CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Large Scale Efficient Data Readout for Vehicle Fleets

Master's thesis in Computer science and engineering

Simon Johnsson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Large Scale Efficient Data Readout for Vehicle Fleets

Simon Johnsson

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Large Scale Efficient Data Readout for Vehicle Fleets
Simon Johnsson

Supervisor: Miquel Pericas, Computer Science and Engineering
Advisor: Dhasarathy Parthasarathy, Volvo Group Trucks Technology
Examiner: Miquel Pericas, Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
Gothenburg, Sweden 2025

Large Scale Efficient Data Readout for Vehicle Fleets
Simon Johnsson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

As vehicles become more technologically advanced, the data generated by a single vehicle reach significant amounts. Diverse data has high potential in use cases such as machine learning by providing insights into different conditions. Currently, there is no clear solution for collecting this data as vehicle systems are restricted in terms of compute, memory, storage, and bandwidth. This thesis investigates the problem of large scale vehicle data readout and presents a solution to it, providing a significant increase by leveraging lossless streaming based compression at low cost. Furthermore, it addresses the architecture necessary in order to sufficiently process the data globally and how best to integrate this efficiently with a massive number of vehicle systems. Lastly, a generalized model is formulated at the micro scale, which establishes the requirements in terms of compute and memory on a single vehicle system based on the findings presented. At the macro scale, the infrastructure required to support the solution is discussed.

Keywords: Data readout, compression, databases, vehicle systems, streaming, embedded systems, master thesis.

Acknowledgements

Firstly, I want to thank Dhas for always giving me interesting problems while considering the bigger picture with his support.

Secondly, I want to thank Miquel for always providing insightful advice and fun conversations that made the challenge of writing a thesis alone less daunting.

Thirdly, I want to thank Szabolcs for all the interesting discussions and thoughtful perspectives.

Finally, I want to thank my family and friends for their endless support and care.

Simon Johnsson, Gothenburg, 2025-06-02

Contents

List of Figures

List of Tables

1

Introduction

During the course of operation, a technologically advanced transport vehicle may internally receive thousands of signals related to its functional attributes every millisecond. These include operational properties such as speed, fuel consumption, gear positions, and GPS location. Such values have the potential to provide valuable insight into the individual vehicle's behavior under different conditions by utilizing statistical analysis and machine learning. However, in order for these data to be of any meaningful use in analytical applications, they must in some way be transferred from the vehicle. The environment in which the vehicle travels changes steadily over time, which means that the conditions for data communication can often be less than favorable. Furthermore the vehicle's system is itself constrained in both storage, bandwidth, and compute capacity, which means that simply storing the data locally would not be sufficient. Vehicles do not therefore currently possess an inherent solution to capture all of this data in an efficient manner, and as a consequence valuable transient signal data gets lost.

In 2004, the Oregon Department of Transportation released a survey on truck data collection methods which recommended that data be collected using intercept interviews and surveys based on mail and fax [1]. Today such methods can arguably be seen as archaic, and the digital landscape in which they were proposed did not have in mind the mass data requirements for machine learning that was to come. Later in 2006, McCormack and Hallenbeck [2] investigated the usage of externally placed transponders and GPS to collect truck movement data along roadway corridors in Washington State. Although useful for evaluating aspects such as road congestion, this form of data collection does not concern the vehicle's own internal state.

A recently emergent motivator of vehicle data collection is the area of self driving cars [3]. These collect numerous different points of information such as GPS, LiDAR, and camera footage, which is thereafter stored and processed on the vehicle. One way of capturing sensor data is through the installation of devices that read communications on the vehicle's internal system [4]. In 2021, Bunting et al. [4] introduced a high performance library for interacting with the so called "Panda" device, developed by Comma.ai. This library aimed to reduce the CPU usage in low-cost hardware such as Raspberry Pis. It is however not always an alternative to install third party devices in transport vehicles and therefore a purely software based solution is needed.

1.1 Aim

The aim is to find a solution that allows for the maximized lossless collection of data points given by the sensors of a truck as part of a vehicle fleet. At the micro scale, a vehicle-local solution is to be presented that collects and transmits signal data in a lightweight and efficient way. At the macro scale, the project aims to investigate the reception of these data streams at the server level from the perspective of managing a large scale vehicle fleet and regarding what solution is most apt to the problem.

1.2 Limitations

While this project aims to construct a solution which takes other external processes into consideration, testing will not consider physical systems currently running in production. This limitation mainly arises from time constraints in terms of testing in real life scenarios, where it would be necessary to establish test environments on automotive roads with limited internet connection. Furthermore additional effort would be required in order to analyze effectiveness from a global perspective over a larger period of time. As such, test environments will be theoretically mocked and appropriated to the extent possible from an academic perspective.

1.3 Requirements

Based on the presented goal, a set of requirements were developed in collaboration with domain experts at Volvo Group. The following requirements were established for the resulting system:

- Capture 4000 double-precision signal data points per sampled time step with minimal transfer size and system load.
- Capture signal data streams with a sampling rate of up to 100 samples per second (100Hz).
- Receive and persistently store captured data from 1 million trucks concurrently successfully.
- Provide an accessible format for data reads.

1.4 Research Questions

> *RQ1: How can vehicle data be compressed efficiently?*
>
> *RQ2: What is the imposed CPU and memory load on the vehicle system in order to accommodate data readout?*
>
> *RQ3: What is the total data communication in terms of bandwidth required to accomodate data readout?*
>
> *RQ4: How can data from vehicles be sent and processed efficiently?*

1.5 Previous Works

The area of mass data readout for entire vehicle fleets is mostly unexplored as a consequence of the sheer amount of data generated. One study investigates data preprocessing along with utilizing a general purpose compression algorithm [5] to increase data readout for the purpose of diagnostics. Here the usage of preprocessing provides favorable results. Although, the study does not address performance implications on the vehicle system of utlizing such a method. For constrained embedded systems, factors like computational capacity and memory limits is a concern.

2

Background

In one way, the readout problem concerns numerous areas from the macro scale in terms of data centers, to the micro scale in terms of the small embedded vehicle computers. This chapter presents aims to establish previous related works and prerequisite theory used in the discussed methodology.

2.1 Properties of Vehicle Data

A major aspect to consider when regarding the question of analysing vehicular data is the complexity in the appearance of data points. For instance, one attribute may be solely concerned with depicting a discrete state of a component in the vehicle system, whereas another is a continuous composition of multiple different factors. In other words the question's intricacy arises from the inherent multivariate nature of these data points.

Capturing physical aspects over time imposes an inherent restriction: the rate of measurement is directly dependent on the sample rate of the corresponding sensor. As a consequence continuous data is captured and represented discretely per time steps. Higher sample rates can thus be inferred to give a higher accuracy in terms of representing state over time, which is benefitial for time-sensitive analysis. On the other hand, utilizing higher sample rates also incurs a higher strain on the capacity for data storage and processing. Additionally due to the data points' multivariate nature, any loss in terms of value precision during storage leads the risk of inaccurately representing the system's state.

2.2 The Controller Area Network

Most operations on data in a modern vehicle involves the usage of the Controller Area Network, also called CAN bus. Developed by Bosh in 1986, the idea of the CAN bus is to allow for the intercommunication between Electrical Control Units (ECUs) without the need for a central host computer [6]. These ECUs act as so called CAN nodes on a network and share information between each other, where a pair of these communicating nodes may be the brakes and engine for example. In other words the CAN bus acts as a viewpoint into the state of the vehicle system.

2.2.1 Electrical Control Units

The ECU is responsible for managing the state of a vehicle's part or attribute(s), such as transmission, steering, and temperatures [6]. An ECU in itself can be seen as the basic unit of computation in the vehicle system [7]. In modern vehicles the number of ECUs can reach over 70+, where each ECU shares information on the CAN bus. This information can include sensor data or control data prepared by an ECU and broadcast on the bus to the entire network. Then, every other ECU accepts the broadcast and checks the data, thereafter making a decision of either receiving or ignoring it. Consequently by broadcasting data to all other nodes on a bus, there is no need for direct complex analog connections between every node pair.

The architecture of an ECU is roughly similar to that of an ordinary personal computer, including a CPU, RAM, flash memory, I/O bridges. It can also extend to more complex configurations utilizing dual processing and Direct Memory Access (DMA) for instance [8]. Some architectures even include shorter vector instructions for the purpose of signal processing. Importantly, while ECUs exist on a network, they are independent units each responsible for their own subsystem of the vehicle, such as airbags, infotainment services, assisted driving, and many more [7]. Additionally subsystems have the potential to be made up of multiple data generating components such as sensors, meaning that the amount of data generated grows rapidly in relation to the expansion and addition of these systems.

2.2.2 CAN Bus Variants

Due to the wide spread difference in the functionalities attached to the CAN bus, there are different variants of CAN. These include Low-speed CAN, High-speed CAN, CAN Flexible Data-rate (FD), and CAN XL [6] (see Table 2.1).

Table 2.1: Simplified overview of CAN bus variants showing maximum baud rate speeds and data payload sizes. Variants include Low-speed CAN, High-speed CAN, CAN FD, and CAN XL.

Property	Low-speed CAN	High-speed CAN	CAN FD	CAN XL
Max baud rate speed	0.125 Mbps	1 Mbps	8 Mbps	20 Mbps
Max data payload size	8B	8B	64B	2048B

Low-speed CAN is most appropriate in scenarios where fault tolerance is critical and is able to continue functioning after sustaining some degree of damage. As a trade-off, the maximum baud rate speed (gross bit rate) is a mere 0.125 Mbit/s, making it mostly useful in devices related to lesser functions such as seat adjustment. Furthermore this variant is increasingly being replaced by the Local Interconnect Network protocol or LIN bus [9]; a supplement to CAN. High-speed CAN is the most common variant and is used for real-time automotive applications, where the difference to Low-speed CAN is most notably its increase in baud rate speed. For applications related to higher data throughput such as Advanced Driver Assistance Systems (ADAS), CAN Flexible Data-rate (FD) and CAN XL increase both payload sizes and baud rate speeds considerably in comparison to the common High-speed CAN.

2.3 The Double Precision Floating Point Format IEEE 754

Representing a wide variety of different vehicle signal data requires a data type with broad range for keeping precision at both the integer and decimal part of a number. Naturally, this infers the usage of some type of floating point implementation. When regarding historical implementations, these floating point systems have typically been developed by each computer manufacturer individually [10] which has lead to inconsistencies between different computer architectures. As a consequence IEEE introduced a standard in 1985 for representing floating point values as 32-bit single words (floats) and 64-bit double words (doubles) (see Figure 2.1) respectively [11], providing a common interface for floating point systems.

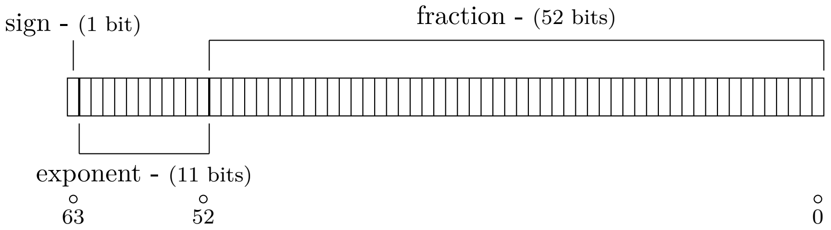


Figure 2.1: Bit sequence representing a double-precision floating point number after the IEEE 754 standard. The sign is represented by one bit, exponent 11 bits, and fraction 52 bits.

Representing a floating point number in bit form according to the IEEE 754 standard requires three different bit segments: the sign, exponent, and fraction (also called mantissa). Firstly the sign bit determines whether the number is signed (bit is 1) or not (bit is 0). Secondly the exponent segment represents an unsigned number in the range of 0 to 2047 in biased form, where a biased exponent means that the number is encoded using an offset-binary representation. In the case of doubles the offset is 1023 and the value represented by the exponent bit segment is calculated as:

$$2^{e-1023} \tag{2.1}$$

where e denotes the decimal value of the exponent bit segment. Finally the fraction bit segment determines the value after the decimal point and the double-precision number is in its entirety thus described as:

$$(-1)^{\text{sign}} \times 2^{e-1023} \times 1.\text{fraction} \tag{2.2}$$

A commonly known and notable problem with the binary representation of floating point numbers, is the inability to represent values exactly. Consequently comparisons of and arithmetic operations on floating point numbers suffer from rounding errors in their results which complicate methods that are not considering the bitwise representation. Utilizing a larger precision as with the double alleviates the problem to some degree as the binary representation in turn becomes closer to the real value.

2.4 Vehicle to Internet Technologies

Intrinsically as vehicles are partly mobile devices in the literal sense, access to the internet is achieved wirelessly. Two ways of achieving this connectivity are cellular networks and WiFi access points [12]. In the case of cellular networks, these connections utilize existing 3G and 4G LTE infrastructures using either brought-in or built-in options. The download and upload speeds for cellular networks is shown in Table 2.2.

Table 2.2: Maximum download and upload speeds for 3G, 4G LTE, and 5G cellular networks according to publicly available information [13], [14]. Notably 4G LTE has the potential to perform better than listed under optimal conditions, although as this is not the general case, only expected speeds are shown.

* 5G upload speeds vary by provider.

Network	Maximum Download Speed	Maximum Upload Speed
3G	1 Mbps	380 Kbps
4G LTE	5 - 100 Mbps	2 - 10 Mbps
5G	20Gbps	10Gbps*

Brought-in refers to the usage of for example smartphones to provide the connection and has the benefit of not requiring a pre-embedded infotainment system, meaning that this connection can later be more easily changed in the future [12]. Comparatively built-in options are embedded into the vehicle itself and has the potential to allow for stronger connections with a larger degree of control in terms of vehicle integration, at the cost of upgradability. Importantly, this issue of upgrading becomes highly relevant as the older technologies 2G and 3G get out-phased in favor of the modern alternatives 4G and 5G [15].

WiFi access points is a tested complement or alternative to cellular network infrastructures [16]. These serve as low cost hot-spots placed around vehicle populated areas such as highways and urban environments [12], allowing for intermittent connections while on the move. While indoor WiFi networks assume a stationary or slowly moving connected device, access points provide service to a vehicles with high mobility. The vehicle environment and behavior imposes a set of challenges however. Firstly, the time to authenticate and connect to an access point is not negligible and thus impacts the total communication possible over the time the vehicle is in the vicinity of an access point. Secondly, buildings and other fixtures in the area pose the risk of deteriorating the signal strength around corners. Finally, the inherent design of the WiFi protocol stack is not adapted to a high mobility environment.

2.5 Big Data and the Internet of Things

As a consequence of the growing demands in monitoring, analyzing, and controlling digital devices, the so called Internet of Things (IoT) has gained considerable popularity [17]. This concept establishes an infrastructure that enables device-to-device communication on a massive, previously unseen level and allows the sharing of real-time information through independent network actors. Aggregating and integrating multiple smaller independent data sources brings the possible benefit of deriving useful information through the usage of for example predictive models [18] that focus on improving real-life use cases. One specific use case for this data is the improvement of energy efficiency in data centers using machine learning, where such approaches have previously seen favorable outcomes [19], [20].

Related to the growth and expansion of IoT networks is the accumulation of large amounts of collected raw data. As such, it is necessary to discuss the relevant and recently arisen subject of Big Data [21] which regards the storage, analysis, and processing of massive, complexly structured datasets. Sensors as part of IoT networks for instance, have the potential to generate drastic amounts of data, some of which provide little or no useful information [22]. Filtering this data however, is a substantial challenge as it is difficult to ascertain what aspects of the data is, or rather could be, useful some time in the future. Data storage becomes an even larger long term issue due to to the sheer amount of data, while the shift from structured to unstructured data makes it unfeasible to store in traditional relational databases [23]. New storage and processing solutions have therefore been necessary to accommodate this shift.

2.5.1 Sensor Data as Time Series

Usage of sensors in complex digital systems introduces a massive data insight in terms of system behavior, where the sensor data production of a Boeing 787 for example, is greater than half a terabyte in a single flight [24]. Typically, sensor data can be represented as a time series, meaning a finite or unbounded sequence of values ordered by time [25]. This way of presenting data allows for analyzing systems during specific points in time, which is both valuable in terms of finding the source of a problem as well as preemptively predicting future problems.

2.5.2 Distributed File Storage Systems

Storage at the scale inferred by Big Data requires the utilization of system resources, properties, and access latency not feasible at the level of a single machine. Therefore, it is necessary to explore concepts such as Distributed File Storage Systems (DFSs). The main idea of a DFS is the usage of a distributed system of multiple machines for file storage, where service activity is spread across the network on multiple independent storage devices [26]. One fundamental property of a DFS is the ability for clients to access files using the same set of file operations applicable to local files. This property is referred to as network transparency and implies that remote files are (to the client interface) indistinguishable from local files. A significant benefit of structuring a system in this non-centralized way is the potential of increased fault-tolerance, meaning that if one machine goes down for instance, the service should still provide its functionality. Additionally, the use of proper file replication on the DFS infers that such an event does not necessarily lead to any data loss.

Throughout the years a number of relevant open source DFS implementations have been presented, where one of these is the Google File System (GFS) [27]. Introduced in 2003 by Google, the GFS sought out to provide a solution that was designed after their needs of large scale data storage. An initial property of the system was the capability of being run on inexpensive commodity hardware, which the authors saw lead to an observed frequency of component failures. Consequently, the adoption of continuous system monitoring, error handling, and automatic recovery was therefore a primary feature of the system proposed. Further design aspects of the GFS were heavily affected by the needs and use cases of Google, such as content aggregation to fewer large GB files rather than a significant number of smaller KB files.

Apache Hadoop [28], [29] is a collection which provides a framework for storage and operations on Big Data, where one of the features of this collection is the Hadoop Distributed File System (HDFS). Importantly, Hadoop has seen wide usage by a number of organizations over the years, where in 2010 it was used by Yahoo! to store 25 petabytes of application data over 25 000 commodity servers [28]. Another benefit of utilizing the HDFS is the close integration with the Hadoop framework which allows for performing computations in parallel closely to the stored data. The HDFS architecture is designed (similarly to other DFSs) using one NameNode which stores and manages file metadata, along with one or multiple DataNodes that store file blocks independently replicated at multiple nodes. These file blocks represent split content, usually into 128 MB each. Applications interact with the HDFS using the HDFS Client which exposes the file system interface using a code library.

2.5.3 Relational and Non-relational Databases

Traditionally data is, and often has, been stored using so called Relational Databases. These excel in the rapid retrieval and storage of structured data [30], making them useful in many applications where it is straightforward to predict the appearance of the database. Additionally, this structured predictability allows for operational features trough the use of query languages such as SQL. There are some drawbacks to managing data in a relational manner however. For instance, the usage of SQL introduces significant complexity when storing unstructured data and many of the features introduced by relational databases are not applicable. Another issue is the incompatibility with high scalability, meaning that in order to support scaling beyond a certain point, distributed databases are required.

In order to meet the demands of large scale data, distributed Non-Relational Databases are a promising alternative [31]. The main advantage of this type of database is the ability to efficiently scale in order to accommodate larger data sizes, as the system is inherently distributed. One example of a non-relational database is Google's Bigtable [32] which is targeted towards massive amounts of structured data, while ergonomically allowing for both up- and downscaling of the distributed cluster.

Another example is Apache HBase (Hadoop database) [33]; based on Bigtable and closely integrated with the Hadoop framework.

2.6 General Purpose Compression

Most often when a chunk of data is compressed on a computer system, general purpose compression is used. These algorithms are tailored to suit a vast number of different types of data by utilizing universal data encodings. Many of the lossless algorithms used today are based on LZ77 [34] and LZ78 [35], proposed by Abraham Lempel and Jacob Ziv in 1977 and 1978 respectively. LZ77 and LZ78 are so called dictionary coders, meaning that they maintain a structure of commonly referenced set of strings (a dictionary) and substitute occurrences of these strings in the data to be encoded with the string's position in the dictionary.

Today, ideas from the Lempel-Ziv algorithms can be found as part of many other ubiquitous algorithms. The GNU Gzip compression utility [36] bundled alongside GNU/Linux for instance, uses the DEFLATE [37] algorithm which is based on LZ77. Some other LZ77 based algorithms include Brotli [38] which is an alternative successor to gzip, LZ4 [39] and Snappy [40] which target compression and decompression speed, and Zstd [41], [42] which targets compression ratio and decompression speed. A prominent adaptation of LZ77 is LZMA (Lempel-Ziv-Markov-Chain) developed by Igor Pavlov for the 7-Zip archiver [43], later used in the XZ-Utils compressors [44]

2.7 Lossless Floating Point Compression

In the fields of scientific computation and IoT a considerable amount of floating point data gets exchanged between nodes. Consequently, compression methods targeted towards floating point values specifically have been developed in recent time, with the goal of being sufficiently fast as not to impact data throughput. One early such compressor is FPC [45], which was developed with exactly this intention.

More specifically, algorithms targeting high data throughput are sometimes described as 'streaming-based' and are mostly useful for time series data by working at the bit level. A prominent streaming based algorithm proposed by Facebook in 2015 is Gorilla (explained in Section 2.7.1), which was developed alongside their time series management system with the same name [46]. By leveraging XOR operations, Gorilla can utilize the directly previous value to maintain information about the following by observing the leading and trailing zeroes in the XOR result. Seven years after its proposal, researchers at the Athens University of Economics and Business identified what they believed to be a flaw with Gorilla, and based on this, proposed CHIMP [47]. The following year in 2023 the so called Elf algorithm introduced a method of erasing the last few bits of the floating point value, which was thought to maintain trailing zeroes in a better way [48]. In 2024 another approach was considered in the Camel algorithm, where the authors made the choice to separate the integer and decimal part of the floating point value and compress these separately [49].

2.7.1 The Gorilla Algorithms

Gorilla proposes two algorithms: one for compressing time stamps and another for floating point values [46].

Time Series Compression

The algorithm utilizes delta-of-delta encoding and is comprised of the following steps:

1. Denote the starting time stamp (without value) t_{-1} as the header and store it using 64 bits. Store the first time stamp t_0 as the delta from t_{-1} in 14 bits.
2. For the following time stamps, t_n calculate the delta of delta:

   $D = (t_n - t_{n-1}) - (t_{n-1} - t_{n-2})$

   - $D = 0 \rightarrow$ store a single '0' bit
   - $D \in [-63, 64] \rightarrow$ store a '10' followed by the value (7 bits)
   - $D \in [-255, 256] \rightarrow$ store a '110' followed by the value (9 bits)
   - $D \in [-2047, 2048] \rightarrow$ store a '1110' followed by the value (12 bits)
   - Else, store '1111' followed by D using 32 bits

Floating Point Compression

The algorithm utilizes a variable length encoding scheme:

1. Store the first value without compression
2. If XOR with the previous value is zero (same value), store single '0' bit
3. Else, calculate the number of leading and trailing zeroes in the XOR, store bit '1' followed by either a) or b):
   (a) (Control bit '0') If there are at least as many leading zeroes and trailing zeroes as with the previous XORed value, the block position is inferred and just store the meaningful XORed value (center bits).
   (b) (Control bit '1') Store the length of the number of leading zeroes in the next 5 bits, then store the length of the meaningful XORed value (center bits) in the next 6 bits. Finally store the meaningful bits in the XORed value.

2.8 Message Brokers

To facilitate efficient communication between different applications, a Message Broker is often introduced. A message broker (also Integration Broker), is an intermediary that allows for sending messages program to program [50]. Importantly, message brokers decouple processes and services within systems while ensuring that message ordering is exactly equal to order of transmission [51]. In contrast to REST APIs, asynchronous communication is possible, meaning that senders do not have to wait for the receiving service's reply to continue operating. Common alternatives of message brokers include ActiveMQ Artemis [52], Apache Kafka [53], Redis [54], and RabbitMQ [55].

3

Problem

The Large Scale Vehicle Data Readout problem regards the collection, transmission, and reception of CAN bus data signals as found in trucks during their operational usage. An appropriate model for viewing this communicative relationship is the Shannon-Weaver model which details the basic components in a communication chain [56]. The model is shown in Figure 3.1 and describes the process of sending a message over a communication channel; from a source to a destination.

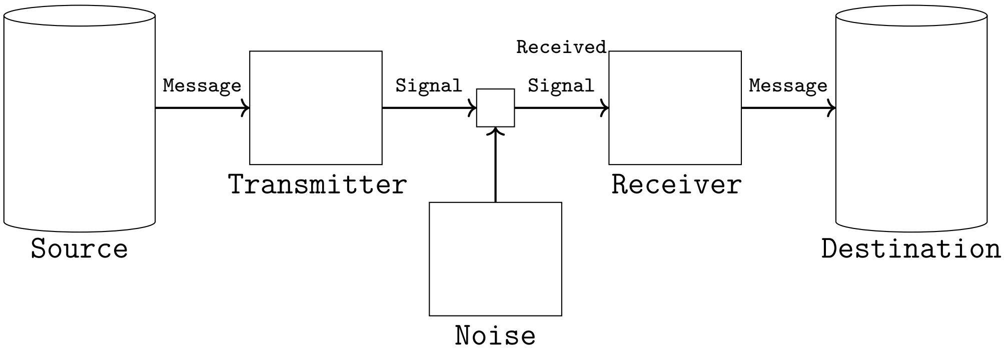


Figure 3.1: Shannon-Weaver model showing how a `Message` gets sent from `Source` using a `Transmitter` as a `Signal`. In the process, `Noise` is introduced before the signal reaches the `Receiver`. The `Receiver` then takes the `Message` and transfers it to the `Destination`.

A major hinder to data collection in vehicles is the lack of sufficient storage to hold larger amounts of data, meaning that signal values collected over time will eventually have to be evicted. Consequently, this presents an inability to store larger amounts of useful historical data as the number of signal dimensions range in the thousands. This leads to an approximately linear scaling in terms of storage. When observing the channel, one must further take into account the limited bandwidth during global travels as conditions might not always be optimal for transferring data. The receiving end additionally faces obstacles due to the large number of vehicles inside a fleet, numbering in the hundreds of thousands. Based on this model, the problem is divided into two distinct sub problems denoted as the Producer Problem and the Receiver Problem.

3.1 The Producer Problem

A single vehicle has the potential to produce a considerable amount of data, generated at regular intervals (samples). The vehicle however, may not possess the necessary storage space to maintain a record of this data over time. Likewise, its network upload speed may fail to meet these samples in real time. Fundamentally this can be regarded as an optimization problem where the maximum amount of data points with a sampling rate of ω Hz and varying dimensionality is achieved based on as few bytes as possible. An initial model of this problem from a storage perspective may be expressed as the score function:

$$
\begin{aligned}
& s(t_\omega, n, b) = \frac{t_\omega \times n}{b} \\
& \text{where:} \\
& \quad t_\omega = \text{number of time steps sampled by}\, \omega \\
& \quad n = \text{number of data attributes} \\
& \quad b = \text{storage required in bytes} \\
& \text{maximize:} \quad t_\omega, n \\
& \text{minimize:} \quad b \\
& \text{subject to:} \\
& \quad 0 < b \leq B,\ B = \text{maximum bytes available}
\end{aligned}
$$

- t_ω and n may be expressed as given input arguments that are to be maximized.
- b corresponds to a function $b(\mathbf{A_f})$ returning the size of bytes required for $\mathbf{A_f}$, where $\mathbf{A_f} \in \mathbb{R}^{n \times t_\omega}$, and corresponds to the n-dimension time series feature set $\mathbf{f}$.

Hence it is necessary to find a function b which gives a sufficiently optimally reduced size of $\mathbf{A_f}$ in order to reach a solution. The arguably most straight-forward approach is to consider b a compression function, which is appropriated for the appearance of the time series found in $\mathbf{f}$. In that case, it is possible to view b in relation to the entropy rate H as proposed by Claude E. Shannon [56] which expresses the fundamental limit at which lossless compression is possible. In information theory, entropy quantifies the amount information associated with the potential state of a variable, where entropy rate is the average entropy change over a sequence of random variables. This imposes an implicit constraint on b as it is mathematically impossible for b to achieve a compression rate of $\mathbf{A_f}$ greater than H.

3.1.1 The computability of b

Another perspective which must be considered when considering compression functions is the strain on the system on which it is being run. While b may be subject to the constraint of fitting in available memory, it also incurs another cost in the form of compute which should not be neglected. Consider a situation where the computation of b infers a relatively high execution time. As a consequence, the intermediate storage may get filled meaning that either new data gets discarded or old data overwritten. In both cases the total sum of data collected over time is lessened, which is not indicated by the current score function s.

An alternative to solving this issue is to consider the worst-case execution time (WCET) of the compression method on the specific hardware, as this would allow ascertaining whether the function has an acceptable complexity. The performance scaling in terms of dataset size is another factor which is to be considered as the compression method might fare well for smaller collections, while drastically slowing down for others. It is then necessary to evaluate this in relation to ω in order to determine if the method is able to compress in time without risking data eviction.

3.2 The Receiver Problem

Consider an aggregate of vehicle fleets numbering in the hundreds of thousands where each member vehicle possesses some data representing a history of its state, S. The aim is for the state to be potentially unloaded at any point in time, at any place in the world. This establishes the need for some form of endpoint with the capacity to receive, decompress, and store the data such that it can be meaningfully represented and retrieved. Such relationship can be observed in Figure 3.2, where a receiver endpoint R sees a number of incoming data streams limited by separate bandwidths.

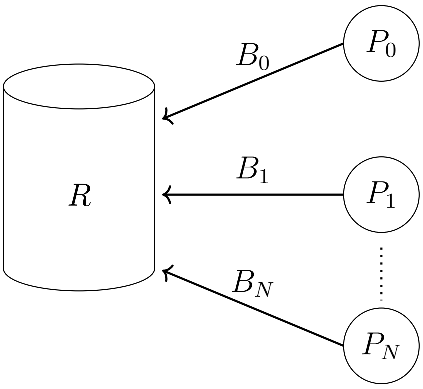


Figure 3.2: Communication topology representing receiver R handling incoming data streams from a number of publishers P_0, P_1, ... , P_N with respective bandwidths B_0, B_2, ... , B_N.

Notably, R symbolizes a graphical streaming entryway and may be comprised of an internal hierarchy consisting of multiple nodes constituting the receiver. R can as such in reality be made up, of for example, a distributed network, rather than existing solely at one point.

The general appearance of the problem is to receive and copy a recorded set of historical states (see Figure 3.3) from a large number of publishers with the constraint that:

$$S' \simeq S \tag{3.1}$$

Essentially meaning that the data should be equivalently preserved. This is especially relevant since the data recorded in S has a certain temporal precision, whereas it may be more favorable to utilize another for S'.

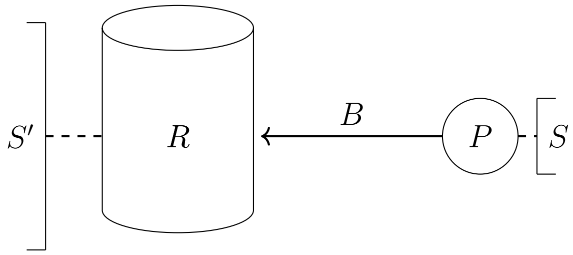


Figure 3.3: Single communication process between receiver R and publisher P where P is streaming its state S to R with a bandwidth of B which stores it as state S'.

4
Methods

This chapter presents the proposed architecture and modeling of a solution to the problem. In order to realistically analyze the solution a simulation model was established based on the presented problems as discussed in Sections 3.1 and 3.2. This model is likewise distinguished as two separate key areas on a micro contra macro scale, where micro refers to the vehicle's internal computations in relation to itself and macro to the external receiver in relation to a vehicle fleet.

4.1 Producer Model

Two main components were identified in the construction of a sufficiently realistic simulation of the vehicle systems: a signal data stream source and a client to receive it (see Figure 4.1).

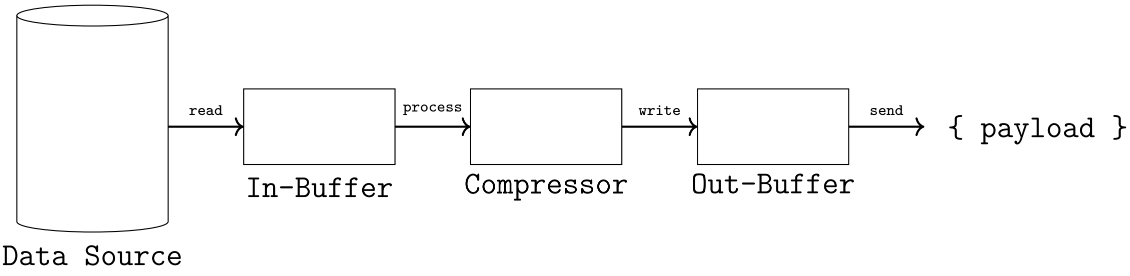


Figure 4.1: Data pipeline from source to output. Initially data is read to `In-Buffer` from `Data Source` and is then processed to the `Compressor`. After compression the data is written to the `Out-Buffer` which formats and encodes the bytes using Base64, sending it as a payload.

4.1.1 Simulating the Data Stream

Accessing and connecting to a real vehicle system was not feasible in relation to the amount time which would have to be allocated for testing purposes. Therefore the data source of the vehicle system had to be simulated. In reality, vehicle data is most often sent by using the CAN bus data protocol, of which there are methods of simulating, using for example CANdevStudio [57]). However, utilizing such a simulation was believed to introduce further complexity in aspects such as bus arbitration and error handling. Different configurations of CAN would have a possible effect on the outcome, although this would depend on manufacturer specifics and is not easily mocked in software. Consequently the concept of the data source was reduced to be viewed as a stream of bytes from pre-collected data files. The abstraction allowed for the utilization of the more generally compatible TCP streaming protocol, meaning that the simulation could involve low-latency remote hosts. The possibility of utilizing remote hosts was especially useful in terms of data access, as the underlying total data source size was too large to feasibly copy to the client machine. An effect of this abstraction was that the in-buffer component of the model operates under ideal conditions and has no bottlenecks in terms of data input, whereas in reality this would not always be the case.

Using TCP, the data simulation stream is performed as follows:

1. Data from a single vehicle's trip is read from a tabular data file on the remote host. Columns represent attribute value sequences and rows are the regularly sampled attribute values at each time step of the trip.

2. Data values are formatted in an array of doubles according to Figure 4.2 and packed as bytes.

3. Sequential chunks of the array are sent with an interval ω Hz equal to the inversion of the time stamp delta. Each chunk corresponds to a time period of size n-attributes × 8B.

Notably, the efficacy of this scheme is dependent on the bandwidth between the remote host and client being equal or higher than the chunk size total sent over time. Assuming a stable connection this dependency was not believed to be a problem, as per the stated requirements (see Section 1.3) this bandwidth would have to be at least 3.2 MB/s; a non-significant number on most internal networks.

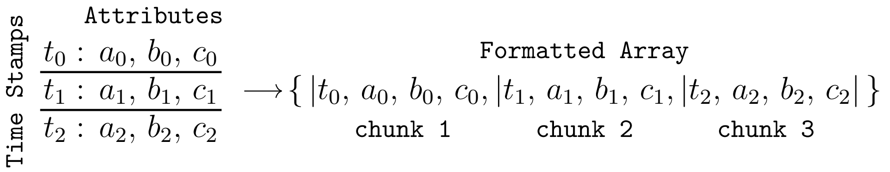


Figure 4.2: Data table consisting of time stamped (t) attribute values (a, b, c) being converted to a formatted array. The array is later read and sent as chunks where the first value of each chunk corresponds to a time stamp followed by an ordered sequence of attribute values.

4.1.2 Client

The client was implemented in C++17, allowing for memory control and embedded compatibility, while providing interfaces for commonly used functionalities. Designing an appropriate structure for the client involved the following considerations:

- Incoming data needs to be buffered as to minimize computational occupancy of the processor over time.

- Compression occurs after a set time frame as to decrease instances of data preprocessing required.
- Outgoing compressed data gets sent after some conditions (such as storage size) are met as to minimize network traffic over time.
- Data is stored exclusively in memory to maximize access speeds.

Initially the process starts its main routine with a blocking `unistd` read from an incoming data stream to the in-buffer, only stopping after the buffer is full. Pessimistically, the input buffer is only allocated the minimum space required to store an interval of data within the set time frame:

$$\texttt{in\_buffer\_size} = \omega \times w_s \times n \times sizeof(double) \tag{4.1}$$

where ω is the sampling rate, w_s the time frame in seconds (e.g. window size), and n the number of attributes. This decision was based on data recency, meaning that more recent incoming data should be prioritized in terms of storage. Data should in other words be stored in its original size for the shortest time possible as to not occupy system memory for longer than necessary. Another notable choice that was made in terms of buffering, is reading the incoming data as unsigned 64-bit integers (bits unchanged) rather than double precision floating point values. Consequently costly floating point operations are avoided, since only the bitwise representations were interesting for the purpose of the selected streaming-based compression methods.

After the in-buffer has been filled, the time steps and attributes are copied and transposed as individual arrays corresponding to the column vectors in Figure 4.2. These arrays are then mapped using indices based on the attribute ordering (see Figure 4.3), i.e. $t \to 0$, $a \to 1$, $b \to 2$, and $c \to 3$. Functionally, these attributes are seen as independent sequences and are therefore compressed individually after the set time frame. This way of storing sequences can essentially be seen as double-buffering, where an additional buffer of equal size is kept for the data to be processed (i.e. compressed). As such writes to the in-buffer can persist while data is being processed. This design essentially makes the in-buffer use double buffering, which means that twice the amount of memory is required since copies are made. One part of the in-buffer waits for new data while another is used for compression. Hypothectically this was thought to improve performance for buffer sizes considerably larger than cache, as keeping separate arrays would exploit spatial locality.

Key	Value
0:	$\{\, t_0, t_1, t_2 \,\}$
1:	$\{\, a_0, a_1, a_2 \,\}$
2:	$\{\, b_0, b_1, b_2 \,\}$
3:	$\{\, c_0, c_1, c_2 \,\}$

Figure 4.3: Key-value attribute map from indices to arrays of time series values, where t_i are time stamps and a_i, b_i, and c_i are attribute values.

Compression is performed by consuming the mapped array values and, for each array, writing to a corresponding output bit sequence (the output buffer) according to some predetermined scheme. This bit sequence is represented by a data structure containing a standard library vector of unsigned bytes, which allowed for dynamic resizing with negligible overhead. Initially the time stamps (t) are compressed using the specific time stamp compression scheme proposed in Gorilla [46]. Thereafter the floating point compression scheme can be selected from two alternatives: Gorilla and CHIMP [47].

The final proposed stage of the client was to stream the output buffer to a Kafka broker using the high-performance `librdkafka` producer API [58]. Two main conditions for emptying the output buffer were identified: reaching a predetermined memory threshold and achieving a sufficient cellular connection strength. More specifically the compressed data rate must be lower than or equal to the network bandwidth rate on average over time, without exceeding a set memory threshold for the output buffer. Consider a varying network bandwidth N (B/s), compressed data rate R (B/s), compression constant C, and a memory threshold of T (B). T is predetermined based on a fraction of the available memory that can be allocated to the process. R is calculated based on the compressed `in_buffer_size` (see Equation 4.1) divided by the set window size w_s:

$$R = \frac{C \times \texttt{in\_buffer\_size}}{w_s} \text{ B/s} \tag{4.2}$$

N is externally dependent on the connection strength and thus the output buffer must be emptied such that when the stored data size approaches T, N is on average significantly higher than R.

4.1.3 Selection of the Compression Algorithms

Establishing a suitable compression method was found to require evaluating alternatives in terms of a set number of criteria. Firstly, the method must be able to compress double precision floating point values efficiently as this is the broadest representation of data in numeric form. Secondly, due to the unpredictability in the importance of decimals the method must be able to achieve lossless compression. While the data may be later chosen by the end receiver to be stored at a loss, its initial capture should accurately reflect vehicle state. Finally, execution of the algorithm should be sufficiently scalable such that it is able to timely compress values in relation to a sample rate ω (Hz) with $\omega \in [1, 100]$ for 4000 attributes at each time step.

Initially one may approach the problem with a general purpose compression algorithm capable of lossless compression of floating point data. A common subset of these includes Xz, Brotli, LZ4, Zstd, and Snappy which all provide favorable compression ratios with regards to floating point time series data [47][49]. Although, another aspect to consider beyond compression ratio, is the cost in terms of computation time required to execute the algorithm. This is an area where most general purpose algorithms are at a disadvantage in comparison to more specialized streaming-based algorithms, where Xz for instance is roughly 40x slower on average in contrast to some streaming based approaches [47]. Thus the choice was made to disregard evaluating general purpose alternatives for the purpose of this solution, as their hypothetical execution time and impact on the system load were deemed too great.

Based on the given criteria for a compression algorithm, streaming-based alternatives were seen as suitable as of their adaptability to persistent compression over time. The Gorilla floating point compression algorithm [46] is one such algorithm proposed by Facebook for usage in a time series database. Consequently, the proposed compression method has seen wide usage in multiple other database alternatives [47] such as InfluxDB [59], IotDB [60], Prometheus [61], TimeScaleDB [62], and M3 [63]. This usage was seen to demonstrate the compression scheme's capacity in handling time series floating point data, making it the first algorithm used in the compressor. Finally, Gorilla also proposes an efficient encoding for time stamps which was selected and used separately in addition to the floating point compression scheme.

Other state of the art streaming-based alternatives were also considered in terms of their performance in compressing data similar to that of vehicles. A later method built on a similar idea to that of Gorilla is CHIMP [47], where the authors describe it as ”outperforming all earlier techniques with regards to both compression and access time” at the time of writing. As a consequence of these results, the CHIMP encoding scheme was selected as the second alternative to be compared alongside Gorilla. Camel [49] was also an even more recent algorithm promising favorable results, compressing the integer and decimal part separately. However, initial testing revealed that these additional operations increased compression time substantially with lesser compression ratios in comparison to Gorilla and CHIMP for the tested vehicle data. As such the Camel algorithm was disregarded in terms of being an alternative for the compressor. This lead to the decision to only consider Gorilla and CHIMP as these were considered similar and sufficient in their performance contra compression.

4.1.4 Compressed Byte Encoding

At some point after compression, the data had to be transferred from the vehicle and into some form of more persistent storage. However, seeing as the data communication payload had the form of a JSON string, a problem arose in regards to how to represent the compressed sequence of values. Naively the bytes could be packed as integers and stored as such, although this leads to problems in cases where the number of bytes is not divisible by 4. Additionally it is difficult to guarantee datatype consistency between all stages in the pipeline as a JSON string does not inherently allow for datatype specification. Accordingly the idea of representing the bytes as single byte UTF-8 characters arose.

Directly encoding unsigned byte values is largely a trivial operation in the case where the value falls within $[0, 127]$, as these can directly be translated into traditional ASCII symbols representable by a single byte. As compressed byte values correspond to arbitrary bit sequences however, the value falls within $[0, 255]$. Consequently a different encoding to UTF-8 had to be considered. To solve this, the binary-to-text encoding scheme Base64 [64], [65] was introduced as a compatibility layer for the bit sequences. A caveat to using Base64 is the associated overhead, as on average, four characters are required to represent three bytes. This adds an additional storage of roughly 33%, which is arguably costly. An alternative is to utilize a different encoding such as basE91 [66] that reduces the maximum storage overhead from Base64 to 23%, and as such the equivalently use this format as well.

4.2 Receiver Model

By observing the problem scenario proposed in Section 3.2 a simulation model was formulated at the macro scale constrained by the requirements given in Section 1.3, as can be seen in Figure 4.4. The main concern of this model is the ability to handle a very large number of data streams (1M) concurrently across the globe, where each stream originates from a mobile vehicle producer.

Figure 4.4: Macro model represented by signal streams coming to a long term storage database

A rough calculation of the gross required total bandwidth can be made by establishing the data production bandwidth for one vehicle (B_i) and then scaling it to the entire vehicle fleet (B). Initially, no other properties than the raw data storage was considered and thus B_i is given by taking the product of the number of signals $n_{\max}$, the maximum sampling rate $\omega_{\max}$, and the number of bytes required to store a signal:

$$B_i = n_{\max} \times \omega_{\max} \times sizeof(double) \tag{4.3}$$

The factors to this calculation were derived from the previously established requirements and are thus: $n_{\max} = 4000$, $\omega_{\max} = 100\text{Hz}$, and $sizeof(double) = 8\text{B}$. This resulted in an initial $B_i = 3.2$ MB/s and B equal to the sum over these:

$$B = \sum_{i=0}^{N} B_i \tag{4.4}$$

where $N = 10^6$, being the number of total trucks in the vehicle fleet. Only considering raw data, this amounted to a significant $B = 3.2$ TB/s. It must however be noted that this number is drastically exaggerated as it does not take into account more complex real life factors such as activity distribution throughout time zones and continents. Additionally, seeing as vehicles are utilizing a mobile connection, they are restricted in terms of this bandwidth as well, meaning that a low quality connection will directly imply a smaller load for the receiving end. Nevertheless this number was used as a "worst-case" situation guideline, even though all of this data likely will not reach the receiver.

Naturally, the derived B is for most systems not a feasible amount of data to process efficiently, and as such further measures were necessary to approach the assumed problem size. An intuitive solution is to introduce compression at the producer level (discussed in Section 4.1.3), which limits the amount of information to be transferred initially and thus the bandwidth required to contain this. Another approach is to consider the public architectures of established streaming platforms such as Netflix [67]. A notable recurring feature in the streaming architecture of Netflix is the usage of event message brokers to store, read, and analyze streaming data efficiently. The utilization of such brokers arises from the inability to process events on arrival, instead postponing the consumption of messages until a later point in time. Introducing a message broker to the model was thought to increase the concurrent data throughput without overloading requests to the database

As a consequence of the high data bandwidth sent to the receiver, the persistent long-term storage of high volume data was believed to be a relevant topic of consideration. In Big Data and IoT, the solution to storing such high volumes is typically to utilize an efficient distributed storage mechanism [23], as opposed to traditionally relational databases. This solution was found to apply in the case of large scale vehicle data readout as well and as such a distributed storage file system was examined. For this purpose there were a number of open source alternatives available, such as the Google File System (GFS) [27], the Hadoop Distributed File System (HDFS) [29], and Terrastore [68], among others. HDFS was investigated as file system for the receiver due to its integration with the well regarded data processing framework Apache Hadoop [29], [69], [70]. Furthermore, it has historically seen high efficacy in previous data storage use cases of large volume data [28].

4.2.1 Selection of the Database and Message Broker

While the distributed file system allows for efficient storage, an additional layer in the form of a non-relational database was seen as necessary. The need arose mainly from the requirement of serving the data in an accessible way, as simply querying the file system provided limited control and structure. One commonly used non-relational database for large data volumes is BigTable [32] developed at Google. Some other choices that were considered include HyperTable [71] (implementing BigTable) and MongoDB [72]. Although, considering that the HDFS was chosen as underlying architecture, an inherent alternative was that of Apache HBase [33] as it runs natively on the HDFS. Based on BigTable, HBase supports the storage of columns in the millions and rows in the billions on top of commodity hardware and has seen usage as a persistent storage database in similar situations such as in the Gorilla system [46].

The most important property when evaluating an alternative for the message broker was seen to be the message throughput, as the data would likely not be needed closely in time after its production and low latency was therefore a secondary concern. Multiple alternatives to serve as message broker were considered, where some of the most popular alternatives include: ActiveMQ Artemis [52], Apache Kafka [53], Redis [54], and RabbitMQ [55]. Based on performance benchmarkings of these [73], [74], Apache Kafka was believed to provide the highest throughput while still delivering acceptable latencies and chosen as message broker for the proposed architecture.

4.2.2 Receiver Architecture

The design of the proposed receiver architecture is shown in Figure 4.5 and set out to solve two main issues: high frequency incoming data throughput and long term data permanence. Initially a payload is produced containing a JSON string of meta-data and a compressed sequence of time series data from a single attribute. This payload constitutes a Kafka message which gets sent to a predetermined topic of a Kafka cluster of brokers. Notably, only a single attribute value sequence gets sent as to not exceed the default 1 MB acceptable message size limit of Kafka brokers [75]. Keeping the message within this limit consequently increases data throughput by reducing network congestion and preserving system resources.

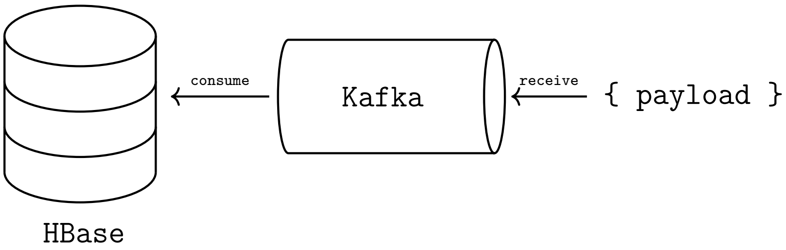


Figure 4.5: Receiver architecture diagram showing a produced payload getting received by an Apache Kafka broker, after which it is consumed into an Apache HBase. The payload contains a message consisting of a value sequence generated by a producer. `Kafka` and `HBase` represent distributed network systems of the associated services.

The main reason why Kafka brokers were introduced into the architecture, is that they decouple direct client communication with the final database endpoint. Thus concurrent ingestion is reduced while database writes are more restrictive, meaning that faulty or corrupt data can be discarded without being written to the database. Another benefit is that the client becomes agnostic of the database architecture and can asynchronously send data to a broker without dependencies on other APIs than Kafka.

After the message has been received by the cluster, it gets temporarily stored under the topic until the moment of consumption. For this purpose, a Kafka consumer process runs alongside the HBase which periodically consumes all received messages under a given topic and ingests them in the HBase long term. It was found that the consumer process could be implemented in a number of ways, where one of the hypothetically most prominent options for large scale traffic is Kafka Connect [76]. The main benefits of using the Kafka Connect API include established fault tolerance, robustness, horizontal scalability, and integrability with other components [77].

On the other hand, the Kafka API allows for implementing a consumer process in a number of languages such as Python [78], C++ [58], and Java [79] which provide a larger degree of flexibility in terms of implementation. The problem with making a custom implementation of the consumer process however, was seen to be the necessity of providing fault handling and scaling guarantees from scratch rather than relying on a predefined established framework.

An important aspect of Kafka producers and consumers are their configurability. This is especially relevant when considering the volume of messages being sent by a single producer per compression window, as these are number in the thousands. In these cases, optimization options such as batching are available to avoid significant traffic issues while theoretically increasing throughput.

4.2.3 Payload Structure and Storage

Sending data in a general and serializable format provides basis for agnostic compatibility with receiving processes in the data pipeline. Consequently a proposal was made to represent the data payload as a JSON string (see Figure 4.6) since this allows for the inclusion of key/value pair metadata. Firstly, the `identifier` field consists (in this example) of two segments: a unique denomination (`vehicle_1`) followed by a sequential label (`sequence_1`). This structure highlights which specific vehicle the data belongs to as well as the data's absolute ordering in relation to other data sequences for the same vehicle.

```
        {
            "identifier": "vehicle_1-sequence_1",
            "attribute": "speed",
            "value": <Base64 encoded byte sequence>,
            "bits": 164,
        }
```

Figure 4.6: JSON message structure for a produced payload including data for the fields `identifier`, `attribute`, `value`, and `bits`.

Secondly the `attribute` field specifies which attribute the data belongs to, in this case `speed`. Thirdly the `value` field contains a compressed byte sequence; Base64 encoded to be representable by a string with minimal storage overhead. Finally the `bits` field describes the number of bits which are written in the byte sequence, since this is required by the decompressor in order to disregard superfluous bits in the last byte. For this example the payload would contain a Base64 encoded sequence of 21 bytes ($\left\lceil \frac{164}{8} \right\rceil = 21$), meaning that of the 168 bits stored, four are superfluous.

The payload's metadata structure allowed for straightforwardly storing the data inside the HBase. After consumption into the database, the value sequence is decompressed using the bit information to its original state and then recompressed. The reason for this is that the receiving endpoint does not have the same compute and memory constraints as the vehicle, and can therefore utilize a general purpose compression algorithm to achieve higher compression ratios at the cost of system load. One such alternative is Gizp which is natively supported by HBase, although the performance is dependent on the configuration decided by the user and will as such not be evaluated in the scope of this thesis.

5

Evaluation

Preceding testing and benchmarking of a solution to the aim established in Section 1.1, a number of research questions were formulated as to establish an appropriate evaluation in terms of metrics. These questions are:

> *RQ1: How can vehicle data be compressed efficiently?*
>
> *RQ2: What is the imposed CPU and memory load on the vehicle system in order to accommodate data readout?*
>
> *RQ3: What is the total data communication in terms of bandwidth required to accomodate data readout?*
>
> *RQ4: How can data from vehicles be sent and processed efficiently?*

5.1 Experimental Setting

In order to sufficiently evaluate the efficiency and performance of the proposed methodologies and present answers to the formulated questions, a number of different system configurations were required. Primarily for the client process described in Section 4.1, an embedded evaluation board (also referred to as *host*) was used in order to more closely resemble the computational environment as seen in real vehicle systems. The system specifications of this board are described in Figure 5.1. After consultation with domain experts, this was seen as representative of hardware that might be found in a technologically advanced transport vehicle.

Client Evaluation Board

- **Operating System**: 64-bit Embedded GNU/Linux
- **Processor**: ARM Cortex A72
  - **Frequency**: 1.9GHz
  - **Cores**: 8, 1 thread per core
  - **L1 Data Cache**: 256KiB total, 32KiB per core
  - **L1 Instruction Cache**: 384KiB total, 48KiB per core
  - **L2 Cache**: 4MiB (over 2 instances)
- **Memory**: 32GB (4x8GB) LPDDR4 RAM @ 4266MHz
- **Network**: 1Gbps LAN
- **Storage**: 16GB Flash MicroSD

Figure 5.1: Configuration for the client host running on an embedded board. Specifications shown include Operating System, Processor, Memory, Network, and Storage.

Secondarily, all other systems were containerized and running on an on-site cluster. One system was functionally responsible for reading vehicle data on the cluster while simulating a data stream to the board running the client process. Using `iPerf` [80], the bandwidth between the host and cluster was measured to be roughly 850Mbit/s over LAN. The allocated data stream system specifications/resources are described in Figure 5.2.

Data Stream System

- **Operating System**: 64-bit Debian GNU/Linux 12
- **Processor**: AMD EPYC 7642 @ 2.3GHz (4 cores reserved)
- **Memory**: 16GB
- **Storage Method**: SSD

Figure 5.2: Configuration for the system streaming data to the client host. Specifications include Operating System, Processor, Memory, and Storage Method.

The allocated system specifications/resources for the Kafka broker/controller are described in Figure 5.3.

Message Broker System

- **Docker Image**: `apache/kafka:4.0.0-rc0` [81]
- **Processor**: Intel(R) Xeon(R) Gold 6442Y @ 3.25GHz (16 cores reserved)
- **Memory**: 32GB
- **Storage Method**: SSD

Figure 5.3: Configuration for the Kafka broker/controller receiving messages from the client host. Specifications include the Docker Image, Processor, Memory, and Storage Method.

For benchmarking and testing purposes, a number of datasets were used separately as source for the data stream. Primarily, the proprietary *Aevitas* data collection owned by Volvo Group Truck Technologies was used. This collection includes signal data collected during a vast number of trips for different vehicle types such as trucks and buses, establishing a varied data basis.

In contrast to the Aevitas collection the *Zenseact Open Dataset* (ZOD) [82] was utilized to provide an open source reference point. Developed for the purpose of large multi-modal autonomous driving, this dataset includes attributes related to cars specifically. Furthermore the dataset is described as having ”the highest range and resolution sensors among comparable datasets”, inferring that it should provide some meaningful precision in terms of benchmarking floating point compression. While the main focus of the Zenseact Open Dataset is on autonomous driving, meaning that it for example includes data collected by lidar, only the parts labeled ”vehicle data” contained in this dataset was seen as applicable to the aim of this project.

A meaningful aspect to consider when evaluating these datasets is that they utilize different sampling rates (Aevitas 10Hz, ZOD ∼100Hz), and as such have different levels of granularity in terms of the delta between two subsequent time step values. It is therefore necessary to establish that comparisons between these are made in terms of data sizes and number of data values.

5.1.1 Data Preprocessing

For the purpose of establishing a common format at the byte scale each dataset was preprocessed before being streamed to the host client. Initially, the temporal precision of each dataset was set to be in terms of milliseconds as no sampling rate of any dataset exceeded 100Hz. This additionally saw the benefit of allowing for higher compression ratios in terms of timestamp compression, as representing the delta of two timestamps at the scale of milliseconds requires considerably less bits than the scale of nanoseconds for the perceived sampling rates. Beyond altering timestamps, no further preprocessing was performed. The reason for this was to preserve the original appearance of the signal data to allow for a fair evaluation with respect to real conditions.

5.1.2 The Aevitas Collection

The aim in the compositon of the Volvo proprietary Aevitas dataset was to establish a diverse set of vehicle types and trips. Essentially, the dataset consists of five vehicle types where each type has 100 corresponding randomly sampled trips of different lengths. Each trip contains a finite set of attributes sampled with a rate of 10Hz. The vehicle types and their associated characteristics can be seen in Table 5.1.

Table 5.1: The Aevitas collection composed by the vehicle types **CM**, **FH**, **FM**, **RM**, and **RX**. Per vehicle type the min/max number of attributes, geometric mean entropy rate, and min/max trip length in time are shown. Entropy rate is calculated based on the discretized attribute time series using the Freedman-Diaconis rule for binning [83].

Vehicle Type	**Attributes**		**Geometric Mean Entropy Rate**	**Trip Length**	
	min	**max**		**min**	**max**
CM	117	138	1.47e-06	0.3s	7h
FH	19	245	2.98e-06	1.9s	12.5h
FM	10	181	8.36e-06	7.7s	3.9h
RM	20	179	8.63e-06	2.0s	4.5h
RX	138	177	2.88e-06	4.8s	8.5h

5.1.3 Vehicle Data in the Zenseact Drives Collection

The *Drives* dataset was chosen and used from the ZOD collection. More specifically, the vehicle data in the form of `hdf5` files was extracted and aggregated for each frame of the drives. This vehicle data consists of three different categories seen in Table 5.2, these being Ego Vehicle Controls, Ego Vehicle Data, and Satellite.

Table 5.2: Categories from the ZOD Drives collection composed of **Ego Vehicle Controls**, **Ego Vehicle Data**, and **Satellite**. Per category the min/max number of attributes, geometric mean entropy rate, and min/max trip length in time are shown. Entropy rate is calculated based on the discretized attribute time series using the Freedman-Diaconis rule for binning [83].

Category	**Attributes**		**Geometric Mean Entropy Rate**	**Trip Length**	
	min	**max**		**min**	**max**
Ego Vehicle Controls	6	7	2.73e-07	10m	2.6h
Ego Vehicle Data	11	12	8.40e-06	10m	2.6h
Satellite	14	18	4.05e-07	10m	2.6h

Sampling rates:

- **Ego Vehicle Controls**: 100Hz
- **Ego Vehicle Data**: 50-100Hz
- **Satellite**: 10-13Hz

5.2 Experiments

The following section describes the experiments performed that evaluated the proposed methodology. Section 5.2.1 shows the selected approach to evaluating the client, whereas Section 5.2.2 shows an exploration of message payload size in relation to the set requirements.

5.2.1 Client

For evaluating the client, two main experiments were utilized. First, the implemented compression methods were run on the entire Aevitas and ZOD dataset collections in order to assess a reference performance benchmark in terms of execution time and compression. Compression was performed by separating all attribute time series contained in a vehicle trip and compressing each time series individually in sequence. In this benchmark, all values were stored locally to exclude communication cost. The execution time for compressing a trip was measured in microseconds from the start of compressing the first attribute time series until the last, meaning that initial data read times from disk are disregarded. As to remain agnostic of the variance in each trips length and number of attributes, execution time was compared as the average of 1000 values. Compression ratio was simply seen as the original size of the trip divided by the compressed size.

Following the established benchmark, test simulations were performed as to assess performance in more realistic scenarios were additional constraints are imposed. Most importantly not all data is available at the point of compression due to the limit of allocated memory. The simulations were performed using the model proposed in Section 4.1 where the largest trip of each sampled vehicle type was streamed as data source. Based on this, the window size factor w_s in Equation 4.1 was manipulated to establish what compression window produced the most favorable results in terms of execution time and compression ratio. Importantly, these experiments were performed only using the Gorilla algorithm. This was motivated by its results surpassing the CHIMP algorithm in all preliminary benchmarks performed, and as such further comparison was deemed superfluous.

For the experimentation on compression windows, the four longest trips were chosen out of every dataset and a coarse-grained logarithmic increase to the window size was performed, starting from 1 second. The aim of this was to establish an intra-dataset comparison as to observe the compression scaling after X seconds of data. Thereafter when experimenting on the execution time, a fine-grained increase was used in order to establish the impact on scaling in relation to cache sizes. By considering the buffer size (bound to window size), an inter-dataset comparison was possible.

5.2.2 Receiver

A considerable parameter in the integration of the client and the receiver was considered to be the payload size. These payloads are seen from the perspective of Apache Kafka as messages and will be discussed as such. In order to investigate the appropriate message size sent to the Kafka broker, a set of message sizes were derived from the total generated bandwidth for one vehicle (see Equation 4.3, Section 4.2) and the estimated compression ratio of that bandwidth (see Table 6.1, Section 6.1). This gave a total generated bandwidth of $B_i = 3.2$ MB/s, a compression ratio of $C \approx 15$ (using Gorilla), and a final compressed output bandwidth of:

$$B_c = \left\lceil \frac{B_i}{C} \right\rceil = 214 \text{ KB/s} \tag{5.1}$$

Furthermore, the receiver architecture in Section 4.2.2 proposes sending a separate message per attribute, meaning that the message size is derived from the compressed per-attribute bandwidth B_a. This is given by the compressed output bandwidth $B_c = 214$ KB/s divided by the number of attributes $n_{\max} = 4000$:

$$B_a = \left\lceil \frac{B_c}{n_{\max}} \right\rceil = 54 \text{ B/s} \tag{5.2}$$

By observing the behaviour of compression ratio in relation to the window size w_s in Section 6.1, it can be noted that the compression ratio peaks at window sizes of roughly 100 to 1000 seconds. As such the appropriate message size was believed to be found within the interval of $[100B_a, 1000B_a]$.

Based on the derived message sizes, experimentation on the interval was performed using the `kafka-producer-perf-test.sh` script included in the Apache Kafka docker image [81] under the `/opt/kafka/bin/` directory. This script performs a producer stream processing benchmark on Kafka given a total number of messages and a payload file. For this purpose, the payload file was of the format proposed in Section 4.2.3, where the number of bytes in the value sequence corresponded to the message size as the additional storage overhead of identifiers was assumed to be negligible. Specifically the message sizes examined were 5.4KB, 13.5KB, 27KB, 40.5B, and 54KB. For each message size, a total of 21.6GB in data was sent and for simplicity in testing, only one Kafka node was utilized serving as both broker and controller with the default settings.

6 Results

Based on the presented evaluation method, experiments were performed on the proposed methodology. In Section 6.1, the results of experiments on the Client as described in Section 4.1 are presented.

6.1 Client Results

Related questions: *RQ1, RQ2, RQ3*

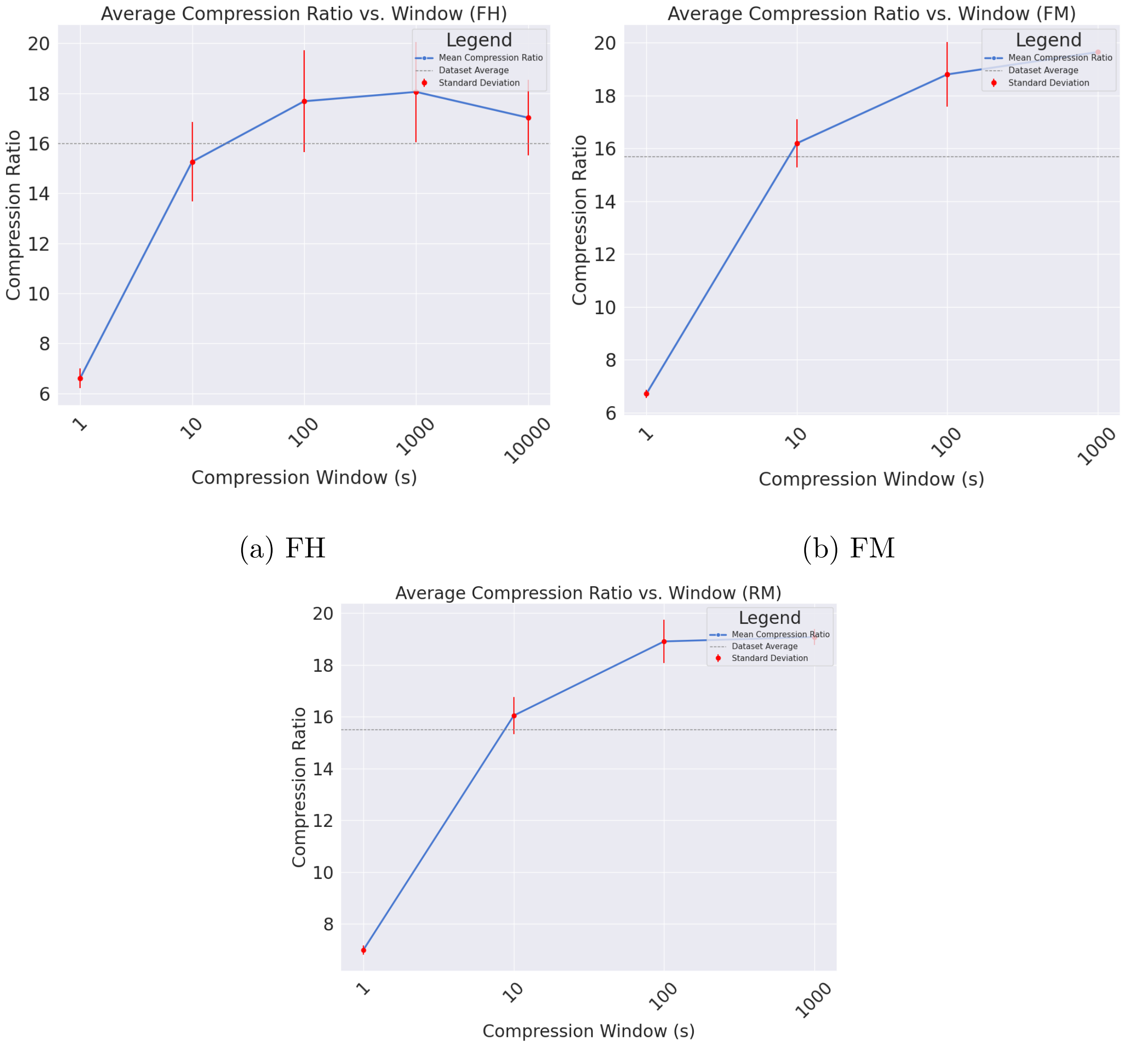
Firstly, with the intention of establishing a simple benchmark for execution time and compression ratios on the tested hardware, the Gorilla and CHIMP algorithms were run serially on all datasets using the host. The results of this benchmarking can be seen in Table 6.1. It was chosen to join all categories of the ZOD drives for this benchmark, as the number of attributes were significantly lower for these in comparison to the Aevitas vehicle types.

Table 6.1: Execution time in microseconds (per 1000 values compressed) and compression ratio ($\frac{\texttt{original}}{\texttt{compressed}}$) for the algorithms Gorilla and CHIMP on vehicle datasets. Results shown correspond to the mean of each dataset where Execution Time uses the harmonic mean and Compression Ratio uses the arithmetic. At the bottom, the arithmetic mean for each algorithm is shown over all datasets.

Dataset	**Execution Time**		**Compression Ratio**	
	Gorilla	**CHIMP**	**Gorilla**	**CHIMP**
CM	15.5	25.5	18.6	15.8
FH	17.6	27.5	16.0	13.8
FM	17.9	27.8	15.7	13.7
RM	17.6	27.2	15.5	13.6
RX	16.9	26.6	16.9	14.5
ZOD	36.0	56.0	6.8	6.38
Total Average	20.3	31.8	14.9	13.0

Gathered from this benchmark, it can be established that the Gorilla compression method is both faster and achieves a higher compression ratio for the selected datasets. Interestingly, the execution time for the ZOD dataset is comparatively higher than all Aevitas datasets, while the compression ratio is considerably lower. It can be argued that this benchmark implies the relationship between how well data compresses and the compression time, as the dataset with the lowest execution time also had the highest compression ratio.

When exploring the impact of window size on compression, performance tended to be around the average for some datasets, which is shown in Figure 6.1. Out of these, the sampled FM (Figure 6.1b) and RM (Figure 6.1a) vehicle types had a considerably low standard deviation, but were also generally above their respective averages as well. FH (Figure 6.1a) showed a higher standard deviation, but still tended to be around its dataset average.



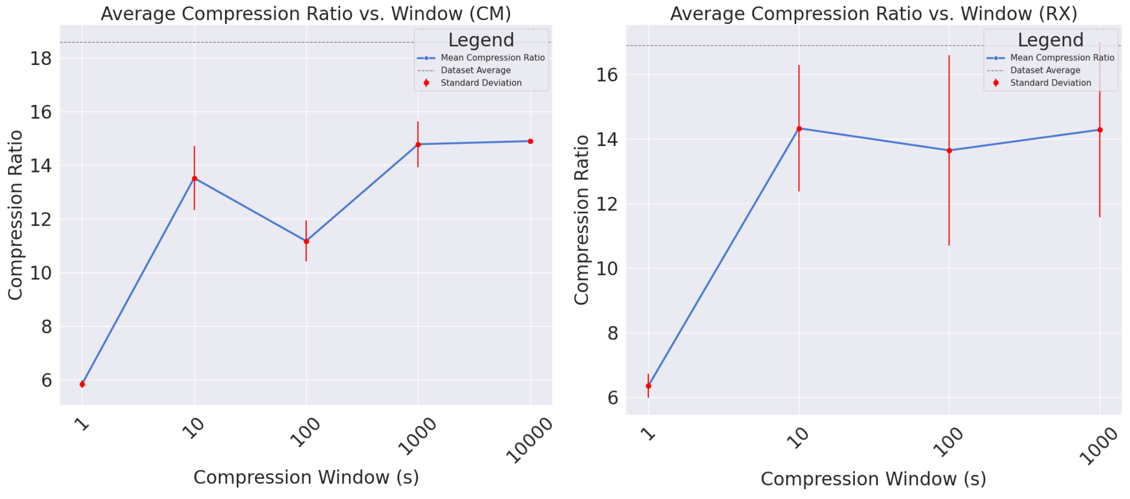
(a) FH

(b) FM

(c) RM

Figure 6.1: Average (arithmetic) compression ratios $(\frac{\texttt{original}}{\texttt{compressed}})$ in relation for a set compression window in seconds for four trips of each vehicle type FH (6.1a), FM (6.1b), and RM (6.1c). A dotted line describes the average compression ratio for each specific vehicle type and standard deviation is shown as a red range and dot. Compresson ratios for the selected window sizes peak at 1000s, while standard deviation shows close to dataset average compression for all three types. FM and RM lack testing for 10000s as none of their available trips were of sufficient length.

Two datasets showed irrational growth in the compression ratio, which is shown in Figure 6.2. More specifically, the performance decreased considerably from 10 to 100 seconds, and increased from 100 to 1000 seconds. A possible explanation for this is that there could be a higher entropy for these vehicle types in window sizes of 100 seconds as compared to 10 or 1000 seconds. Another explanation could be that the vehicles changed thir behavior significantly within these 100 second intervals.



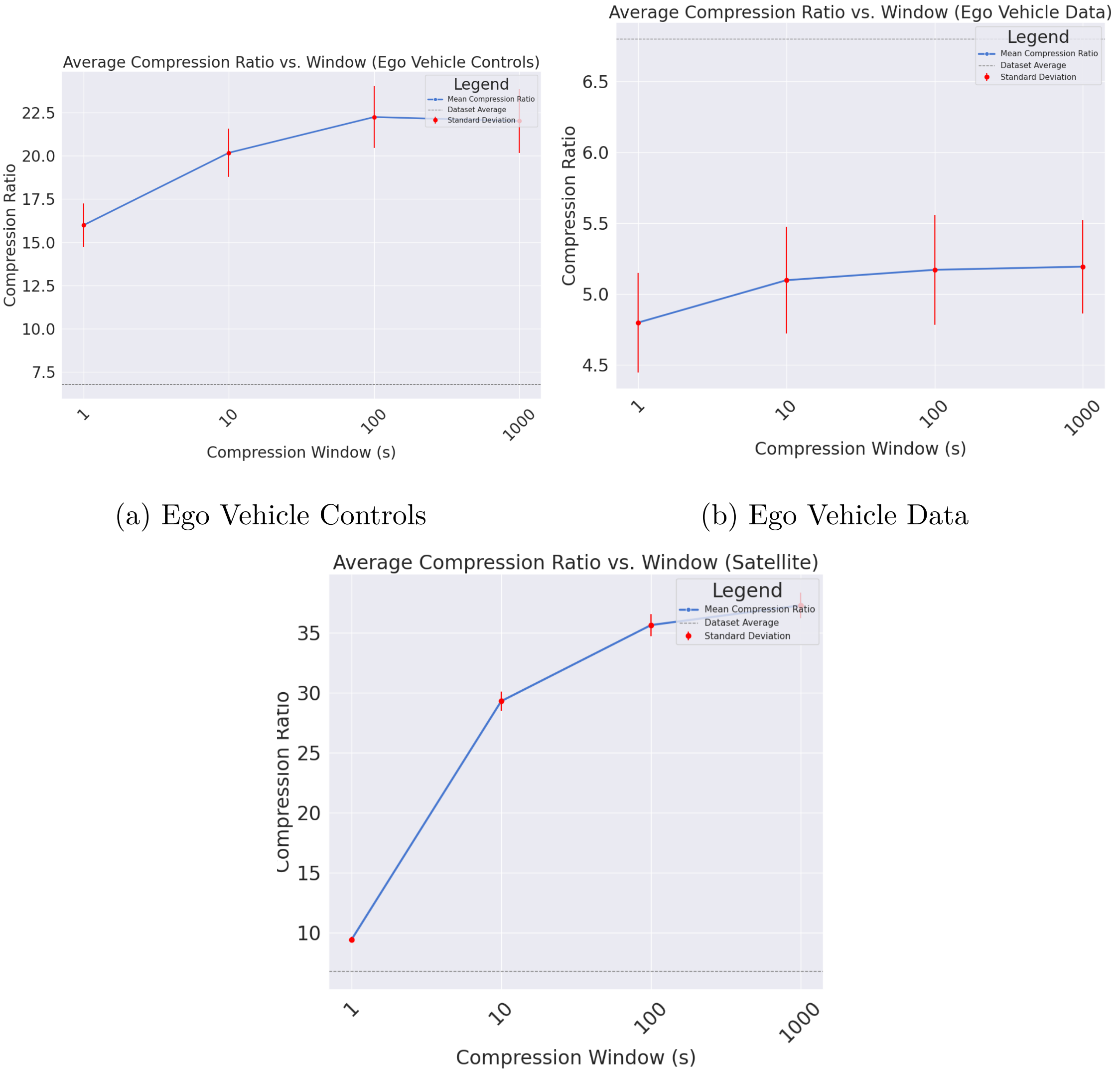
(a) CM

(b) RX

Figure 6.2: Average (arithmetic) compression ratios $(\frac{\texttt{original}}{\texttt{compressed}})$ for a set compression window in seconds for four trips of each vehicle type CM (6.2a) and RX (6.2b). A dotted line describes the average compression ratio for each specific vehicle type and standard deviation is shown as a red range and dot. Performance declines from 10s to 100s, increasing again at 1000s and thereafter converging. The standard deviaton is significantly higher for RX than CM, suggesting that compression is more variable for RX. The selected trips for both types display considerably lower performance than the average in their respective dataset.

The compression window experimentation results of the ZOD categories are shown in Figure 6.3. Notably, Ego Vehicle Controls (Figure 6.3a) and Satellite (Figure 6.3c) showed significantly high compression ratios, whereas Ego Vehicle Data (Figure 6.3b) had comparatively much lower compression. None of these categories were close to the ZOD average, signifying some sort of imbalance in the distribution of data points. This is further supported by the dataset statistics which suggest that Ego Vehilce Data contains a larger number of data points than the rest, based on the sample rate and number of attributes.

In general, it can be stated that compression reaches a maximum after a window of roughly 100 to 1000 seconds for most tested datasets. The reason was believed to be that this interval provides a sufficient amount of historical data such that the storage of the initial values has less of an impact over time.



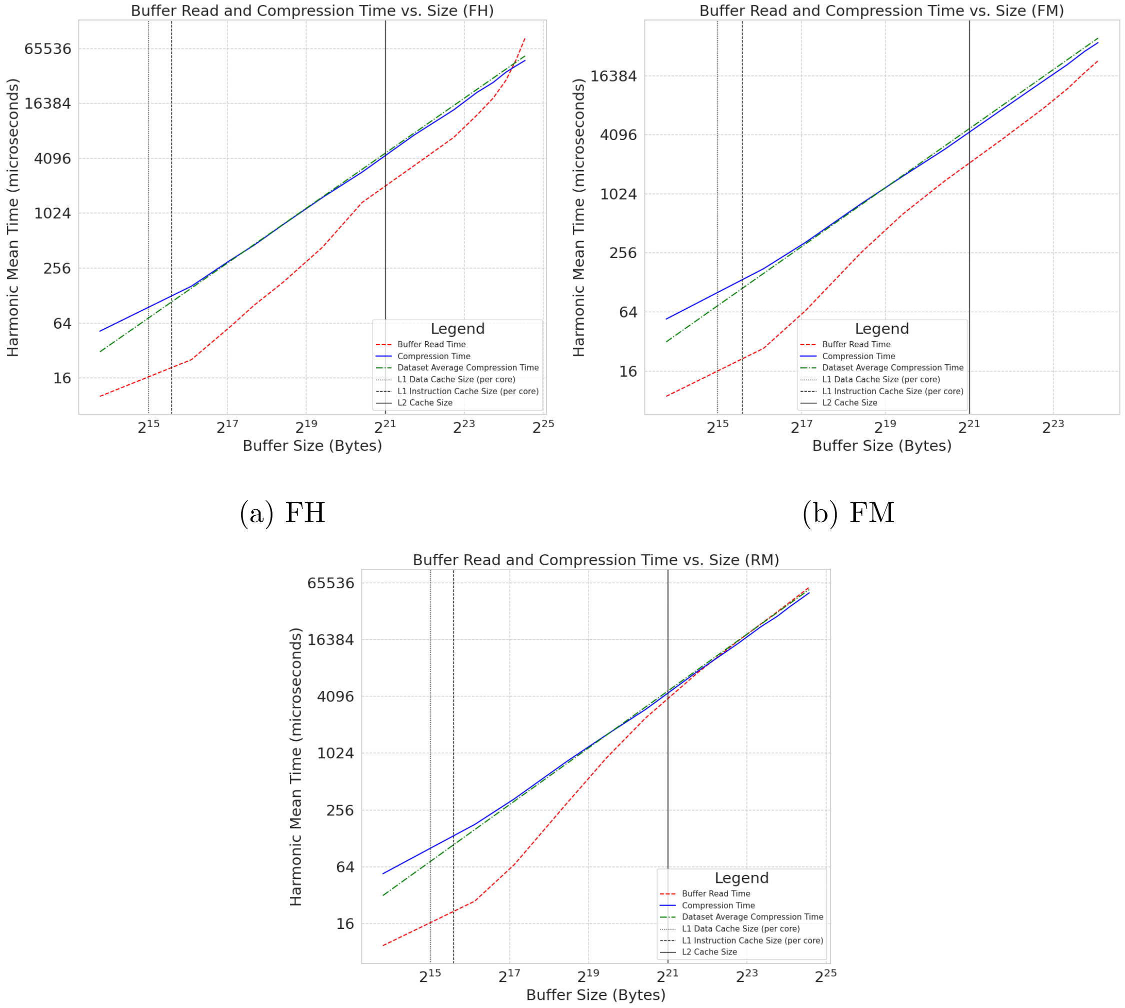
(a) Ego Vehicle Controls

(b) Ego Vehicle Data

(c) Satellite

Figure 6.3: Average (arithmetic) compression ratios ($\frac{\texttt{original}}{\texttt{compressed}}$) in relation for a set compression window in seconds for four trips of each vehicle dataset Ego Vehicle Controls (6.3a), Ego Vehicle Data (6.3b), and Satellite (6.3c). A dotted line describes the average compression ratio for each specific vehicle type and standard deviation is shown as a red range and dot.

For the sake of transparency, the choice was made to divide execution time into buffer read time and compression time. Buffer read time corresponds to the transposition required to process the values in the in-buffer, while compression time is simply the time it takes to run the compression stage. It can be observed that for most experiments on the Aevitas datasets, compression time was always close to the average and scaled linearly with the buffer size. This is trend is apparent in Figure 6.4. Additionally it can be noted that for buffer sizes below the L1 cache size, the compression time is initially higher than the average, but has a lower gradient and eventually converges with the average. This behavior was observed for the vehicle types shown in Figure 6.5 as well.



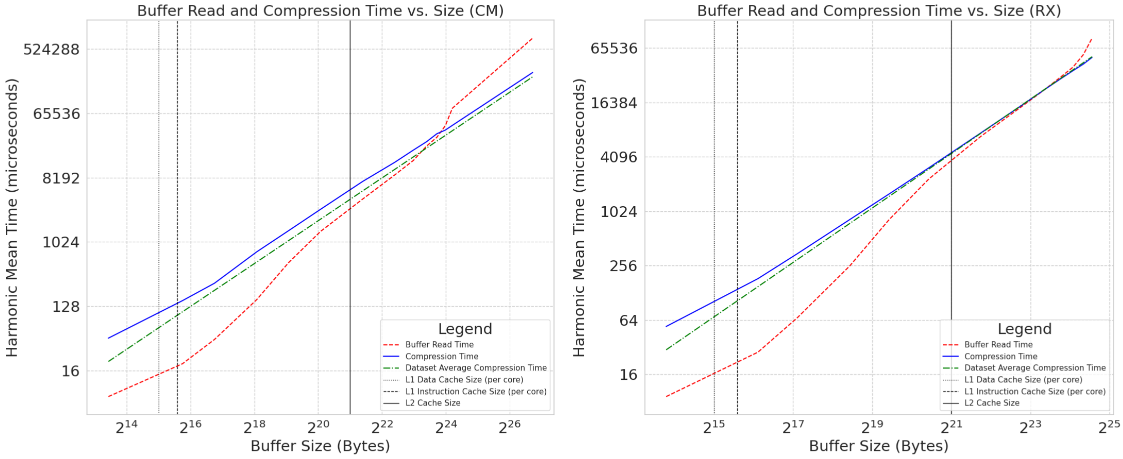
(a) FH

(b) FM

(c) RM

Figure 6.4: Harmonic mean time for buffer read (dotted red) and compression (blue) in microseconds in relation to a set buffer size for one trip of vehicle types FH (6.4a), FM (6.4b), and RM (6.4c). The average compression time for all trips in each dataset is shown as a dashed line (green) and buffer size is shown logarithmically with base 2. L1 data and instruction cache sizes are shown per core as vertical dotted and dashed lines respectively. L2 cache size is shown as a black vertical line.

An interesting aspect to consider is the scaling of the buffer read time. For the Aevitas collection, the buffer read time was considerably lower than the compression time, but after buffer sizes of roughly 2^{16} bytes ($\approx$ 66KB), buffer read time scaled aggressively fast. After around 2^{24} bytes ($\approx$ 17MB), buffer read time surpassed compression time, which is apparent in Figure 6.5a. This behaviour had no straight forward explanation, but one theory is that there is a relation between the number of attributes and buffer read time, which becomes more probable when analyzing the ZOD results.



(a) CM

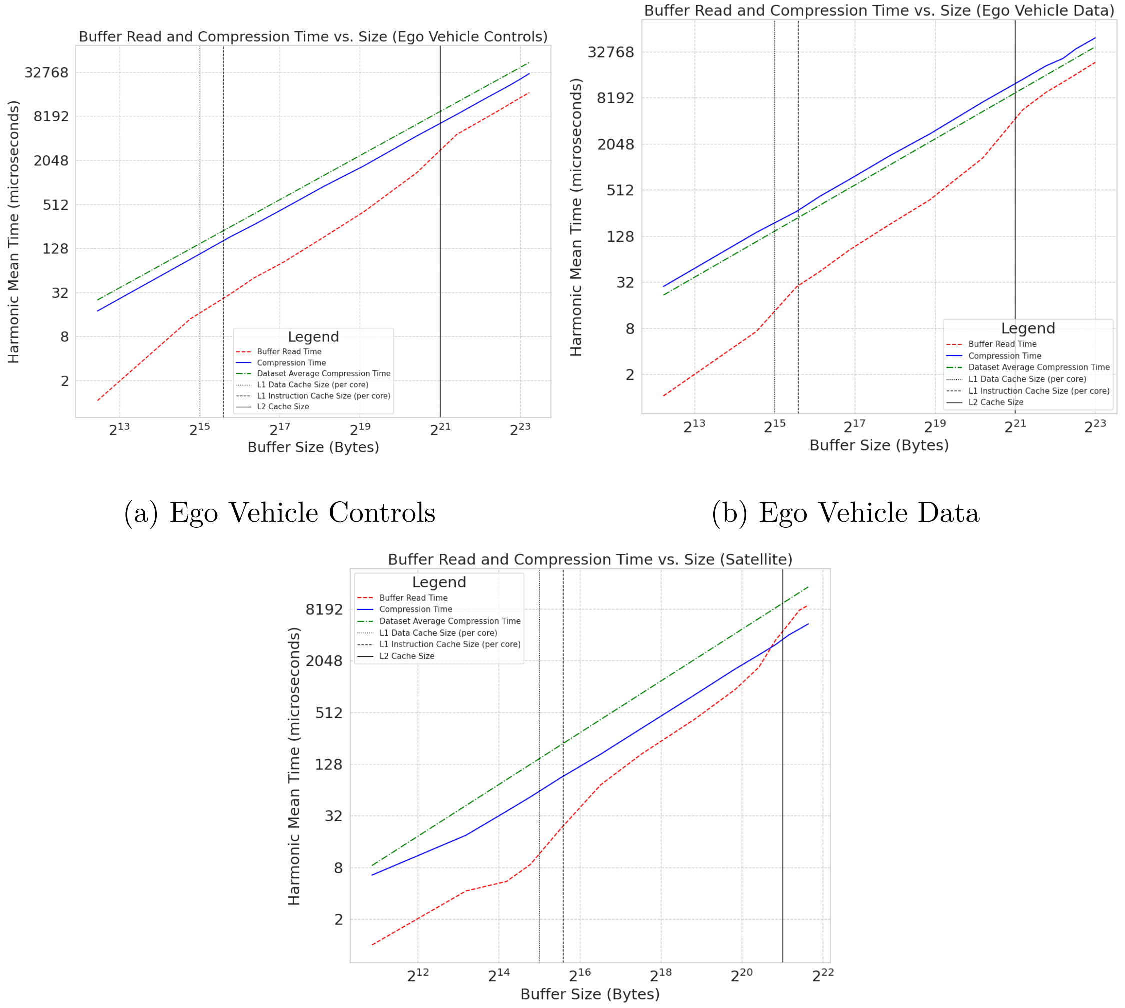
(b) RX

Figure 6.5: Harmonic mean time for buffer read (dotted red) and compression (blue) in microseconds in relation to a set buffer size for one trip of vehicle types CM (6.5a) and RX (6.5b). The average compression time for all trips in each dataset is shown as a dashed line (green) and buffer size is shown logarithmically with base 2. L1 data and instruction cache sizes are shown per core as vertical dotted and dashed lines respectively. L2 cache size is shown as a black vertical line.

The results of buffer size experimentation on the ZOD categories are presented in Figure 6.6. Here, the buffer read times grow aggressively until a certain point, after which its gradient stabilizes once again. Notably this point varies from each category, existing around 2^{21} bytes ($\approx$ 2.1MB) for Ego Vehicle Controls and Ego Vehicle Data, but is considerably more chaotic for Satellite. This arguably hints towards the previously mentioned theory that the number of attributes is correlated with the behaviour, as the buffer read time is also significantly higher for the ZOD categories when comparing the same buffer size for Aevitas.

An observation is how the compression time of Ego Vehicle Controls and Ego Vehicle Data are considerably higher than the average. Additionally, compression time seems almost perfectly linear in contrast to behaviour of changing around the L1 cache size, as seen in the Aevitas vehicles.

A general behavior observed for the majority of datasets is a change in the buffer read time gradient around the L1 and L2 cache sizes. Hypothetically, this is due to an introduced memory fetch which impacts the buffer read time first, while consequently maintaining the compression time as linear.



(a) Ego Vehicle Controls

(b) Ego Vehicle Data

(c) Satellite

Figure 6.6: Harmonic mean time for buffer read (dotted red) and compression (blue) in microseconds in relation to a set buffer size for one trip of each vehicle dataset Ego Vehicle Controls (6.6a), Ego Vehicle Data (6.6b), and Satellite (6.6c). The average compression time for all trips in each dataset is shown as a dashed line (green) and buffer size is shown logarithmically with base 2. L1 data and instruction cache sizes are shown per core as vertical dotted and dashed lines respectively. L2 cache size is shown as a black vertical line.

6.2 Message Broker Results

Related questions: *RQ3, RQ4*

Figure 6.7 shows the measured throughput in MB/s for messages of the examined sizes. Most importantly for these sizes the throughput always increases with size where 54KB had the highest throughput. The largest measured throughput increase can be seen when going from 13.5KB to 27KB, where a similar gain is again noticeable when doubling 27KB to 54KB. This behaviour is likely correlated with Kafka's ability to better process messages of larger sizes, and as such more data overall is processed.

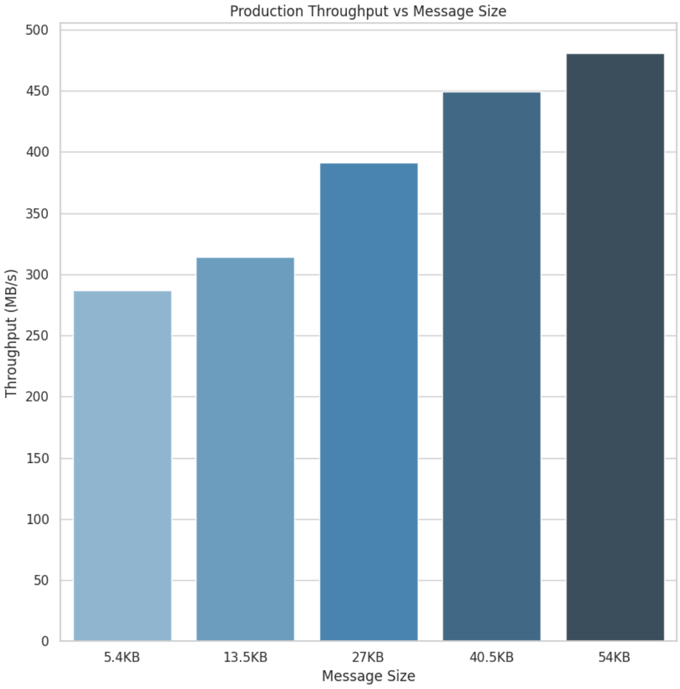


Figure 6.7: Apache Kafka processing throughput (MB/s) for produced messages of the sizes 5.4KB, 13.5KB, 27KB, 40.5KB, and 54KB. Throughput is highest for 54KB and lowest for 5.4KB. For the message sizes shown, throughput always increases with message size where the largest difference is from 13.5KB to 27KB.

Figure 6.8 shows the measured throughput in messages/s for the messages of the examined sizes. Here, the troughput in contrast decreases as the message size increases, where the throughput for 5.4KB is orders of magnitude higher than 13.5KB.

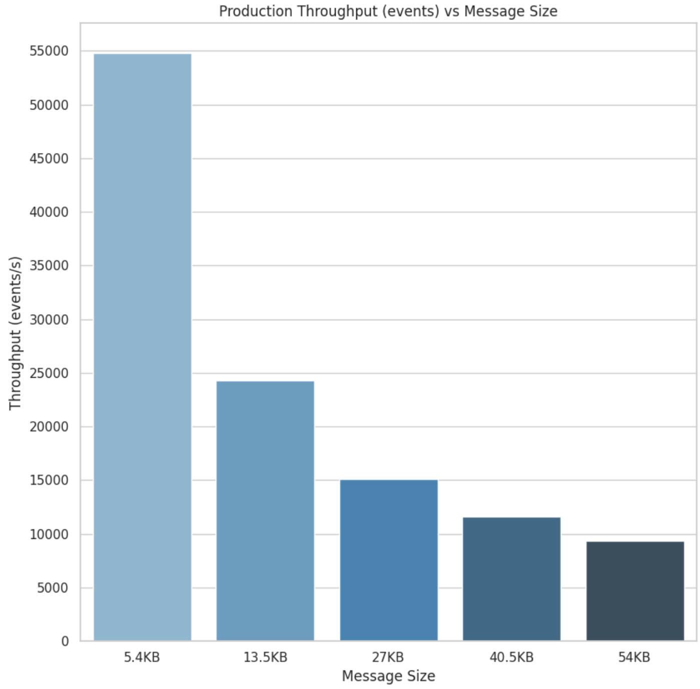


Figure 6.8: Apache Kafka processing throughput (events/s) for produced messages of the sizes 5.4KB, 13.5KB, 27KB, 40.5KB, and 54KB. Throughput is highest for 5.4KB and lowest for 54KB. For message sizes shown, throughput always decreases with message size where the largest difference is from 5.4KB to 13.5KB.

Figure 6.9 show percentiles of latency in milliseconds for different message sizes. The relationship between message size and latency is different from message size and throughput. Here it can be observed that the highest latency is for 13.5KB, followed by 27KB. Notably, 5.4KB yields the third lowest latency whereas 54KB clearly yields the lowest latency.

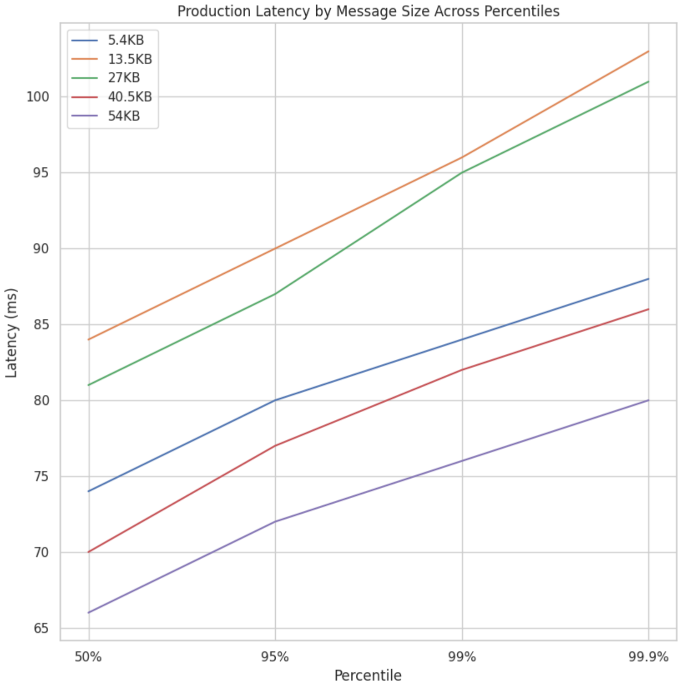


Figure 6.9: Apache Kafka processing latency (ms) for produced messages of the sizes 5.4KB, 13.5KB, 27KB, 40.5KB, and 54KB shown as percentiles. 13.5KB shows the highest latency while 54KB shows the lowest. 5.4KB shows lower latency than 13.5KB and 27KB, suggesting that these sizes approach a local maximum.

7

Discussion

Solving the problem of large scale efficient vehicle data readout in a lossless manner required e Based on the research questions proposed, a number of discussion points were considered. Chiefly described in Section 7.1 is the impact on the client system in terms of system resources. Section 7.2 composes the total cost of computation and memory as a hypothetical mathematical function. Finally, Section 7.3 discusses the requirements for supporting the client in terms of infrastructure globally.

7.1 System Resource Utilization

By analyzing the benchmarks shown in Section 6.1, it can directly be established that the computational requirements of the proposed methodology are significantly low for the tested hardware. Assuming that the method maintains a linear scaling, this means that a single time step consisting of 4000 values can be compressed in less than 100 microseconds based on the Gorilla algorithm. One apparent issue is the displayed buffer read time of the solution, as it out-scaled compression time for some datasets. This is partially attributed to the amount of data being copied in the transposition processing of the initial in-buffer structure, as smaller buffer sizes are considerably more performant in their processing time.

For the presented solution, cache size in relation to buffer size is arguably an aspect that also affected execution time. This is most notable due to the change in gradient around buffer sizes roughly approximate to the different cache levels. System's with smaller cache sizes or fewer cache levels may therefore see less favorable execution times for larger buffer sizes.

When comparing the Aevitas collection with ZOD, it can be observed that Aevitas performed on significantly higher on average. Although, the difference in average compression ratio can likely be explained by the Ego Vehicle Data category displaying very low compression gain by having more data. After more closely examining this dataset, it was seen that the data does not tend to follow a predictable pattern and is as such harder to compress efficiently. Furthermore the results arguably show that the number of attributes and compressibility have an impact on the resulting compression time. For instance, while Ego Vehicle Controls shows favorable compression ratios, the execution time is almost twice as high as FH for the same buffer size. The most notable difference between these are the number of attributes, and as such this suggests a correlation. A possible reason for this correlation is that fewer attributes at the same buffer size will result in each attribute having more data to compress on average.

In terms of memory utilization, the most dominant factors were considered to be the buffers. As an example, utilizing double buffering with compression windows of 1000 seconds would require an input buffer of roughly 6.4GB in size, which is a non-trivial number for less equipped systems. The non-triviality arises from the constant memory occupation, and in some situations even 10% of total available memory can be seen as too great. In this case, trade-offs can be made as smaller windows around 10 to 100 seconds still provide favorable compression results. The cost of this is more frequent writes to the output buffer and a higher risk for interference with other processes on the system. Furthermore, this also has a risk of decreasing the total data throughput of the receiver.

Bandwidth is a factor that is arguably highly variable in terms of connection strength and coverage. It is consequently more interesting to discuss bandwidth as a relation between the compression output bandwidth and the cellular connection bandwidth. Given the compression output bandwidth calculated in Section 5.2.2, this value is roughly 214KB/s based on the set requirements. A 3G cellular connection is naturally unfeasible and the system must utilize either a 4G LTE or 5G network, which means that systems without such compatibility can not adhere to the established requirements. When considering a 4G LTE connection, the lower bound of the maximum upload speed is roughly 250KB/s (2Mbps), being only slightly higher than the compressed output bandwidth. This is problematic as it infers that the system must maintain a constant stable upload connection to avoid running out of memory. Consequently, the proposed solution is more viable in environments that can achieve cellular bandwidths marginally higher than 2Mbps as to accommodate connection strength variability.

Mitigating the problem of bandwidth bottlenecking is a difficult challenge and there are some measures of at least moderating this impact. Firstly, the most apparent solution is to either disregard new data or discard old data. This is a very lossy approach and some heuristic might be required in order to assess some form of importance score to the collected data. Secondly, data can be sampled less frequently for some period in order to alleviate the strain on connection quality. In some cases this might make the data less compressible, but if attributes change slowly over time, this impact will be less significant. Finally, balancing the compression window in terms of connectivity may also see some benefits. Larger windows are more sensitive to differences in connection quality as timing with well connected zones becomes more difficult. On the other hand, smaller windows are on average less compressible and will as such increase the rate at which memory gets filled.

7.2 Model Formulation

The investigated results were formed from the basis of a specific set of requirements. For general compatibility, a hypothetical mathematical model was discussed as to provide some form of compatibility layer between different configurations. This was evaluated in terms of compute and memory. Notably, this is based on the observed performance of the Gorilla algorithm.

Compute

The most significant computational process in the client is the compression stage. For simplicity, the buffer read time and compression time were assumed to be equal. This gave the computational model:

$$T_{exe} = \frac{2 \times 20.3 \times \omega \times w_s \times n}{1000}$$

where T_{exe} is the cyclic execution time in microseconds (e.g. every w_s seconds).

Memory

Two main factors govern memory usage of the solution: one static and one dynamic. The static factor is the input buffer, determined as a factor of the chosen sampling rate, compression window, and number of attributes. The dynamic factor is the output buffer, determined as a growth rate function g over the compression window as a product of compression, sampling rate, and number of attributes. Furthermore, the output buffer is also affected by a flushing function $f(t)$ over time in seconds, that is either 0 or equal to the network bandwidth. This gave the memory model:

$$M = 2 \times \omega \times w_s \times n \times sizeof(double) + g - f(t)$$

where M is the total memory used in bytes.

7.3 Digital Infrastructure Requirements

A significant portion of solving the proposed receiver problem is directly related to throughput and network capacity. Based on the prior calculations in Section 5.2.2, the initial total accommodated bandwidth in compressed form is roughly 214GB/s. In comparison, the findings in Section 6.2 present a maximum throughput of roughly 500MB/s. This difference is substantial, however there are a two points that are worth considering. Firstly, it is unfeasible for all vehicles to connect to the same broker as they are spread out globally, thus making region based alternatives a necessity. Secondly, the measured throughput relates to testing on a single producer, and arguably the this may hold for a larger number through horizontal scaling. The study did not however have the capacity to investigate this at a larger realistic scale.

Although an effort was made to prioritize throughput in MB/s, the presented solution also shows favorable numbers in terms of latency for larger message sizes. This becomes more relevant in scenarios were accessing data in real time (or close to) is necessary, such as monitoring. Similarly to throughput, these results can not accurately depict larger scale performance, but nonetheless provide insight into how latency is effected in the system.

Establishing a realistic testing environment for the long term persistent storage was seen as difficult due to the considerable amount of data and resources required.

Furthermore, the way in that data is ordered and stored might vary considerably between different use cases and is as such difficult to predict. In terms of storage size, the data generation rate discussed amounts to hundreds of gigabytes per second which is a massive amount of data. However, it must be established that it is unfeasible for all vehicles in a fleet to generate data at all times. There are some periods where the vehicle must be stationary and/or turned off, and as such the total generation rate will vary over time. Using efficient general purpose compression may also alleviate this further over time. Finally, data must be evaluated in terms of its relevance as it will grow massively over time and it is unlikely that all data collected can be stored forever.

8

Conclusion

This chapter concludes the project and presents its learning outcomes in Section 8.1. Section 8.2 brings up the ethical considerations associated with the project and Section 8.3 considers areas with potential for future work.

8.1 Learning Outcomes

Based on the findings presented in this project, it can firstly be established that leveraging streaming based compression is a significantly effective method of reducing the computational load on constrained vehicle systems. The compression performance of Gorilla also allows for the collection of up to 19 times more data while still meeting real time demands; a feat that was not possible with previous methods.

It can be established that if possible, larger windows of data are generally to be preferred. The reasons include higher compression ratios, better receiver throughput, and longer distances between compression. Additionally, a relationship was observed between the compression time and the final compression ratio, where lower compression time tended on average to yield higher compression ratios. It is therefore important to consider the compressibility of the selected attributes, as these can introduce a higher strain on the system both in terms of execution time but in memory usage increase also.

Receiver infrastructure is a complex issue which is not easily solved without field testing, as numerous external unseen factors may complicate the process. The evaluated data throughput seems to be favorable under controlled environments however, and utilizing horizontal scaling is one hypothetical solution to making the infrastructure sustainable over a larger number of connected clients. There are also numerous configuration options that may increase performance further, and as such Apache Kafka is a seemingly stable choice for offloading data.

Cellular connections pose an inherent bottleneck and restriction to the proposed architecture, one which is not easily solved. By leveraging higher bandwidth networks it is feasible to offload data efficiently, although without constant guarantee and availability is therefore a concern that has to be addressed for this to function efficiently. At the same time in regards to the vehicle itself, the most popular CAN variants do not have sufficient bandwidth to accommodate large streams of generated data and as such measures are required at this end as well.

8.2 Ethical Considerations

Dealing with a large number of data points created indirectly by actions of individuals give rise to an inherently ethical problem. Namely the question of individual privacy and consent towards being subject data collection.

More specifically this project would, allow for the accumulation of a significant amount of data generated partly from the behavior and patterns of individual drivers. For instance it would be possible to infer if a vehicle was speeding at any point in time, but also the efficiency of a driver in terms of vehicle maneuvering. The problem arises from the prospect of using data in unforeseen or malicious ways. A solution to this may perhaps be to anonymize data such that it can not be used to identify any one individual. At the same time, such anonymization impacts usage areas where it may be beneficial to do so and could inadvertently hinder these. As a result it is left outside the scope of this project to solve these issues as negative use cases are unpredictably outside controllable countermeasures.

By contrast, collecting vehicle data can arguably infer noble aspects as well. Monitoring and optimizing the usage of vehicles provides a foundation for reducing emissions, improving the longevity of components, and generally benefiting the sustainability of cargo transportation. In terms of the benefits to the environment, the project from this point of view be seen as ethically defensible.

8.3 Future Work

An interesting area which was not explored in this thesis is the utilization of lossy compression. Hypothetically, such a method would allow for even higher compression ratios in relation to lossless compression. The challenge arises from domain knowledge at the attribute level in terms of what is acceptable loss, which is arguably different for individual use cases. Another related aspect to consider at the producer level is to investigate a lesser precision for the data values with 32-bit or even 16-bit floating point numbers.

Investigating alternative payload formats at the bit level may lead to increased performance for offloading data. A suggestion may be for the receiver to keep a dynamic record for each vehicle in regards to what attributes are being sent, which could decrease the payload size by some margin. The difficulty is integrating such a format efficiently with the message broker, and storage savings are likely not signficiant, but an investigation is relevant nonetheless.

A challenging problem is migitating the risk of not offloading data in time. Strategies are heavily dependent on the requirements and aims of fleet owners, meaning that there is no one straight-forward solution to this problem. An investigation on this may be feasible to establish what works while minimizing aspects such as data loss.

While it was seemingly apparent based on surveys that Apache Kafka likely is the highest throughput message broker, additional software configurations may increase the performance further. These can be at the producer, consumer, and broker level, and warrant a more in-depth investigation on how they effect the performance for vehicle data readout.

Considering different alternatives for the final storage system is an important area that was not evaluated in this project. For instance, the performance implications of using some other file system and/or databasis an aspect that has long-term effects on the data's final usage, although poses challenges in terms of the specific use cases of the data.

Bibliography

[1] E. Jessup, K. L. Casavant, C. T. Lawson, et al., "Truck trip data collection methods," Oregon. Dept. of Transportation. Research Group, Tech. Rep., 2004.

[2] E. McCormack and M. E. Hallenbeck, "Its devices used to collect truck data for performance benchmarks," *Transportation Research Record*, vol. 1957, no. 1, pp. 43–50, 2006. DOI: `10.1177/0361198106195700107`. eprint: `https://doi.org/10.1177/0361198106195700107`. [Online]. Available: `https://doi.org/10.1177/0361198106195700107`.

[3] C. Bloom, J. Tan, J. Ramjohn, and L. Bauer, "Self-driving cars and data collection: Privacy perceptions of networked autonomous vehicles," in *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, Santa Clara, CA: USENIX Association, Jul. 2017, pp. 357–375, ISBN: 978-1-931971-39-3. [Online]. Available: `https://www.usenix.org/conference/soups2017/technical-sessions/presentation/bloom`.

[4] M. Bunting, R. Bhadani, and J. Sprinkle, "Libpanda: A high performance library for vehicle data collection," in *Proceedings of the Workshop on Data-Driven and Intelligent Cyber-Physical Systems*, ser. DI-CPS'21, Nashville, TN, USA: Association for Computing Machinery, 2021, pp. 32–40, ISBN: 9781450384452. DOI: `10.1145/3459609.3460529`. [Online]. Available: `https://doi.org/10.1145/3459609.3460529`.

[5] L. Görne, H.-C. Reuss, A. Krätschmer, and R. Sauerwald, "Smart data preprocessing method for remote vehicle diagnostics to increase data compression efficiency," *Automotive and Engine Technology*, vol. 7, no. 3, pp. 307–316, Dec. 2022, ISSN: 2365-5135. DOI: `10.1007/s41104-022-00113-9`. [Online]. Available: `https://doi.org/10.1007/s41104-022-00113-9`.

[6] CSS Electronics. "CAN Bus Explained - A Simple Intro [2025]," Accessed: Mar. 20, 2025. [Online]. Available: `https://www.csselectronics.com/pages/can-bus-simple-intro-tutorial`.

[7] C. Bernardini, M. R. Asghar, and B. Crispo, "Security and privacy in vehicular communications: Challenges and opportunities," *Vehicular Communications*, vol. 10, pp. 13–28, 2017, ISSN: 2214-2096. DOI: `https://doi.org/10.1016/j.vehcom.2017.10.002`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S2214209617300803`.

[8] M. Wolf, "Chapter 9 - automotive and aerospace systems," in *Computers as Components (Fifth Edition)*, ser. The Morgan Kaufmann Series in Computer Architecture and Design, M. Wolf, Ed., Fifth Edition, Morgan Kaufmann, 2023, pp. 437–452. DOI: `https://doi.org/10.1016/B978-0-323-85128-2.00009-`

8. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780323851282000098.

[9] CSS Electronics. "LIN Bus Explained - A Simple Intro [2025]," Accessed: Apr. 7, 2025. [Online]. Available: https://www.csselectronics.com/pages/lin-bus-protocol-intro-basics.

[10] M. Overton, *Numerical computing with IEEE floating point arithmetic*. SIAM, 2001.

[11] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019. DOI: 10.1109/IEEESTD.2019.8766229.

[12] N. Lu, N. Cheng, N. Zhang, X. Shen, and J. W. Mark, "Connected vehicles: Solutions and challenges," *IEEE Internet of Things Journal*, vol. 1, no. 4, pp. 289–299, 2014. DOI: 10.1109/JIOT.2014.2327587.

[13] 1NCE. "What Are the 5G Speeds and How It Differs from 2G, 3G, and 4G Networks?" Accessed: Apr. 7, 2025. [Online]. Available: https://www.1nce.com/en-eu/resources/iot-knowledge-base/iot-connectivity/cellular-networks/5g-vs-4g-speed.

[14] Things Mobile. "What is the speed of 2G, 3G, 4G and 5G networks?" Accessed: Apr. 7, 2025. [Online]. Available: https://www.thingsmobile.com/q-a/what-is-the-speed-of-2g-3g-4g-and-5g-networks-.

[15] TechSverige. "Sweden turns off 2G and 3G - Switch networks now," Accessed: Apr. 8, 2025. [Online]. Available: https://techsverige.se/en/2024/01/sverige-slacker-2g-och-3g-byt-nat-nu/.

[16] V. Bychkovsky, B. Hull, A. Miu, H. Balakrishnan, and S. Madden, "A measurement study of vehicular internet access using in situ wi-fi networks," in *Proceedings of the 12th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '06, Los Angeles, CA, USA: Association for Computing Machinery, 2006, pp. 50–61, ISBN: 1595932860. DOI: 10.1145/1161089.1161097. [Online]. Available: https://doi.org/10.1145/1161089.1161097.

[17] Y. Hajjaji, W. Boulila, I. R. Farah, I. Romdhani, and A. Hussain, "Big data and iot-based applications in smart environments: A systematic review," *Computer Science Review*, vol. 39, p. 100 318, 2021, ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2020.100318. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013720304184.

[18] C. Dupont, M. Sheikhalishahi, A. R. Biswas, and T. Bures, "Iot, big data, and cloud platform for rural african needs," in *2017 IST-Africa Week Conference (IST-Africa)*, 2017, pp. 1–7. DOI: 10.23919/ISTAFRICA.2017.8102386.

[19] J. L. Berral et al., "Towards energy-aware scheduling in data centers using machine learning," in *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, ser. e-Energy '10, Passau, Germany: Association for Computing Machinery, 2010, pp. 215–224, ISBN: 9781450300421. DOI: 10.1145/1791314.1791349. [Online]. Available: https://doi.org/10.1145/1791314.1791349.

[20] K. Haghshenas, A. Pahlevan, M. Zapater, S. Mohammadi, and D. Atienza, "MAGNETIC: Multi-Agent Machine Learning-Based Approach for Energy Efficient Dynamic Consolidation in Data Centers," *IEEE Transactions on*

Services Computing, vol. 15, no. 1, pp. 30–44, 2022. DOI: 10.1109/TSC.2019.2919555.

[21] S. Sagiroglu and D. Sinanc, "Big data: A review," in *2013 International Conference on Collaboration Technologies and Systems (CTS)*, 2013, pp. 42–47. DOI: 10.1109/CTS.2013.6567202.

[22] A. Labrinidis and H. V. Jagadish, "Challenges and opportunities with big data," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 2032–2033, Aug. 2012, ISSN: 2150-8097. DOI: 10.14778/2367502.2367572. [Online]. Available: https://doi.org/10.14778/2367502.2367572.

[23] A. Siddiqa, A. Karim, and A. Gani, "Big data storage technologies: A survey," *Frontiers of Information Technology & Electronic Engineering*, vol. 18, no. 8, pp. 1040–1070, Aug. 2017, ISSN: 2095-9230. DOI: 10.1631/FITEE.1500441. [Online]. Available: https://doi.org/10.1631/FITEE.1500441.

[24] J. Ronkainen and A. Iivari, "Designing a data management pipeline for pervasive sensor communication systems," *Procedia Computer Science*, vol. 56, pp. 183–188, 2015, The 10th International Conference on Future Networks and Communications (FNC 2015) / The 12th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2015) Affiliated Workshops, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2015.07.193. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050915016749.

[25] S. K. Jensen, T. B. Pedersen, and C. Thomsen, "Time series management systems: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 11, pp. 2581–2600, 2017. DOI: 10.1109/TKDE.2017.2740932.

[26] E. Levy and A. Silberschatz, "Distributed file systems: Concepts and examples," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 321–374, Dec. 1990, ISSN: 0360-0300. DOI: 10.1145/98163.98169. [Online]. Available: https://doi.org/10.1145/98163.98169.

[27] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03, Bolton Landing, NY, USA: Association for Computing Machinery, 2003, pp. 29–43, ISBN: 1581137575. DOI: 10.1145/945445.945450. [Online]. Available: https://doi.org/10.1145/945445.945450.

[28] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[29] Apache Software Foundation. "Apache Hadoop," Accessed: Mar. 10, 2025. [Online]. Available: https://hadoop.apache.org/.

[30] N. Jatana, S. Puri, M. Ahuja, I. Kathuria, and D. Gosain, "A survey and comparison of relational and non-relational database," *International Journal of Engineering Research & Technology*, vol. 1, no. 6, pp. 1–5, 2012.

[31] U. Bhat and S. Jadhav, "Moving towards non-relational databases," *International Journal of Computer Applications*, vol. 1, no. 13, pp. 40–47, 2010.

[32] F. Chang et al., "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, Jun. 2008, ISSN: 0734-2071. DOI:

10.1145/1365815.1365816. [Online]. Available: https://doi.org/10.1145/1365815.1365816.

[33] Apache Software Foundation. “Apache HBase,” Accessed: Mar. 14, 2025. [Online]. Available: https://hbase.apache.org/.

[34] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977. DOI: 10.1109/TIT.1977.1055714.

[35] J. Ziv and A. Lempel, “Compression of individual sequences via variable-rate coding,” *IEEE Transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978. DOI: 10.1109/TIT.1978.1055934.

[36] Free Software Foundation, Inc. “GNU Gzip: General file (de)compression,” Accessed: Apr. 14, 2025. [Online]. Available: https://www.gnu.org/software/gzip/manual/gzip.html.

[37] Peter Deutsch. “DEFLATE Compressed Data Format Specification version 1.3,” Accessed: Apr. 14, 2025. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc1951.

[38] J. Alakuijala et al., “Brotli: A general-purpose data compressor,” *ACM Transactions on Information Systems*, 2019. [Online]. Available: https://dl.acm.org/citation.cfm?id=3231935.

[39] Y. Collet. “LZ4 - Extremely fast compression,” Accessed: Feb. 24, 2025. [Online]. Available: https://github.com/lz4/lz4.

[40] J. Dean, S. Ghemawat, and S. H. Gunderson. “Snappy, a fast compressor/decompressor,” Accessed: Feb. 24, 2025. [Online]. Available: https://github.com/google/snappy.

[41] Y. Collet. “Zstandard - Fast real-time compression algorithm,” Accessed: Feb. 24, 2025. [Online]. Available: https://github.com/facebook/zstd.

[42] Y. Collet. “Zstandard Compression and the ’application/zstd’ Media Type,” Accessed: Apr. 15, 2025. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8878.

[43] I. Pavlov. “7-Zip,” Accessed: Apr. 15, 2025. [Online]. Available: https://www.7-zip.org/.

[44] The Tukaani Project. “XZ Utils,” Accessed: Feb. 24, 2025. [Online]. Available: https://tukaani.org/xz/.

[45] M. Burtscher and P. Ratanaworabhan, “Fpc: A high-speed compressor for double-precision floating-point data,” *IEEE Transactions on Computers*, vol. 58, no. 1, pp. 18–31, 2009. DOI: 10.1109/TC.2008.131.

[46] T. Pelkonen et al., “Gorilla: A fast, scalable, in-memory time series database,” *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1816–1827, Aug. 2015, ISSN: 2150-8097. DOI: 10.14778/2824032.2824078. [Online]. Available: https://doi.org/10.14778/2824032.2824078.

[47] P. Liakos, K. Papakonstantinopoulou, and Y. Kotidis, “Chimp: Efficient lossless floating point compression for time series databases,” *Proc. VLDB Endow.*, vol. 15, no. 11, pp. 3058–3070, Jul. 2022, ISSN: 2150-8097. DOI: 10.14778/3551793.3551852. [Online]. Available: https://doi.org/10.14778/3551793.3551852.

[48] R. Li, Z. Li, Y. Wu, C. Chen, and Y. Zheng, "Elf: Erasing-based lossless floating-point compression," *Proceedings of the VLDB Endowment*, vol. 16, no. 7, pp. 1763–1776, 2023.

[49] Y. Yao, L. Chen, Z. Fang, Y. Gao, C. S. Jensen, and T. Li, "Camel: Efficient compression of floating-point time series," *Proc. ACM Manag. Data*, vol. 2, no. 6, Dec. 2024. DOI: 10.1145/3698802. [Online]. Available: https://doi.org/10.1145/3698802.

[50] Gartner. "IB (Integration Broker)," Accessed: May 26, 2025. [Online]. Available: https://www.gartner.com/en/information-technology/glossary/ib-integration-broker.

[51] IBM. "What is a message broker?" Accessed: May 30, 2025. [Online]. Available: https://www.ibm.com/think/topics/message-brokers.

[52] Apache Software Foundation. "ActiveMQ Artemis," Accessed: Apr. 23, 2025. [Online]. Available: https://activemq.apache.org/components/artemis/.

[53] J. Kreps, "Kafka : A distributed messaging system for log processing," 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:18534081.

[54] Redis Inc. "Redis," Accessed: Apr. 23, 2025. [Online]. Available: https://redis.io/.

[55] Pivotal Software. "RabbitMQ," Accessed: Apr. 23, 2025. [Online]. Available: https://www.rabbitmq.com/.

[56] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[57] GENIVI. "CANdevStudio," Accessed: May 27, 2025. [Online]. Available: https://github.com/GENIVI/CANdevStudio.

[58] M. Edenhill. "librdkafka - the Apache Kafka C/C++ client library," Accessed: Mar. 18, 2025. [Online]. Available: https://github.com/confluentinc/librdkafka.

[59] InfluxData. "InfluxDB," Accessed: Feb. 25, 2025. [Online]. Available: https://github.com/influxdata/influxdb.

[60] C. Wang et al., "Apache iotdb: A time series database for iot applications," *Proc. ACM Manag. Data*, vol. 1, no. 2, Jun. 2023. DOI: 10.1145/3589775. [Online]. Available: https://doi.org/10.1145/3589775.

[61] Prometheus Authors. "Prometheus," Accessed: Feb. 25, 2025. [Online]. Available: https://github.com/prometheus/prometheus.

[62] Timescale Inc. "TimescaleDB," Accessed: Feb. 25, 2025. [Online]. Available: https://github.com/timescale/timescaledb.

[63] J. Jagielski, M. Mao, P. Rungta, R. Skillington, and M. Way. "M3," Accessed: Feb. 25, 2025. [Online]. Available: https://github.com/m3db/m3.

[64] Mozilla Foundation. "Base64," Accessed: Mar. 14, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Base64.

[65] Simon Josefsson. "The Base16, Base32, and Base64 Data Encodings," Accessed: Mar. 14, 2025. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc4648.

[66] J. Henke. "basE91," Accessed: May 28, 2025. [Online]. Available: https://sourceforge.net/projects/base91/.

[67] Netflix. "Evolution of the Netflix Data Pipeline," Accessed: Mar. 13, 2025. [Online]. Available: https://netflixtechblog.com/evolution-of-the-netflix-data-pipeline-da246ca36905.

[68] Terrastore. "Terrastore—Scalable, Elastic, Consistent Document Store," Accessed: Mar. 13, 2025. [Online]. Available: http://code.google.com/p/terrastore.

[69] J. Nandimath, E. Banerjee, A. Patil, P. Kakade, S. Vaidya, and D. Chaturvedi, "Big data analysis using apache hadoop," in *2013 IEEE 14th International Conference on Information Reuse & Integration (IRI)*, 2013, pp. 700–703. DOI: 10.1109/IRI.2013.6642536.

[70] V. K. Vavilapalli et al., "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13, Santa Clara, California: Association for Computing Machinery, 2013, ISBN: 9781450324281. DOI: 10.1145/2523616.2523633. [Online]. Available: https://doi.org/10.1145/2523616.2523633.

[71] HyperTable. "HyperTable," Accessed: Mar. 14, 2025. [Online]. Available: http://hypertable.com/documentation/.

[72] MongoDB. "MongoDB Architecture Guide (White Paper)," Accessed: Mar. 14, 2025. [Online]. Available: https://www.mongodb.com/lp/white-paper/architectureguide?%20jmp=docs&_ga=1.165918654.1239465962.14309%2078187:%20MongoDB%20%5BAccessed%20on%20May%207,%202015%5D..

[73] R. Maharjan, M. S. H. Chy, M. A. Arju, and T. Cerny, "Benchmarking message queues," *Telecom*, vol. 4, no. 2, pp. 298–312, 2023, ISSN: 2673-4001. [Online]. Available: https://www.mdpi.com/2673-4001/4/2/18.

[74] S. N. Raje, "Performance comparison of message queue methods," English, Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2023-06-21, Ph.D. dissertation, 2019, p. 69, ISBN: 9781392820513.

[75] I. Confluent. "What is Kafka Message Size Limit?" Accessed: Mar. 13, 2025. [Online]. Available: https://www.confluent.io/learn/kafka-message-size-limit/#:~:%20:text=Apache%20Kafka%20key%27s%20default%20maximum,%20help%20brokers%20manage%20memory%20effectively..

[76] Confluent, Inc. "Kafka Connect," Accessed: Mar. 14, 2025. [Online]. Available: https://docs.confluent.io/platform/current/connect/index.html.

[77] Å. Hugo, B. Morin, and K. Svantorp, "Bridging mqtt and kafka to support c-its: A feasibility study," in *2020 21st IEEE International Conference on Mobile Data Management (MDM)*, 2020, pp. 371–376. DOI: 10.1109/MDM48529.2020.00080.

[78] Powers, Dana and Arthur, David. "kafka-python," Accessed: Mar. 18, 2025. [Online]. Available: https://kafka-python.readthedocs.io/en/master/.

[79] Apache Software Foundation. "Apache Kafka API Documentation," Accessed: Mar. 18, 2025. [Online]. Available: https://kafka.apache.org/documentation/#api.

[80] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer, and K. Prabhu. "iPerf - The ultimate speed test tool for TCP, UDP and SCTP," Accessed: Apr. 3, 2025. [Online]. Available: https://iperf.fr/.

[81] The Apache Software Foundation. “Apache Kafka Docker Image,” Accessed: Apr. 3, 2025. [Online]. Available: `https://hub.docker.com/r/apache/kafka`.

[82] M. Alibeigi et al., “Zenseact open dataset: A large-scale and diverse multimodal dataset for autonomous driving,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023, pp. 20 178–20 188.

[83] D. Freedman and P. Diaconis, “On the histogram as a density estimator: L2 theory,” *Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete*, vol. 57, no. 4, pp. 453–476, 1981, ISSN: 1432-2064. DOI: `10.1007/BF01025868`. [Online]. Available: `https://doi.org/10.1007/BF01025868`.