



(a) Latency vs. Batch Size (b) Throughput vs. Batch Size

9

Performance and Scalability

As described in 1.1, this work compares the performance and scalability of Flink and OpenWhisk framework. The results present in Sec. 8.6 are discussed in this section.

9.1 Discussion of Flink's performance and scalability

Looking at the results presented in Sec. 8.6, Flink shows a maximum latency of less than 50ms across all performed tests. The throughput is linearly increasing with size, with no lost packages. However, a dip in the latency relating to the last size configuration has been observed across multiple tests and runs. The current theory is memory allocation, as the jump between two of the last sizes is not comparable to the size increase between other points. It could indicate that a minor increase requires little additional memory allocation and incurs less cost than otherwise during the memory allocation as the first bigger tuple is received.

9.2 Discussion of OpenWhisk's performance and scalability

Results presented in Sec. 8.6 show how OpenWhisk performs regarding latency and throughput. A maximum latency of 500ms can be observed across the tests. The throughput shows a primarily linear relationship with the size of packets sent. Some package loss occurs during a Twitter test set to 1100ms interval. During this test, some packages are denied by the OpenWhisk framework as the rate at which packets arrive can not be sustained regardless of their size, as the rejection is based on the actions per second metric. As the packet size is increased past the allowed range, the OpenWhisk framework will reject the package as the request is too long to be processed, even if the stream rate is low.

9.3 Comparison of the performance and scalability

Comparing the results presented in Sec. 8.6, we notice Flink performs better than OpenWhisk in latency and throughput. The latency observed is, on average, times

or a magnitude lower for applications deployed with Flink. Sustainable data rates are also higher for Flink, and its limitation on packet size is less restrictive than OpenWhisk's.

10
Conclusion

Data volumes are rapidly increasing, primarily due to the increasing usage of IoT devices. This can be observed in Fig. 1.1 and 1.2. Stream processing has been established as an effective way to process data streams continuously. As steams evolve, the computational needs and requirements associated with processing them change. The traditional SPEs lack the necessary elasticity to efficiently adapt to changing requirements and optimize computational resource usage. As more data is generated, costs associated with the analysis and processing of it increase. This means that traditional SPEs are prone to over-provisioning and not utilizing available resources to a high degree, and it can become a problem requiring reconsidering available deployment options to keep the costs down.

This work has implemented a prototype SPE-like API for OpenWhisk to be utilized similarly to the traditional SPEs. It has shown that streaming applications can be implemented in both frameworks and highlighted the differences in outcomes. It was shown that with the correct code and logic, an API similar to that of Flink could be similarly implemented in OpenWhisk. With the pay-per-use model, costs can be reduced as the framework runs idle.

The evaluation of the frameworks has yielded findings that are outlined in Section 8. Tests performed included a stateless application utilizing *map()* API of Flink and the prototype equivalent for OpenWhisk. For a stateful application test, the second test utilized *windowAll()* and watermarks API and prototype equivalents. The results indicate that Flink possesses greater capacity and performance for workloads comparable to OpenWhisk's. It is easier to write applications due to the provided APIs and existing resource base consisting of communities and applications written for Flink. It can be a good choice for use cases that look for robust and well-established solutions.

Looking at the results for stateless applications, the latency increase in the case of OpenWhisk has been around 500-900% increasing with the batch size. The bandwidth has been equivalent for both Flink and OpenWhisk, meaning that both frameworks handled the data rate as expected, although only up to a specific size, after which OpenWhisk denied the request due to it being too long. In the case of stateful application, a 300-400% increase in latency was noticed. During the most intense part of the test, bandwidth was about 50% lower for OpenWhisk due to many denied requests due to the action limit set in OpenWhisk's standalone mode.

Future work in this direction could explore more generic implementations of the APIs related to OpenWhisk that could be more easily used in applications, leading to faster development time while relying on these APIs. Development of such API would significantly close the gap between the traditional and serverless frameworks regarding development coding effort.

In conclusion, both Apache Flink and Apache OpenWhisk offer valuable options for stream processing, with their respective strengths and considerations. The choice between the two frameworks depends on the project's requirements, the need for advanced features, the expected development and coding effort, and the financing strategy.

Bibliography

[1] M. Mohammadi, A. Al-Fuqaha, S. Sorour, and M. Guizani, "Deep learning for iot big data and streaming analytics: A survey," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 4, pp. 2923–2960, 2018.

[2] V. Gulisano, "Motivations and challenges for stream processing in edge computing," in *Companion of the ACM/SPEC International Conference on Performance Engineering*, 2021, pp. 17–18.

[3] D. Reinsel, J. Rydning, and J. F. Gantz, "Worldwide global datasphere forecast, 2021–2025: The world keeps creating more data—now, what do we do with it all," *IDC Corporate USA*, 2021.

[4] *Exploding topics - iot stats*, `https://explodingtopics.com/blog/iot-stats`, Accessed: June 19, 2023.

[5] *It spending and staffing benchmarks—it budget and cost key metrics by industry and company size*, `https://www.computereconomics.com/it-spending-and-staffing-benchmarks/`, Accessed: June 19, 2023.

[6] M. R. Ahmed, M. A. Khatun, A. Ali, and K. Sundaraj, "A literature review on nosql database for big data processing," *Int. J. Eng. Technol*, vol. 7, no. 2, pp. 902–906, 2018.

[7] R. Kraleva, V. Kralev, and N. Sinyagina, "Design and analysis of a relational database for behavioral experiments data processing.," *International Journal of Online Engineering*, vol. 14, no. 2, 2018.

[8] S. Madden, "From databases to big data," *IEEE Internet Computing*, vol. 16, no. 3, pp. 4–6, 2012. DOI: `10.1109/MIC.2012.50`.

[9] V. K. Fabian Hueske, *Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications*, 1st ed. O'Reilly Media, 2019, ISBN: 9781491974292.

[10] R.-Y. Chen, "An intelligent value stream-based approach to collaboration of food traceability cyber physical system by fog computing," *Food Control*, vol. 71, pp. 124–136, 2017.

[11] H. Najdataei, V. Gulisano, P. Tsigas, and M. Papatriantafilou, "Pi-lisco: Parallel and incremental stream-based point-cloud clustering," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, 2022, pp. 460–469.

[12] V. Gulisano, M. Papatriantafilou, Z. Chen, E. Hryha, and L. Nyborg, "Towards data-driven additive manufacturing processes," in *Proceedings of the 23rd International Middleware Conference Industrial Track*, 2022, pp. 43–49.

[13] Wikipedia, *Serverless computing*, `https://en.wikipedia.org/wiki/Serverless_computing`, [Accessed on June 20, 2023], 2023.

[14] L. Rao, "Picloud launches serverless computing platform to the public," *TechCrunch*, Jul. 2010, [Accessed on June 20, 2023].

[15] V. Gulisano, "Streamcloud: An elastic parallel-distributed stream processing engine," Ph.D. dissertation, Universidad Politécnica de Madrid, 2012.

[16] *Unix time*, Wikipedia, The Free Encyclopedia, [Accessed: June 20, 2023]. [Online]. Available: `https://en.wikipedia.org/wiki/Unix_time`.

[17] M. Hirzel, S. Schneider, and K. Tangwongsan, "Sliding-window aggregation algorithms: Tutorial," in *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '17, Barcelona, Spain: Association for Computing Machinery, 2017, pp. 11–14, ISBN: 9781450350655. DOI: `10.1145/3093742.3095107`. [Online]. Available: `https://doi.org/10.1145/3093742.3095107`.

[18] V. Gulisano, H. Najdataei, Y. Nikolakopoulos, A. V. Papadopoulos, M. Papatriantafilou, and P. Tsigas, "Stretch: Virtual shared-nothing parallelism for scalable and elastic stream processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4221–4238, 2022. DOI: `10.1109/TPDS.2022.3181979`.

[19] D. Taibi, N. El Ioini, C. Pahl, and J. R. S. Niederkofler, "Patterns for serverless functions (function-as-a-service): A multivocal literature review," 2020.

[20] J. Verwiebe, P. M. Grulich, J. Traub, and V. Markl, "Survey of window types for aggregation in stream processing systems," *The VLDB Journal*, pp. 1–27, 2023.

[21] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu, "General incremental sliding-window aggregation," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 702–713, 2015.

[22] V. Gulisano, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas, "Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join," *IEEE Transactions on Big Data*, vol. 7, no. 2, pp. 299–312, 2016.

[23] V. Botev, M. Almgren, V. Gulisano, O. Landsiedel, M. Papatriantafilou, and J. van Rooij, "Detecting non-technical energy losses through structural periodic patterns in ami data," in *2016 IEEE International Conference on Big Data (Big Data)*, IEEE, 2016, pp. 3121–3130.

[24] R. Duvignau, B. Havers, V. Gulisano, and M. Papatriantafilou, "Querying large vehicular networks: How to balance on-board workload and queries response time?" In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, IEEE, 2019, pp. 2604–2611.

[25] I. Walulya, D. Palyvos-Giannas, Y. Nikolakopoulos, V. Gulisano, M. Papatriantafilou, and P. Tsigas, "Viper: A module for communication-layer determinism and scaling in low-latency stream processing," *Future Generation Computer Systems*, vol. 88, pp. 297–308, 2018.

[26] J. van Rooij, V. Gulisano, and M. Papatriantafilou, "Tintin: T ravelling in t ime (ifn ecessary) to deal with out-of-order data in streaming aggregation," in *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*, 2020, pp. 141–152.

[27] V. Gulisano, D. Palyvos-Giannas, B. Havers, and M. Papatriantafilou, "The role of event-time order in data streaming analysis," in *Proceedings of the 14th*

ACM International Conference on Distributed and Event-based Systems, 2020, pp. 214–217.

[28] M. Yu, T. Cao, W. Wang, and R. Chen, "Following the data, not the function: Rethinking function orchestration in serverless computing," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 1489–1504.

[29] *Apache Flink documentation*, `https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/overview/`, Accessed: 2023-09-15.

[30] M. Sciabarrà, *Learning Apache OpenWhisk: Developing Open Serverless Solutions*, 1st ed. O'Reilly Media, 2019, ISBN: 9781492046165.

[31] *Iris flower data set*, Wikipedia, The Free Encyclopedia, [Online; accessed 24-June-2023]. [Online]. Available: `https://en.wikipedia.org/wiki/Iris_flower_data_set`.