Balancing strict performance requirements and trade-offs for efficient data handling in unbounded flows

Design considerations for a proof-of-concept stream processing pipeline for vehicular data validation

Master's thesis in Computer science and engineering

MÅNS JOSEFSSON
CARL-MAGNUS WALL

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024
www.chalmers.se

Master's thesis 2024

Balancing strict performance requirements and trade-offs for efficient data handling in unbounded flows

Design considerations for a proof-of-concept stream processing pipeline for vehicular data validation

MÅNS JOSEFSSON
CARL-MAGNUS WALL

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Balancing strict performance requirements and trade-offs for efficient data handling in unbounded flows
Design considerations for a proof-of-concept stream processing pipeline for vehicular data validation
MÅNS JOSEFSSON
CARL-MAGNUS WALL

© MÅNS JOSEFSSON, CARL-MAGNUS WALL, 2024.

Supervisor: Marina Papatriantafilou, Department of Computer Science and Engineering, Industrial supervisor: Binay Mishra, Volvo Group Trucks Technology, Assistant supervisor: Martin Hilgendorf, Department of Computer Science and Engineering, Examiner: Vincenzo Massimiliano Gulisano, Department of Computer Science and Engineering

Master's thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Balancing strict performance requirements and trade-offs for efficient data handling in unbounded flows
Design considerations for a proof-of-concept stream processing pipeline for vehicular data validation
MÅNS JOSEFSSON, CARL-MAGNUS WALL
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In the current era of ever-growing networks of sophisticated sensors and smart devices, there are vast volumes of data created at every moment. Big Data environments such as these necessitate scalable and efficient data pipelines in order to provide near real-time analytics and monitoring. However, with especially high, unbounded data rates, traditional (store-then-process) database procedures and batch-based processing methods are struggling to remain performant. To this end, processing streams of data continuously has become an increasingly appealing approach, targeting low latency, high scalability and real-time data processing. Stream processing tools enable the execution of both stateless and stateful computations on data as it is being transferred through the pipeline, providing opportunities to increase the efficiency of processing procedures such as, e.g., data validation. However, stream processing pipelines can be complex, especially in multi-tenant settings, and it is not always clear how to efficiently approach their implementation. A process such as data validation might be subject to a multitude of requirements that all affect pipeline design and considerations. To investigate such requirements and give detailed insight on how to approach the design of a stream processing pipeline for efficient data validation on unbounded flows of data, a proof of concept pipeline is developed and tested in a case study at Volvo Trucks. The case study involves multi-tenant automotive testing, where the proposed pipeline enables near real-time validation of data, for purposes such as monitoring vehicle sensor behavior. The pipeline is comprised of Apache Kafka, for persistent event storing, Apache Flink, for continuous stateful analysis, and Apache Druid, for data serving. Evaluation of the pipeline is performed from the perspective of a set of metrics, namely data completeness, sustainable throughput, latency, scalability and fault tolerance. In order to harmonize the requirements of the pipeline and discern how trade-offs affect performance, various tool-tuning experiments and stress tests are performed. Performance evaluation of the pipeline reveals that in a controlled environment, with limited resources, the minimum throughput requirement of the use case can be sustained, while still achieving sub-second latencies and offering a degree of fault tolerance. The pipeline also shows promise of adapting well to different levels of scale, providing enough headroom for a tenfold increase in data volumes over current demands.

Keywords: Data pipelines, stream processing, data completeness, latency & throughput, fault tolerance, scalability, data validation

Acknowledgements

We would like to thank our supervisors, Marina Papatriantafilou, Martin Hilgendorf and Binay Mishra, as well as our examiner, Vincenzo Massimiliano Gulisano, for their guidance, expertise and patience throughout this thesis. We would also like to extend gratitude to all the team members at Volvo Trucks for giving us such a warm welcome and assistance during this time. Finally, to our very own "Village People", thank you so much for your insights and feedback. It helped us find our purpose with this thesis.

Måns Josefsson, Gothenburg, June 2024 Carl-Magnus Wall, Gothenburg, June 2024

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

4PP	4 parallelism pipeline
8PP	8 parallelism pipeline
8PP3B	8 parallelism pipeline, 3 brokers
16PP	16 parallelism pipeline
AMD	Advanced Micro Devices, Inc.
API	Application Programming Interface
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DDR5	Data Rate 5 Synchronous Dynamic Random-Access Memory
GPS	Global Positioning System
JAR	Java Archive
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
NaN	Not a Number
RAM	Random Access Memory
SPE	Stream Processing Engine
UI	User Interface
VM	Virtual Machine
WSL	Windows Subsystem for Linux

Contents

List of Acronyms **ix**

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**
1.1 Problem context 2
1.2 Challenges 3
1.3 Contributions 3
1.4 Overview 4

2 Background **5**
2.1 Pipeline definition 5
2.2 Stream Processing 6
2.2.1 Stateless and stateful streaming analysis 7
2.2.2 Processing semantics 8
2.3 Fault tolerance 8
2.4 Data validation 8
2.5 Pipeline components and tools 9
2.5.1 Apache Kafka 9
2.5.2 Apache Avro 10
2.5.3 Schema Registry 10
2.5.4 Apache Flink 11
2.5.5 Apache Druid 12
2.5.6 Docker containers 13

3 Problem **15**
3.1 Designing for timeliness and adaptable scale 16
3.2 Case study - Vehicular sensor data validity monitoring 17
3.3 Evaluation metrics 19
3.3.1 Latency 19
3.3.2 Throughput 19
3.3.3 Resource utilization 19
3.3.4 Scalability 20

4 Challenges **21**

4.1 Requirement precedence and tool selection 21
4.2 Data validation through stateful analysis 22
4.3 Schema evolution resilience 22
4.4 Use-case motivations and limitations 23

5 Approach **25**
5.1 Declaration of requirement precedence 25
5.2 Pipeline design motivations and expectations 25
5.3 Pipeline design overview 26
5.4 Stateful data validation job 27

6 Evaluation **31**
6.1 Evaluation setup 31
6.2 Testing rationale 32
6.3 Evaluation methodology 33
6.3.1 Parameter tuning and data simulation 33
6.3.2 Latency measurements 34
6.3.3 Determining saturation rates 35
6.3.4 Resource utilization measurements 35
6.3.5 Monitoring tools 36
6.4 Scalability performance 37
6.5 Latency performance and distribution 40
6.6 Resource utilization performance 42
6.7 Performance observations 46
6.8 Operating in a controlled environment 47

7 Related work **49**

8 Conclusion and future work **53**

List of Figures

2.1 A basic pipeline structure. 5
2.2 A ParDo operation that increments the value of each tuple by three. . 6
2.3 The Kafka architecture. 10
2.4 Overview of how the schema registry facilitates serialization between a data producer and a data consumer. 11

3.1 High-level overview of the field testing pipeline structure. 17

5.1 Overview of the pipeline structure with its corresponding layers. . . . 27
5.2 Overview of the dataflow graph for the stateful stream processing job. The KafkaSource ingests from a Kafka topic and KafkaSink produces to a separate topic. 28
5.3 Data types used in the pipeline. LDT is short for the Java object type LocalDateTime. Each LocalDateTime contains a timezone-aware complete date and time down to the granularity milliseconds. Padding is an optional field that can be used to experiment with how payload sizes impact latency. 28

6.1 Overview of the pipeline run environments. 32
6.2 Overview of the pipeline structure with indexes indicating where the five separate steps of the latency measurement process are performed. 35
6.3 The Flink job as viewed from the Flink Web UI. The colors indicate how busy and backpressured each group of operators is. 36
6.4 Scalability results for each of the four pipeline configurations. 38
6.5 Comparison of the scalability results from each of the four pipeline configurations. 39
6.6 Comparison of the pipeline latency percentiles for the minimum requirement and saturation rates with a parallelism degree of 16. 41
6.7 Comparison of the pipeline latency percentiles for the saturation rate at parallelism degree four and 16 respectively. 41
6.8 Comparison of the pipeline latency percentiles for the saturation rate at parallelism degree of eight, with one and three brokers respectively. 42
6.9 Comparison of the resource utilization at the minimum requirement rate with parallelism eight using 1 and 3 brokers respectively. Here, JM and TM are short for JobManager and TaskManager respectively. K, K0, K1 and K2 represent Kafka brokers. 43

6.10 A screenshot showing the behavior of the heap memory in the Kafka brokers. 44
6.11 Comparison of the resource utilization at the saturation rate with a parallelism degree of 8, using 1 and 3 brokers respectively. 45

List of Tables

6.1 Default parameters for testing. 34
6.2 Description of the tested pipeline configurations. 37
6.3 Skewness values for all pipeline configurations. 40

1

Introduction

The advancements in technology over the past few years have improved the ability to gather and analyze data [1]. Devices have decreased in size and become increasingly efficient. Concurrently, device interconnectivity has seen a rise in both relevance and facilitation. Smaller devices, like sensors, can now easily be placed closer to data sources, providing valuable information, including overall system status and performance metrics. Consequently, the number of sensors has increased, especially in industrial settings.

Drastic influxes of sensors lead to significant increases in collectible time-series data. While larger volumes of data can prove useful for analyses and decision-making, the process of transmitting, storing and extracting value from the data keeps introducing new challenges [2]. To accommodate large flows of data from edge devices, and provide end users such as data analysts with timely, high-quality data, it is necessary to implement an efficient and scalable *data pipeline* [3].

A common way of handling data in data pipelines is to process it in batches [4, 5]. However, with especially large *unbounded* flows, where data sets do not have an explicit endpoint, (post-)processing becomes a difficult task [6]. Depending on the amount and timeliness of data that is required for specific analyses, systems such as databases can be pushed beyond their capabilities [2]. This often leads to unavoidable trade-offs between various factors, including latency and data quality.

As an alternative approach, *stream processing engines (SPEs)* [1, 7], which are characterized by their ability to handle data in a continuous manner, are prime examples of how pipeline design can impact performance and functionality [6]. Continuous processing ensures that data is delivered and processed with minimal latency, while also allowing for granular transformation of data down to single data points. Both benefits are desirable from a data analysis standpoint since they can provide quick responses in suitable data representations [8].

Pertaining to data analytics, one key factor is that of *data validity* [9]. Valid data helps to provide accurate, high-quality results from various analyses. Nevertheless, with increasing data volumes, low-latency data validation becomes cumbersome. To mitigate this problem, this thesis showcases how a stream processing pipeline can be designed, constructed and utilized to enable continuous validation of large volumes of data. Specifically, the pipeline aims to alleviate problems inherent to the

validation of unbounded flows of data.

As these unbounded flows continue to grow and industrial workflows become increasingly data-driven, batch-based solutions are lagging in both performance and versatility. However, since these solutions have historically been the de facto standard in certain industrial contexts [8], complete infrastructure overhauls can be difficult to perform. Thus, it is often necessary to slowly introduce new pieces of infrastructure while still maintaining current solutions. As a consequence, resource contention could become a problem. Therefore, it is important to not only define clear minimum requirements for any new processing pipeline but also ensure that it can adapt well to both small- and large-scale operations. This thesis studies such considerations to form a sound frame of reference for efficient pipeline design and implementation.

1.1 Problem context

The issues with large volumes of data are manifested in various industrial settings, including vehicular data transfers, monitoring and analysis [10, 11]. Within these settings, data pipeline design choices can have a significant impact on overall performance in terms of timeliness, throughput and scalability. For a purpose such as data validation, it is also important that the pipeline performs reliably, e.g. through failure recovery capabilities. Data validation is a crucial step to ensure that data complies with the requirements of a particular system environment [12]. However, such procedures can be time-consuming, especially when done in a batch-based fashion. By utilizing stream processing pipelines, data validation can be performed with high efficiency [9], but it is important to note that such pipelines also introduce a myriad of new design considerations that can have a direct effect on resulting pipeline performance. In this thesis, such considerations are studied in a case study at Volvo Trucks. Here, enormous amounts of vehicular sensor data are collected from different vehicles and vehicle parts daily. This data is subsequently sent in batches to a central storage location, from which the data can be fetched for various types of post-processing, including data validation. With ever-increasing data rates, due to increases in vehicles and sensors, this procedure is growing increasingly unmanageable for both engineers and certain systems. In particular, databases have been severely affected. For instance, storing all incoming data in a time-series database has proven to be problematic[1]. Treating the incoming data as *bounded*[2] sets, and using either batch-processing or more conventional (store-then-process) database procedures, is unsustainable from a latency perspective [4, 6]. Conversely, the continuous nature of stream processing pipelines is intended to accommodate unbounded, high data rate scenarios, and there is an increasing need for clear guidelines on how to approach the design and construction of such pipelines.

[1]These particular issues were revealed through internal discussions with data engineers at Volvo Trucks.

[2]As opposed to unbounded sets, bounded sets have well-defined endpoints.

1.2 Challenges

The transfer and processing of large data volumes put great demands on the scalability of a pipeline [11]. Workloads should preferably be distributable across nodes, which in turn usually need to be managed by orchestration tools. In *multi-tenant*[3] scenarios, the data pipeline could become especially difficult to operate and maintain. While each tenant expects timely service from the pipeline, it is possible that their behavior can lead to unpredictable computing loads, which the pipeline must be able to sustain.

The pipeline design and implementation also needs to consider data timeliness and high, varying data flow rates, i.e. throughput, to make it useful to data analysts [8]. Timely data is vital for allowing data analysts to rectify problems with faulty or misconfigured sensors quickly. With high data flow rates, it should be possible to provide multiple analysts with such timeliness concurrently. However, sustaining high rates could depend on several factors, including data variety. Additionally, without guarantees of data completeness and some degree of fault tolerance, the results of downstream analyses could be affected negatively due to reduced reliability.

Modularity and maintainability are also essential attributes for the pipeline to be useful in many different scenarios. Although open-source tools commonly provide community-developed updates and allow for extensions, any changes still need to be addressed by pipeline developers. To minimize the need for custom coding solutions, the pipeline components must have well-established inter-compatibility. Depending on pipeline requirements, the choice of components could be limited.

1.3 Contributions

The main contribution of this thesis is the study of how to approach the design, construction and evaluation of a scalable, proof-of-concept stream processing pipeline for efficient data validation. The design targets data completeness guarantees as well as high, sustainable throughput rates, while also achieving sub-second latencies and offering a degree of fault tolerance. The pipeline is tested and evaluated in a case study at Volvo Trucks, where a system for monitoring invalid data from testing vehicles, at rates of at least one billion data tuples per day, is of interest. The pipeline acts as an alternative to the current batch-based solution and aims to present how the validation procedure can be improved by performing continuous stateful analysis to monitor the frequency of invalid data points. Stress testing of the proposed pipeline is performed to determine performance limits. As part of the testing, resource utilization is monitored to further assess how adaptable the pipeline is in terms of scalability. In a controlled testing environment, with limited resources, results show that the pipeline can not only satisfy the minimum throughput demand but even exceed it by a factor of ten, without sacrificing either data completeness, sub-second latency performance, or fault tolerance.

[3]Historically, multi-tenancy has been defined in a multitude of ways [13]. In this thesis, the term refers to single application instances serving concurrent requests from several different users.

1.4 Overview

The thesis is structured as follows: Chapter 2 details concepts regarding data pipelines and stream processing, as well as tools used for the construction of the pipeline. Chapter 3 formulates the problem of remaining performant with increasing data rates and how this directly affects pipeline design and requirement considerations. Here, the vehicular data validation case study at Volvo Trucks is introduced. The chapter also lists the metrics of interest in relation to pipeline requirements and evaluation. Chapter 4 lists the challenges of addressing the problem through the construction of a stream processing pipeline. Specifically, it details the intricacies of balancing pipeline requirements and how this affects tool selection and configurations. It also lists challenges with stateful data validation and data variety, as well as those pertaining to the use case at Volvo Trucks. Chapter 5 presents how the pipeline requirements are given precedence and how the pipeline is structured. It also provides motivations for which tools are used in the pipeline, along with performance expectations. Finally, it describes how the stateful data validation is performed. Chapter 6 describes the evaluation setup and methodology, and also presents and discusses the results of performed benchmarks. Chapter 7 provides insight into related work within the realm of large scalable pipelines, alternative processing methods, as well as data validation through stream processing. Chapter 8 concludes the thesis and outlines its main findings. It also discusses possible future directions of the study, along with corresponding extensions of the pipeline and its surrounding infrastructure.

2

Background

In this chapter, background information for concepts relevant to the thesis is described. First, definitions for pipelines and stream processing are given. Then, the concept of data validation is described. Finally, the components and tools used in the pipeline are described.

2.1 Pipeline definition

Data pipelines can assume a multitude of forms, ranging from pure data transfer systems to sophisticated processing structures where data is conformed to satisfy predefined requirements before reaching its final destination. With regards to pipeline architecture design, there are two key types of components, commonly referred to as *sources* and *sinks*, which represent data generation points and data distribution points respectively [14, p. 130-131]. Between these points, it is common to place various processing components, which can take inputs as well as produce outputs. [15]. A common way of representing how a pipeline is constructed from these components is to draw a *directed acyclic graph (DAG)*, as shown in Figure 2.1.

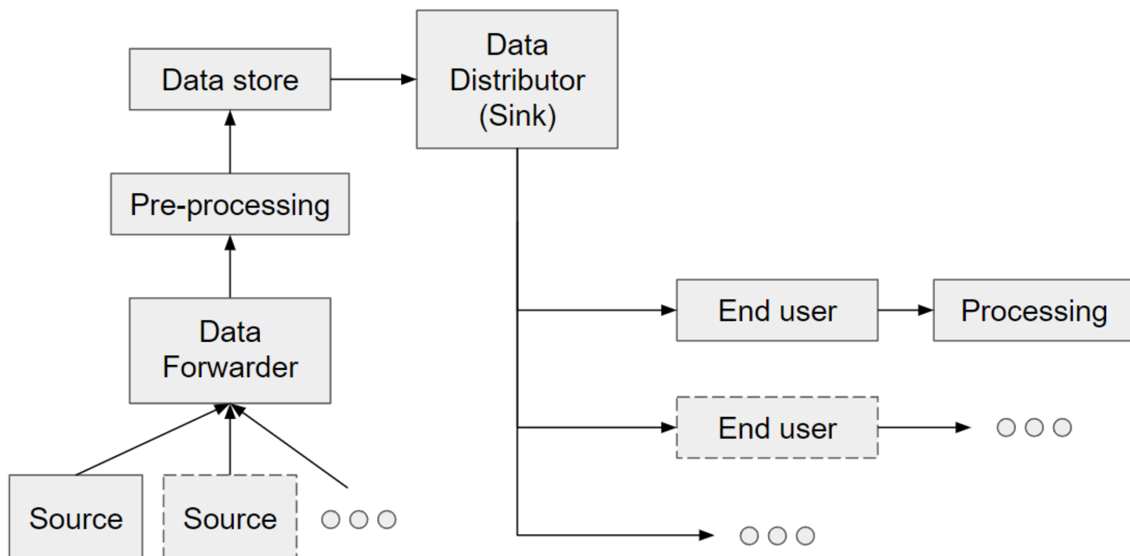

Figure 2.1: A basic pipeline structure.

2.2 Stream Processing

Stream processing of data has become a capable alternative approach to traditional *batch processing* methods [16]. In batch processing, the processing is performed in periodically scheduled jobs, consisting of bounded chunks of amassed data, whereas stream processing operates on data in a continuous flow [17, p. 390]. The continuous flow of stream processing enables more granular control of data, which facilitates distributed computation at a high scale. Additionally, as a consequence of avoiding batching data before sending it, one can expect lower overall latency from this approach [16], [17, p. 390].

The semantics of stream processing can be explained with the formal Dataflow model [4]. In this model, data is defined as tuples, each consisting of a list of attribute-value pairs $(\tau : v_0, A_1 : v_1, ..., A_n : v_n)$ [18]. The value of attribute τ is the timestamp of the tuple. The timestamp is used when the stream processing computations take time into consideration. Tuples can be transformed using different kinds of operations, such as the ParDo operation [4]. ParDo takes a number of tuples as input and performs a user-defined function on it, resulting in zero or more tuples as output. An example of a ParDo operation can be seen in Figure 2.2. Since the ParDo operation performs tuple-wise transformations, its functionality can be generalized to cases where there is an unbounded sequence of tuples to process. Such a sequence is referred to as a *stream* [19]. A stream of tuples generally starts with a data source and ends with a data sink, similar to the components of Figure 2.1. Along the path to the sink, operations like ParDo are performed on the tuples by *operators* to transform them in meaningful ways. The resulting structure of streams entering, being operated on and leaving the processing forms a DAG.

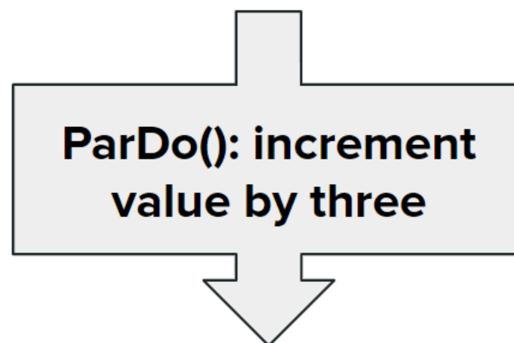

Figure 2.2: A ParDo operation that increments the value of each tuple by three.

2.2.1 Stateless and stateful streaming analysis

In stream processing, the concept of state management is central. Operations are commonly referred to as being either *stateful* or *stateless* [17], where the former indicates that past events are accounted for when performing computations, as opposed to the latter. To support stateful computations on unbounded data, streaming systems divide the stream into *windows* [20]. The basis for how the unbounded data is divided into bounded windows varies across different window types. For this thesis, the focus will be on windows based on time.

The two main data time domains in data processing systems are *event time* and *processing time* [14, p. 9]. Both these time domains can be used in windows [14, p. 17]. Event time reflects the actual time when the event that generated the data occurred, whereas processing time reflects the time when the system first observes the data. If there is no congestion in the system and if data is processed immediately when its event occurs, then the event time progresses at the same pace as the processing time. However, this is very unlikely in complex, real-world systems [14, p. 59].

Tumbling windows and sliding windows are examples of window assigners based on time [21]. A tumbling window will divide time into equally large windows of a given time interval. When a window closes, another is created and there is no overlap between windows. Sliding windows include a window slide value beside the time interval, which is used to control how often a new window will be created. If the window slide is smaller than the time interval, then there will be overlaps between windows. To be able to use windows in the stateful analysis, each tuple must include a timestamp field. Otherwise, the processing will be able to discern to which window the tuple should be assigned. A keyed window will, in addition to separating data into windows of time, also separate data based on their contents [21]. The operation that splits the data into keyed streams based on their key is referred to as a *KeyBy* operation. The keying of a data stream separates it into distinct streams that can be processed independently from each other.

In distributed settings with multiple data sources, data is frequently delayed and processed out of order [22]. This makes it difficult to know when all the data belonging to a time window has arrived. One method for counteracting this problem is the use of *watermarks*, which gives *data completeness* guarantees for windows [14, p. 27]. The term data completeness refers to the degree to which the elements belonging to an aggregation element (such as a windowed operation) are present in an instance of the aggregation element [23]. For instance, a windowed operation with no missing data is considered complete. Watermarks are created when external data is introduced to the streaming DAG [22]. Each watermark consists of an event time w, which indicates that all data from the corresponding external source with an event time preceding w has been sent. This informs stateful operators that they can safely close their window(s) and perform their operations on the already gathered data.

2.2.2 Processing semantics

The processing semantics of a system determine how failures will impact the processing output [24]. In stream processing, *exactly-once* semantics ensures that no data is lost or duplicated, even when the processing encounters failures [25]. With these semantics, every event in the stream will affect the state exactly once. Weaker processing guarantees exist in the form of *at-least-once* and *at-most-once* semantics [24]. In the former, all messages will be delivered with reservation for duplication of output, and in the latter data may be dropped but not duplicated. Before exactly-once semantics became more established as a concept in the landscape of stream processing, the most common data delivery guarantee was at-least-once [14, p. 121]. As a consequence of avoiding necessary precautions in relation to exactly-once processing, systems utilizing a weaker processing guarantee trade accuracy for efficiency.

The approach of using stream processing to get approximate results lead to the creation of the Lambda architecture. In the Lambda architecture, both stream processing and batch processing are performed on ingested data [14, p. 121]. Stream processing is used at the time of data arrival to achieve prompt, but tentative results, which can be used for real-time analysis that is less susceptible to inaccuracies. Later, after enough data has been gathered, batch processing of the same data will provide the exact results. While the Lambda architecture can be useful in certain scenarios, increased system complexity from simultaneous maintenance of two pipelines and the fact that inaccuracies often are too large to allow the prompt data to be useful invalidates the approach for many use cases.

2.3 Fault tolerance

A system is *fault-tolerant* if it can continue to perform its functions in spite of faults [26]. One type of fault is the fail-stop failure, where a process stops working and all its data is lost [27]. A common approach to achieve fault tolerance in a system is to implement *redundancy* [26]. With space redundancy, the system is provisioned with additional resources that are only used in cases of faults. This kind of redundancy can be applied to hardware, software or information. *Checkpoint and restart* is a software redundancy mechanism that is used to recover from faults. The mechanism creates snapshots of the current state at different points of execution. These snapshots can be used to restore the system to a certain point in time before the event of a fault occurring. *Replication* is another redundancy mechanism, which targets high availability. Replication is the creation of multiple copies of an object that could possibly be subject to modification [28]. When implemented correctly, accessing a replicated object should be indistinguishable from accessing a non-replicated object.

2.4 Data validation

Data validation is the act of checking whether data meets the requirements defined by the system in which it is being processed [12]. For instance, a data object

with missing data in a field that is deemed mandatory by the system should be invalidated. Manual validation can be possible in some scenarios but can become infeasible when data items and requirements on these increase drastically. Apart from being time-consuming, manual validation requires careful attention to detail to avoid errors. Introducing automated or semiautomated checks can increase the speed and accuracy of validation.

2.5 Pipeline components and tools

Data pipelines are often complex to construct, which necessitates the use of multiple pre-built tools. The landscape for data pipeline tools is extensive and many of the most prominent open-source tools offer native connectors that allow them to communicate seamlessly with each other. In this section, a selection of open-source components is presented. The use of stable, well-established open-source components can facilitate pipeline implementation and testing.

2.5.1 Apache Kafka

Apache Kafka is an open-source distributed messaging system for transferring large volumes of log data with low latency [29]. Figure 2.3 shows the architecture of a Kafka cluster, along with its three main components: *producers*, *brokers* and *consumers*. A number of producers collect *log* data from data sources and publish them to message streams called *topics*. The topics are stored as log lists in brokers, which enable consumers to set up subscriptions to topics that are of interest. Consumers read from subscribed topics by setting up a message stream and traversing it sequentially. Overall consumption rate can be made faster if more consumers are introduced to a topic, through point-to-point or publish-subscribe approaches. In point-to-point, consumers can take turns reading from streams, whereas in publish-subscribe each consumer processes the data in its own separate *partition*. A partition is a bucket that receives a subset of the data for a certain topic [30]. Partitions are created to allow multiple consumers to read from (and producers to write to) brokers simultaneously. In order to ensure that all the events relevant to a certain key (e.g. vehicle) can be reached from one location, events with the same key are always written to the same partition.

Two of the key design choices that contribute to the scalability of the Kafka architecture are stateless brokers and fault tolerance. While the brokers maintain log lists, each consumer is responsible for keeping track of what has been read and what should be read next. Each consumer does this by maintaining an *offset* for each specific topic partition. The offsets are protected with a sort of checkpoint and restart mechanism. Periodically, consumer offsets are checkpointed to a designated offset topic, which is accessible by all consumers [31]. This way, the system can mitigate the processing delay caused by fail-stop failures occurring in consumers. When a consumer crashes, a redundant consumer can fetch the latest checkpointed offset and continue from where the previous consumer halted operation. Fail-stop failures can be mitigated on the Kafka broker level as well, given that redundant brokers

are set up and kept up to date using replication.

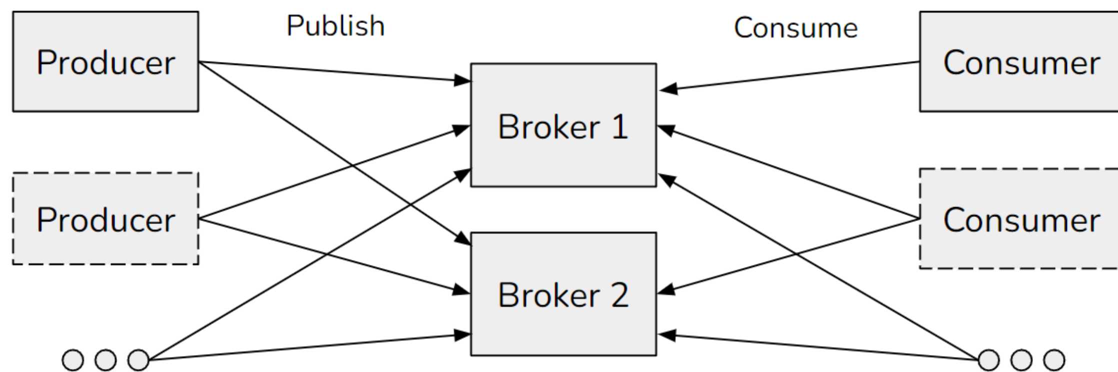

Figure 2.3: The Kafka architecture.

2.5.2 Apache Avro

Serialization is the process of converting the state of an object into a format that is suitable for transmission or storage [32]. Apache Avro is a data serialization system that is specifically designed for providing compact and fast conversions of objects, to and from a binary format [33]. Serialized Avro data does not contain any readable structural information needed for deserializing, making its binary representation small [34]. Instead, Avro utilizes data schemas defined using JSON formatting to instruct systems how serialization and deserialization are to be performed [33]. If the same schema is used at both stages, the conversion will be successful. Avro data is always accompanied by the schema, making the data self-describing.

Avro supports container files for persisting Avro data as records [33]. A record is a complex type defined in the schema, with a name and multiple fields [34]. Each field is required to have a name and a type. Records are stored in a file, with one data block containing one record. Included at the top of the file is the schema that is used to serialize and deserialize the data blocks. Both processes are performed by traversing the schema depth-first, left-to-right. The software project management and comprehension tool Maven [35] can be used to generate Java classes from Avro schemas, facilitating Avro usage in the Java language [36].

2.5.3 Schema Registry

A *schema registry* is a centralized repository that manages and validates schemas [37]. The schema registry can be configured to handle the serialization of data between data producers and data consumers in Kafka topics. When a data producer sends a message to the Kafka broker, it can include the schema to register it to the registry. Once the schema is registered, it is mapped to a schema ID which will be used instead of the full schema [29]. From this point onwards, any data consumer receiving a message can fetch the schema that the message adheres to from the registry. Besides guaranteeing successful serialization by using the same schema in

both ends of the process, sending an ID on a communication link is generally more efficient (the ID is smaller in size) than sending the full schema, saving resources [37]. Due to the registry hosting schemas in a single place, the complexity of managing and updating them can be abstracted away from the data producers and consumers. This can facilitate the development and maintenance of a data pipeline.

The validation provided by the schema registry ensures that data always is consistent, even when schemas change over time [37]. For instance, all the data being produced to and consumed from a system such as Kafka is validated against the schema registry to ensure that data conforms to its registered schema structure. This reduces the risk of data becoming corrupted or being lost. In cases where the initial data model eventually must change to meet new demands, *schema evolution* is possible [38]. Schema evolution allows for the creation of new versions of schemas, which might be backward or forward-compatible. The use of schema evolution can ensure that the data model can be updated for all connected systems at once, without losing consistency guarantees.

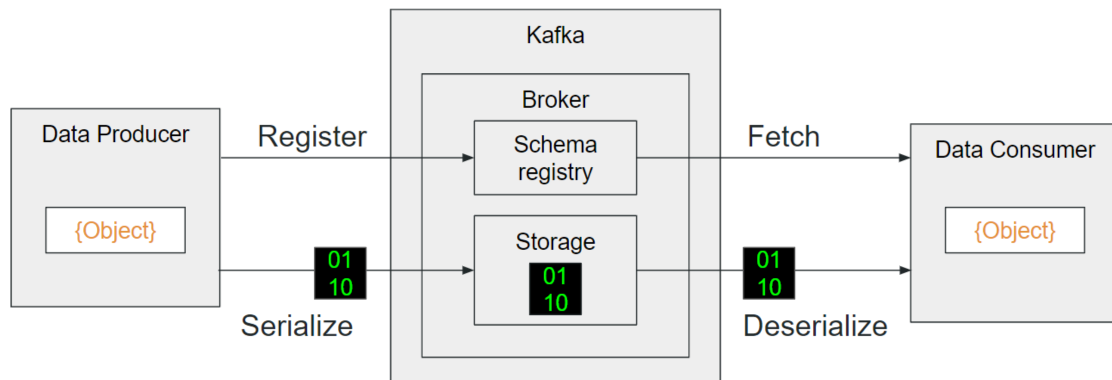

Figure 2.4: Overview of how the schema registry facilitates serialization between a data producer and a data consumer.

2.5.4 Apache Flink

Apache Flink is an open-source system for processing unbounded streaming and bounded batch data [39]. In both cases, the data is treated like streaming data in windows of time. To support batch processing, the stream is considered to be finite and the order and time of arrival of elements are ignored. The first step of the Flink workflow is the Flink Client, which converts Flink programs with processing logic into a dataflow graph. A dataflow graph is a DAG consisting of stateful operators and data streams that connect them. The dataflow graph is delivered by the Flink Client to a *JobManager* that is tasked with supervising the distributed execution of the graph.

The actual execution of operators and the data stream handling is performed by *TaskManagers* and their assigned *Task Slots* [40]. Each TaskManager has at least one Task Slot, which corresponds to the smallest unit of resource scheduling in Flink. Task Slots are assigned a fixed subset of the memory available to the TaskManager.

Flink can utilize data parallelism to split the processing across Task Slots [39]. Although separation into Slots allows for parallel execution and memory isolation, some sharing between Slots is possible [40]. This improves resource utilization and minimizes per-task overhead.

Flink has multiple features that make it suitable for stateful stream processing [39]. Users can define windows and watermarks for out-of-order event processing as well as customized data structures for state. Multiple notions of time are available, including event time, processing time and ingestion time. Furthermore, it utilizes *checkpointing* and offers exactly-once processing, given that data is consumed from a persisted source with replay capabilities, like Apache Kafka. In Flink, checkpointing involves saving the current state of operators at regular intervals [41]. These two features ensure prompt fault recovery for the system. In cases with unbounded streams, checkpointing is especially important, since recomputing the arbitrarily long series of previous computations would be an infeasible task. An additional feature of Flink is its web interface, the Flink Web UI [40]. The Web UI, which is maintained by the JobManager, presents the status of current and past Flink jobs, allowing for easy monitoring.

2.5.5 Apache Druid

Apache Druid is an open-source real-time analytical data store that targets low latency data ingestion and quick sub-second queries on large data sets [42]. In Druid, data is stored in data sources, which consist of timestamped events that are partitioned into data segments. Each segment is uniquely identified by its data source identifier, timespan and version string. If multiple segments match the timespan of a query, then the segment with the highest version string will be fetched. This segment corresponds to the latest view of the data. The storage layout of Druid is column-oriented, meaning that it only scans and loads columns required for the aggregating query. This orientation can be less CPU-intensive than row-oriented layouts where a specific column can only be loaded by scanning all rows in which it occurs. Moreover, Druid utilizes compression to reduce the size of ingested data, effectively increasing the data storage capacities of hosted clusters.

The design of Druid is distributed, with dedicated nodes fulfilling different purposes. Ingestion and serving of real-time data are handled by real-time nodes. These nodes make the newest data within a limited timespan available for real-time queries before the data is persisted in deep storage. Data in deep storage is fetched into cache memory by historical nodes, from where it can be queried. Queries to Druid are routed to the correct real-time nodes and historical nodes by broker nodes. The cluster is controlled by a coordinator node which monitors the state of the cluster and is in charge of data management. The coordinator instructs historical nodes on how they should operate, based on efficiency, replication and load balancing.

The role-based node architecture of Druid provides various benefits, including scalability and fault tolerance. Historical nodes operate in a shared-nothing architecture, where each node has its dedicated resources [43]. This allows nodes to independently

serve requests, avoiding data contention. With each node operating independently, it is also easier to provide fault tolerance by maintaining replicas. Moreover, the scalability of the system is improved since the system's capabilities can easily be expanded by adding additional independent nodes.

2.5.6 Docker containers

Docker is an open-source platform for running applications in lightweight, isolated environments called containers [44]. Along with the application code, containers can also include any required dependencies or programs that are needed to run the applications. This makes the deployment, distribution and management of applications easier. Docker containers acquire resources from the operating system to be able to run their applications. The amount of resources given to a certain container can be configured, facilitating scalability. Since Docker containers have all of the programs and dependencies required to run their applications, it is portable by design. When multiple services and containers are to be used in conjunction, a multi-container application can be used [45]. It is recommended that the applications be split into separate containers to have more granular control of resource utilization [46].

3
Problem

With ever-increasing data volumes, originating from various sources, data-driven tasks become increasingly difficult. To accommodate such volumes, engineers must not only consider how to perform efficient data transfers but also where and how to process and store data. This is directly connected to pipeline design and implementation strategy. As resource requirements grow, it is imperative to ensure that the purpose of a pipeline is clear. Constructing a general-purpose pipeline might impede the implementation process since it would require tuning and evaluations from the standpoint of a high number of different use cases. Instead, it is often preferable to define a specific purpose that the pipeline aims to satisfy, in order to gain a better understanding of processing and performance requirements.

Data validation is one such purpose, which has been shown to be possible to perform efficiently for unbounded flows [9, 47]. However, pipelines needed to perform such procedures often consist of multiple different components, examples of which are listed in Section 2.5. When working with a limited amount of resources (e.g. due to contention) and the prospect of scaling operations up, it can be unclear how to properly combine and configure such tools in order to accommodate efficiency, reliability and adaptability. Such a design process requires more than a simple step-by-step tool guide; there is a need for a more encompassing frame of reference for how to motivate design choices and approach implementation.

While stream processing pipelines intend to further automate workflows and enhance efficiency over batch-based solutions, they introduce a set of complexities, including balancing fault-tolerance measures and processing semantics with overall throughput and latency. Depending on the strictness of certain pipeline requirements, the process of making trade-offs can become difficult. Thus, the main research question of this thesis is:

Q How and to what extent are various pipeline performance factors impacted by imposing strict requirements?

Determining which requirements take precedence over others could facilitate making trade-offs. Nevertheless, the on-off property of certain factors, such as fault tolerance, could pose a problem when balancing the requirements. Imposing strict minimums for each requirement is likely to degrade the scalability and adaptability of the pipeline.

3.1 Designing for timeliness and adaptable scale

In order to support quick and efficient data analysis and monitoring, data must be made available as close to the time of its generation as possible. For complete pipeline structures, there are multiple factors that affect how timely data can be. These include pipeline ingestion delays (e.g. waiting for batches to fill up), network bandwidth(s), queueing delays or *backpressure*[1] between pipeline components, as well as total processing time.

To quantify timeliness, it is typical to measure the overall *latency* of data. This describes the time delay that is imposed on data as it moves from one point of the pipeline to another (e.g. from the pipeline source(s) until a result is delivered to a sink). As mentioned, in multi-tenant settings, a high number of users desire quick responses to data queries, and to satisfy such demands, it is crucial to ensure that the pipeline can sustain high data rates, i.e. high *throughput*. This factor reflects the quantity of data that can be transferred and processed over a certain amount of time. In the field of stream processing pipelines, the quantity is often referred to as the average amount of bytes per second, tuples per second, or both. It should be noted that high throughput does not inherently imply low latency. For instance, within batch processing, depending on the strategy used, large batches can cause long wait times for smaller subsequent batches. This is typically referred to as a convoy effect and is a common problem in First-In-First-Out (FIFO) scheduling scenarios [10]. Within the context of stream processing, when parallel workloads are not distributed evenly across operators, data skews might occur, leading to certain data items having higher latencies than others.

Pertaining to parallel workloads, this can be directly tied to the concept of pipeline scalability. A pipeline can be considered to be scalable if it can sustain higher data rates as it is assigned more resources. For instance, with a higher degree of parallelism, more cores can be utilized at once, in turn allowing for more data to be processed simultaneously. Another facet of scalability is to maintain similar performance even as workloads increase. To only have slight increases in latency while the data rate is growing is an example of this. During the pipeline design phase, it is important to consider what the current situation demands in terms of both latency and throughput. Nevertheless, it is equally important to take into account any possible computational overhead. This does not only include future demands, but also occasional spikes in data rates or other similar disruptions to normal pipeline operation. Further, it is also worth considering the overall adaptability concerning pipeline scaling. In times of low activity or heavy resource contention, it is often beneficial to scale down accordingly to reserve or balance resources.

Delivery guarantees and fault tolerance

While the concept of optimizing a pipeline for low latency and high throughput is enticing, it is crucial to also consider what guarantees must be placed on the data

[1]Backpressure occurs when data rates through the pipeline exceed the consumption capabilities of downstream operators [48].

processing as well as the pipeline as a whole. Ensuring that data is not processed multiple times, through exactly-once semantics, often introduces throughput and latency overhead, and to ensure end-to-end compliance, all tools in the pipeline must agree on the same semantics strategy. In terms of pipeline guarantees, it is common to enforce some method(s) of fault tolerance into the system. As an example (and as mentioned in Section 2.5.4), the persistent storage capabilities of Kafka along with checkpointing can enable Flink to recover from failures. However, failure recovery affects the time it takes for a certain data item to be processed. It might also cause an increase in backpressure since events might still be produced into Kafka. Further, pertaining to an event store like Kafka, in the case of broker failure, it is necessary to maintain copies of event logs in order to provide fault tolerance. This is done by replicating logs across a number of different brokers. In this thesis, this type of fault tolerance acts as the main example for investigating performance impact. Since this method requires maintaining more data and brokers in the Kafka cluster, it is expected to affect overall pipeline performance, especially in resource-limited scenarios.

3.2 Case study - Vehicular sensor data validity monitoring

To investigate how a context-specific stream processing pipeline can be constructed and evaluated, a vehicular data validation scenario is considered. At Volvo Trucks, an increasing number of vehicles (and vehicle parts) undergo testing daily. Vehicles are equipped with numerous sensors, all gathering operational data, including engine temperatures, speeds and GPS coordinates. The sensors are connected to a logging device that gathers incoming data and sends it over the network in batches to a central storage location for further processing, as seen in Figure 3.1.

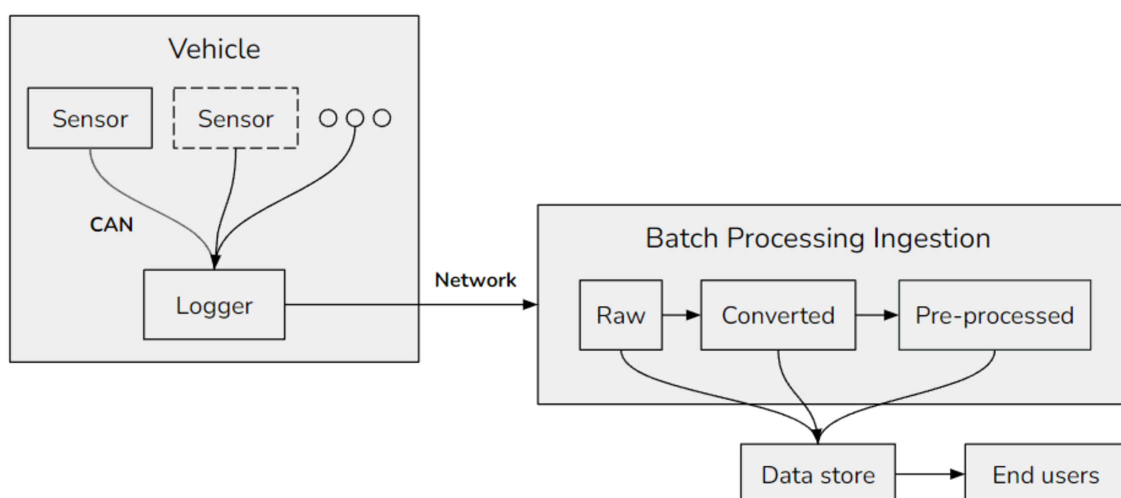

Figure 3.1: High-level overview of the field testing pipeline structure.

Upon data store ingestion, the raw data is first converted from its binary log format to CSV files and then finally processed into Parquet[2] files. While all representations of the data are made available to end users, it is the Parquet files that are typically accessed for analysis. In the current state, data is made available in Parquet representations approximately one hour after ingestion. The daily amount of ingested data corresponds to approximately one billion tuples, and the entire procedure (from vehicle to storage) aims to satisfy high degrees of data completeness.

As an example of how these Parquet files are processed, data validation can be performed. One of the more common validation procedures is to observe the rate at which various sensors report faulty values. In the Parquet files, faulty numbers are represented as *Not a Number*, or *NaN* for short. When a sensor value is reported as NaN, it indicates that the sensor did not produce a value at a certain point in time. Tracking NaN rates can provide valuable insight into the overall quality of a data set. High rates over a specific timespan indicate that the data set is not apt for various analyses or business decision purposes. Further, if a sensor is repeatedly reporting NaN values, then it could be misconfigured or even malfunctioning and require replacement. This process is done in a batch-based fashion, where data from multiple vehicles, spanning up to four days, is sifted through and analyzed. However, with the rapid influx in data volumes, this process is becoming increasingly difficult. The databases simply cannot cope with the size that these Parquet files are assuming. In these scenarios, the current strategy is to filter out or discard certain data, in order to adhere to the capabilities of the database. As a consequence, some data is not accounted for during the NaN-value checking in these cases. This is in stark contrast with the pressing demand for data completeness and could have negative effects on the accuracy of subsequent analyses.

Use-case requirements

In the context of this particular case of NaN monitoring, it is of great importance that engineers receive as current information as possible. In this scenario, different vehicles or vehicle groups could be divided among several engineers or teams. Therefore, it is essential for the case study to not only ensure low latency but also high throughput in the pipeline. Furthermore, with increasing numbers of vehicles and sensors, the pipeline should not only be able to withstand sudden data rate spikes but should also be able to sustain activity of some order of magnitude greater than what is currently required. As mentioned, the current daily data rate is approximately one billion tuples. As such, this dictates the minimum throughput requirement of the pipeline. Moreover, for the specific type of data validation, it would be of great interest to ensure exactly-once processing, since the quality of the procedure could suffer if certain data is processed more than once. Fault tolerance through e.g. replayability and replication would also benefit the use case, due to the high demand for data completeness.

[2]Apache Parquet [49] is a columnar storage format that allows for smaller file sizes and faster query times compared to CSV.

3.3 Evaluation metrics

In this section, descriptions of the evaluation metrics are given. The focus is on what is measured, with details of how exactly these measurements have been performed being addressed in Section 6.3.

3.3.1 Latency

Considering the importance of providing data in useful representations in a timely manner, the pipeline's latency, denoted L_p, is considered. Calculating the latency caused by windowed aggregations is a non-trivial problem due to the latency of a windowed operator being impacted by the waiting times of incoming tuples [50]. The approach used in this thesis is to define latency as the time from the moment that all the data that contributes to a given window instance is made available to the pipeline source(s) to the moment the corresponding result is delivered to the sink. In Equation 3.1, the latency L_p of a window instance is calculated as the difference between these moments in time. For clarity, t_{latest} in the equation below refers to the last event that contributes to a window instance.

$$\text{Pipeline latency } L_p = t_{sink} - t_{latest} \text{ (milliseconds)} \tag{3.1}$$

3.3.2 Throughput

Since the pipeline targets large volumes of data, its instantaneous throughput T_p will be measured. The minimum throughput requirement states that the pipeline should be capable of rates of at least one billion tuples per day. Thus, the overall throughput is calculated per Equation 3.2, where n is the number of tuples that are processed over t seconds.

$$\text{Instantaneous throughput} \quad T_p = \frac{n}{t} \quad \text{(tuples/second)} \tag{3.2}$$

3.3.3 Resource utilization

In terms of resources, CPU and RAM usage is monitored. The CPU usage $u_{cpu,t}$ is measured as the percentage of a set maximum hardware capacity c_{max} at an instant t. Here, physical and virtual threads are considered to be equal. The heap and non-heap memory $c_{mem,t}$ considers the percentage of a set dedicated capacity c_{ded} at an instant t. The percentage of utilized memory at an instant t is referred to as $c_{used,t}$. These calculations are performed according to Equations 3.3 and 3.4, which both produce results in the range of 0 to 1.

$$\text{CPU utilization } u_{cpu,t} = \frac{c_{used,t}}{c_{max}} \tag{3.3}$$

$$\text{Memory utilization } u_{mem,t} = \frac{c_{used,t}}{c_{ded}} \tag{3.4}$$

3.3.4 Scalability

As mentioned in Section 3.1, pipeline scalability can encompass a number of different aspects, including data rate handling capabilities as well as maintainability of similar timeliness performance even as data rates increase. It can also describe the ability to improve timeliness as more resources are provisioned. In this thesis, the scalability of the pipeline is determined by its *saturation rate*, which represents the highest data rate T_p that the pipeline can sustain without degraded latency performance. The saturation rate is determined by calculating the latency L_p for increasing data rates T_p. The resulting latency indicates whether performance has been degraded as a result of an increase in data flowing through the pipeline. To test different data rates in the pipeline, the production rate of each producer is adjusted. The production rate is the same for each producer to facilitate the load-balancing in Kafka. By testing different degrees of parallelism in the pipeline and the impact this has on the saturation rate, it should be possible to see if the pipeline can sustain higher data rates as it is assigned more resources.

4

Challenges

As mentioned, given the intricacies of both hardware and software interoperability, it is often necessary to pre-define a limited set of functionalities and requirements when designing a data processing pipeline. Further, by employing an iterative workflow, and ensuring basic operation of each individual tool, the implementation process can be facilitated. In this chapter, the challenges pertaining to achieving the desired performance of the proposed pipeline are described in detail.

4.1 Requirement precedence and tool selection

Ensuring timely delivery with guarantees of data completeness and some degree of fault tolerance is not as straightforward as merely selecting tools that advertise such functionality. Pipelining tools, such as the ones listed in Section 2.5, often ship with default settings that might not satisfy all demands inherently. As an example, since Kafka version 3.0.1 [51], default settings specify that producers are idempotent[1] and await an acknowledgment from the Kafka cluster leader that tuples have been successfully replicated (to a specified minimum amount of brokers) before considering data items to have been written successfully to a topic. While these are prerequisites for exactly-once processing, they do not inherently guarantee that the consumer will abide by the same rules. In order to make the produce-consume procedure atomic, both parties must utilize a transactional messaging system [52]. This means that before sending a tuple (or batch of tuples), the producer must begin a transaction. Once all tuples in a transaction have been sent, the producer then commits the tuples to the topic, marking the transaction as finalized. To ensure that the consumer abides by this procedure, it has to be set up to only read committed tuples. This guarantees consistency, meaning that tuples are only read once, even in the case of failure (e.g. if a transaction is aborted). Regarding failures, the replication procedure mentioned above (as well as in the previous chapter) is also commonly used to provide a degree of fault tolerance to a Kafka cluster. By ensuring that data items are replicated across multiple brokers, the cluster can still operate even if a certain broker shuts down unexpectedly. Nevertheless, without a proper understanding of such operational configurations, the resulting pipeline performance could become unreliable.

[1] Idempotence ensures exactly-once delivery, even in the case of a retry.

In order to facilitate the implementation process, it is often beneficial to define a precedence list in regard to performance and functionality demands. If there is a high demand for data completeness and therefore stricter delivery guarantees, it is necessary to select tools that can offer such functionality. However, in certain scenarios, it can be difficult to decide on required trade-offs, especially pertaining to latency and processing semantics. If there is a strict requirement of only a few milliseconds of latency on certain data, then it might be necessary to make sacrifices in terms of delivery guarantees. For instance, when defining checkpointing parameters in Flink, it is possible to set whether checkpointing should be done in exactly-once or at-least-once mode [53]. The latter allows for consistent, low-millisecond latencies, but naturally lacks protection against duplication. A major challenge with these matters is finding a fitting balance between all requirements, with regard to both requirement precedence and the amount of available resources. This process often requires careful monitoring of the performance of each tool individually, as well as the pipeline in its entirety.

4.2 Data validation through stateful analysis

One of the main drawbacks of using batch processing or more traditional processing methods in unbounded, high data rate scenarios is potentially high latencies. As mentioned, data validation as a procedure intends to ensure that data complies with a particular system's requirements. Depending on what the requirements are, the manner in which the procedure is performed can vary. One method is to monitor data over certain time intervals in order to procure aggregates that describe the overall data validity within said interval. With a constant, high-velocity flow of data, this becomes increasingly troublesome, especially depending on the desired frequency of aggregations. By instead utilizing stream processing techniques, namely windowing and continuous stateful analysis, such aggregations can be performed with lower latency and at higher levels of granularity. However, depending on the size of state that needs to be maintained, the minimum amount of resources required can vary greatly. In cases where multiple different aggregations can be of interest, the complexity of the analysis implementation can increase.

Within the context of the use case, it could be of interest to monitor NaN values at different levels of granularity concurrently. To address this and ensure efficient use of resources, the method by which windowing is performed at each granularity level should be handled in such a way that the number of windows that must be open simultaneously is reduced as much as possible.

4.3 Schema evolution resilience

In multi-tenant scenarios, data variety is likely to exist. It is also possible that requirements change over time, including how data should be organized. Schemas offer an efficient way of enforcing contracts between various components in a data pipeline. This becomes important when adding new sensors to an existing system.

However, including entire schemas for every data item in a stream is not efficient. As mentioned in Section 2.5.3, a schema registry can remove the responsibility of schema handling from producers and consumers of a pipeline. This is especially beneficial in a streaming scenario since the amount of data producers is generally high. Abstracting away the schema handling allows both parties to use their resources more efficiently, while still promoting data governance.

For the use case, this type of functionality is of great interest, since different vehicles might have different types of sensors and therefore a different schema structure. It also allows for new sensors and vehicles to be added to the system, without the concern of, e.g., requiring manual alteration of (de)serialization methods.

4.4 Use-case motivations and limitations

As presented in Section 3.2, the current processing infrastructure at Volvo Trucks is entirely batch-based. There exists, however, an incentive to incorporate data streaming at a large scale. In order to support efficient workflows, the infrastructure must be able to cope with increases in data volume. While batch processing can be used for efficient transfers of large amounts of vehicular data [10], it is lacking in processing granularity. As mentioned, extended granularity through continuous streaming can facilitate parallelism, which can be used to avoid waiting for certain batches to finish. This provides opportunities to increase the adaptive capabilities of the pipeline, e.g., by enabling fine-tunable data prioritization in the pipeline. It also introduces the possibility of using stream processing tools to continuously perform any desired transformations on the data. However, to which extent data can be streamed depends on hardware capabilities. For the testbed of the use case at hand, only some testing equipment is capable of streaming data directly from the source, but there does not exist a baseline streaming solution for this equipment as of yet. Inevitably, this affects the real-world testing and performance comparisons of the proposed pipeline. Despite this, since the current daily data ingestion corresponds to approximately one billion tuples, the resulting pipeline can still be evaluated from the standpoint of this minimum throughput requirement. It is important to note that this number includes data from sources that are not (yet) capable of streaming. Thus, if the pipeline can comfortably sustain such rates, it should not only be capable of handling the demands from streaming-capable sources but also be future-proof to this extent.

5
Approach

This chapter first declares which requirements are considered for the design of the pipeline, along with their precedence and how they affect pipeline testing. Then, the motivations and performance expectations for the pipeline components are discussed. Subsequently, an overview of the pipeline design is presented. Finally, a description of the stateful data validation job used for testing the pipeline is given.

5.1 Declaration of requirement precedence

For the design of the pipeline, the following requirements are considered:

R1 Ensuring data completeness and fault tolerance

R2 Sustaining high throughput, satisfying at least an expected daily rate

R3 Achieving sub-second latencies and high scalability

The list above also represents the ordering of precedence of the requirements. For data validation, it is of utmost importance that all data is accurately accounted for, warranting high precedence of ensuring data completeness in the pipeline. With this in mind, the sustainability, with regard to the minimum data rate requirement, is then determined. Once the sustainability of the minimum rate is confirmed, the pipeline is tested for latency performance and scalability. To further assess the performance of the pipeline, fault tolerance measures are then applied. Comparisons between tests with fault tolerance and those without are performed to determine overall performance impact.

5.2 Pipeline design motivations and expectations

Given the requirements presented above, it is important that the tools selected for the pipeline can scale to data rates beyond the minimum requirement while still achieving sub-second latencies. The following tools are expected to satisfy these demands, provided they have access to enough resources. For stream processing purposes, Flink (1.18.1) is chosen due to its support for continuous stateful analysis as well as exactly-once semantics. To achieve exactly-once processing in Flink, its sources must be replayable and the sinks must be transactional [25]. Kafka (3.7.0)

achieves both these goals, while also being flexible with connections to other tools. Among these tools is Druid (29.0.1), which can consume Kafka data natively. Druid is also especially fitting for the pipeline since it specializes in time-series data and real-time analytics. Additionally, it is configured to only read committed data by default, ensuring exactly-once processing [54].

The main motivation for implementing a schema registry in the pipeline is to have uniform data serialization and deserialization between all tools. If different parts of a pipeline use different implementations for this purpose, the complexity of managing and debugging the communications would increase. As for the file format, Avro is the best-supported data format for serialization in both Kafka and Flink, making it a good choice for the pipeline.

In terms of fault tolerance, the log replication capability of Kafka is utilized, as mentioned previously. Since this fault tolerance measure introduces additional brokers, it is expected to affect overall pipeline performance negatively due to an increase in overhead. Fault induction tests (i.e. aborting brokers) are not performed, and the pipeline is expected to remain stable during testing. The main interest here is to study how pipeline performance is impacted by the inclusion of some degree of fault tolerance.

For scalability purposes, Kafka and Flink can be adjusted on a partition and parallelism level. Higher degrees of parallelism allow Flink to host a higher number of task slots, which in turn can be assigned dedicated CPU cores, typically leading to improved performance in terms of overall latency and throughput. For pipeline testing, this enables (limited) simulations of setting thresholds pertaining to accessible resources. However, it should be noted that with a higher number of partitions, data skews can occur, meaning that certain partitions are more populated than others. This could degrade overall latency performance in the pipeline.

5.3 Pipeline design overview

The pipeline design consists of three main layers: the data generation layer, the processing layer and the data-serving layer, as shown in Figure 5.1. The data generation layer consists of a Kafka producer, implemented in Java using the Kafka Application Programming Interface (API). This Java program creates Java objects based on an Avro schema. The data is serialized based on the schema and is sent to the processing layer, where it is stored in a topic inside the Kafka cluster. Concurrently, the schema used for serialization is ingested into the schema registry. From there, it can be accessed by pipeline components in any subsequent layer that needs to handle or deserialize the Avro data. Once data is produced to a Kafka topic, a Kafka consumer in Flink fetches the data, starting at the earliest offset. Using the schema from the schema registry, Flink deserializes the data back into Java objects, on which it performs stateful processing.

The results of the processing are aggregations that count the number of encountered records with *NaN*-values during the last minute and hour respectively. The results

are serialized and produced to a second Kafka topic, which is being consumed by Druid inside the data-serving layer. Druid ingests this data and persists it for the duration of a specific run of the pipeline. A Javascript program running the Druid API queries the data in order to fetch the latest aggregates and display these in a dashboard.

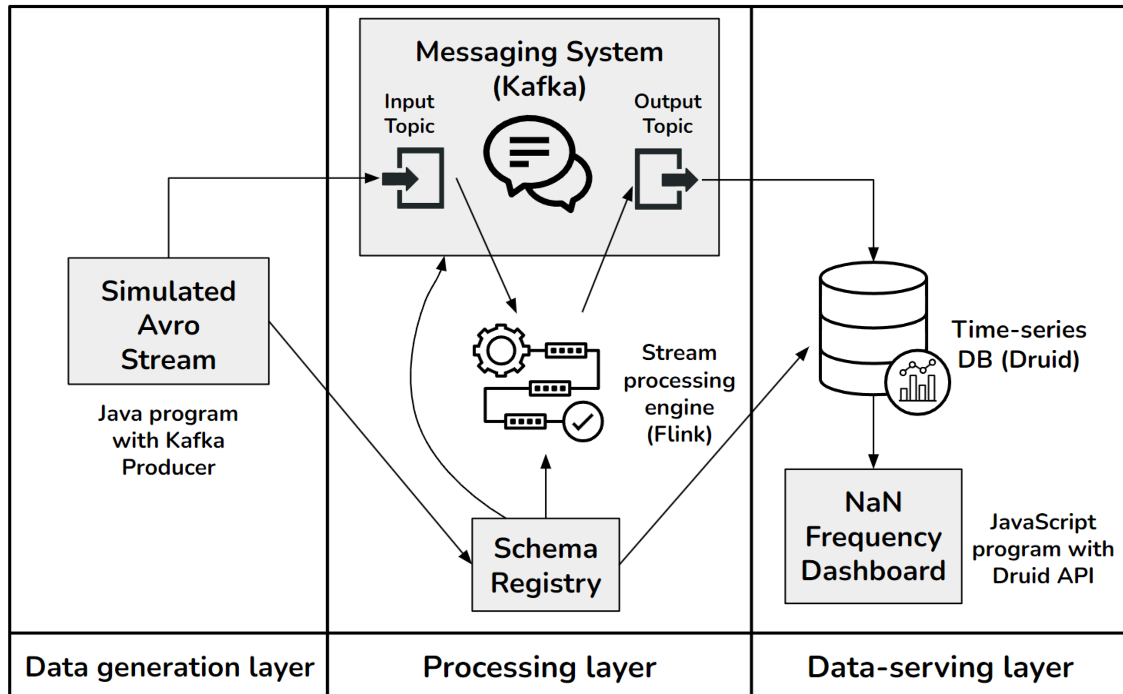

Figure 5.1: Overview of the pipeline structure with its corresponding layers.

5.4 Stateful data validation job

In order to illustrate how stateful stream processing can be performed by the pipeline, a data validation Flink job is constructed. The goal of this job is to detect NaN-values within vehicle signal objects and maintain the frequency of NaN-values that have occurred during the last minute and hour respectively. An overview of the dataflow graph created for the job, including its source, operations and sinks, is shown in Figure 5.2. Each operator in the graph can be executed at a fixed degree of parallelism, which is defined before the job is submitted to the Flink JobManager. The job is implemented in Java, utilizing Kafka and Flink APIs.

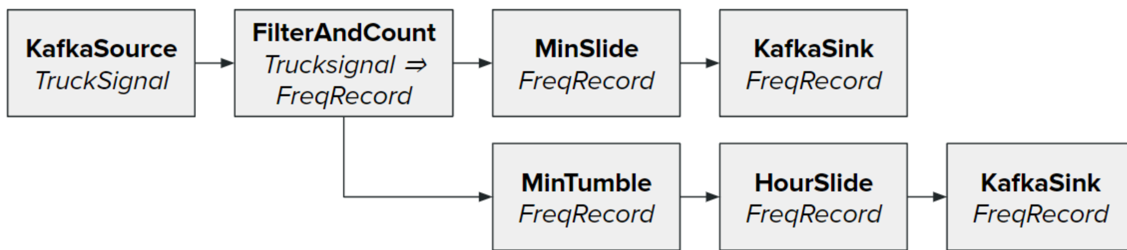

Figure 5.2: Overview of the dataflow graph for the stateful stream processing job. The KafkaSource ingests from a Kafka topic and KafkaSink produces to a separate topic.

The first step of the dataflow graph is the *KafkaSource*, which consumes data from a Kafka topic using a number of partitions. Each Kafka record has a key, which determines the partition to which a tuple should be assigned. In this pipeline, the key is a data type referred to as *TruckKey*. The tuples also contain a value of the data type *TruckSignal*. Once ingested, the value of each tuple is sent to the *FilterAndCount* operation. FilterAndCount filters the TruckSignals and only keeps those with a NaN-value in its value field. These TruckSignals are converted into *FreqRecords*, which have additional fields used for storing NaN-frequencies at different levels of granularity. Figure 5.3 details the structure of the data types used in the pipeline.

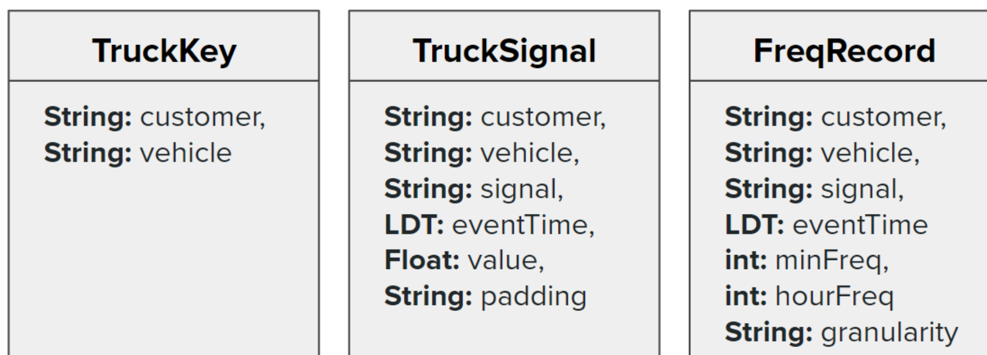

Figure 5.3: Data types used in the pipeline. LDT is short for the Java object type LocalDateTime. Each LocalDateTime contains a timezone-aware complete date and time down to the granularity milliseconds. Padding is an optional field that can be used to experiment with how payload sizes impact latency.

The FreqRecords created by FilterAndCount are then sent to two separate operators. The first of these is referred to as *MinSlide*, which aggregates FreqRecords into summaries using sliding windows. These windows are one minute long and are created once every second. Each window counts the ingested values and sends a frequency summary when it closes. Since a new window is opened each second, an update with a new frequency summary for the last minute will be sent at this granularity. The updates are then ingested into a Kafka topic using a *KafkaSink*,

which is set up to use transactions and exactly-once processing. The topic can be subscribed to in order to continuously receive new updates.

The second operator ingesting FreqRecords from FilterAndCount is referred to as *MinTumble*, which utilizes tumbling windows with an interval of one minute. Once a window closes, the summary is ingested into the *HourSlide* operator. HourSlide conforms to the same sliding window logic as MinSlide does, but instead has an interval of one hour and a sliding offset of one minute. In order to ensure that the following ingestion and transactions work correctly, HourSlide has its own KafkaSink with its own individual transactional ID.

The tumbling nature of MinTumble is suitable from a resource and performance perspective. By creating minute summaries in MinTumble, the amount of data contributing to the windows in HourSlide will decrease. The concern is also split across the two operators, providing an opportunity for threads to distribute their work more granularly.

6 Evaluation

The goal of the evaluation is to determine how and to what extent various pipeline performance factors are impacted by imposing strict requirements, as stated in the research question $\boldsymbol{Q}$ in Chapter 3. In this chapter, the evaluation setup, testing rationale and evaluation methodology are first described. The benchmarking results are then presented and discussed. Finally, observations of the results are discussed.

6.1 Evaluation setup

All pipeline components and external helper programs are run and tested on a single Windows 11 (23H2) desktop computer, powered by an AMD Ryzen 9 7900X 12-core (24-thread) CPU with 32 GB of DDR5 RAM. For storage, a 2 TB NVMe drive (running on 4 fourth-generation PCI-E lanes) is used. Most of the pipeline components are required to be run in a Linux environment, motivating the choice of hosting a virtual machine (VM) on the desktop. For this purpose, WSL2[1] with Ubuntu version 22.04.4 LTS is used. The VM is given access to all cores as well as 22 GB of RAM. An overview of the execution environment of each component is shown in Figure 6.1. The code is available at the following **repository**[2].

Within the confines of the VM, multiple components are run. The Kafka producer is run in the Integrated Developer Environment (IDE) IntelliJ. To enable monitoring of pipeline output and latency, a Node.js HTTP server that fetches stream processing results from Druid is hosted. To display the JavaScript dashboard, a web browser (Firefox) is used. The web browser is also used to display the Flink Web UI, where the state of the Flink job is monitored. To run all the pipeline components, a Docker multi-container instance, hosting Druid, Kafka, the Schema Registry as well as the Flink JobManager and TaskManager is set up. The multi-container approach facilitates the handling of component configurations and interdependencies. Before the HTTP server can fetch data, Druid must be configured to ingest data from the correct topic. This setup also includes specifying the schema registry address, data format (Avro) as well as which columns to include. Here, it is important to assign the Kafka ingestion timestamp to a dedicated column, since it is necessary for latency

[1] WSL denotes Windows Subsystem for Linux, which allows Windows operating systems to host lightweight Linux virtual machines [55]. The 2 denotes version number.

[2] `https://github.com/CWTED/PoC-stream-processing-pipeline`

calculations. This configuration is handled through the Druid web console. For the Flink cluster, Application Mode [56] is used. This mode is the most straightforward to use since it only requires the cluster to start, execute one job until it is finished, or, as in the case of this thesis, continuously execute one job until the cluster is shut down. The prerequisite of needing to include the job at cluster startup is not an issue, since the job is prepared in advance. The job itself is packaged as a JAR file and is specified along with the Flink JobManager in the multi-container setup.

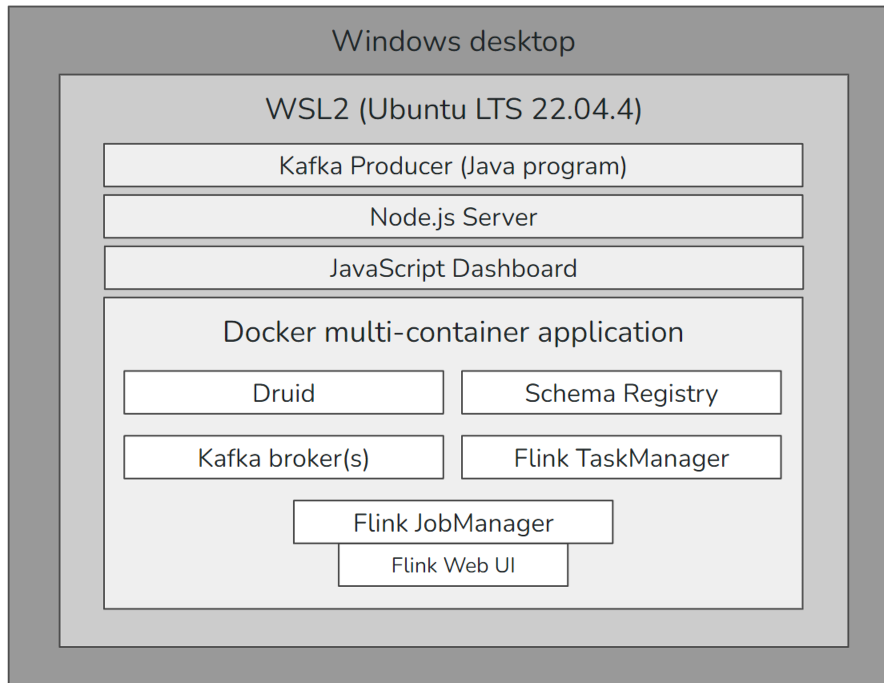

Figure 6.1: Overview of the pipeline run environments.

6.2 Testing rationale

To investigate the research question, tests for scalability, latency and resource utilization are performed. These tests also ensure that the strict requirements (***R1***, ***R2***, ***R3***) of the case study, described in Section 5.1, are satisfied. Regarding requirement ***R1***, all pipeline tests are run with exactly-once guarantees across all components. Other types of processing semantics are not considered, since the goal is to ensure that data sets are as complete as possible. For the fault tolerance aspect, comparison tests, with and without such measures applied, are performed. Here, the impact of Kafka broker redundancy through log replication is investigated. To address requirement ***R2***, all tests are performed at the minimum required rate or higher.

The tests cover both facets of scalability as defined in Chapter 3.1, addressing the latter part of requirement ***R3***. Varying degrees of parallelism are tested to see whether an increase in resources allows the pipeline to sustain higher data rates. Moreover, varying data rates are tested to determine whether similar performance

can be sustained as workloads increase. The sub-second latency part of requirement ***R3*** is addressed by performing latency distribution tests. If pipeline latency consistently is below one second, the pipeline satisfies the requirement.

Resource utilization tests give an additional perspective on the fault tolerance and scalability requirements. For fault tolerance, the addition of more Kafka brokers to the pipeline is expected to impact the amount of resources being used. Concerning scalability, it is desirable to see if the resource utilization is proportionate to the data rate being processed.

6.3 Evaluation methodology

This section describes the methodology used to perform the evaluation. First, parameter tuning is discussed. Then, the pipeline monitoring method and how performance results are collected are described.

6.3.1 Parameter tuning and data simulation

In order to keep tests consistent, a set of default parameters is defined. These parameters are shown in Table 6.1. The messages being sent from the Kafka producer are 80B in size. They are sent in batches of size 65536B with a batch linger of one millisecond. Batches are sent either when the batch size is reached or when the linger duration ends. The chosen batch size is higher than the default setting (16384B) in order to reduce memory overhead and accommodate higher data rates. A low linger duration is desirable from a latency standpoint since it ensures that messages are sent as quickly as possible in the event that batches are not being filled. The checkpointing interval determines how frequently checkpointing will be performed on operator state. A short checkpointing interval increases overhead from frequent checkpointing operations, whereas a longer one leads to longer fault recovery times. The main reason for having checkpoints enabled is to ensure exactly-once processing while avoiding transactional ID re-use, which prevents the KafkaSinks from producing aggregations into Kafka. Checkpoint snapshots are made durable in the JobManager's heap memory, and the interval is kept short at 300ms. Theoretically, this should allow the job to recover quickly enough to maintain sub-second latencies. It should also provide some insight into how increases in workloads affect resource usage.

The degree of parallelism and the number of partitions in the pipeline are set to different values depending on the test being performed. The degree of parallelism determines how many Flink source operators can ingest data from Kafka topic partitions at once. To achieve optimal performance it is recommended to keep the number of Kafka topic partitions and Flink source operators reading from said partitions equal [57]. If there are more Flink source operators than there are Kafka partitions, then some operators will become idle. With fewer Flink source operators than there are Kafka partitions, some operators will need to subscribe to multiple partitions simultaneously.

The selection of 50 customer-vehicle combinations and ten signals for the proof-of-concept pipeline is considered to be a reasonable starting point, in accordance with discussions with data engineers at Volvo Trucks. This number also allows for flexibility in regard to the number of Kafka partitions. If the number of unique keys, which in this case is determined by the number of customer-vehicle combinations, is low, then there exists a possibility that some partitions stay idle during pipeline operation.

Customer-vehicle combinations (#)	50
Signals per vehicle (#)	10
Kafka message size (B)	80
Kafka batch size (B)	65536
Kafka batch linger (ms)	1
Flink checkpoint interval (ms)	300

Table 6.1: Default parameters for testing.

The data used to test the pipeline is simulated. This is mainly due to the hardware limitations mentioned in Section 4.4. The pipeline as described in this thesis assumes that data will be able to be streamed from vehicles and converted to the Avro format. The use of simulated data is also useful for testing purposes since it is smaller in size and has a less complex structure than real-life data, facilitating debugging during testing. With respect to this, the pipeline should be able to also manage real-life data, given that the schema and Flink processing logic are adapted to its structure. To ensure that the pipeline can successfully sustain realistic data rates, initial tests are based on the expected daily rate of the case study, i.e. one billion tuples, which corresponds to 11575 tuples per second.

6.3.2 Latency measurements

The pipeline latency L_p is based on the latest event time included in the aggregation t_{latest} and the ingestion time into Kafka t_{sink}. Refer to Figure 6.2 for a visual indication of where each step of the latency measurement is performed inside the pipeline. The event time of each tuple is set in the data generation layer inside the Kafka producer when the tuple is generated (step 1). When tuples are aggregated into FreqRecords, the latest included event time is saved as the event time of the FreqRecord (step 2). This logic ensures that at the end of processing, each FreqRecord leaving Flink has event time t_{latest}. The t_{sink} value of a FreqRecord, on the other hand, is its Kafka timestamp. The Kafka timestamp is equal to the time that the result is produced to the Kafka topic after leaving Flink (step 3). The difference in milliseconds between these two times is calculated and persisted with CSV formatting in the data-serving layer inside the JavaScript dashboard (step 4). The calculation is performed every second and each run is executed for at least 1000[3]

[3]Tests were run for 20 minutes, resulting in 1200 values. However, it is deemed to be easier to interpret the latency percentile results with 1000 values.

seconds. A Python script is used to create a list from the first 1000 latency values and calculate the percentiles as well as the average and the median values (step 5). These values are used to create both percentile latency plots for singular data rates as well as scalability plots. For the scalability plots, each data rate is tested three times, with the minimum latency, maximum latency and average latency of these three tests being shown.

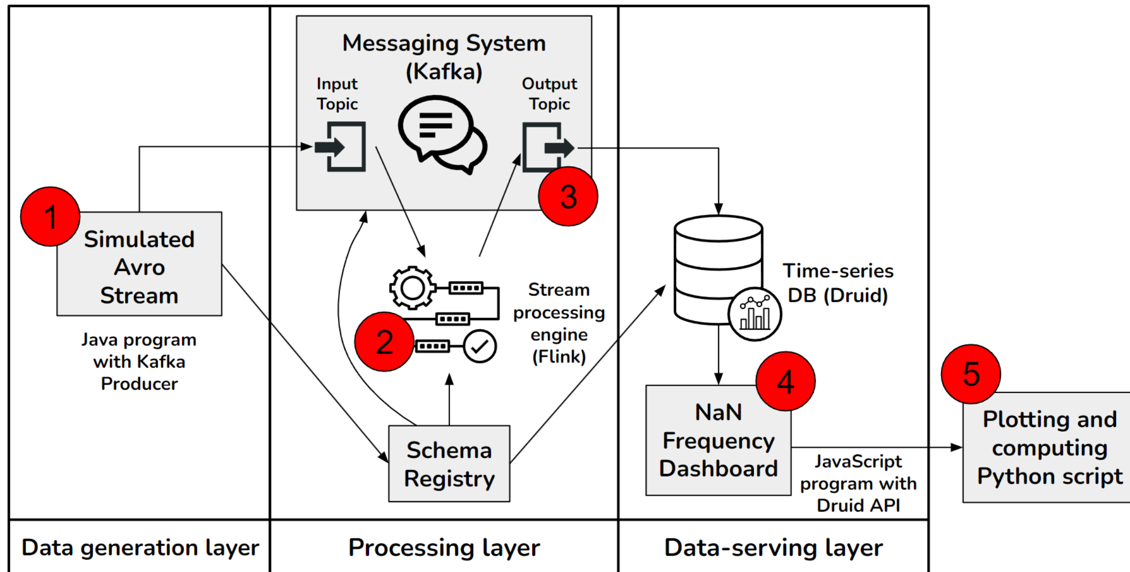

Figure 6.2: Overview of the pipeline structure with indexes indicating where the five separate steps of the latency measurement process are performed.

6.3.3 Determining saturation rates

To find the saturation rates of each pipeline, a two-step method is used. In the first step, a data rate is tested and if the pipeline can sustain it, the data rate is doubled. This ensures that a crashing rate can be found quickly. Once the pipeline crashes, the second step starts. In this step, the mean value of the crashing point rate and the last sustainable rate are used. This binary search-inspired approach reduces the number of required tests to find the saturation rate.

6.3.4 Resource utilization measurements

Resource utilization values are collected using the monitoring tool JConsole [58]. This tool is able to gather performance and resource utilization information from applications running in a Java Virtual Machine (JVM), such as Flink and Kafka. Readings of CPU, heap and non-heap memory are created every four seconds and the results can be exported to CSV files. The Kafka broker, the Flink JobManager and the Flink TaskManager are all monitored during a run of the pipeline. The resulting CSV files are entered into a Python script that samples 300[4] consecutive

[4]This number corresponds to test run-times of 20 minutes, which in turn corresponds to the run-times of the performance tests.

values and calculates the resource utilizations $u_{cpu,t}$ or $u_{mem,t}$ for each instant t depending on if c_{max} or c_{ded} is used in the computation. The one exception to this testing process is the Kafka broker non-heap memory. It does not have values for c_{max} and its values for c_{ded} are not static, making it difficult to calculate its usage with JConsole. The tests are run with default heap and non-heap resource limits. The maximum dedicated heap memory of the Kafka brokers and the JobManagers is 1GB. The corresponding limit for the TaskManager is 0.5GB. The non-heap limit was 744MB for both the JobManager and the TaskManager. JConsole is also used to measure the data rates (throughput) of the pipeline. The values shown as data rates in the figures are the tuples per second entering the Kafka topic from the data generation layer.

6.3.5 Monitoring tools

The dashboard and the Flink Web UI are used for monitoring the state of the Flink job while it is running. The Flink Web UI view of the job can be seen in Figure 6.3. Each rectangle represents a group of operations being performed by different threads. The current workload of each thread in an operator is expressed by two separate percentage values, namely *busy* and *backpressured*. These represent the time that the thread is actively working and the time spent as being backpressured respectively. Meanwhile, the JavaScript dashboard continuously queries the stream processing results from Druid. The dashboard ingests the latest aggregation result, including its values for l_{sink} and l_{latest}, which are used to determine the current latency of the pipeline. Besides giving a live update on the current latency of the pipeline, the values also indicate whether the pipeline is experiencing degraded performance. For instance, if there is a short interval between l_{sink} and l_{latest}, but the difference between l_{sink} and the current time is growing, then the pipeline is not able to keep up with the current data flow. The queries are performed via HTTP more than once per second to ensure that no values are missed.

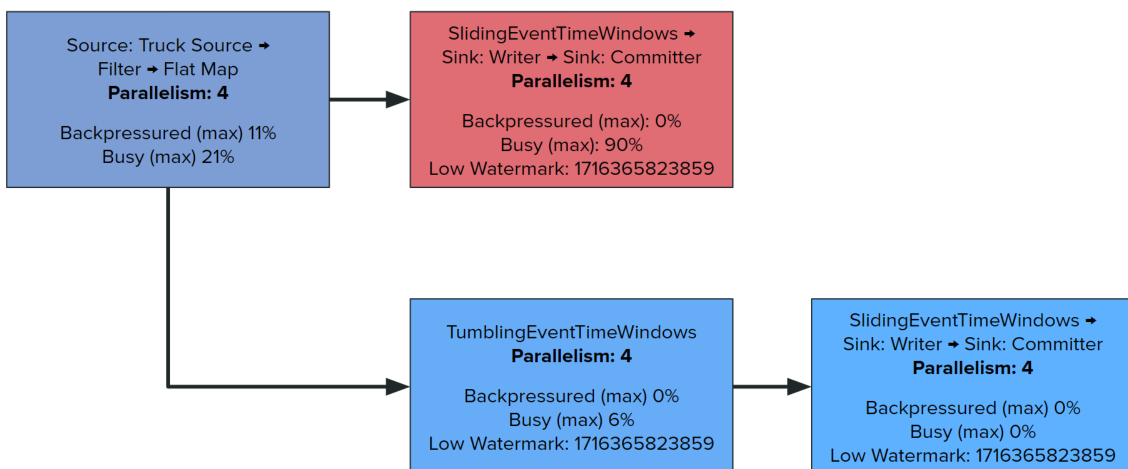

Figure 6.3: The Flink job as viewed from the Flink Web UI. The colors indicate how busy and backpressured each group of operators is.

6.4 Scalability performance

In order to confront the requirement in ***R3*** in Section 5.1, scalability tests are performed. The degree of parallelism in Flink and the data rate of the pipeline is changed in order to address both facets of scalability mentioned in Section 3.1. Testing shows that the highest degree of parallelism that can run the Flink job is 16 and that the lowest degree is four. The reason for 16 being the highest possible degree might be due to the VM not being able to spare more cores for the processing while all other containers are running. The lower limit can be explained by there being four groups of operations in the Flink job (as shown in Figure 6.3) that each needs to be run simultaneously in order to avoid pipeline bottlenecks.

Scalability tests are performed on four different pipeline configurations with varying degrees of parallelism and number of Kafka brokers. The attributes of each parallelism configuration can be seen in Table 6.2. The name of each pipeline configuration contains XPP, where X is its degree of parallelism and PP stands for parallelism pipeline. One of the pipeline configurations also has XB in its name, indicating that it has X Kafka brokers. The role of the additional brokers is to increase the likelihood of tolerating fail-stop failures at the Kafka broker level. All pipeline configurations use exactly-once semantics for data completeness.

Pipeline configuration	Degree of parallelism	Number of Kafka brokers
16PP	16	1
8PP	8	1
8PP3B	8	3
4PP	4	1

Table 6.2: Description of the tested pipeline configurations.

The test results for each pipeline version are shown in Figure 6.4. The saturation rate in each pipeline version is marked with a black dot. Common for all pipeline configurations is that latencies start to spike beyond the saturation rate. The error bars in the figure show that latencies measured up until the saturation rate generally are very close across multiple runs, while the data rates that follow the saturation rate have larger differences between their minimum and maximum latency. Past the highest plotted rate in each pipeline configuration, latencies increase indefinitely. The occurrence of latency spikes can be explained by Flink inducing backpressure when data can not be sent to downstream operators due to them already working at full capacity. A possible explanation for why increases are indefinite after a certain rate is that the full capacity operators are preventing the pipeline from producing results each second as it is instructed to, continuously piling up workloads indefinitely.

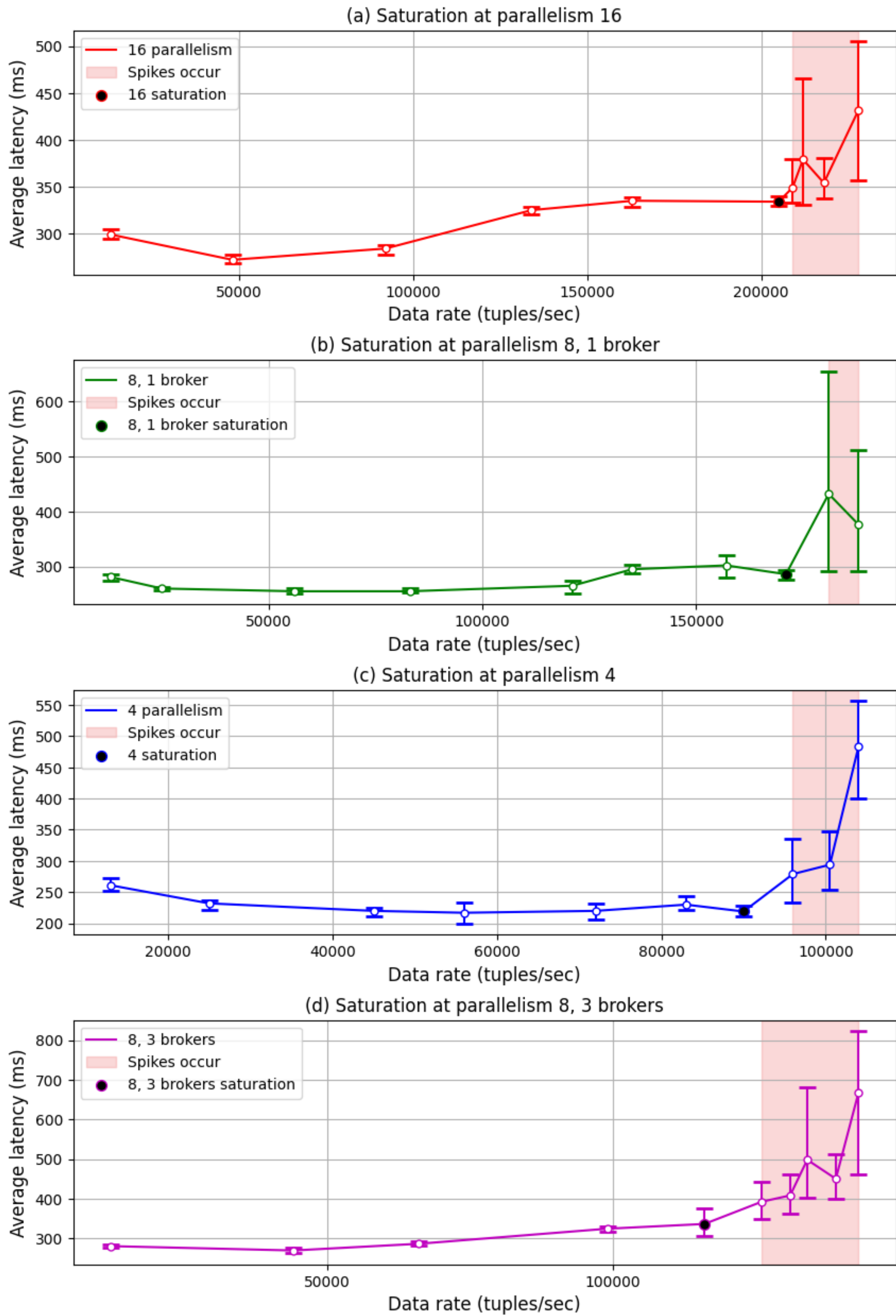

Figure 6.4: Scalability results for each of the four pipeline configurations.

Pipeline configurations with lower degrees of parallelism have lower saturation rates. This pattern can be seen in Figure 6.5. The decrease in saturation rate can likely be attributed to the fact that configurations with lower degrees of parallelism do not have as many available resources for simultaneous computations as configurations with higher degrees. As expected, this means that the throughput that can be sustained is lower for pipeline configurations with lower degrees of parallelism. With that said, the difference between the saturation rate in the 8PP and the 16PP is not that great considering that the 16PP can utilize double the CPU at once. This could be an indicator that the Flink job is not benefiting from more CPU resources and that another resource is the bottleneck. The saturation rate of the 8PP3B being lower than that of the 8PP confirms the expectation of fault tolerance negatively impacting the amount of data that can be processed at once by the pipeline. The saturation rate is 32.2% lower in the 8PP3B than in the 8PP.

Unexpectedly, Figure 6.5 indicates that the latency of a pipeline configuration benefits from having a low degree of parallelism up until the respective saturation rate. Possibly, this relation between the parallelism degree and latency is caused by higher parallelism degrees causing higher computational overhead. The 8PP3B has the highest overall latency of all pipeline configurations, confirming that the inclusion of fault tolerance has a negative impact on latency as well. Its latency is 14.9% higher than that of the 8PP.

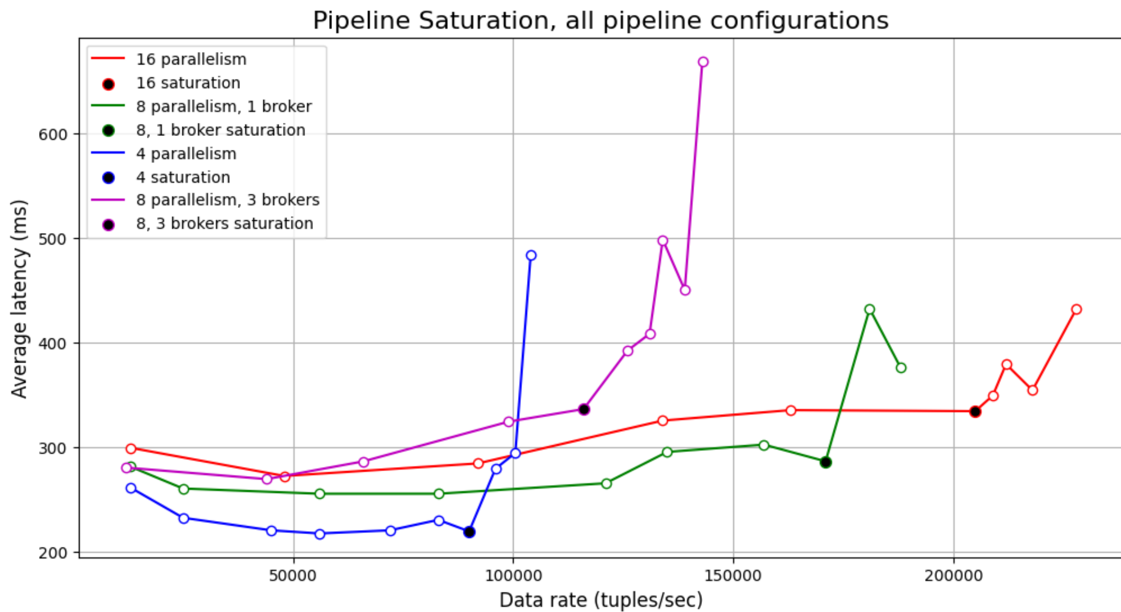

Figure 6.5: Comparison of the scalability results from each of the four pipeline configurations.

Figure 6.4 shows that all pipeline configurations except for the 8PP3B have higher latencies at the minimum required throughput compared to measurements in the 25000-100000 range. These results are reproducible and can be explained by the use of transactions combined with the fact that the method in which tuples are produced

is not completely deterministic. While the type of vehicle signal is cycled through for each tuple produced into Kafka, the vehicle ID is randomized. Since latency is calculated for a specific vehicle ID and signal, it is inevitably affected by the fact that at lower production rates, each committed transaction has a lower chance of containing a specific vehicle-signal combination. Because of this, delays that can be expected for low data rates due to randomization are gradually becoming less impactful as the data rate increases.

Data skewness

For higher degrees of parallelism, a higher number of Kafka partitions is set. In such scenarios, there exists a possibility that data distribution over partitions can become skewed, as mentioned in Section 5.2. This can cause higher latencies for data belonging to highly populated partitions, and thus affect overall pipeline latency.

By sampling data distribution across Flink KafkaSource operators for each pipeline configuration and computing the Fisher-Pearson coefficient of skewness [59] for each resulting dataset, the skewness values of the pipeline configurations are compared. The skewness values are shown in Table 6.3. The data shows that the 16PP does indeed have the highest skewness value of all pipeline configurations. However, the value for the 4PP is higher than that of the 8PP and 8PP3B, indicating that data skewness is not a relevant contributor to pipeline latency performance. This is all the more apparent when comparing the skewness values of the 16PP and 8PP3B with their respective latency numbers.

Pipeline configuration	Skewness value
16PP	0.817
4PP	0.633
8PP3B	0.592
8PP	0.591

Table 6.3: Skewness values for all pipeline configurations.

6.5 Latency performance and distribution

The latency of each data rate, shown in Figure 6.4, is studied to address the first requirement in ***R3***. For the pipeline to qualify as being able to handle a data rate at sub-second latency consistently, its 99th percentile should be under one second. Figure 6.6 presents the latency percentiles for the tests performed on the minimum required and saturation rates of the 16PP. While the average only is marginally higher in the saturation rate, the 99th percentiles show an increase of more than 50ms. The two rates have similar distributions, but the distribution of the saturation rate is shifted further to the right. The scalability results for the 16PP in Figure 6.4 convey that this increase was not sudden but gradual. At both data rates, the 16PP pipeline is below a second in its 99th percentile by a good margin.

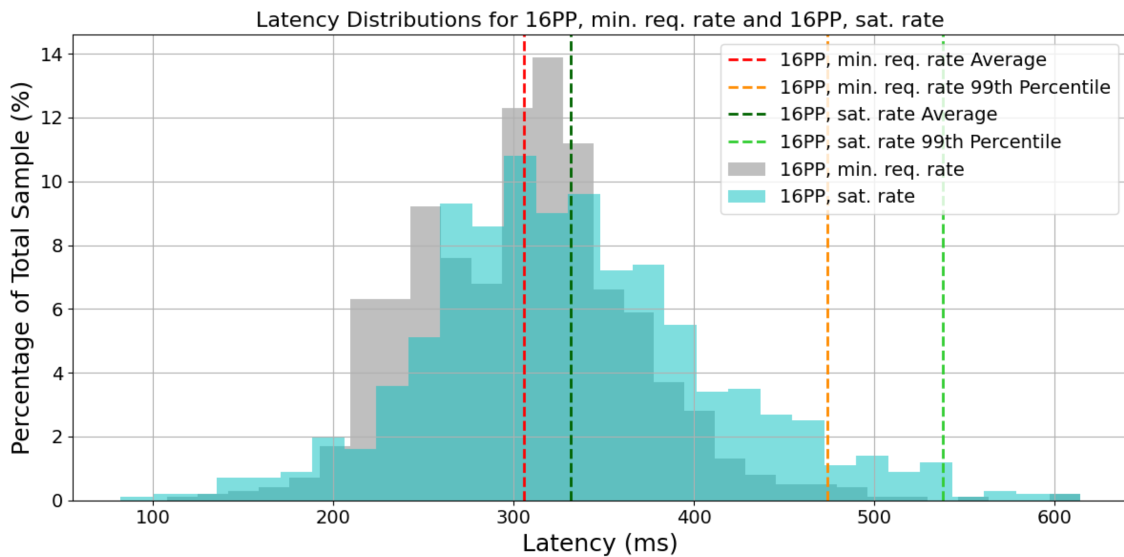

Figure 6.6: Comparison of the pipeline latency percentiles for the minimum requirement and saturation rates with a parallelism degree of 16.

The latency percentiles of the 4PP and the 16PP at their respective saturation rates are shown in 6.7. Overall, the latency of the 4PP saturation rate is significantly lower than that of the 16PP saturation rate. It could be argued that the reason for this is that the 4PP saturation rate only has half the data rate of the 16PP saturation rate, but this is invalidated by the results in Figure 6.5. There, it is shown that the latency of the 4PP is consistently lower than the 16PP latency. Thus, the 4PP can be considered to perform better for the data rates up until its saturation rate.

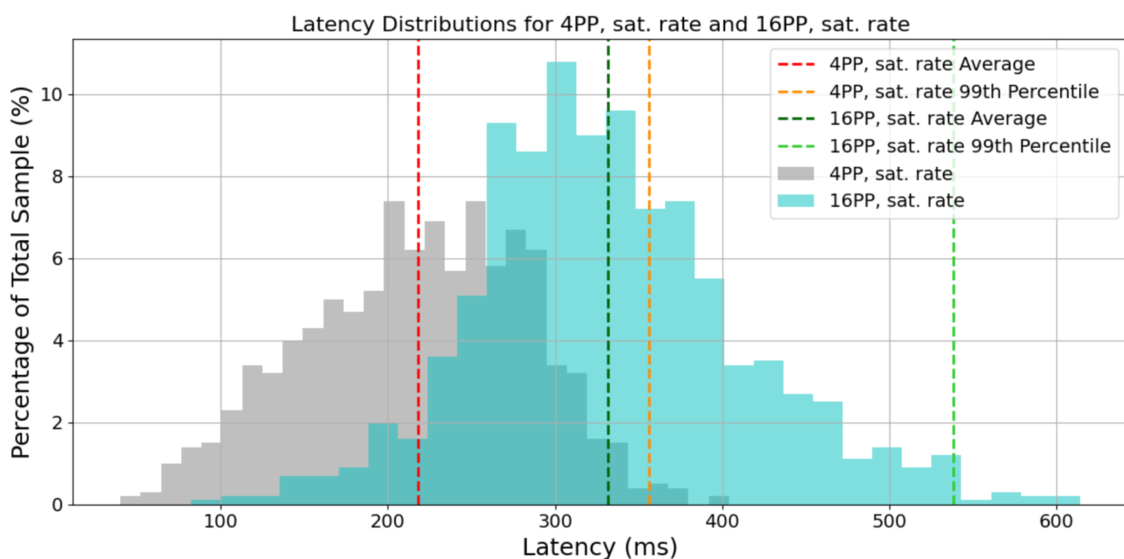

Figure 6.7: Comparison of the pipeline latency percentiles for the saturation rate at parallelism degree four and 16 respectively.

Figure 6.8 shows how the distribution of latencies in the saturation rate is impacted

by introducing fault tolerance to the pipeline. The higher average and 99th percentile of the 8PP3B show that its latencies generally are higher. With this said, the 99th percentile is still well below one second in latency. In comparison to the 99th percentile in the other pipelines, the 8 parallelism pipelines are in the middle of 4PP and 16PP. This means that the 8PP can be augmented with fault tolerance but still has a lower average latency than the 16PP at the data rates up until its saturation rate.

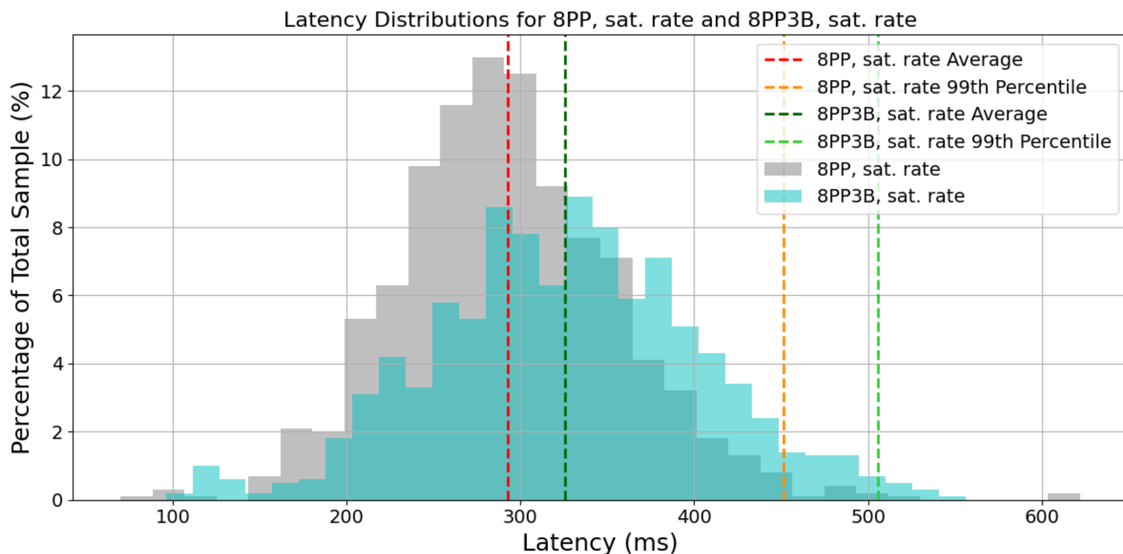

Figure 6.8: Comparison of the pipeline latency percentiles for the saturation rate at parallelism degree of eight, with one and three brokers respectively.

6.6 Resource utilization performance

From the scalability and latency tests it is shown that the 8PP has lower latencies than the 16PP and a similar saturation rate. In this section, the resource utilization of the 8PP is compared to that of the 8PP3B in order to study the effect of fault tolerance on resource utilization. The results contribute to addressing the scalability part of ***R3***. The types of resources that are measured are heap memory, non-heap memory and CPU.

The resource utilization of each pipeline at the minimum required data rate is shown in Figure 6.9. The main difference between the two pipelines in terms of available resources is that the 8PP3B has two additional Kafka brokers. The Kafka brokers each use approximately the same amount of heap memory. The distribution of utilization inside each broker over time is even due to the sharply fluctuating behavior of the heap memory usage. This behavior can be seen in Figure 6.10. Both JobManagers use similar amounts of heap memory, but the 8PP3B uses slightly more resources for its TaskManager. The difference in non-heap memory usage between the two pipelines is marginal. The percentage of CPU used is greater in the 8PP3B for each measured container. A likely explanation for this is that the 8PP is not at its CPU limit at this rate. If it were, then the overall CPU utilization for the two

pipelines would be equal. This means that the 8PP3B has a higher CPU utilization already at the minimum requirement rate.

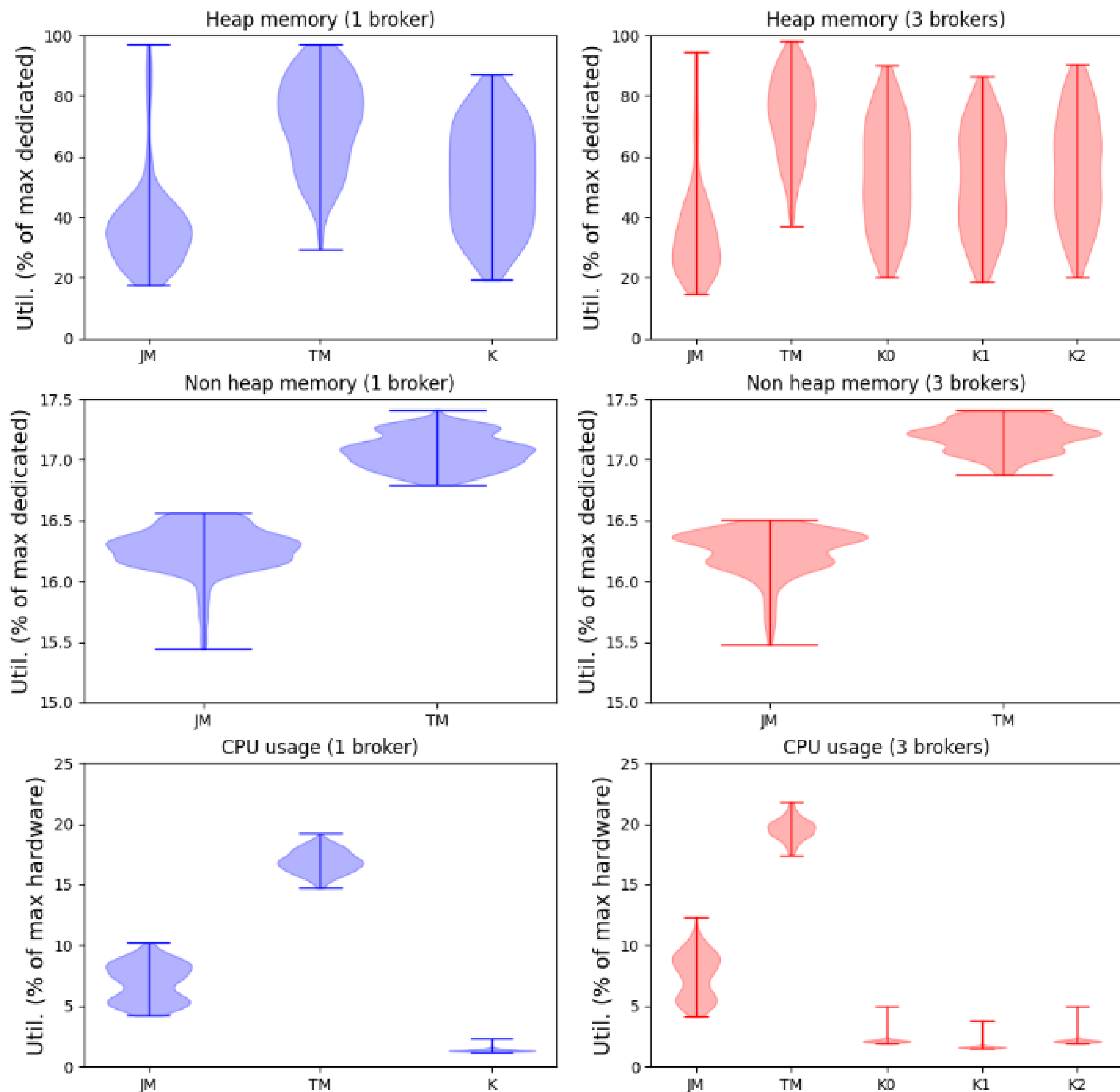

Figure 6.9: Comparison of the resource utilization at the minimum requirement rate with parallelism eight using 1 and 3 brokers respectively. Here, JM and TM are short for JobManager and TaskManager respectively. K, K0, K1 and K2 represent Kafka brokers.

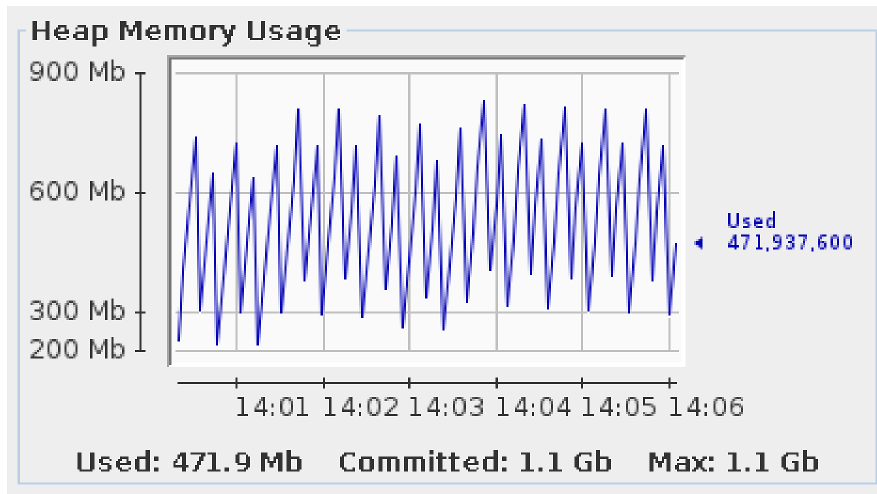

Figure 6.10: A screenshot showing the behavior of the heap memory in the Kafka brokers.

Figure 6.11 presents the resource utilization of the two pipelines at their respective saturation rates. The heap memory usage of the JobManagers at this rate has barely changed from the percentage that was used at the minimum required data rate. Both TaskManagers have lower minimum values for the utilization than at the minimum requirement rate. Utilization of heap memory in K0, K1, and K2 has altered slightly, but this could be due to small irregularities in the distribution of workload among them. The non-heap memory results are similar to that of the minimum requirement rate, with the exception of the 8PP3B JobManager having a more condensed distribution at this data rate. The metric where higher rates made the most difference is the CPU. At this rate, the CPU utilization of the 8PP JobManager and Kafka broker is about the same as before. However, the TaskManager CPU usage has increased to 50% of the maximum hardware capacity. Meanwhile, the 8PP3B has increased JobManager CPU usage and doubled its TaskManager CPU usage. That the 8PP3B TaskManager has gone from higher CPU usage than the TaskManager of 8PP to lower CPU usage indicates that at this rate, the added overhead of the additional Kafka brokers impacts Flink job performance. It can be seen that roughly the same amount of CPU resources is used per broker regardless of how many are running simultaneously. This, together with the additional CPU required for the JobManager, is likely what degraded the performance in the TaskManager of the 8PP3B.

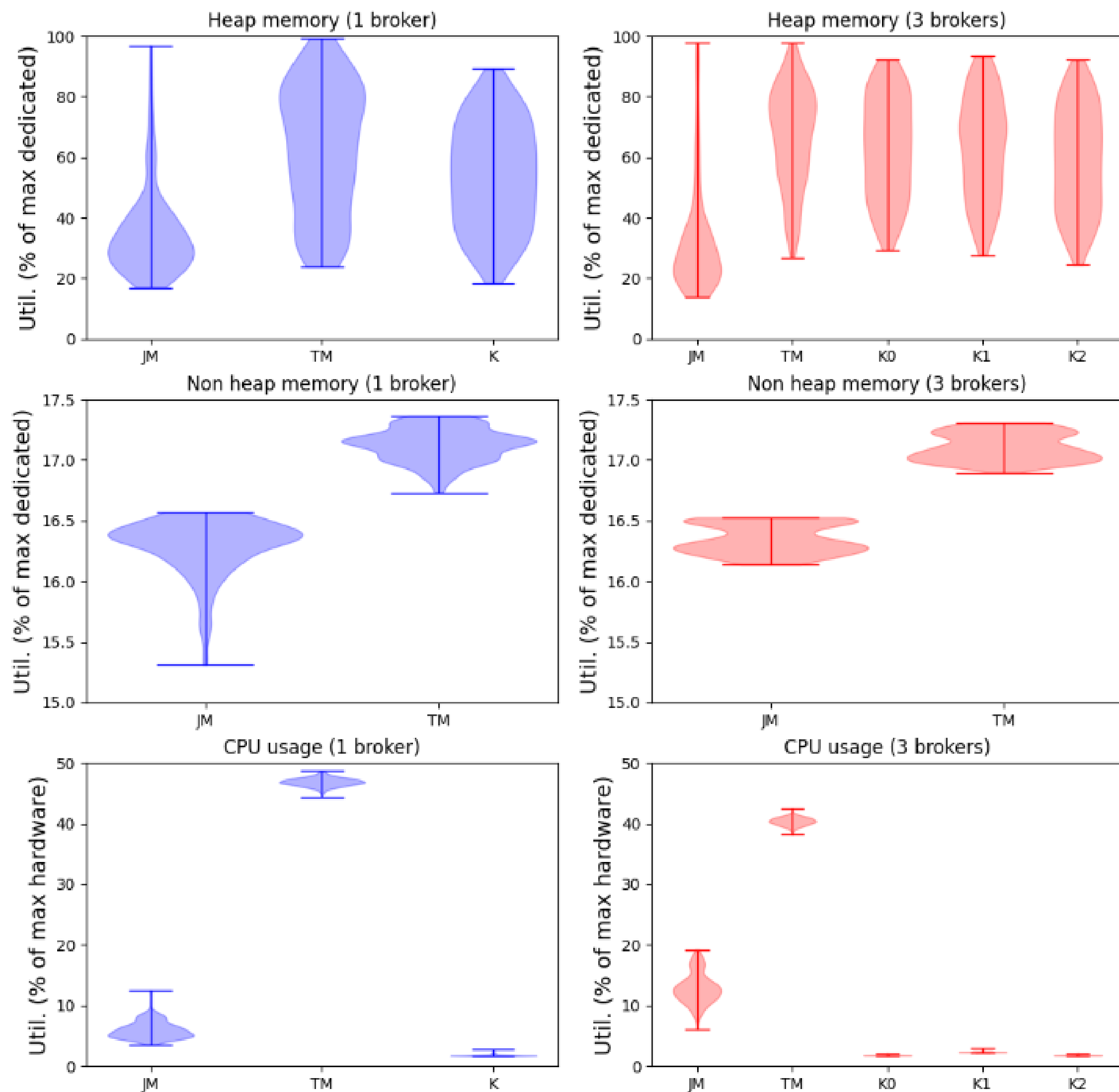

Figure 6.11: Comparison of the resource utilization at the saturation rate with a parallelism degree of 8, using 1 and 3 brokers respectively.

Disk utilization

During testing, disk speeds are not causing bottlenecks. While Kafka persists logs on disk and does not guarantee sequential writes and reads, it utilizes larger contiguously structured segments and the OS page cache to enforce sequential access as much as possible [60]. The disk used in the experiments reports the following sequential write and read speeds:

```
Sequential write:

dd if=/dev/zero of=/tmp/test bs=1M count=65536 oflag=direct conv=
   fdatasync
65536+0 records in
65536+0 records out
68719476736 bytes (69 GB, 64 GiB) copied, 11.5927 s, 5.9 GB/s

Sequential read:

dd if=/tmp/test of=/dev/null bs=1M count=65536
65536+0 records in
65536+0 records out
68719476736 bytes (69 GB, 64 GiB) copied, 12.3117 s, 5.6 GB/s
```

Listing 6.1: Sequential write and read tests.

These speeds exceed the throughput of the pipeline by a large margin, considering that the highest rate that the pipeline can sustain is 205000 tuples per second or just below 16MB per second.

6.7 Performance observations

To further reflect upon the results presented above, this section ties them to the research question ***Q***, stated in Chapter 3. One major observation is the fact that the log replication for fault tolerance introduces enough overhead to affect both latency and scalability performance. It would seem that the requirement of ensuring that a minimum amount (two in this case) of replications have been created and are in sync within the Kafka cluster is causing a bottleneck. The producers (including the KafkaSinks in Flink) all have to await acknowledgments from the Kafka cluster that this procedure has been performed before committing any transactions, which could in part explain the increased latency even at lower data rates. Resource contention also starts to affect the overall performance in this scenario, which is reflected by the 32.2% shift of the saturation point and the 14.9% increase in latency at this point. In the minimum requirement setup, with the 8PP, each slot in the TaskManager is intended to have one CPU core dedicated to it. The same principle applies to the JobManager, the single Kafka broker, and Druid. With two additional brokers, Kafka requires access to more RAM and cores in order to maintain all partitions (which are all single-threaded) across the brokers. While the overall CPU utilization of Kafka is modest, the RAM activity has roughly tripled. However, with an increase in elements vying for resources, more frequent context switching and higher process synchronization efforts could be reasons for the overall degradation of pipeline performance. It should be noted that the overhead induced by the multi-container Docker instance is not accounted for in the testing. With two additional brokers, Docker has to maintain two additional Kafka containers, which also affects overall resource requirements. Needless to say, the evaluation environment does not only run the pipeline components. Instead, a more interesting conclusion that can be drawn here is that pipeline performance can indeed be affected greatly depending

on which requirement(s) take high precedence. In this case, the importance of fault tolerance causes a noticeable decline in scalability.

The 8PP3B managed to satisfy all the pipeline requirements. Here, data completeness is achieved through exactly-once semantics, while fault-tolerance is assured through log replication on the Kafka level (requirement ***R1*** in Section 5.1). As mentioned in Section 6.5, the 99th percentile dictates whether overall latency performance is acceptable, and through testing it can be seen that the 8PP3B can sustain data rates at sub-second latencies at both the minimum requirement and saturation rates (requirements ***R2*** and ***R3*** in Section 5.1). By prioritizing fault tolerance the highest, some sacrifices were made in terms of latency and scalability.

During testing of different degrees of parallelism, it can be seen that the latency performance is degraded at higher degrees, even at low data rates. As presented, data skews do not seem to be the main malefactor in this regard. However, much like with an increase in brokers, an increase in parallelism degree most certainly leads to heightened synchronization efforts within the system.

6.8 Operating in a controlled environment

The controlled environment in which the pipeline is run avoids factors that are more than likely to affect performance in real-life scenarios. Due to all containers being run on the same device, network bandwidth is a negligible factor. As mentioned, disk speeds also did not affect performance in any meaningful way. If the pipeline were to be distributed across multiple devices in order to increase the available resources, then these factors can potentially become bottlenecks, especially with even larger volumes of data. On the topic of data, the use of simulated data contributed to the controlled environment as well. During testing, the throughput was stable, with no activity spikes being simulated. This removed any reason to perform tests such as moving average analyses on the throughput capabilities of the pipeline.

Regarding tuple size, the average size of 80 bytes is relatively small. In order to test the impact of larger tuples in the pipeline, the padding field mentioned in Figure 5.3 was used to increase the size of each tuple to approximately 1 KB. While testing did, as expected, result in higher throughput in terms of bytes per second but lower in tuples per second, the logic was not updated to handle more types of data than before. This means that these tests more accurately depict the scenario where the value of each tuple is large, not the scenario where there is an increased number of interesting fields (i.e. vehicle signals) to consider in each tuple. This did not fit well with the use case and therefore these test results were deemed unsuitable for the thesis.

To have full control over the data rate during testing, all tuples were set to have NaN values. While this configuration is apt for measuring how many tuples can be handled per second by the pipeline, it does in fact represent the worst-case scenario in terms of data rate. For the use case, the majority of values will not be NaN. Thus, for a system with a NaN-value rate of 5%, the pipeline would only keep one

in twenty tuples for windowing. Since the main bottleneck of the pipeline is tied to the minute-wise sliding windowing, this should result in a workload decrease of 95%. The magnitude of this decrease depends on the rate of NaN values, which means that the pipeline can be expected to handle significantly higher data rates in systems with low NaN rates.

7

Related work

Systems and pipelines for large data volumes have been studied for a variety of use cases, such as smart grids and mobile cellular networks. This section lists a selection of such use cases in order to provide insight into how this thesis fits within the context.

A large-scale system for validation of Big Data streams has been built for the domain of energy grids [9]. In energy grids, the eChiDNA system aims to collect and validate data from an Advanced Metering Infrastructure (AMI) containing millions of smart energy meters. Here, it is desirable to collect energy data automatically due to the distributed nature of the smart meters. Furthermore, automatic readings allow for more granular sampling periods, improving the accuracy of load forecasting and billing. Increased autonomy and data volumes necessitate monitoring and validation of the data in order to detect erroneous values caused by, for instance, malfunctioning hardware, unlawful manipulation of energy values and software issues. The eChiDNA system is implemented using Kafka for data ingestion and the SPE Apache Storm [61] for data validation. Validation is performed based on a configurable number of validation rules that can consider both single and composite events. Low latency and high throughput were achieved during experiments performed on millions of devices reporting values on an hourly basis.

Similarly to the Lambda architecture mentioned in Section 2.2.2, the data processing approach referred to as micro batching attempts to leverage the benefits of both stream and batch processing. This method attempts to achieve lower latency than regular batch processing by buffering data in short intervals into small batches of data that are processed together [62]. The smaller size of batches decreases the latency penalty incurred by the length of the buffering interval. Meanwhile, the throughput of the computation is still high, due to computation being performed in batches of data. Although the latency for micro-batching can be considered to be low at a few seconds, it is still significantly larger than the sub-second latency of stream processing approaches [62, p. 582].

Mobile cellular network data has been monitored using statistical methods to detect anomalies such as system failures and missing values [63]. This pipeline is designed to scale for high-velocity streams containing multidimensional data with a daily data volume in the order of terabytes. The motivation for this is based on a demand for providing nontrivial insights from these large data volumes in real

time. For dynamically detecting anomaly detection, the statistical metrics Pearson correlation and relative entropy were used. These maintain a baseline, built from non-anomalous data, that is used to determine whether incoming data is anomalous or not. The implementation of the Big Data pipeline starts with a pre-processing infrastructure called Firehose. Firehose collects binary signal events, parses them and then serializes events to the Avro format. The Avro data is then written to different Kafka queues, depending on the structure of the signal. The SPE Spark Structured Streaming [64] reads the data from the Kafka queues and performs anomaly detection. The metric values are written continuously to a dashboard and alerts are also triggered when anomalies are detected. The size of the Spark Streaming micro-batches is chosen experimentally based on the data volumes in the pipeline as well as which type of data is used (e.g. 2G, 3G). Parallel consumers are used in the streaming and these are configured to receive every event only once. Experiments performed on events from millions of mobile devices show that the system is efficient and effective for large data volumes. The system has a batching component as well, which means that it conforms to the Lambda architecture.

The use of validation as a separate step of pre-processing before any further stream processing is performed on data has been explored [65]. The motivation is that data that is not validated will compromise the otherwise useful results of computations given by stream processing. By validating first, data consistency and integrity are ensured. The use of stream processing for validation additionally avoids the issue of first storing unbounded data by instead processing it in real-time. The proposed system architecture ingests data from different sources into a data validator implemented in Flink. Data is then analyzed in different ways both statelessly (e.g. missing data in fields) and statefully (e.g. identifying duplicated data). The data that is deemed valid is allowed to leave the pre-processing step and will be forwarded for further stream processing implemented in the client library Kafka Streams [66]. Kafka Streams was chosen due to its interoperability with other applications, fault tolerance, scalability and processing options. The system is evaluated from a use case for validating data from energy meters in smart energy grids.

Stream processing has been utilized to detect malfunctioning smart meters based on their voltage streams [47]. The system in question, called LoCoVolt, leverages the fact that smart meters with common contexts should report similar values. If data from a smart meter deviates over a certain threshold, this will trigger an alert. It is important that these alerts are timely, to ensure that faulty meters can be exchanged quickly. Otherwise, it can result in incorrect billing as well as damages to people and equipment. Moreover, from a business value perspective, the data must have good quality in order to be useful. The stream processing was implemented using Flink. Tuples were counted in windows of an hour, matching the rate at which the smart meters reported values. Testing was performed on 4 million readings collected from 939 smart meters. A resource-restricted device with similar resources to those used in the field was able to achieve a throughput rate of 600 tuples per second with a latency of two seconds. The evaluation showed that the detection was accurate at the tested rates.

The challenge of balancing performance factors and trade-offs has previously been studied through the FORTE framework [10]. Here, the task of optimizing transfers of large batches of vehicular data is addressed through lossless compression and in-memory processing techniques. The paper highlights the complexities of handling a large number of parameter choices during the optimization process. By harnessing CPU and RAM resources more efficiently, the addition of the aforementioned compression and in-memory processing allows FORTE to outperform a baseline approach by up to 1.8 times in terms of sustainable data rate. FORTE also enables a set of qualitative properties, enhancing data veracity, governance and security, and its efficient use of resources suggests benefits from an energy-consumption perspective, as well as in terms of scheduling and orchestration.

The DRIVEN framework implements online, lossy compression using stream processing to allow for the analysis of large amounts of positional data from connected vehicles [67]. In the framework, data analysts in analysis centers can define stream processing queries over vehicle sensor data. A query defines the maximum error that can be introduced from the compression as well as what vehicles, timespan and set of sensors should be considered. Multiple maximum error values, network speeds and use cases are considered to evaluate trade-offs in average error, compression ratio and gathering time. The framework prototype, implemented using Apache Flink, can achieve a tenfold improvement in transmission speeds and a compression ratio of up to 20:1 while incurring an accuracy loss of below 10%.

8

Conclusion and future work

This thesis studies design principles and considerations of scalable stream processing pipelines. As various industrial contexts are creating ever-increasing amounts of data and are becoming progressively data-driven, the need for efficient data processing pipelines cannot be overstated. The design of such pipelines demands careful attention to detail, especially in regard to balancing performance requirements. Replacing existing systems for data handling can be a daunting undertaking, but the pressing need for better solutions within various industrial contexts could make this a necessary course of action. This thesis depicts to what extent strict requirements such as fault-tolerance, high scalability, sub-second latencies and the ability to handle high throughput of massive volumes of data have on the performance of a pipeline. To exemplify this, the thesis showcases the design and construction of a stream processing pipeline for data validation, and performs thorough comparisons of different pipeline configurations, with and without fault-tolerance applied, in order to assess how requirement precedence can affect performance. These comparisons reveal that the inclusion of fault tolerance can have noticeable effects on overall pipeline latency and scalability, showing a decrease of 32.2% in regard to pipeline saturation rate and an average latency increase of 14.9% at this rate. The fault-tolerant pipeline still manages to comfortably sustain data rates of 11575 to 116000 data tuples per second, corresponding to a range of 1-10 billion tuples per day, all while achieving latencies of just above 500 ms within the 99th percentile. These numbers exceed the current demands of a use-case at Volvo Trucks, where data rates are steadily increasing and the need for a new, more efficient data processing infrastructure is growing rapidly. The results highlight the importance of careful capacity planning and consideration of requirement precedence when designing data processing pipelines for Big Data environments.

Future work

As revealed in the study, there is a cost inherent to the scalability of data processing pipelines, especially when designing around strict requirements. As a consequence, there exists an incentive to investigate how performance can be optimized. Such investigations can consider processing logic, tool tuning, as well as suitable extensions to the pipeline. In the following paragraphs, a selection of future considerations for the proof-of-concept pipeline is presented.

Counting NaNs inside of data is currently the sole focus of the validation logic of the proof of concept pipeline, but the functionality can easily be extended. By adapting the data ingested into Flink as well as the processing logic in Flink itself, arbitrary validation rules can be implemented. The limitation of what can be done in the processing logic is the resources available to the system running it.

If data was pre-processed and aggregated in each vehicle first, then the pipeline resources could be used more efficiently. An object as large as a truck likely has room for the computing resources required to do lightweight processing of the data originating from its sensors. The compressed nature of the aggregated data would also mean that more data can be sent on the network link to where the central data pipeline is located.

Another way to increase pipeline resource utilization would be to deploy the resources of the pipeline in a distributed cluster. The pipeline components, including Flink, Kafka and Druid are all distributable and scalable. This allows them to be controlled by orchestration tools such as Kubernetes [68]. A distributed setup generally improves the resource utilization of the pipeline and makes it adapt to fluctuating data volumes over time more efficiently.

In the pipeline, exactly-once semantics and log replication facilitate fault tolerance, but there are other levels at which this can be addressed as well. Flink is currently only using a single JobManager. This makes the JobManager a single point of failure for the processing. If it shuts down, the incoming workload will not be handled until a new JobManager is introduced. If the downtime is long, the latency of the pipeline can suffer. To address fault tolerance at this level, additional JobManagers should be on standby, waiting for a chance to take over the computation when a JobManager is shut down.

The pipeline does not persist results across runs in its current implementation, which means that historical queries only can be performed in Druid on data from an active run. However, if Druid is configured to persist data, it would theoretically be possible to query the frequencies for a specific customer, vehicle and signal over an arbitrary timespan at the granularity of one minute. This could be done by selecting the last aggregation for each minute (which should contain the exact frequencies of the last minute) and then summing the values into the total frequency over the timespan of choice.

Alerts could also be implemented into the design. For instance, a monitoring tool with alert functionality, such as Grafana [69], could consume results from the Kafka output topic and make decisions based on this data. Alerts could, for example, be triggered by a stateless threshold for single values or from a stateful threshold, which can take a number of previous values into consideration.

All of the directions covered above introduce further considerations and challenges. Any extension or modification to the pipeline could impose new requirements. Each of these needs to be treated with the same amount of attentiveness as any existing requirements, to ensure that pipeline operation remains reliable and performant.

Bibliography

[1] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *SIGMOD Rec.*, vol. 34, no. 4, p. 42–47, dec 2005. [Online]. Available: https://doi.org/10.1145/1107499.1107504

[2] A. Jacobs, "The pathologies of big data," *Communications of the ACM*, vol. 52, no. 8, pp. 36–44, 2009.

[3] H. Foidl, V. Golendukhina, R. Ramler, and M. Felderer, "Data pipeline quality: Influencing factors, root causes of data-related issues, and processing problem areas for developers," *J. Syst. Softw.*, vol. 207, p. 111855, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:261705942

[4] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt *et al.*, "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, 2015.

[5] S. Shahrivari, "Beyond batch processing: Towards real-time and streaming big data," *Computers*, vol. 3, no. 4, pp. 117–129, 2014. [Online]. Available: https://www.mdpi.com/2073-431X/3/4/117

[6] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, and S. Khan, "A survey of distributed data stream processing frameworks," *IEEE Access*, vol. 7, pp. 154 300–154 316, 2019.

[7] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker, "Fault-tolerance in the borealis distributed stream processing system," *ACM Trans. Database Syst.*, vol. 33, no. 1, mar 2008. [Online]. Available: https://doi.org/10.1145/1331904.1331907

[8] K. Goodhope, J. Koshy, J. Kreps, N. Narkhede, R. Park, J. Rao, and V. Y. Ye, "Building linkedin's real-time activity data pipeline." *IEEE Data Eng. Bull.*, vol. 35, no. 2, pp. 33–45, 2012.

[9] J. van Rooij, J. Swetzén, V. Gulisano, M. Almgren, and M. Papatriantafilou, "echidna: Continuous data validation in advanced metering infrastructures," in *2018 IEEE International Energy Conference (ENERGYCON)*, 2018, pp. 1–6.

[10] M. Hilgendorf, V. Gulisano, M. Papatriantafilou, J. Engström, and B. Mishra, "Forte: an extensible framework for robustness and efficiency in data transfer

pipelines," in *Proceedings of the 17th ACM International Conference on Distributed and Event-based Systems*, 2023, pp. 139–150.

[11] M. Zhang, T. Wo, T. Xie, X. Lin, and Y. Liu, "Carstream: an industrial system of big data processing for internet-of-vehicles," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1766–1777, 2017.

[12] R. Abo and L. Voisin, "Formal implementation of data validation for railway safety-related systems with ovado," in *Software Engineering and Formal Methods: SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS, FM-RAIL-Bok, MoKMaSD, and OpenCert, Madrid, Spain, September 23-24, 2013, Revised Selected Papers 11*. Springer, 2014, pp. 221–236.

[13] J. Kabbedijk, C.-P. Bezemer, S. Jansen, and A. Zaidman, "Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective," *Journal of Systems and Software*, vol. 100, pp. 139–148, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121214002313

[14] T. Akidau, S. Chernyak, and R. Lax, *Streaming systems: the what, where, when, and how of large-scale data processing.* " O'Reilly Media, Inc.", 2018.

[15] M. Drocco, C. Misale, G. Tremblay, and M. Aldinucci, "A formal semantics for data analytics pipelines," *arXiv preprint arXiv:1705.01629*, 2017.

[16] D. Palyvos-Giannas, *Explainable and Resource-Efficient Stream Processing Through Provenance and Scheduling.* Chalmers Tekniska Hogskola (Sweden), 2022.

[17] M. Kleppmann, *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems.* " O'Reilly Media, Inc.", 2017.

[18] D. Palyvos-Giannas, K. Tzompanaki, M. Papatriantafilou, and V. Gulisano, "Erebus: Explaining the outputs of data streaming queries," in *Very Large Data Base*, vol. 16, no. 2, 2023, pp. 230–242.

[19] D. Palyvos-Giannas, V. Gulisano, and M. Papatriantafilou, "Haren: A framework for ad-hoc thread scheduling policies for data streaming applications," in *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*, 2019, pp. 19–30.

[20] J. Verwiebe, P. M. Grulich, J. Traub, and V. Markl, "Survey of window types for aggregation in stream processing systems," *The VLDB Journal*, vol. 32, no. 5, pp. 985–1011, 2023.

[21] Apache Flink™, "Windows." [Online]. Available: https://nightlies.apache.org/flink/flink-docs-release-1.19/docs/dev/datastream/operators/windows/

[22] T. Akidau, E. Begoli, S. Chernyak, F. Hueske, K. Knight, K. Knowles, D. Mills, and D. Sotolongo, "Watermarks in stream processing systems: Semantics and comparative analysis of apache flink and google cloud dataflow," Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), Tech. Rep., 2021.

[23] B. Pernici and M. Scannapieco, “Data quality in web information systems,” in *International Conference on Conceptual Modeling.* Springer, 2002, pp. 397–413.

[24] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos, “A survey on the evolution of stream processing systems,” *The VLDB Journal*, pp. 1–35, 2023.

[25] Apache Flink™, “Fault tolerance via state snapshots.” [Online]. Available: https://nightlies.apache.org/flink/flink-docs-release-1.19/docs/learn-flink/fault_tolerance/

[26] E. Dubrova, *Fault-tolerant design.* Springer, 2013, vol. 8.

[27] Z. Chen and J. Dongarra, “Algorithm-based fault tolerance for fail-stop failures,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 12, pp. 1628–1641, 2008.

[28] B. Charron-Bost, F. Pedone, and A. Schiper, *Replication.* Springer, 2010.

[29] J. Kreps, N. Narkhede, J. Rao *et al.*, “Kafka®: A distributed messaging system for log processing,” in *Proceedings of the NetDB*, vol. 11, no. 2011. Athens, Greece, 2011, pp. 1–7.

[30] A. Kafka, “Documentation.” [Online]. Available: https://kafka.apache.org/documentation/

[31] Confluent, “Kafka Consumer Design: Consumers, Consumer Groups, and Consumer Offsets.” [Online]. Available: https://docs.confluent.io/kafka/design/consumer-design.html

[32] K. Grochowski, M. Breiter, and R. Nowak, “Serialization in object-oriented programming languages,” in *Introduction to data science and machine learning.* IntechOpen, 2019, pp. 1–18.

[33] Apache Avro™, “Apache Avro 1.11.1 documentation.” [Online]. Available: https://avro.apache.org/docs/1.11.1/

[34] ——, “Apache Avro 1.11.1 specification.” [Online]. Available: https://avro.apache.org/docs/1.11.1/specification/

[35] The Apache Software Foundation, “Welcome to Apache Maven.” [Online]. Available: https://maven.apache.org/index.html

[36] Apache Avro™, “Getting Started (Java).” [Online]. Available: https://avro.apache.org/docs/1.11.1/getting-started-java/

[37] Confluent, “Schema Registry Overview.” [Online]. Available: https://docs.confluent.io/platform/7.1/schema-registry/index.html

[38] ——, “Schema Evolution and Compatibility for Schema Registry.” [Online]. Available: https://docs.confluent.io/platform/current/schema-registry/fundamentals/schema-evolution.html

[39] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, “Apache Flink™: Stream and Batch Processing in a Single Engine,” *The Bulletin of the Technical Committee on Data Engineering*, vol. 38, no. 4, 2015.

[40] Apache Flink®, “Flink architecture.” [Online]. Available: https://nightlies.apache.org/flink/flink-docs-master/docs/concepts/flink-architecture/

[41] ——, “Checkpointing.” [Online]. Available: https://nightlies.apache.org/flink/flink-docs-release-1.19/docs/dev/datastream/fault-tolerance/checkpointing/

[42] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli, “Druid: A real-time analytical data store,” in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 157–168.

[43] P. Valduriez, *Shared-Nothing Architecture*. Boston, MA: Springer US, 2009, pp. 2638–2639. [Online]. Available: https://doi.org/10.1007/978-0-387-39940-9_1512

[44] B. B. Rad, H. J. Bhatti, and M. Ahmadi, “An introduction to Docker®and analysis of its performance,” *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 17, no. 3, p. 228, 2017.

[45] Docker Inc., “How compose works.” [Online]. Available: https://docs.docker.com/compose/compose-application-model/

[46] ——, “Multi container apps.” [Online]. Available: https://docs.docker.com/get-started/07_multi_container/

[47] J. van Rooij, V. Gulisano, and M. Papatriantafilou, “Locovolt: Distributed detection of broken meters in smart grids through stream processing,” in *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*, 2018, pp. 171–182.

[48] M. Hanif, H. Yoon, and C. Lee, “A Backpressure Mitigation Scheme in Distributed Stream Processing Engines,” in *2020 International Conference on Information Networking (ICOIN)*. IEEE, 2020, pp. 713–716.

[49] Apache Parquet, “Overview.” [Online]. Available: https://parquet.apache.org/docs/overview/

[50] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, “Benchmarking distributed stream data processing systems,” in *2018 IEEE 34th international conference on data engineering (ICDE)*. IEEE, 2018, pp. 1507–1518.

[51] Apache Kafka®, “Notable changes in 3.0.1.” [Online]. Available: https://kafka.apache.org/30/documentation.html#upgrade_301_notable

[52] ——, “Class KafkaProducer<k,v>.” [Online]. Available: https://kafka.apache.org/37/javadoc/org/apache/kafka/clients/producer/KafkaProducer.html

[53] Apache Flink®, “Checkpointing.” [Online]. Available: https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/dev/datastream/fault-tolerance/checkpointing/

[54] Apache Druid®, “Consumer properties.” [Online]. Available: https://druid.apache.org/docs/latest/ingestion/kafka-ingestion/#consumer-properties

[55] Microsoft Learn, “What is the windows subsystem for linux?” [Online]. Available: https://learn.microsoft.com/en-us/windows/wsl/about

[56] Apache Flink®, "Deployment." [Online]. Available: https://nightlies.apache.org/flink/flink-docs-master/docs/deployment/overview/

[57] R. Metzger, "Kafka + flink: A practical, how-to guide." [Online]. Available: https://www.ververica.com/blog/kafka-flink-a-practical-how-to

[58] Oracle, "Using JConsole." [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/management/jconsole.html

[59] T. S. community, "scipy.stats.skew." [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.skew.html

[60] Apache Kafka®, "Don't fear the filesystem!" [Online]. Available: https://svn.apache.org/repos/asf/kafka/site/083/design.html

[61] Apache Storm™, "Apache Storm." [Online]. Available: https://storm.apache.org/index.html

[62] N. Tantalaki, S. Souravlas, and M. Roumeliotis, "A review on big data real-time stream processing and its scheduling techniques," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 35, no. 5, pp. 571–601, 2020.

[63] L. Rettig, M. Khayati, P. Cudré-Mauroux, and M. Piórkowski, "Online anomaly detection over big data streams," *Applied Data Science: Lessons Learned for the Data-Driven Business*, pp. 289–312, 2019.

[64] Apache Spark™, "Structured Streaming Programming Guide." [Online]. Available: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

[65] P. Baban, "Pre-processing and Data Validation in IoT Data Streams," in *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*, 2020, pp. 226–229.

[66] Apache Kafka®, "Streams." [Online]. Available: https://kafka.apache.org/0102/documentation/streams/

[67] B. Havers, R. Duvignau, H. Najdataei, V. Gulisano, M. Papatriantafilou, and A. C. Koppisetty, "Driven: A framework for efficient data retrieval and clustering in vehicular networks," *Future Generation Computer Systems*, vol. 107, pp. 1–17, 2020.

[68] D. Rensin, *Kubernetes*. O'Reilly Media, Incorporated, 2015.

[69] Grafana, "About Grafana." [Online]. Available: https://grafana.com/docs/grafana/latest/introduction/

DEPARTMENT OF SOME SUBJECT OR TECHNOLOGY
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY