CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Efficient industrial big data pipeline for lossless transfer of vehicular data

Synergetic effects of data compression and in-memory processing on latency and throughput

Master's thesis in Computer science and engineering

MARTIN HILGENDORF

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Efficient industrial big data pipeline for lossless transfer of vehicular data

Synergetic effects of data compression and in-memory processing on latency and throughput

MARTIN HILGENDORF

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Efficient industrial big data pipeline for lossless transfer of vehicular data
Synergetic effects of data compression and in-memory processing on latency and throughput
MARTIN HILGENDORF

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

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

Efficient industrial big data pipeline for lossless transfer of vehicular data
Synergetic effects of data compression and in-memory processing on latency and throughput
MARTIN HILGENDORF
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In the age of big data and growing product complexity, it has become common to monitor and record many aspects of a product or system in order to extract well-founded intelligence and draw conclusions to continue driving innovation. Automating and scaling data transfer and analysis processes in pipelines becomes essential to keep pace with increasing data volumes and rates generated by such practices. Further, industrial big data pipelines are subject to a number of requirements and challenges: data veracity, security, and governance, alongside overall pipeline performance and scalability. To address these challenges in a case study at Volvo Trucks, a general big data pipeline design is developed to serve as a framework for enabling efficient transfer of large data volumes from remote test sites to data centres. Synergetic effects of data compression and in-memory processing as techniques to improve pipeline performance, both in terms of throughput and end-to-end latency, are studied and evaluated. An implementation of a pipeline based on the proposed design is carried out on Apache Airflow to explore latency and throughput performance as well as other aspects such as efficiency and scalability of the design. Various general-purpose lossless data compression algorithms are evaluated and compared in order to balance compression effectiveness and compression time in the pipeline. Performance evaluation of the proposed pipeline with data compression is carried out, achieving an average throughput uplift of 38.8 % over the current solution in use today, while also providing desired functionality which was previously missing such as integrity verification, logging, monitoring and traceability, as well as cataloguing of ingested data. Further, a variation of the pipeline design using shared memory processing to alleviate an identified hardware bottleneck is demonstrated, achieving 82.6 % higher average throughput than the current solution using identical infrastructure and hardware resources.

Keywords: Data pipelines, big data, latency & throughput, data compression, data governance, data veracity, compression, Apache Airflow, workflow orchestration

Contents

List of Figures

List of Tables

1
Introduction

The design and implementation of big data pipelines is more relevant than ever as data grows in prevalence, volume, and completeness. Increasingly complex systems with more advanced sensing capabilities produce larger quantities of data that must be processed and analysed in order to extract intelligence and provide value and insight to engineers and businesses. However, this data may be generated at remote locations with limited processing power. In this situation, *edge computing* has established itself as a popular paradigm to harness these resources to perform pre-processing of the raw data, for example using summarisation algorithms to reduce the required transfer bandwidth. Even then, data must be consolidated in a single location to enable a holistic view. To this end, *data transfer pipelines* have emerged as a common approach to automate reliable transfer of large data volumes. In the age of big data, these data transfer pipelines must scale in order to provide the required throughput and latency performance.

1.1 Motivation

As a motivating example, which also serves as the main case study in this thesis, the following scenario at Volvo Trucks is studied. Data from a fleet of field test vehicles is gathered at distant locations, such as test tracks and customer sites, to enable data-driven product development. In order to analyse and extract insight from these volumes of data, data must first be transferred to data centres, where resources and tools necessary for processing are available. During a test, raw data is captured in test vehicles, often at rates of up to 1 GB/min per vehicle. Once a test is complete, the logged data must be transferred to data centres and ingested into storage systems and processing clusters. As new field test sites and customer collaborations are established more frequently, automated ingestion pipelines become an essential tool to improve efficiency of data management operations. To meet the recurring need for data pipelines with similar challenges and requirements, demand for a reference design of an efficient data transfer pipeline supporting best practices in data management has increased.

1.2 Challenges

This project aims to study the challenges involved in increasing transfer pipeline performance in industrial environments. Reducing delays between data recording and analysis is critical for engineering efforts such as at Volvo Trucks, which faces these

challenges and provides a major use-case for this work. The primary requirement is thus developing a performant and scalable transfer pipeline to support growing data volumes within existing hardware systems. Primary performance metrics of such a pipeline are *throughput* and *latency*, and the pipeline must make efficient use constrained resources such as network bandwidth, particularly as demand and contention increases with growing data volumes and addition of further processing steps. Due to the complexity of such a system, a large number of design choices must be made, resulting in a combinatorial explosion of parameter choices to optimise pipeline performance.

1.3 Contributions

The main contribution of this thesis is the study of synergetic effects of *data compression* and *in-memory processing* techniques to enhance data transfer pipeline performance. To this end, a prototype implementation of such a data transfer pipeline is performed as part of a case study at Volvo Trucks. Effects and various aspects of integrating transparent data compression into data transfer pipelines are studied, with the aim of increasing pipeline throughput across transfer bottlenecks by transferring compressed data. The trade-off between compression time and transfer time is studied in detail, as well as the reduction in communication bandwidth requirements. Further, resource utilisation of the pipeline system in the case study is evaluated to detect resource utilisation imbalance, which is addressed through the implementation of in-memory processing techniques to improve global pipeline throughput by 82.6 % on average over the previous solution.

1.4 Overview

The content of this thesis is presented as follows. Chapter 2 introduces background pertaining to data compression and workflow orchestration platforms, Apache Airflow in particular, used for implementation of the pipeline. Chapter 3 describes the problem of performant and scalable big data transfer in more detail, along with requirements and evaluation metrics for a solution. Chapter 4 presents the challenges involved in constructing a big data transfer pipeline to address the problem, and explores possible options for managing throughput, latency, and scalability. Particular challenges of the case study at Volvo Trucks are also presented here. Chapter 5 describes how the pipeline is implemented and evaluation methods of the selected techniques for improving pipeline performance. In Chapter 6, results from performance measurements are presented and discussed. Detailed benchmarking a variety of compression strategies is presented in order to select a strategy for use in the full pipeline. Then, performance of the proposed pipeline is evaluated, as well as individual pipeline components, and exhibited behaviours such as queueing delays when waiting for resources. In-memory processing is then evaluated as a technique to address an identified performance bottleneck, and the achieved performance improvement is discussed with relation to the altered resource utilisation pattern. Chapter 7 presents a selection of related works in the domain of data

pipelines, and which study associated problems. Finally, Chapter 8 summarises the main conclusions of the work and mentions new challenges and some possibilities for further studies and approaches.

2
Background

This chapter provides background information on topics related to the work conducted in this thesis. A core idea used for improving efficiency in data pipelines is data compression, and a variety of its aspects are introduced here. Further, an implementation of a data pipeline is required for evaluation in a realistic setting. To this end, a workflow management tool is used as a platform and framework with relevant features and functionality needed for implementing a data transfer pipeline. A variety of tools exist within this domain today. Apache Airflow is selected for the pipeline implementation for its flexibility, robustness, and ease of use.

2.1 Data compression

Many tools and algorithms for compressing data have been developed. Algorithms consist of a compression and a corresponding inverse algorithm for decompression. To maximise compression performance for a particular application, the most suitable tool must be found. However, the term performance is itself ambiguous in the context of data compression. Compression algorithms have differing resource requirements when compressing a given amount of data, and achieve varying throughput and compression ratios. Further, the achieved performance may vary depending on the input data, as different types of data may compress differently well. In general, data compression techniques can be classified into two categories: lossless and lossy compression.

2.1.1 Lossless compression

In lossless compression, a decompression algorithm can recover the exact input data from the compressed data. This makes lossless compression algorithms suitable for transparent integration into systems, as the eventual consumers of the data can be fully unaware while the underlying system can operate more efficiently. Example applications include transparently storing data in compressed disk blocks [1], system memory [2], or during high-speed network communications within computing clusters [3]. Today, a wide variety of lossless compression algorithms are available for various purposes.

2.1.1.1 Application-specific compression

Compression algorithms can be designed for specific applications, such as within multimedia. Well known techniques which are commonly used today include the

H.264/H.265 [4], [5], VP9 [6], and AV1 [7] codecs from the domain of video data, or FLAC [8] and ALAC [9] for lossless audio compression. These are relatively advanced algorithms, often based on more general techniques which have been specifically selected and combined to perform well in these domains, but making the algorithm unsuitable for other types of data.

2.1.1.2 General purpose compression methods

General purpose compression methods aim to be applicable on a wider variety of input data, or make fewer assumptions about data structure or format. Various families of methods for lossless compression can be identified, generally based on the main techniques used. Within the scope of this work, four algorithms are of particular interest: GZIP, bzip2, LZ4, and Zstandard, and background information for them is presented below.

GZIP First released in 1992, gzip is a common compression format that is still in use today. Specified in RFC 1952 [10], it is a mature data encoding based on the DEFLATE algorithm (itself specified in RFC 1951 [11]) which consists of two steps. The first step, a technique known as LZ77, finds repeated sequences in the input and replaces them with a reference to the location of the previous repetition. In the second step, a more intricate technique called Huffman coding is employed to replace the most common sequences with shorter representations. To avoid prohibitively high complexity with large inputs, the first step utilises a sliding window over the input data to determine how much time should be invested into finding a match. The size of this window can be controlled by the user through the compression level parameter, where a higher value indicates that a larger window should be used. This is useful if better compression is desired, at the cost of slower compression times.

Bzip2 Bzip2 was initially published in 1996 [12], and has since established itself as a stable tool. Compared to gzip, it generally achieves more effective compression at the cost of higher resource requirements and compression time. In scenarios where data is compressed once but decompressed often, bzip2 can be a suitable candidate due to its faster decompression times. The relatively slow but effective compression is due to a stack of nine different compression techniques applied in sequence [13].

LZ4 As its name suggests, the LZ4 algorithm is related to the LZ77 repetition-finding stage used by gzip in the first stage of the DEFLATE algorithm [11]. Released in 2011 by Yann Collet [14], LZ4 targets the opposite end of the compression level-vs-time tradeoff than bzip2. While compression efficiency is generally worse than Deflate, compression throughput is significantly higher. Thus, LZ4 has quickly gained traction despite its relative novelty, with native support in the Linux kernel and Apache Hadoop, a framework for distributed storage and processing of big data. Due to its higher compression speed and ability to process continuous, potentially unbounded, data streams, LZ4 is also suitable for various real-time applications. While gzip and Bzip2 generally operate on complete files compressed in blocks, LZ4

defines a frame format allowing it to compress arbitrarily long sequences of input data [15].

Zstandard Also developed by Yann Collet, the Zstandard algorithm was released in 2016 [16] and described in RFC 8788 [17]. With its arrival, Zstandard has managed to dethrone the long-reigning DEFLATE algorithm as the leading algorithm with a similar balance in compression level and time. Since the release of gzip and its subsequent growth as one of the most common DEFLATE implementations, newer algorithms have usually improved either compression speed or effectiveness, rarely both. While not as fast as Collet's earlier LZ4, Zstandard outperforms gzip/DEFLATE in compression ratio, compression speed, as well as decompression speed. With its background at Facebook, Zstandard is designed for modern big data operations and scalability requirements. This is achieved through a similar compression level setting as other tools, allowing users to select the compression speed and level needed for the application.

2.1.1.3 Limitations

A limitation of lossless compression algorithms is that there will always be some input which is not compressible to a smaller size. This means that a general, lossless compression algorithm which can successfully compress any given input is impossible [18, Section 3.3]. In order for a lossless compression algorithm to be reversible, a necessity for decompression, there must be a single unique input for every possible output. However, as the space of possible inputs is larger than the space of compressed outputs, by the pigeonhole principle, a compression algorithm cannot successfully compress every possible input.
To counteract this problem, existing lossless compression tools exploit additional assumptions about the input data. If the format of input data is known in advance, e.g. an algorithm for lossless video compression, certain patterns and redundancies can be expected to exist, and the algorithm can specifically target these structures for compression. This restricts the inputs which can be effectively compressed to those matching these assumptions, while allowing the algorithm to compress these specific files significantly better. In more general purpose compression algorithms, such as the ones described above, fewer such assumptions are made regarding the input data. Thus, a wider variety of data files can potentially be compressed, but less effectively than if purpose-built compression algorithms were applied.

2.1.2 Lossy compression

While lossless compression aims to be fully reversible, allowing the input data to be reconstructed completely, there are many applications which do not have this requirement. In these cases, so-called lossy compression algorithms produce a significantly smaller output by approximating the input data. While this inexact output is not reversible to recover the original input data, it allows for much stronger compression at the cost of losing detail. Lossy compression is commonly employed in situations where exactness is not required, but the achieved reduction in data size

or processing time are desired. Common examples include JPEG image compression or MP3 audio compression, which are specifically developed for their respective domains. Within data pipelines, particularly in continuous data streaming applications, performing such compression on-the-fly can provide the desired low latency needed to support real-time applications. For example, Havers *et al.* [19] demonstrate the use of piecewise linear approximation (PLA) within the *DRIVEN* data management framework to compress data before transmission over the limited bandwidth available on vehicular networks, building upon earlier work by Duvignau *et al.* [20] which adapts the PLA technique to a continuous data streaming context. PLA constructs linear segments of points, which can be communicated more efficiently than the underlying raw data, while controlling loss of precision by guaranteeing a maximum error bound. However, within the requirements of the work conducted during this thesis, lossy compression is not applicable specifically because the focus is on pipelines that allow recovering the original, unaltered data.

2.2 Apache Airflow

Apache Airflow [21], hosted by the Apache Software Foundation, is an open-source orchestration platform for implementing and executing workflows. Various workflow orchestration tools exist in the open-source software ecosystem, such as Apache Oozie[1], Azkaban[2], Luigi[3], or Argo Workflows[4]. However, Oozie, Azkaban, and Luigi all focus purely on orchestrating data processing workflows within Apache Hadoop clusters, limiting their applications outside this environment. Similarly, the Argo suite of tools, which includes Argo Workflows, specialises in orchestration of containerised workloads in Kubernetes environments [22]. As such, Airflow is selected for the implementation of the pipeline due to its flexibility, as it integrates with a wide variety of systems and protocols, and scalability due to its modular architecture. As an *orchestration platform*, Airflow is then responsible for the scheduling and execution of workflows. However, Airflow itself is not processing system, and in its role as an orchestrator, delegates such workloads to dedicated external systems. Orchestration provides a single centralised point of scheduling to coordinate various such external systems and workflow steps, as opposed to distributed job coordination with choreography protocols.
Airflow workflows are defined in code, making them highly flexible and dynamic. This allows Airflow to model, schedule, and execute arbitrarily complex workflows from within a single tool. Further, a web-based front end allows users to observe and monitor workflow executions in a clear manner. This section introduces some technical background on Airflow, including the workflow modelling technique as well as the main components of the Airflow system. Main components include the *scheduler*, responsible for scheduling tasks for execution, and the *worker*, where said tasks are executed. Further, a message queue is used for communication between scheduler and worker, and a metadata database is used to store persistent data.

[1] `https://oozie.apache.org/`
[2] `https://azkaban.github.io/`
[3] `https://github.com/spotify/luigi`
[4] `https://argoproj.github.io/workflows/`

Figure 2.1 shows an overview of the Airflow architecture; the various components are described in more detail below.

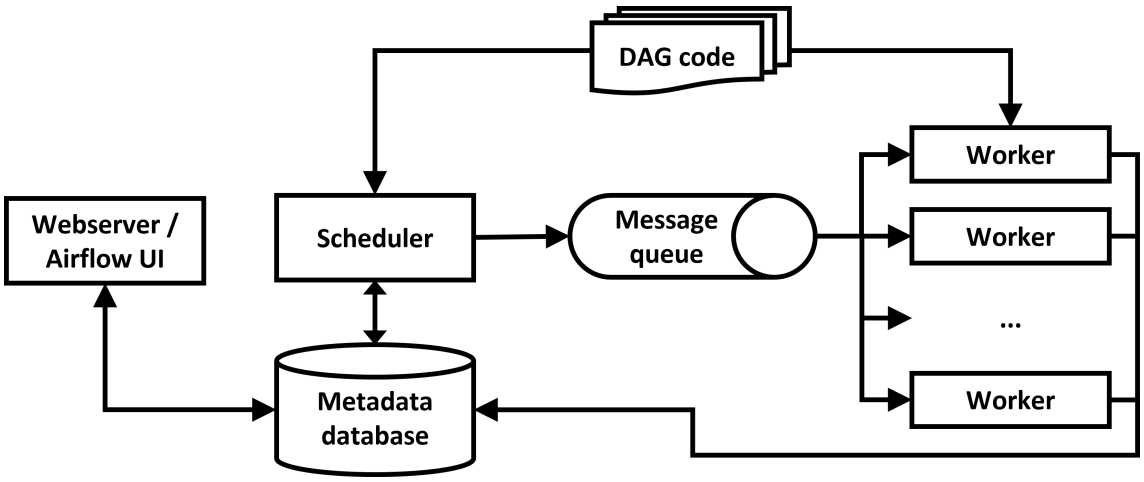


Figure 2.1: Airflow architecture overview, showing the main components of the system and their interactions. Derived from [23].

2.2.1 Modelling workflows

To enable reasoning about the structure of a complex workflow, the latter can be modelled as a set of tasks and the dependencies between pairs of these tasks. Airflow makes use of *directed acyclic graphs* (DAGs) to build a complete model of the workflow from these tasks and their dependencies. The workflow graph is composed of nodes for each task and directed edges to encode dependencies. Edges are directed, such that for any two tasks A and B connected by an edge $A \rightarrow B$, the execution of task B depends on task A to be completed successfully. B is said to be downstream of task A, which, in turn, is said to be upstream of task B. Task B cannot begin execution until task A has completed successfully so that its output or produced results are available. Because the graph is acyclic, dependency loops between tasks are forbidden. Thus, by resolving dependencies of all tasks, a well-behaved execution schedule can be derived algorithmically. This schedule guarantees that tasks will run in a valid order such that all dependencies of a task are met before it is executed.

2.2.2 Scheduling

The Airflow scheduler is a centralised component responsible for checking whether any DAGs should be executed at regular intervals. DAG execution can be triggered by various different triggers, such as automated scheduling similar to `cron` on UNIX-like operating systems, Airflow REST API calls, or manually via the web interface. The scheduler then creates a *DAG run*, an instantiation of the abstract workflow model DAG. This DAG run is associated with a unique DAG run identifier, making it possible to trace detailed workflow history, logs, and enables observability for users. Once triggered, the DAG run is added to the queue of currently running

DAG runs persisted in the metadata database, and is shown in the web interface as having a status 'running'.
A second scheduling loop is then used to check whether there any running DAG runs contain tasks whose dependencies are all met. If such a task is found, a task instance is created and submitted for execution. Once execution completes, an exit status for the task is determined and the scheduler checks whether any other tasks have met dependencies. The DAG model thus allows Airflow to schedule arbitrarily complex workflows, while providing a flexible scheduling platform for users.

2.2.3 Execution

When the scheduler submits a task for execution, a message is submitted to a message queue system, such as Celery [24] or Redis® [25]. Airflow worker processes are connected to this message queue and one of them will retrieve the message and execute the corresponding task. Workers are independent processes, capable of executing multiple tasks simultaneously. This allows worker processes to be scaled dynamically, for example by using auto-scaling features of the Kubernetes platform [22], a prominent system for managing containerised applications. The auto-scaling feature can add and remove workers to scale the Airflow system elastically based on current load, facilitated by the underlying message queue system for coordination.
As Airflow mainly aims to be an orchestration platform, workers are not expected to be performing intensive computational tasks, unlike typical computing systems such as Slurm [26] or Apache Spark [27] which aim to collect and coordinate the combined processing resources of multiple servers into a single, more powerful processing system. Instead, workers merely provide a managed execution unit to perform simpler tasks, or to dispatch heavier processing jobs to dedicated external systems. For example, a common use-case for Airflow is the training of deep learning models. The workflow can involve heavy data preprocessing steps such as ingestion, structuring, and cleansing, followed by training and evaluating models, and finally storing the model with the best performance. Importantly, intensive tasks such as model training do not run directly in the Airflow worker; rather, the Airflow scheduler will submit a task to the Worker, which in turn will submit a training process to a dedicated system with the necessary software and hardware components for the task.

2.2.4 Resource management

Because scheduling and execution of tasks can be somewhat detached from the target environment in which the workload executes, uncontrolled scheduling of tasks can overwhelm the target system or drain particular resources. Thus, Airflow provides several mechanisms to control scheduling behaviour in situations where resource management is necessary. Amongst the simpler techniques is limiting parallelism within and across instances of a particular workflow DAG. This restricts how many tasks of a particular DAG run can be running in parallel, but also across all DAG runs of the same DAG. In this way, overloading due to excessively many parallel

tasks can be prevented.
However, to enable more powerful management of resources, Airflow provides *pools*. Pools are used to encapsulate the state of each resource that needs to be managed. Each task which utilises these resources is assigned to the corresponding pool. Pools, effectively semaphores, have a specified capacity and any task assigned to the pool must wait until enough capacity is available before it can be executed. Task are queued in first-in, first-out (FIFO) order, which prevents starvation scenarios where a particular task would never get access to the resource. This allows workflow parallelism to be controlled based on resources used, and the same pool can also be used for tasks from different DAGs that all utilise the same resources.

2.2.5 Reliability and scalability

As an orchestration platform, Airflow both aids in designing reliable workflows, but must itself also be a reliable and scalable platform. As described previously, Workers can be scaled on demand to support orchestrating several thousand tasks. However, the Scheduler as a centralised component can become a bottleneck and point of failure. To this end, Airflow supports running multiple Scheduler instances, to enhance both fault-tolerance as well as performance. In systems with large workflows and many tasks, a single scheduler may not be capable of processing all scheduling decisions. In these situations, the workload can be distributed across additional instances of the Scheduler component. To correctly coordinate these schedulers, Airflow relies on synchronisation primitives provided by the metadata backend database used to store persistent data. By locking particular rows, a Scheduler instance can protect a critical section before making scheduling decisions, thereby preventing data races. While Airflow itself must be reliable and scalable to support arbitrary workflows, it also provides features for improving reliability of the implemented workflows. Automated monitoring of the executed tasks allows detecting failed tasks and cancelling downstream tasks as their dependencies are not met. Thus, workflows can be designed to never process partial or incomplete output produced by a failed upstream tasks. Automatic failure notifications can be emitted to alert operators and engineers of such workflow failures, who can then utilise the recorded logging information from the failed task to resolve the issue. The DAG run can then be resumed from the point of failure, rather than re-executing all tasks which have already been successfully completed.

3
Problem

Data is often generated and processed in different locations, and transferring large volumes of data between systems and geographical locations is a long-standing problem. Now, growing data volumes in the age of Big Data further highlight the importance of timely and scalable data transfer systems. As resource requirements needed to process Big Data volumes begin to exceed processing capabilities and communication bandwidth of computation systems, new distributed processing paradigms such as cluster, edge, and cloud computing have increased in prevalence.
A common life cycle for data involves a sequence of steps: generation or capture at a source location, transfer to centralised processing infrastructure where is it analysed, and finally optional storage or archival for further offline processing. In practice, such data management workflows are commonly implemented by data pipelines, which automate the execution of various steps such as transfer, integrity verification, or analysis.

3.1 Case study: "The Brønnøy Project"

As a motivating example, which also serves as the main case study in this thesis, a Big Data pipeline for vehicular data is studied. In the Brønnøy project, Volvo Trucks and Volvo Autonomous Solutions, have partnered with Brønnøy Kalk AS to develop an autonomous transport solution to enhance the efficiency of the Brønnøy Kalk limestone mine in Velfjord, Norway [28], [29]. Autonomous trucks will transport limestone from the mining pit along a predefined 5 kilometre route to a port for crushing and shipping. As the project nears its delivery date, multiple trucks outfitted with autonomous driving capabilities have been delivered to the customer site, where regular tests are performed. These test runs aim to verify various aspects of the autonomous driving system, such as behaviour, safety, and performance.
During these tests, a vehicle records detailed data from various systems and sensors. This data is used by developers and engineers for debugging and verification, such as by simulating and replaying particular scenarios. However, the Volvo staff and data centres are primarily located in Sweden rather than Norway. In order for the data to be available for analysis and processing by engineers, it must first be transferred from the remote site to Volvo's data centres.
A high-level illustration of the data flow in the case study is shown in Figure 3.1. Data is recorded on board the vehicles onto removable disks. Once logging is completed, data is transferred to an upload station, a computer located at the remote site with a large storage capacity. Once data is stored on the upload station, it must

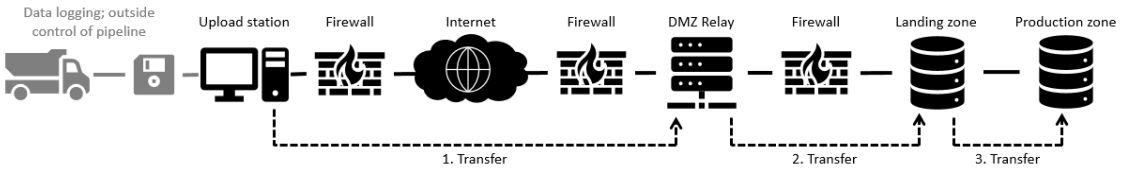


Figure 3.1: High-level overview of a data flow from a remote site to data centres; using the Brønnøy project case study as an example.

be transferred to a data centre where it can be processed and analysed. Managing transfer of such large volumes of critical data as part of a major industrial project involves a wide variety of challenges.

3.2 Timely data delivery and pipeline throughput

Processing data and extracting knowledge as soon as possible, possibly in real-time, is a common requirement for various applications, such as IoT-based control systems or advanced driver assistance systems in vehicles (challenges treated in e.g. [30]–[33], approaches surveyed in [34]–[36]). To enable short lead times between data generation and analysis, the transfer must deliver data in a timely manner, i.e. with minimal additional time introduced by the data transfer. For a particular data item, the duration of this transfer from end-to-end is referred to as *latency* or *pipeline latency* in the context of transfer pipelines. In a data transfer process under sustainable load, average latency of a transfer is inversely proportional to *transfer throughput*. Thus, to reduce latency for high-volume data transfer pipelines, throughput of the transfer must increase.
In the context of the case study at Volvo, pipeline latency is an important matter for developers and engineers working on the Brønnøy project. Shortening the lead time between occurrence of potential issues during testing and engineers having access to captured data is important to support development of the product. Thus, improving throughput in the transfer pipeline is one of the primary targets to reduce pipeline latency (reliant on an appropriate scheduling and queueing discipline [37]) of this case study. However, all data must be transferred over the Internet, which places some restrictions on available bandwidth, and complicates timely delivery.

3.3 Maintaining performance under increasing load

As data volumes grow, higher demand is placed on existing computational resources. These resources therefore need to be utilised efficiently in order to uphold system performance to an acceptable level. This allows the system to remain functional and performant as the workload scales up.
In the case of the Brønnøy project, data arrives at the upload station in a daily burst. Even during testing with a reduced number of vehicles and operating hours, these bursts can exceed 1 TB. As further vehicles are taken into operation, this vol-

ume is expected to increase further, accordingly placing higher demand on existing hardware infrastructure to process the larger and more frequent data.

3.4 Qualitative requirements

Alongside the primary problem of transferring data in an efficient and scalable manner, enterprise data operations often involve additional requirements in data pipelines. Data is becoming an increasingly important asset in organisations employing data-driven product development. For example, the development and enhancement of autonomous driving systems such as the one in Brønnøy requires vast amounts of data. Managing and safeguarding this data is critical for such businesses.

Data veracity To this end, an important characteristic of a Big Data is its veracity. Data veracity has to do with the perceived reliability of the data in a big data environment. Veracity is closely related to data quality — low data quality reduces the overall value of a big data set and, by extension, the value of conclusions derived from the data. Analysis built around inaccurate data is less likely to be trustworthy and reliable, which reduces data veracity.

To support data veracity within the context of a data transfer pipeline, strong indicators for reliability are data integrity and pipeline robustness. Data should remain intact throughout the transfer, without experiencing degradation or corruption. Maintaining the integrity of the transferred data is an essential part of any data transfer system, and automatic verification can provide strong guarantees in this regard, thereby improving veracity. Beyond integrity validation, a robust pipeline with few outages is more likely to be perceived as trustworthy.

Data governance Data governance is another critical aspect of a successful big data management operation. Controlling and managing data at all stages of its lifecycle, from recording, through transfers and processing, analysis, and archiving or removal, is essential to manage the volumes present in modern big data systems. Recording the processing history, commonly referred to as *lineage*, of data allows tracing the full lifecycle of a data file. Additional metadata can be used to prepare indexes for searching for particular data more efficiently than scanning the complete big data set for every query.

Data security Further, due to the value of data as an asset to an organisation, corresponding security measures are required in order to protect the data. In the context of data transfer pipelines, data needs to be protected from various potential threats both at rest and in transit. Unauthorised access, modification, or deletion of data would case severe problems and incur high costs. Particularly, large-scale transfer pipelines routing data over the internet or through public clouds could expose this data to a large number of potential adversaries.

3.5 Evaluation metrics

This section presents a number of evaluation metrics for analysing the performance of a data transfer pipeline. A pipeline consists of a sequence of N *processing steps* which data must pass through. Data may travel through the steps of a pipeline either continuously as a stream of separate data items, or in *batches* consisting of several data items. Further, each pipeline step has an individual *service rate*, measured in MB/s, describing the average rate at which it can process data. Each pipeline step processes a single item or batch at once, but different steps may process different batches in parallel. Figure 3.2 illustrates the execution timeline of a pipeline consisting of three steps for three batches. Note the *queueing delay* in the execution timeline of batch B before each task, as batch A has not completed processing by those tasks yet. For the purposes of this thesis, the queueing discipline of such queues is assumed to be first-in, first-out (FIFO) order.

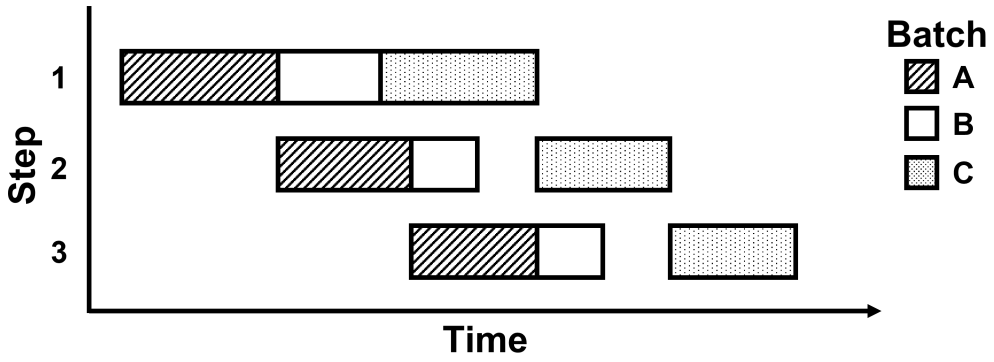


Figure 3.2: Execution timeline of a batched pipeline with three tasks ($N = 3$) and three batches of data. Batch B is smaller in size than batches A and C.

In order to judge pipeline delivery timeliness, *batch latency* of a batch b is measured as the time in seconds taken for the batch to traverse the complete pipeline. Batch latency, denoted l_b, is equivalent to the sum of the queueing delay before and time spent executing each step of the pipeline:

$$\begin{aligned} \text{Batch latency } l_b &= \text{Time}_{\text{End}} - \text{Time}_{\text{Start}} \\ &= \sum_{i=1}^{N} q_i(b) + \sum_{i=1}^{N} t_i(b) \end{aligned}$$

where $q_i(b)$ is the queueing delay (in seconds) experienced by batch b before beginning processing by step i, and $t_i(b)$ is the *processing time* (also in seconds) taken for batch b to be processed by step i. Figure 3.3 visualises this calculation for a batch in the example timeline from Figure 3.2.
Batch latency is directly related to the *batch size*, the total size of data in a batch. Therefore, a normalised metric is necessary in order to enable comparison between batches of different sizes. To this end, the *batch throughput*, commonly measured in MB/s, for a batch b is defined as:

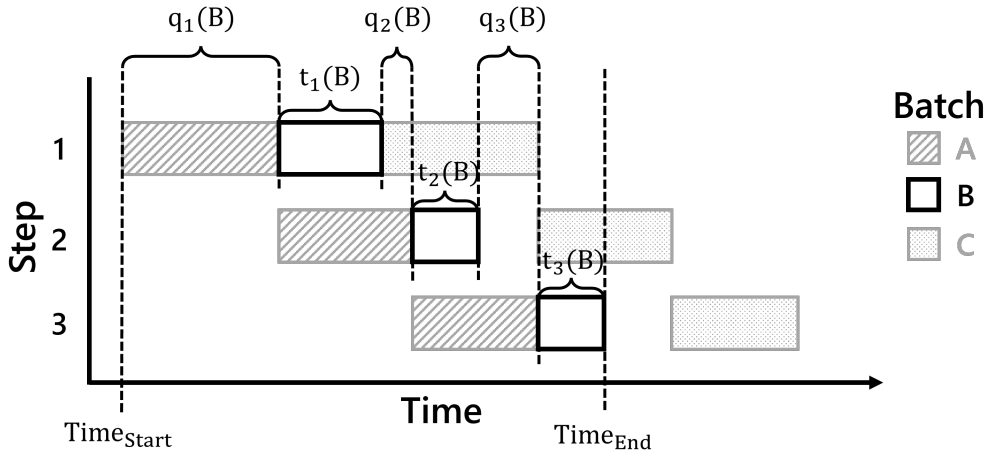


Figure 3.3: Batch latency calculation for batch B.

$$\text{Batch throughput } R_b = \frac{\text{Size}(b)}{l_b}$$

Similarly, the *pipeline throughput*, also measured in MB/s, describes the average rate at which the pipeline transfers data during a specific time interval. Consider a period of continuous pipeline operation of duration T during which B batches are transferred. The real time duration T starts as the first batch begins processing by the first task, and ends when the final batch completes processing by the final task — see Figure 3.4 for a visual representation. The total data volume is the sum of the sizes of all B data batches processed during the time interval T. Pipeline throughput during this period is then defined as:

$$\begin{aligned}\text{Pipeline throughput } R_P &= \frac{\text{Total data volume}}{\text{T}} \\ &= \frac{\sum_{i=1}^{B} \text{Size}(b_i)}{\text{Time}_{\text{End}} - \text{Time}_{\text{Start}}}\end{aligned}$$

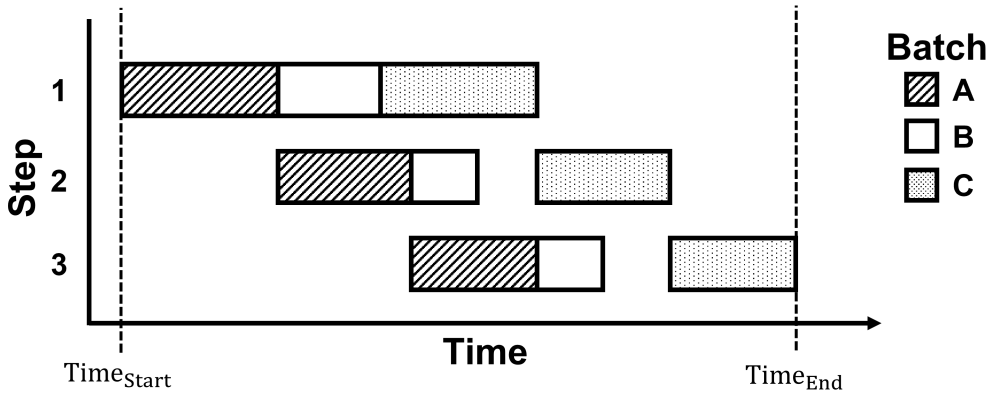


Figure 3.4: Real time duration to process all three data batches, for determining pipeline throughput.

By considering R_P for the duration of 24 h, an upper bound on the data volume which can be transferred by the pipeline during a day can be estimated. This *maximum daily data volume*, in GB, can be compared against the expected daily volume of generated source data in a pipeline application, and allows classifying whether pipeline performance is sufficient for the application.
In order to systematically address performance bottlenecks in a pipeline, the underlying cause for the bottleneck needs to be identified. Resource *utilisation* measurements of the various hardware resources used by pipeline steps are an important tool when diagnosing performance limitations. Relevant metrics include utilisation levels of the processor, system memory, disk bandwidth, and network bandwidth. Utilisation of these resources is measured as how much of their maximum capacity is consumed by the workload at a particular instant.

4

Challenges and possibilities

As presented, the problem of efficient big data transfer involves a large number of challenges from a wide variety of technical domains. The complexity of such a system cases a combinatorial explosion of the parameter space for design choices. Thus, for the purpose of this thesis, focus is placed on a selection of techniques for maximising pipeline performance in terms of throughput and latency.

4.1 Pipeline performance

To ensure timely delivery of large volumes data, a data transfer process must sustain a suitably high throughput. Data transfer throughput in computer systems is affected by a large number of factors, primarily the available bandwidth of the communication link. Maximising the bandwidth utilisation to achieve the highest possible data transfer rates also requires the sender and receiver to be able to process the transmitted data at this same rate. Modern computer systems contain a variety of hardware and software components to manage the resources involved in such tasks. Making efficient use of these components and resources is essential for maximising data transfer rates in the complete system.
As pipeline load increases when data volumes grow, the system will eventually reach one or more resource limitations preventing it from scaling further. Identifying and addressing these bottlenecks is therefore essential for scaling a data pipeline. Generally, there are two main options for scaling a processing system: hardware scaling and performance tuning.

Hardware scaling Hardware scaling simply introduced more resources to the system where a need has been identified. This allows increasing the workload further by providing more resources which can be consumed. A distinction is made between adding resources horizontally (scaling out), or vertically (scaling up). In horizontal scaling, additional complete nodes are added to an existing system. In practice, this may involve deploying additional servers to a storage cluster to increase capacity or throughput, or installing a secondary upload station at a customer site. This approach inherently requires an increases parallelism in order to make use of the additional nodes, as the workload is now distributed over more nodes. On the other hand, vertical scaling introduces resources into existing systems, such as upgrading a CPU, system memory, or network connection. This allows sequential processes to consume more resources, thereby avoiding additional complexity of parallelising the workload.

Performance tuning Besides introducing further resources to scale a processing system, performance tuning aims to tune an existing system to require fewer resources to perform the same amount of work. By making more efficient use of existing resources, more work can be performed in the same system. However, performance tuning requires detailed understanding of the system and various complex interdependencies between its components. Identifying and addressing bottlenecks requires studying various trade-offs within the system. On the other hand, hardware scaling is often simpler and cheaper to implement, but may yield diminishing returns.

4.2 Zooming in to case-study

Within the scope of this thesis and the case study in the Brønnøyproject, a number of known constraints and performance bottlenecks guide the direction of explored options for achieving high performance and efficiency in the developed pipeline.

4.2.1 Systems and resources

Figure 4.1 provides a high-level overview of the various systems and transfer hops involved in the pipeline.

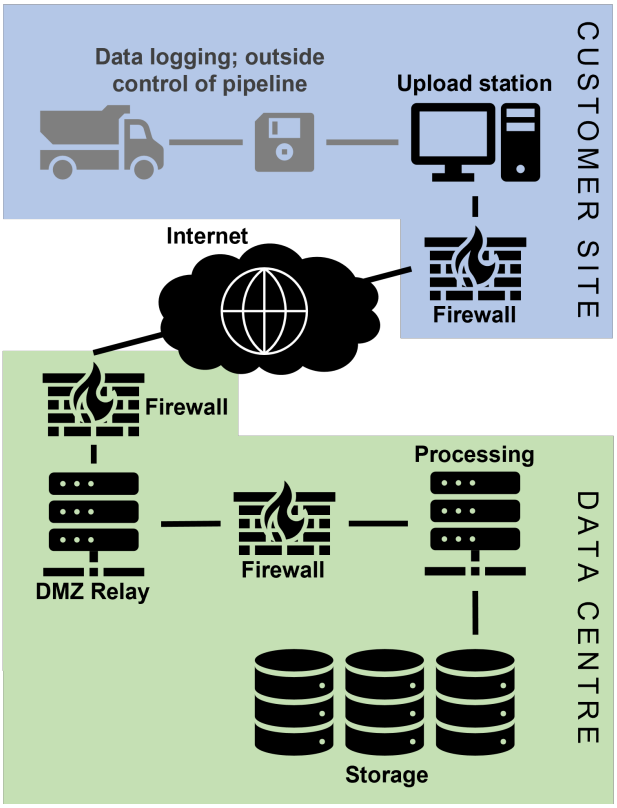


Figure 4.1: Overview of systems comprising the data transfer infrastructure.

At the remote customer site, the installed upload station is effectively finalised and

hardware has been tested and validated. The deployed hardware includes enough resources for the existing workload, but may potentially reach bottlenecks as data volumes grow. Although hardware scaling would be an option to address this, the focus of this work lies in performance tuning within an existing system. Thus, any performance enhancements at the remote site should be achieved by utilising present resources more efficiently. On the other hand, processing performed within Volvo data centres is more easily scaled as hardware resources are readily available and modern virtualisation technologies facilitate rapid scaling of workloads.

Alongside the systems between which the data moves, transfer performance evaluation must also consider characteristics of the network links over which these transfers occur. The first such transfer in the pipeline takes place from the upload station, over the internet, to Volvo data centres. Despite installation of a dedicated 1 Gbit/s fibre-optic link to the Brønnøy site in Norway, preliminary tests under realistic loads show sustained data transfer rates of only around 65 MB/s (0.52 Gbit/s, corresponding to a utilisation of 52 % of the available bandwidth). The other transfer hops, which take place within the data centre, achieve 2 to 5 times higher throughput. As such, the limited bandwidth along the first transfer forms a significant performance bottleneck which must be appropriately addressed by a pipeline solution.

4.2.2 Baseline approach

As the Brønnøy project has already been under development for several years, the demand for transfer of logged data has accompanied it for equally long. Currently, the transfer is performed using an ad-hoc solution consisting of several manual steps. First, a data management engineer begins the transfer from the upload station to a host functioning as a relay in the demilitarised zone (DMZ)[1]. Once this transfer is complete, a second step is invoked to transfer data from the DMZ relay to a landing zone for arriving data in the data centre. Finally, a scheduled process structures the incoming data files and places them in the final storage location where engineers can access it.

This approaches serves as a natural baseline for performance comparisons, as it purely involves transferring the data as-is from source to target. However, due to its simple nature, the approach suffers from a variety of problems which affect performance. Primarily, a substantial amount of manual effort is required to perform the various steps in the correct order. As arrival times for data batches are not known in advance, there may be extraneous delays between data arriving and the manual start of the transfer process. Automation of the transfer can reduce incurred waiting times and improve latency. Further, the pipeline only performs transfer of data files. This is not only problematic from the perspective of data veracity, governance, and security, but also because there exist various areas where efficiency can be improved to achieve better pipeline performance.

[1] A demilitarised zone is a network segmentation technique used to isolate an internal network from direct access by untrusted hosts on the internet, while allowing external access only to particular services deployed within the DMZ.

4.3 Possibilities for efficiency improvements

Focus in this thesis is placed on two approaches to improving efficiency and performance in data transfer pipelines. Inspired by the shortcomings identified in the straightforward baseline implementation seen in the case study, the main techniques for improving efficiency are *data compression* and *in-memory processing*. Introducing data compression involves studying and balancing various trade-offs to improve throughput, and in-memory processing for better efficiency and scalability.

4.3.1 Data compression

Data compression is a common technique to improve throughput in systems with communication bandwidth constraints. By applying a compression algorithm to the input data, a compressed output of smaller size is obtained. Compressing data before transferring it can increase the throughput of the transfer operation beyond the physical bandwidth of the channel. Although the traffic rate is identical when transmitting compressed data, the effective volume of raw data transferred can be much higher.

Different compression algorithms achieve varying compression and decompression throughput (measured in MB/s), different compression ratios, and different resource utilisation. Compression and decompression throughput are the rate at which raw input data is consumed by the compression algorithm, and recovered data is emitted by the decompression algorithm, respectively. The *compression ratio* describes the ratio between the size of the raw and compressed data, and is used to describe how effectively the algorithm has compressed the data. A compression ratio $r > 1$ indicates that the compression has achieved a reduced the size of the data by factor r. Compression ratio is heavily dependent on the nature of the input data and the algorithm used, as certain algorithms are able to compress particular types data better than others.

Data in the pipeline studied at Volvo consists of various kinds of data from different sensors and signals which are collected in larger, heterogeneous files. This suggests the need for selecting a general purpose compression algorithm which compresses this mixture of data, rather than a specialised solution for a single specific kind of data. Further, the tools used by developers and engineers require the data files to be in the original format, which necessitates a lossless compression technique.

When integrating compression into a data pipeline, pipeline latency and throughput are primarily affected by the throughput and compression ratio of the chosen algorithm on the data present in the pipeline. The challenge lies in identifying a suitable algorithm and compression level which balances the *trade off* between additional resource and time costs for compression and decompression against the performance improvements of the transfer operation.

4.3.2 In-memory processing

The described compression and decompression operations rely on a variety of resources, including processor time, system memory, and storage bandwidth. The

available capacity of these resources for compression and decompression work is dependent on system load by other tasks. Contention may be high when a pipeline processes multiple batches in different steps in parallel. Thus, avoiding usage of such demanded resources by modifying the pipeline to harness unused capacity of another resource is a central strategy for performance tuning. These heavily utilised resources, which create performance bottlenecks, must be identified and addressed. In modern computer systems, CPU and system memory operate at significantly higher data rates than high-capacity storage disks. Thus, a pipeline of processing steps operating on the same data will suffer if input data must be read from slower storage. Various possible options exist to alleviate the performance impact of slow storage. Hardware scaling of storage by upgrading to a faster medium with higher bandwidth is one such option, but is not possible within the scope of the case study. An option employed by many operating systems to improve disk access performance is to use free system memory to cache data previously read from the disk — a technique known as *page caching*. However, page caching is not suitable for processing volumes of data larger than the cache capacity, as early data will have been evicted by the time it is used again, and will result in a high rate of continued cache misses while performance appears as if no caching was present at all. Instead, another option is to let processing steps communicate directly via *pipes* — in-memory buffers for inter-process communication. In doing so, intermediate disk reading and writing operations can be avoided, as data is shared between cooperating processes directly in memory. Upcoming Section 5.2.3 describes this technique further as well as implementation of such pipe-based in-memory processing into a data pipeline.

5
Approach

This chapter describes the chosen approach for addressing the previously described challenges and evaluating possible solutions. The case study within the Brønnøy project at Volvo Trucks takes a central role to explore and evaluate techniques for enhancing pipeline performance.

5.1 Pipeline construction

Based on the challenges and requirements presented, a pipeline design is developed to implements the desired features. The developed pipeline design is based on the requirements identified as part of the case study at Volvo Trucks. However, the design also needs to achieve necessary flexibility to enable the evaluation of various performance improvement techniques for this work.

5.1.1 Structure

The principal pipeline structure is illustrated in Figure 5.1, based on the desired features and identified requirements in the case study.

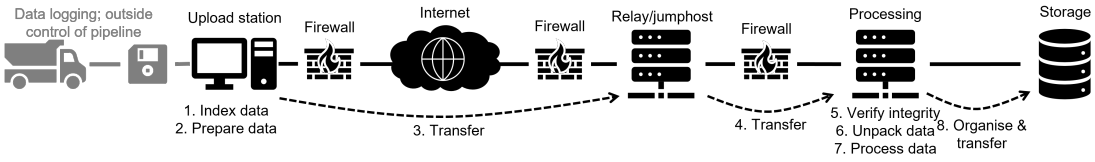


Figure 5.1: High-level overview of pipeline steps.

The pipeline structure is a linear sequence of tasks, which can be categorised into three main groups based on their function and location: preparation on the upload station, transfer, and processing in the data centre. Data in the pipeline system is processed in batches in a so-called *batched streaming* model. This model allows for improved parallelism compared to a batch processing system, where all data is processed as a singular batch. On the other hand, a streaming system continuously transfers smaller volumes of data. While a streaming model inherently encourages parallelism, data must be in a format suitable for streaming in such smaller segments. However, in the case study, data arrives in larger bursts of opaque files. Thus, as data is not continuous and the data file format can vary between different test sites and logging systems, a full streaming model is not suitable. Rather, an inbound

chunk is split into batches, which are then submitted for transfer in the designed pipeline.

Preparation First, all data in a batch on the upload station is indexed and prepared for transfer. This includes registering every file into a cataloguing database for traceability and data governance purposes. The recorded metadata about each file includes size, modification time, and a checksum to enable integrity verification at the final stage of the pipeline. Checksums are computed using the OpenSSL's SHA256 digest functions for their ability to utilise SHA hardware instructions on modern x86 CPUs to accelerate SHA256 computations efficiently [38]. The next steps of preparation include compression using a selected compression strategy and encryption. The method used to select an appropriate compression strategy is described in Section 5.2.2. Compression takes place before encryption to reduce the overall processed data volume; by compressing first, there is less data volume to encrypt, and encryption will therefore take less time. Encryption takes place to protect data from unauthorised reading during the transfers.The compression step is placed before encryption in the pipeline in order to improve effective throughput of the encryption process by operating on smaller input data. Finally, another checksum of the compressed and encrypted file is taken before it is ready for transfer. This additional checksum allows verifying data integrity without requiring decrypting and decompression of potentially altered data.

Transfer Once data files in a batch have been prepared, transfer can begin. For the particular application studied in the case study, data must be transferred over the internet. A variety of cybersecurity measures are required in order to protect data and systems. Alongside cryptographic protocols to protect data packets in transit, a relay host in a demilitarised zone (DMZ) can be used. The DMZ is a separate network containing hosts and services which should be accessible from untrusted hosts on the Internet. Hosts in the DMZ network cannot access the internal network which is to be protected. The reverse can be permitted by firewall configuration, and then allows internal systems to retrieve data files stored on the DMZ relay. This introduces an additional isolation layer between an internal network and the Internet. Thus, for the data stored on the DMZ relay to arrive in the data centre, a second transfer from the DMZ relay is necessary. While this practice requires additional steps and increases latency, it is a necessary network security measure which cannot be avoided in an industrial data pipeline deployment. Data arriving from the DMZ relay is placed in a landing zone on network-attached storage, from where subsequent processing tasks can retrieve files for processing.

Processing & storage Once data files have reached the landing zone, integrity after the transfers must be verified before any processing takes place. The previously recorded checksums of the compressed and encrypted files are verified, and pipeline processing only continues if all data is valid. Next, data is decrypted and decompressed to restore the original raw data files. Finally, data is organised in order to improve accessibility for developers. Files are grouped by the vehicle identifier of

the vehicle they were captured on, as well as the date of the recording. Data is then placed in the final target location and made accessible to users.

5.1.2 Implementation

The designed workflow can trivially be modelled as a set of tasks and dependencies, which makes it well suited for Apache Airflow. Airflow is selected as the orchestration platform for the implementation from the various tools present in the domain of free and open source workflow management systems. As noted in Section 2.2, other workflow management tools tend to focus on workflows in a particular environment such as Hadoop or Kubernetes. Airflow instead aims to be a general purpose tool with high flexibility. Integrations for a wide variety of systems are provided, including various types of databases, HTTP calls, or remote access to systems via SSH. These integrations make Airflow a powerful framework for implementing data pipelines, alongside functionality for monitoring and alerting, credentials management, and a powerful user interface. Further, various Airflow as a Service offerings by several large companies, including Google Cloud Platform[1] and Amazon Webservices[2], serves as another marker for the quality, maturity, and scalability of Airflow as a workflow orchestration platform.

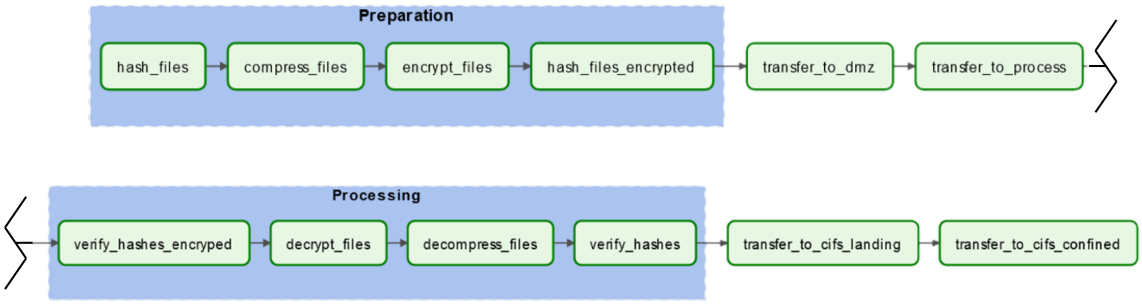


Figure 5.2: Airflow DAG implementing the proposed transfer pipeline. Tasks are grouped (blue boxes) to visually highlight the stage of the pipeline they belong to.

Following the Airflow paradigm of pipelines-as-code, the data transfer pipeline workflow is implemented as an Airflow DAG, shown in Figure 5.2. Each group in the previously described structure is divided into several steps, which correspond to tasks in this DAG. The tasks implement the following functionality of the pipeline, while also updating corresponding metadata in the file cataloguing database:

`Preparation` — Prepare data for transfer

`hash_files` Calculate the SHA256 checksum of each file in the batch.

`compress_files` Using the selected compression strategy, compress each data file in the batch.

`encrypt_files` Encrypt all data files in the batch.

[1] Google Cloud Composer: `https://cloud.google.com/composer`

[2] Managed Workflows for Apache Airflow: `https://aws.amazon.com/managed-workflows-for-apache-airflow/`

hash_files_encrypted Calculate the SHA256 checksum of each encrypted file in the batch.

transfer_to_dmz Transfer data files from Brønnøy to the Volvo DMZ relay.

transfer_to_processing Transfer data files from the DMZ relay to the internal network.

Processing — Unpack and structure arriving data.

verify_hashes_encrypted Verify integrity of inbound data files before proceeding by comparing SHA256 checksums.

decrypt_files Decrypt all data files in the batch.

decompress_files Decompress data files.

verify_hashes Compare SHA256 checksum with original data to verify integrity after all processing steps.

transfer_to_cifs_landing Transfer data to a landing zone network attached storage.

transfer_to_cifs_confined Transfer data to the destination zone network attached storage.

The Airflow scheduler triggers a run of this DAG for every batch of data to be transferred. In order to control concurrency between these parallel DAG runs, Airflow pools are employed to control access to certain resources such as the CPU on the upload station, various network links, or the CPU on the processing server. All pools are initialised with a capacity of 1 slot in this project. This limits execution parallelism to one active instance per task, as tuning such parallelism is beyond the scope of this work.
A guiding principle during the implementation of the pipeline is to use widely available, stable utilities. Examples include using OpenSSL [39] for checksum computations, GnuPG [40] for data encryption, rsync [41] for data transfers, or OpenSSH [42] for remote authentication and interaction with hosts. As these tools are well-established in their respective fields, feature extensive documentation, and are well tested, they are very suitable for building a robust and efficient pipeline. Further, as these tools are widely available, deployment of the pipeline system in an industrial setting is further simplified by avoiding installation of additional tools and components.

5.2 Performance evaluation methodology

Once the pipeline is implemented and its functionality verified, the achieved performance level is evaluated. Maximising throughput in the transfer pipeline is one of the primary targets of this case study. However, all data has to be transferred via the internet, which places some restrictions on bandwidth as described in Section 4.2.2. As the full 1 Gbit/s bandwidth of the connection cannot be reached, transfers from Brønnøy to the DMZ are limited to around 65 MB/s. This degrades throughput and latency of these transfers and complicates timely delivery of data.

The pipeline aims to address this bottleneck by compressing data before transfer, but must in turn ensure that the time and resource investment into compression provides a benefit for performance.

5.2.1 Comparison with baseline solution

In order to put any quantitative performance measurements into perspective, performance of the current, naive baseline solution is also measured. The baseline solution consists only of the necessary transfers, without any of the desired additional "bells and whistles" such as logging, monitoring, or cataloguing transferred files for traceability purposes. Focus of the baseline solution is purely on maximising bandwidth utilisation of the bottleneck link (Brønnøy–DMZ) while there is data to be transferred. In doing so, performance is left on the table due to inefficient use of other available resources and strong boundness to a single resource, the physical bandwidth of the network transfer.
The new advanced pipeline includes several steps before and after the transfers to implement the various industrial requirements. Every additional step, such as checksum computation or encryption, must process the complete data set one more time per step. This introduces significant additional latency, thereby increasing the total duration of the transfer pipeline for every batch of data. Accordingly, overall pipeline performance will decrease compared to the baseline, unless measures are taken to improve performance again.

5.2.2 Selecting a data compression strategy

One key method for improving pipeline performance is data compression. By trading more processing time to compress data before transmission, reliance on bandwidth is reduced as effective throughput is increased by transmitting compressed data. Various data compression strategies exist, each with varying compression speeds and compression ratios for the particular type of input data. A variety of such strategies must be evaluated in order to select one which maximises the achieved performance improvement.
The main target area for performance tuning and resource efficiency improvements is the remote upload station, due to the difficulty in expanding available hardware resources at remote sites. The processing stage of the pipeline takes places within a data centre where resources are easier to scale, both horizontally and vertically. Thus, compression performance is more critical to evaluate than decompression performance, as the latter can more easily be scaled in the data centre in the future to improve performance.
Four compression algorithms are selected to be evaluated for use in the pipeline: (1) gzip, (2) bzip2, (3) LZ4, and (4) Zstandard. Selection of these particular algorithms is motivated by multiple reasons, summarised in Table 5.1, and is based on desired properties described in Section 4.3.1. Gzip is a very prevalent standard utility for data compression due to its prevalence and generally good balance between compression time and effectiveness. Bzip2 and LZ4 are included here because the on-board system which records the data on the vehicles in the Brønnøy project has

native support for these compression algorithms. As the vendor of the data logging solution includes these algorithms for compressing recorded data files, it suggests that these algorithms are suitable and perform well for the type of data at hand. The data logging system in fact supports compressing data directly on-the-fly while logging using either of these algorithms, but this feature cannot be used for other technical reasons. This implies that Bzip2 and LZ4 are suitable candidates for the type of data present in the case study pipeline. While Bzip2 is significantly slower than LZ4, it generally achieves better compression. Finally, Zstandard is a modern competitor to gzip, also aiming for a balance between compression speed and effectiveness but with better performance both in time-efficiency and compression ratio.

Algorithm	**Remarks**
Gzip	Balance between compression speed and effectiveness. Common standard utility.
Bzip2	Slow but strong compression. Natively available for the logging system.
LZ4	Fast but weaker compression. Also available natively, like Bzip2.
Zstandard	Balance between speed and compression, like Gzip. Modern competitor to Gzip with higher performance (both speed and compression ratio).

Table 5.1: Useful properties of the selected compression algorithms.

A *compression strategy* refers to the combination of an algorithm and a selected compression level. To make a well-founded selection for the strategy to use in the data transfer pipeline, the performance of each strategy must be determined. By using each strategy to compress the same input dataset and recording compression time and output size, the strategies can be compared based on speed and compression ratio. The best performing strategy can then be integrated in the data pipeline and used for analysing whether overall throughput has improved compared to the baseline scenario.

5.2.3 In-memory processing

The introduction of additional processing steps on the upload station also places higher demand on available resources. Checksum computation, compression, and encryption all require CPU time to perform their tasks. However, these processes must also read input and write output data. As several pipeline instances may run in parallel for separate batches, it is possible for several pipeline steps to perform disk input/output operations simultaneously — initial checksum calculation, compression, encryption, secondary checksum computation, as well as the first transfer all

need to read and/or write data from disk. Recording system resource utilisation is essential to evaluate pipeline requirements and bottlenecks. For these measurements, the System Activity Report (`sar`) utility of the `sysstat` performance monitoring suite [43] is used, because it enables nonintrusive data collection on the same host due with negligible resource footprint while active.

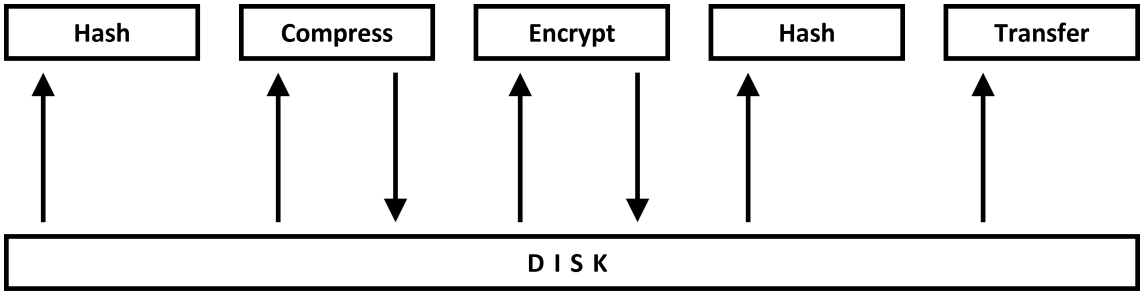


Figure 5.3: Read and write operations performed by the advanced pipeline.

One resource of particular concern is the data storage disk. An overview of the pipeline preparation steps and the read and write operations they perform is given in Figure 5.3. A total of 7 high-volume data read and write operations are issued while executing these five tasks concurrently. As pipeline instances each process different batches, the read and write operations issued to the disk access widely different files and regions rather than a sequential, contiguous region. This access pattern adversely impacts performance of hard disk drives, as each access to a different location requires physical repositioning of the disk arm. Solid state disks utilising underlying Flash storage chips do not contain any moving parts, and can therefore service such "random access" much more efficiently [44]. As the upload station within the case-study at Volvo consists of a large array of hard disk drives, the benefits of a Flash-based storage medium does not apply. Further, the big data volumes effectively defeat disk caching techniques such as the Linux kernel's page cache. Continuous eviction of cached data before it can be used again renders the cache ineffectual as every operation misses the cache.

To alleviate contention for disk bandwidth, the preparation processes can instead be directly connected via streams. A possible design using pipes is shown in Figure 5.4. Instead of repeatedly reading the same data from disk, each data file is now only read once. Intermediate data is not written back to disk, but is instead sent to consuming processes via in-memory buffers. This requires all preparation processes for a file to run simultaneously, as the buffers cannot grow indefinitely and must be emptied by the downstream/consuming processes. This differs from the original design for the advanced pipeline, where concurrent pipeline instances would be processing separate data files.

Implementation of this technique is performed using standard shell features such as pipes, which are used to connect output of one program to the input of another. However, the complete pipeline of data preparation commands must now be run simultaneously, so that data can be consumed as soon as it is produced by the previous process. Therefore, these commands must now be part of the same Apache Airflow task, implementing the complete data preparation process. This reduces the

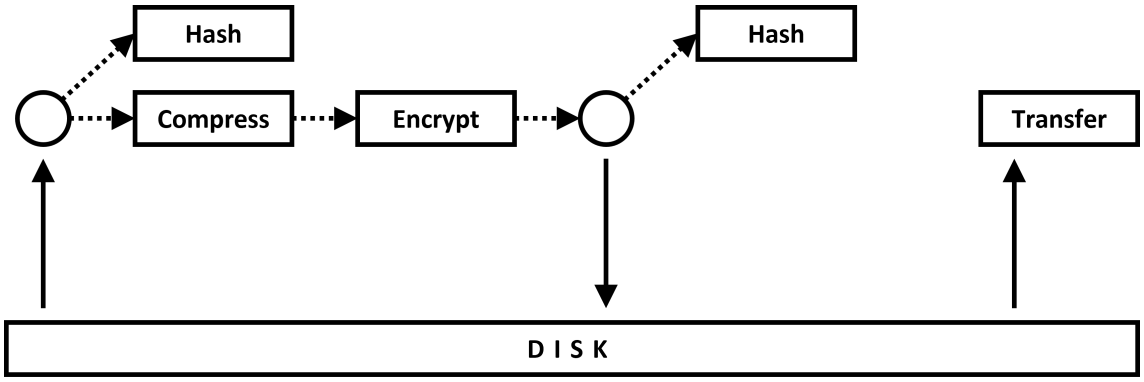


Figure 5.4: Read and write operations performed by the advanced pipeline with in-memory processing. Dashed arrows indicate in-memory data transfers, which do not involve the disk.

granularity at which errors in the pipeline can be caught and reported. Previously, if an error occurred during the encryption task, only the encryption task and any downstream tasks had to be restarted to retry the pipeline. However, with the integrated in-memory implementation, the initial checksum and compression must be repeated as well. The design choice of implementing in-memory processing in this way promises better efficiency, while losing the ability to restart the transfer pipeline in a more fine-grained manner to avoid repeating work.

6
Evaluation

To evaluate the performance of the selected techniques compared to the baseline approach, they have been implemented into pipeline, deployed, and evaluated in a realistic environment. This directly provides relevant data and insight regarding the effect on pipeline performance, as no extrapolation is necessary to transfer insights from a synthetic experimental setup to the real world. Pipeline latency and throughput, as described in Section 3.5, are the primary metrics of the evaluation, as they describe the overall performance of the full pipeline. Further, the maximum daily data volume that a pipeline with a certain pipeline throughput can transfer is then used to classify whether pipeline performance is sufficient for the application. To better understand pipeline behaviour as well as the effects of compression and in-memory processing, batch latency and throughput is also measured. This provides insight into how data flows through the batched-streaming pipeline design proposed in Section 5.1.1. Further, resource utilisation levels are measured to demonstrate the effect of the proposed techniques on resources in the system and, in turn, their impact on overall pipeline performance. The selection of a compression strategy for performance evaluation of the full pipeline is performed by benchmarking various strategies. Key properties of data compression in pipelines, presented in Section 4.3.1, are utilised to determine the best performing strategy.
This chapter proceeds to describe evaluation environment, in terms of systems and infrastructure as well as a controlled test data set used throughout the evaluation. Next, benchmark results for a variety of data compression strategies are presented in order to make a well-founded selection for use in the complete pipeline. Then, performance metrics of the baseline, the completed pipeline with compression, as well as a version using in-memory processing, are presented and compared.

6.1 Evaluation environment

The evaluation is performed in the target environment at Volvo Trucks, using the same systems and infrastructure components that a production deployment of the pipeline would utilise. Next, a suitable evaluation data set is constructed for use in performance benchmarks.

6.1.1 Systems and infrastructure

The evaluation environment consists of multiple systems and network links, geographically and logically spread out over a variety of regions. Figure 6.1 illustrates

the full system.

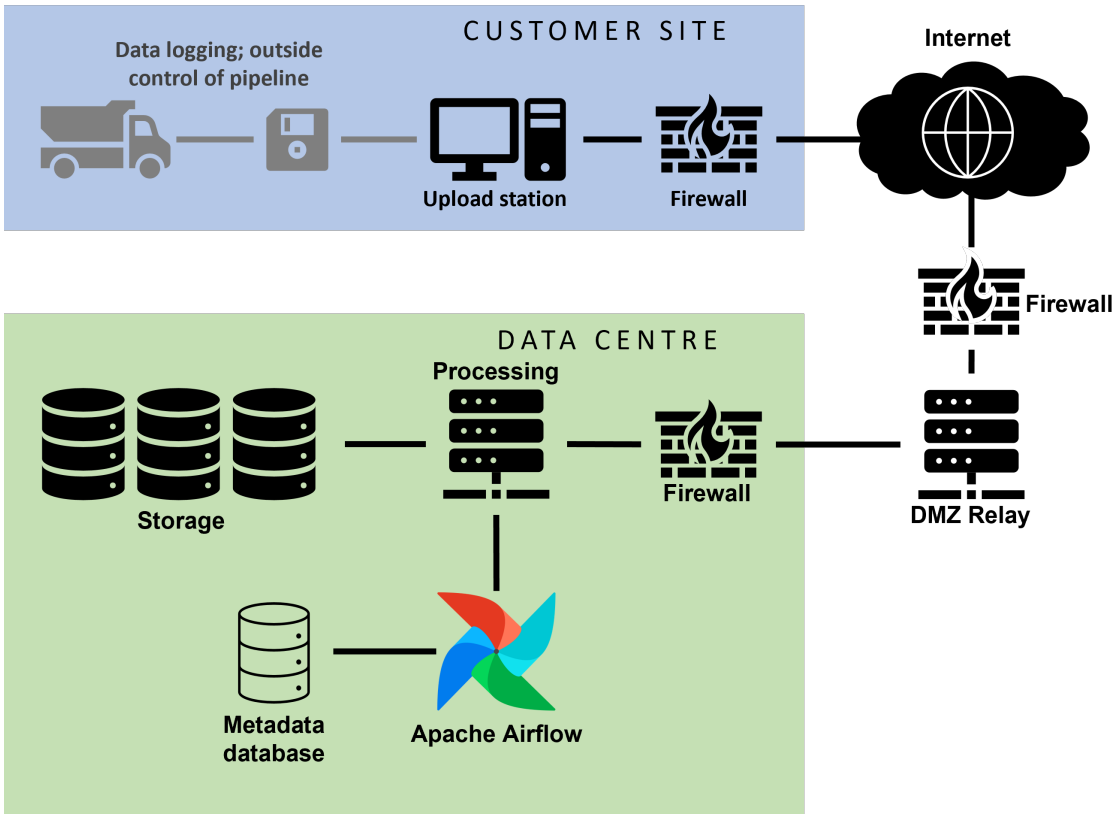


Figure 6.1: Systems present in the evaluation environment.

The pipeline begins at the remote upload station at the customer site in Norway. This system has an Intel® Xeon® E-2236 12-core CPU operating at 3.40 GHz, 64 GB RAM, a 1 Gbit/s network link to its gateway router, and is running Ubuntu 20.04.4 LTS.
From the upload station, data is transferred over the internet to Volvo networks. Data protection during this transfer is guaranteed by common network security techniques, such as VPN tunnelling and SSH for remote authentication and access. These protocols introduce additional, but unavoidable, communication overhead, thereby further limiting throughput.
The next host in the pipeline sequence is the relay host in the demilitarised zone. This is a virtualised host, with access to 4 cores of an AMD EPYC 7601 32-Core processor, 10 Gbit/s network connection, and running Ubuntu 20.04.4 LTS. This host functions as a buffer for inbound data on the way to hosts on the internal network, and does not perform any processing tasks itself. Rather, is serves to prevent direct transmission from external internet-connected devices to internal systems and networks. As such, network bandwidth is the most significant resource at this node in the pipeline.
A dedicated physical server is used for any processing tasks after data has been pulled from the DMZ relay. This processing node has an Intel® Xeon® E5-2690 28-core CPU running at 2.60 GHz, 256 GB of RAM, and runs Red Hat Enterprise Linux Server 7.9. Data is stored on a centralised storage cluster to enable access from

anywhere within the data centre at high speeds. This design simplifies scaling out processing work to multiple servers in the future. The processing server is connected to this storage with a 10 Gbit/s network link which provides high-speed access to data.
Alongside the systems comprising the data pipeline hops, the deployment of the pipeline requires some further infrastructure. Apache Airflow version 2.1.2 and Python 3.8 are used for orchestration and implementation of the pipeline. An instance of Airflow is deployed within the data centre, and consists of the scheduler, the web server, a single worker node, alongside a dedicated PostgreSQL database for persistent metadata storage and a Redis® instance used as a message broker between the scheduler and worker processes. Further, a PostgreSQL 12.9 database server is used by the pipeline to store file metadata catalogue.

6.1.2 Evaluation data set

A controlled test data set for pipeline performance evaluation is constructed by sampling 337 GB of real data collected during the Brønnøy project. The availability of such real data for this evaluation allows for more accurate evaluation and conclusions as it reflects attributes of the real data that could be missing in a synthetic data set.
A separation of data into batches is required by the proposed pipeline. A suitable attribute for this purpose is present in the data set, as each vehicle start-up generates a new unique directory into which data for that run is placed. This directory structure provides a natural separation of data into batches without requiring further pre-processing.
Batches contain two types of data files: a constant set of plain-text log files, and a series of data files, referred to as "bag files", storing sequences of timestamped signals. These bag files are continuously written while data is recorded, and each bag file contains data for a tumbling window of 60 s of vehicle operation, resulting in a file size between 500 MB to 1 000 MB. The log files are considerably smaller, with an average size around 350 kB each.
The benchmark data set is created from 14 such batches of data. Batch sizes in the data set vary, which reflects a variation in size of data batches seen in the project. This allows examining pipeline performance and behaviour under realistic loading conditions. Figure 6.2 illustrates the distribution of batch sizes in the input data set.
The distribution of file types in the test data set is shown in Table 6.1. Bag files account for the majority of the data volume at 99.97 % of total volume, while the comparatively minuscule 90 MB of log files make up a quarter of files in the pipeline by quantity. Because the effectiveness of data compression is directly dependent on the type of input data, it is important to select a compression strategy which is capable of compressing bag files particularly well.

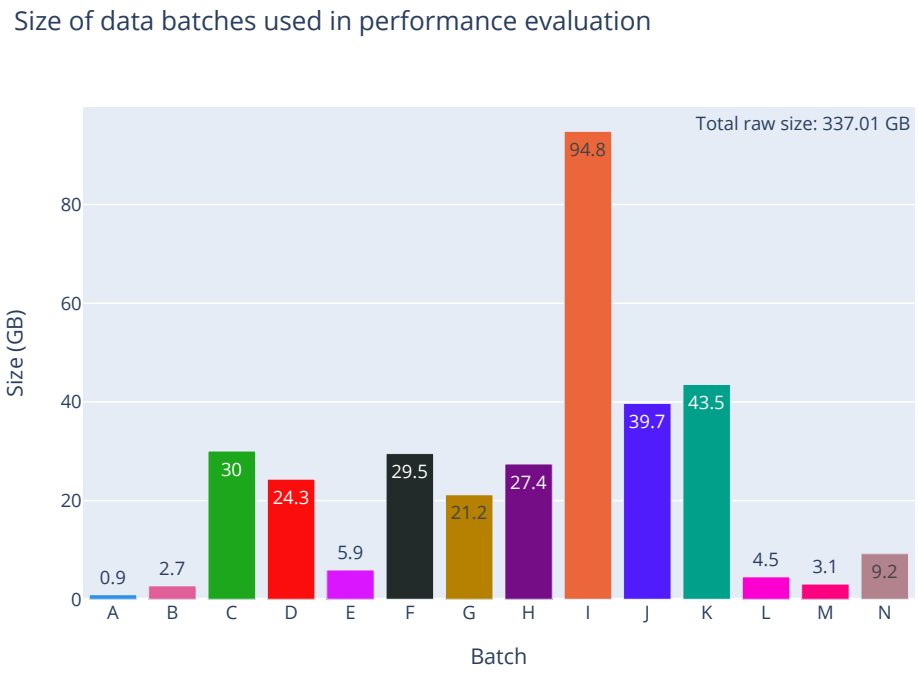


Figure 6.2: Batch size of each batch (labelled A–N) in the evaluation data set.

Type	Count	Size	Proportion by count (%)	Proportion by size (%)
Log	252	89.59 MB	27.85	0.03
Bag	653	336.92 GB	72.15	99.97
Total:	905	337.01 GB	100.00	100.00

Table 6.1: Evaluation data set composition by data file type

6.2 Data compression performance

Before evaluating the proposed pipeline design, a suitable data compression strategy must be selected. The following compression algorithms are benchmarked as part of this evaluation:

For each algorithm, all compression levels were evaluated, yielding a total of 47 compression strategies. Compression performance for each strategy is measured by compressing a subset of the pipeline evaluation data set. This subset consists of 100 bag data files, and reduces the data volume to 54 GB. This is done in order to

[1] `https://www.gnu.org/software/gzip/`
[2] `https://sourceware.org/bzip2/`
[3] `https://lz4.github.io/lz4/`
[4] `https://facebook.github.io/zstd/`

Algorithm	Implementation	Version	Compression levels
Gzip	GNU `gzip`[1]	1.10	`-1 .. -9`
Bzip2	`bzip2`[2]	1.0.8	`-1 .. -9`
LZ4	`lz4`[3]	1.9.2	`-1 .. -12`
Zstandard	`zstd`[4]	1.4.4	`-1 .. -17`

Table 6.2: Evaluated compression algorithms.

accelerate benchmarking of all strategies as the input data is almost 7 times smaller but remains suitably large to demonstrate the performance of each strategy.
Note that decompression performance is not measured here. There are multiples reasons for this: (1) decompression is significantly faster than compression as it requires less work, and therefore has a lesser impact on pipeline performance, (2) decompression speed for the algorithms studied here is generally constant and independent of compression level [45], and it is therefore not useful to measure it for each strategy, and (3) data in this pipeline is decompressed in the data centre, where hardware resources are not a constraint and large-scale parallelisation is feasible, further reducing the potential impact of decompression speed on pipeline performance.

6.2.1 Compression benchmarks

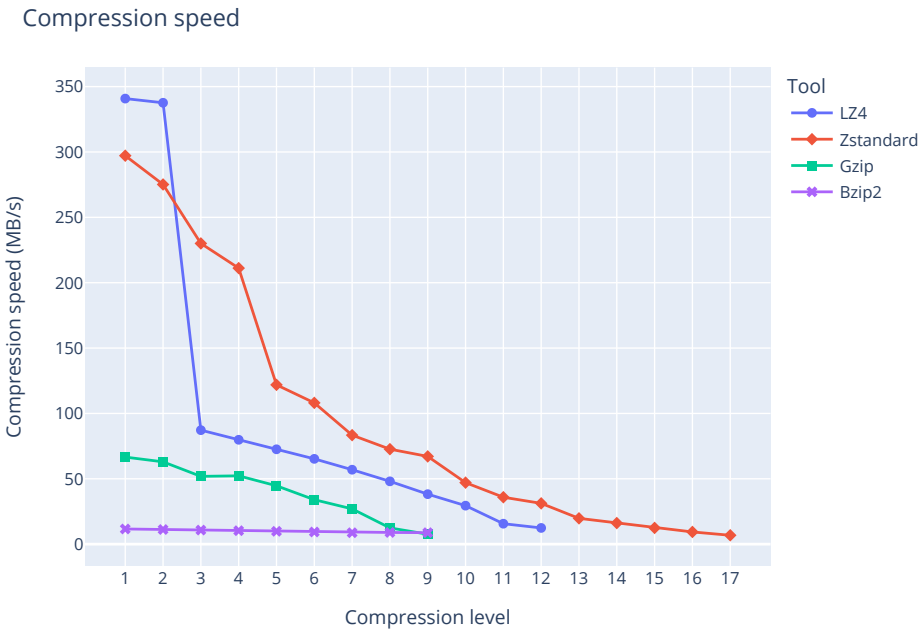
The files in compression benchmark data set were compressed sequentially using each strategy. Contents of the page cache are dropped before each benchmark to eliminate effects of this cache by ensuring that data is always read from and written to the disk. Elapsed time of the compression operation for all tiles and the total size of the output files are recorded, and used to derive compression speed and ratio, respectively, for each strategy.
Performance of each strategy in these metrics is illustrated in Figure 6.3. A distinct cluster of high-speed compression strategies is seen in Figure 6.3a, consisting of LZ4 levels 1 and 2 and Zstandard levels 1 to 4. Throughput of these strategies exceeds 200 MB/s, while nearly all other strategies reach speeds less than half of this.
While these LZ4 strategies are particularly fast, the achieved compression ratio of 1.84 (Figure 6.3b) is the lowest of any strategy. Both Gzip and Bzip2 achieve a compression ratio between 2.2 to 2.3, which is consistently higher than LZ4 at any level. Zstandard is the only algorithm able to exceed a compression ratio of 2.4, but achieved compression effectiveness plateaus while speed decreases until extremely high compression levels are used. Lower compression levels of Zstandard achieve compression speeds in excess of 200 MB/s and compress the test data set in 3 to 5 minutes, while the highest levels slow down to levels similar to Bzip2 at less than 10 MB/s and require over 2 hours to complete.
As the compression speed of Zstandard decreases significantly beyond level 4, with achieving only minor gains in compression effectiveness despite the additional time investment, Zstandard at low compression levels appears to be a promising set of strategies. Detailed performance data for these strategies is shown in Table 6.3, along with relative performance compared to the highest speed and ratio amongst

these 4 strategies.

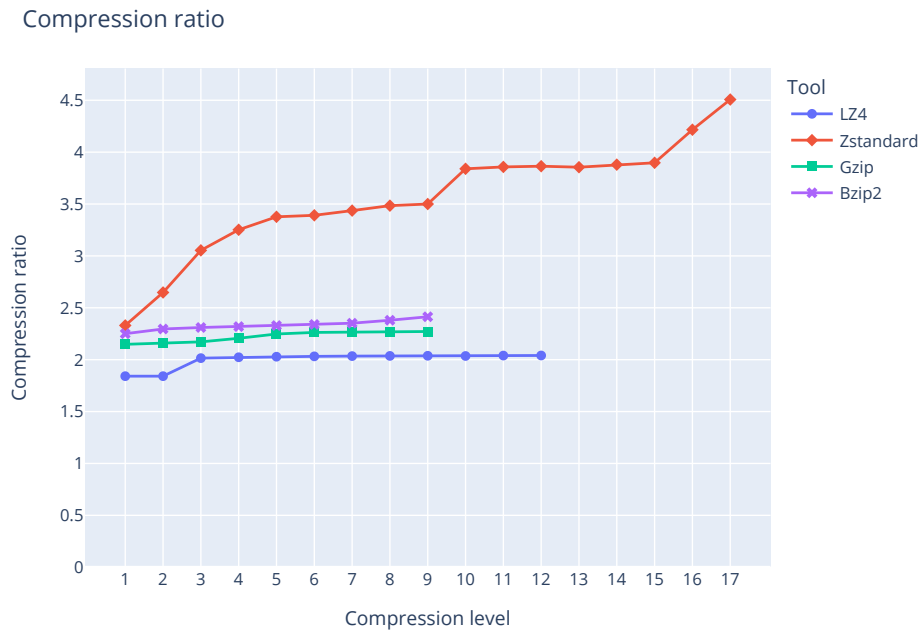
Strategy	**Speed** (MB/s)	**Delta from best speed** (%)	**Compression Ratio**	**Delta from best ratio** (%)
`zstd -1`	297.1		2.330	−28.35
`zstd -2`	275.1	−7.41	2.648	−18.58
`zstd -3`	230.0	−22.59	3.054	−6.10
`zstd -4`	211.2	−28.94	3.252	

Table 6.3: The final four most promising compression strategies with best expected performance in the transfer pipeline application. The final decision is to select a balance point between speed and compression ratio.

For implementation of the proposed transfer pipeline, Zstandard with compression level 3, the default, is selected as it exhibits a good balance between compression speed and effectiveness.



(a) Compression speed of every strategy. The rate at which uncompressed input data is processed by the compression tool at the given compression level setting.



(b) Compression ratio achieved on the test data set by each strategy. Higher compression ratio implies better compression effectiveness.

Figure 6.3: Compression performance of all 47 compression strategies. Every point on a series corresponds to the tool's performance at the particular compression level. Each strategy was used to compress the same 54 GB dataset.

6.3 Pipeline performance

Here, performance metrics of three designs of the data pipeline are compared. The three designs are:

Baseline The baseline solution as described in Section 4.2.2.

Advanced The proposed Airflow-based batched streaming pipeline, including additional functionality and data compression.

In-memory A further enhancement of the advanced pipeline utilising in-memory processing.

Each pipeline was used to transfer the full test data set, and the results for each pipeline are evaluated below.

6.3.1 Baseline solution

The baseline pipeline is not batch-aware, which means that each step operates on the complete data set at once as a single large batch. As such, per-batch metrics is not applicable in this scenario. The processing time for each of the three steps is measured and shown in Table 6.4 alongside the average service rate of the transfers. Total end-to-end latency until the full data set is available in the target location is 12 542 s and pipeline throughput $R_{Baseline}$ of the complete pipeline is calculated:

$$R_{Baseline} = \frac{337.01\,\text{GB}}{12\,542\,\text{s}} = 26.87\,\text{MB/s}$$

Additional delays between pipeline steps due to operator inattentiveness and lack of automation are not included, but could lead to lower pipeline performance in practice.

Pipeline step	**Processing time** (s)	**Average throughput** (MB/s)
Transfer to DMZ	5 779	58.32
Transfer to landing zone	3 272	103.01
Transfer to storage	3 491	96.59
Full pipeline	12 542	26.87

Table 6.4: Latency and throughput for transfer of 337 GB test data set using baseline solution.

Maximum daily volume

If this pipeline throughput is sustained, the maximum data volume transferable per day using the baseline pipeline is:

$$\text{Maximum daily volume} = 26.87\,\text{MB/s} \cdot 86\,400\,\text{s} = 2.322\,\text{TB}$$

As the field test vehicles produce data at an average rate of 500 MB/min, this daily limit would support transferring up to

$$\frac{2.322\,\text{TB}}{500\,\text{MB/min}} = 4643 \text{ minutes} = 77.4 \text{ hours of log data}$$

This capacity is sufficient to transfer data generated by up to 4.8 trucks operating for two 8-hour shifts per day. In the current stages of the Brønnøy project, only four trucks are used for testing, and only for single shifts, and thus, the baseline pipeline is able to keep up with this volume. However, as the project scales up in number of vehicles and shifts, data volumes will grow beyond this limit, and additional pipeline throughput performance will be needed. With potential for up to 7 vehicles to be operating for 16 hours per day, data volumes may reach up to 3.4 TB per day. Further, reducing transfer times to support more timely delivery provides engineers and developers with access to necessary data sooner. Finally, the desired additions of data integrity verification and encryption introduce additional work which would reduce pipeline performance further.

6.3.2 Data compression

Performing data compression before transmitting data over a bandwidth-limited link increases the effective rate at which raw data is transferred beyond the physical bandwidth of the communication channel. An execution timeline of the advanced pipeline is shown in Figure 6.4. This pipeline includes the additional desired features that are not present in the baseline, such as integrity verification and encryption. Each batch is now also processed by a separate pipeline instance, which allows for the inter-batch parallelism seen in the execution timeline.
The pipeline completes transfer of all test data batches in 9 002 seconds, which results in an average pipeline throughput of:

$$\text{Pipeline throughput} = \frac{337.01\,\text{GB}}{9\,002\,\text{s}} = 37.44\,\text{MB/s}$$

This entails an improvement of 38.8 % over the baseline (26.97 MB/s). If this average throughput is sustained, a theoretical limit on the maximum data volume transferable per day is

$$\text{Daily volume} = 37.44\,\text{MB/s} \cdot 86\,400\,\text{s} = 3.234\,\text{TB}$$

This enables to pipeline to service a data volume equivalent to 6.7 trucks operating for double 8-hour shifts. However, this value is not sufficient to support all seven vehicles, and further performance improvements must be made.

Observations

A number of observations about pipeline behaviour and performance can be made from the execution timeline in Figure 6.4. As batches vary in size, the processing time in each step varies proportionally. Larger batches such as C, F, and most prominently batch I, take longer to process, and thereby induce larger queueing

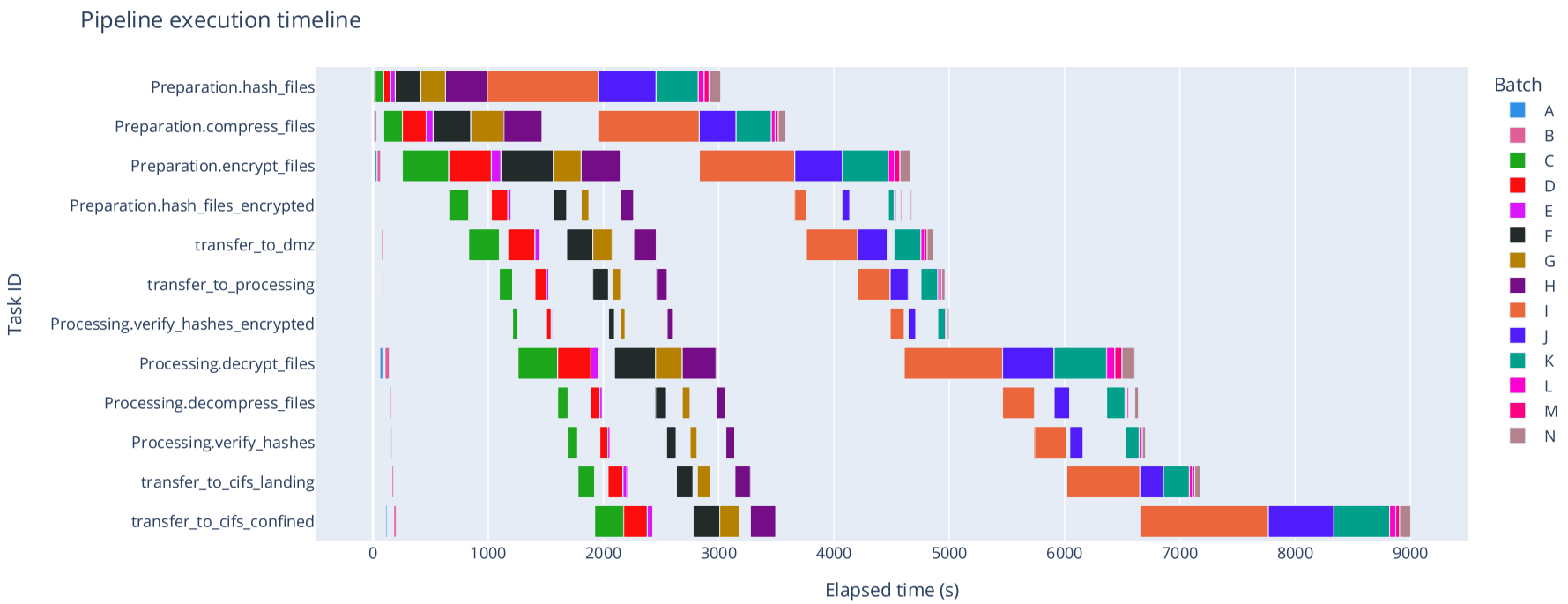


Figure 6.4: Execution timeline of Advanced pipeline with compression for all batches (A–N) of the test data set. Note the extensive queueing endured by all batches following batch I. Particularly the small batches L, M, and N spend a significant portion of their total transfer duration waiting.

delays for subsequent batches. This convoy effect is a drawback of the FIFO queueing discipline employed by Apache Airflow. As batches cannot change processing order, smaller batches are effectively stuck behind larger ones and cannot overtake them to reduce their queueing time. At the same time, this also prevents starvation of a batch.

Pipeline task performance

Average service rates for all tasks in the pipeline are shown in Figure 6.5. Note the service rate of the `transfer_to_dmz` task, which corresponds to the first transfer stage in the Baseline pipeline. While the physical data transfer rate over the link is still limited, the pipeline now transfers compressed data at this rate. Accounting for the achieved compression ratio of 3.089, the `transfer_to_dmz` task effectively transfers raw data at a significantly higher rate. Similarly, effective service rates for all other tasks operating on compressed data are also higher than their physical rate by the same factor. Compression alleviates limited service rates of tasks such as encryption (`Preparation.encrypt_files`) or transfers (`transfer_to_dmz`). Due to the high service rate of the chosen compression strategy alongside the achieved compression ratio, latency winnings from processing compressed data in these particularly slow tasks are far greater than the latency introduced by compression and decompression. Therefore, the investment in data compression provides a net benefit for pipeline performance.

Per-batch performance

The impact of queueing delays and the convoy effect on per-batch performance is visualised in Figure 6.6. Processing time of each pipeline step for all batches alongside the total waiting time incurred by each batch are shown.
Batches J–N suffer particularly much as they wait for batch I to progress through the pipeline. This significantly increases their batch latency as listed in Table 6.5, and as a consequence, batch throughput is accordingly low. By improving the service rate of the slowest tasks, the impact of the convoy effect can be alleviated, as the faster processing times help to reduce queueing delays for subsequent batches.

6.3.3 In-memory processing

The in-memory pipeline replaces the preparation steps of the Advanced transfer pipeline with an in-memory processing pipeline. This allows for more efficient data flow between these various processing steps, while reducing the amount of read and write operations performed against the storage disks.
The execution timeline of the in-memory pipeline is shown in Figure 6.7, and completes in 6 842 s. Pipeline throughput for transfer of the full test data set is then:

$$\text{Pipeline throughput} = \frac{337.01\,\text{GB}}{6\,842\,\text{s}} = 49.26\,\text{MB/s}$$

This is a performance improvement of 31.6 % over the Advanced pipeline (37.44 MB/s), and an 82.6 % improvement over the baseline (26.97 MB/s). With this pipeline

Figure 6.5: The service rate of each task, i.e. the rate at which is processes its input, is shown in blue. For tasks operating on compressed data (all tasks from `Preparation.compress_files` until `Processing.decompress_files`), the effective service rate over raw data is shown in red. Notably, the `transfer_to_dmz` now reaches an effective rate of over 150 MB/s over the bottleneck link.

throughput, the maximum data volume transferable per day becomes

$$\text{Daily volume} = 49.26\,\text{MB/s} \cdot 86\,400\,\text{s} = 4.256\,\text{TB}$$

This volume allows for transferring logged data for approximately 142 operating hours every day, which is sufficient for full-time operation of 5.9 trucks, or 8.8 trucks operating for double shifts every day. The in-memory pipeline exhibits significantly lower turnaround time and higher pipeline throughput, with enough performance overhead to support the expected growth in data volume in the Brønnøy project.

6.3.4 Observations and comparisons across pipelines

While the in-memory pipeline still performs identical processing steps as the Advanced pipeline, the number of tasks is lower in order to implement the in-memory processing steps as Airflow tasks.

Figure 6.8 illustrates the processing time of each pipeline step for all batches. Compared to the same metrics for the Advanced pipeline (Figure 6.6), batches now complete faster but the convoy effect remains present. Although waiting times are shortened, their proportion of the overall batch latency remains similar to before.

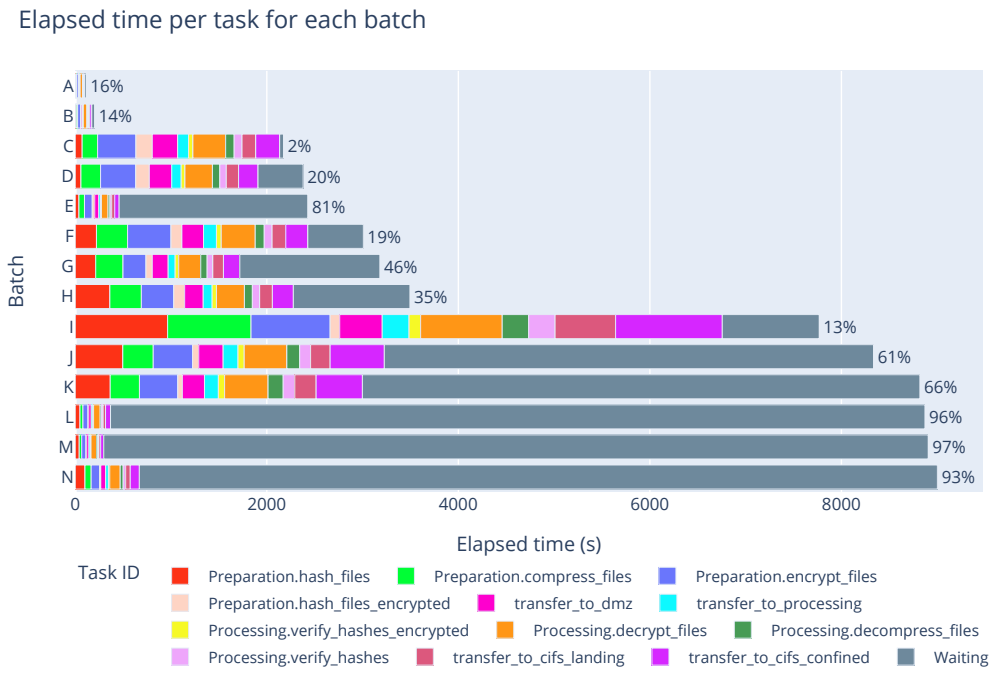


Figure 6.6: Time spent executing each tasks or queueing (waiting) during each pipeline instance. Waiting times are collapsed into a single block for clarity, and annotated with the proportion of the total batch latency that was spent waiting.

Task performance and resource efficiency

By utilising idle processing resources, the advanced pipeline achieves significantly higher performance than the baseline. Distribution statistics for utilisation levels of the most significant four hardware resources on the upload station are shown in Figure 6.9. Data is collected at a rate of 1 Hz while the upload station is actively used by the pipeline. Once all batches have progressed past the `transfer_to_dmz` step, upload station resources are no longer utilised by the pipeline and would skew the distribution of resource utilisation data.

The baseline solution, seen in blue, reads data files from disk and transfers them immediately. As such, CPU and RAM requirements are low, while network utilisation is at maximum throughput and disk bandwidth utilisation is consistently around 80 %. The performance bottleneck due to transfer throughput becomes evident, as other resources are not fully loaded.

By harnessing untapped CPU resources for data compression, the advanced pipeline is able to achieve higher performance. When inspecting the resource utilisation of the Advanced pipeline, seen in red, the additional CPU utilisation is visible. RAM utilisation is essentially unaffected in the Advanced pipeline, and network throughput fluctuates between 0 and the bandwidth limit. This indicates that data preparation is not fast enough to saturate the bandwidth of the transfer, as data is not always available for transfer. Preparation performance is not limited by CPU, as utilisation levels generally lie below 40 %, but rather by disk bandwidth. As multiple pipeline tasks attempt to read and write various data files concurrently,

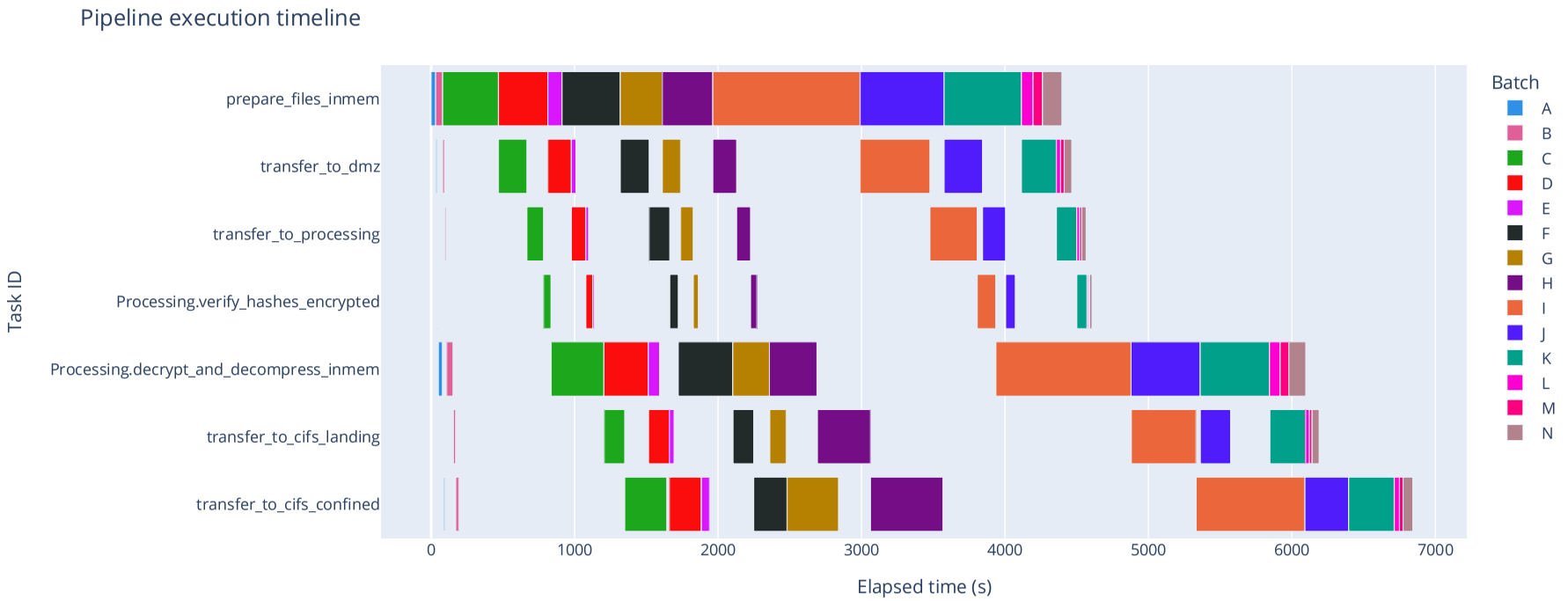


Figure 6.7: Execution timeline of Advanced pipeline with compression for all batches (A–N) of the test data set. The preparation steps, which appeared as separate tasks in the Advanced pipeline, have been grouped into a single task to simplify implementation of the in-memory data preparation. Similarly, decryption, decompression, and integrity verification on the processing side have been grouped into a single task. Again the convoy effect due to the bigger, slower batch I is clearly visible.

Batch	**Latency** (s)	**Throughput** (MB/s)
A	114	8.13
B	201	13.58
C	2 174	13.82
D	2 379	10.23
E	2 427	2.45
F	3 008	9.82
G	3 180	6.66
H	3 494	7.85
I	7 768	12.21
J	8 335	4.76
K	8 819	4.94
L	8 872	0.51
M	8 906	0.35
N	9 002	1.03

Table 6.5: Latency and throughput performance of each batch.

disk bandwidth utilisation is consistently very high with a median value of 99.2 %, thereby throttling performance of the processing tasks.
The in-memory pipeline, shown in green, addresses this problem by restructuring pipeline steps in order to remove several intermediate reading and writing operations. By transferring intermediate data via in-memory buffers and only reading and writing data at the beginning and end of the processing chain, disk bandwidth utilisation is reduced to around 70 %. This allows for much more consistent operation of the preparation processes. CPU utilisation is reduced and much more consistent, as processes spend less time waiting for I/O operations to be serviced by the disk. Network utilisation still fluctuates as preparation is not as fast as transferring, but as the higher Q3 (75^{th} percentile) compared to the Advanced pipeline indicates, more time is spent transferring at high rates than previously.

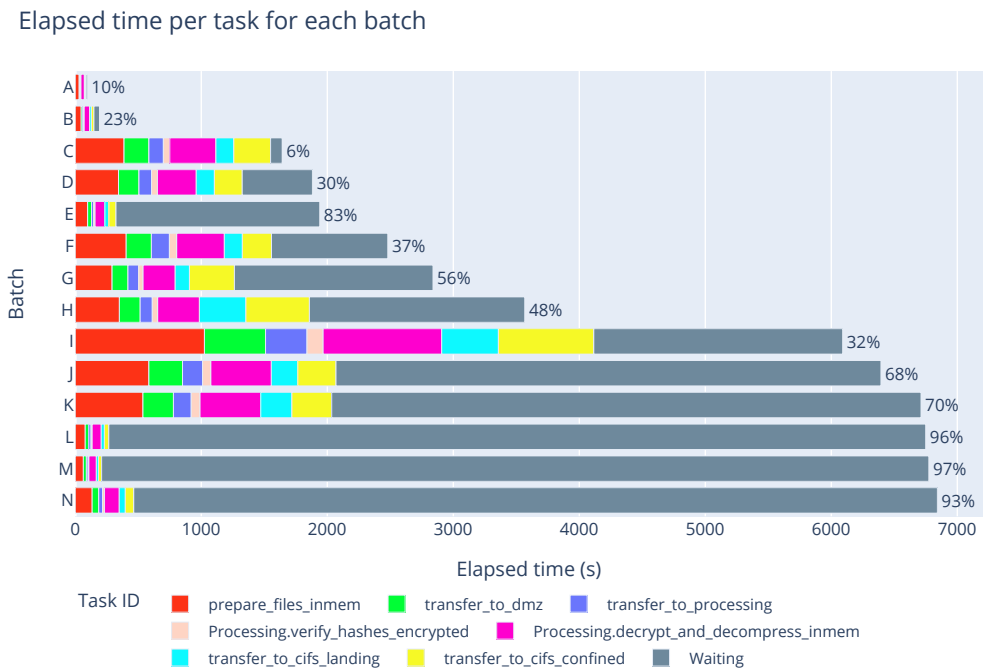
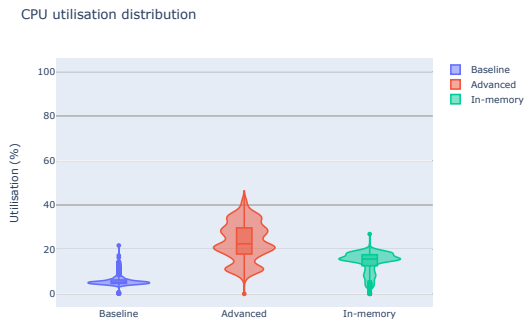
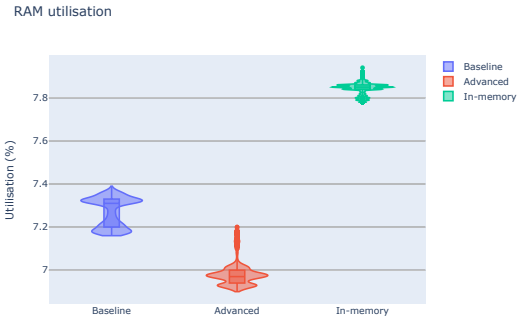


Figure 6.8: Time spent executing tasks or waiting for each batch during execution of the in-memory pipeline. Even though the pipeline processes data much faster than the Advanced pipeline, the smaller batches L, M, and N are still seen waiting and queueing for over 90 % of their execution time.



(a) Distribution of CPU utilisation on the upload station with each pipeline.



(b) Distribution of RAM usage on the upload station with each pipeline. Note the enhanced y-axis scaling.

(c) Distribution of network bandwidth utilisation on the upload station while transferring data using each pipeline. 100 % corresponds to a transmit rate of 68 MB/s, the maximum achievable bandwidth on the route between Brønnøy and the DMZ. The median for both Advanced and In-memory is 0.01, but note the higher frequency of high utilisation for the In-memory pipeline.

(d) Distribution of disk bus utilisation. Corresponds to the proportion of time that the disk is busy servicing requests. At 100 % utilisation, the disk is never idle and additional incoming requests beyond the maximum service rate of the disk will need to wait in queue.

Figure 6.9: Distribution of utilisation level for hardware resources on the upload station during transfer of the test data set for each pipeline. The box within each violin illustrates the span of Q1 and Q3 quartiles (25^{th} and 75^{th} percentile) of the data; the line bisecting the box designates the median value.

6.4 Discussion of added qualitative properties

Beyond ascertaining sufficient pipeline performance and scalability to support growing volumes of data, the industrial data pipeline implementation also implements a number of qualitative features.

Data veracity As the main focus of the data pipeline is transfer, rather than processing, data quality can solely be negatively impacted by corruption occurring in various transfer stages. To alleviate this, the pipeline implements automatic verification of data integrity upon arrival in the data centre. This verification by comparing file hashes with the expected value is possible as the pipeline transfers the full raw data files. In pipelines performing summarisation or other processing, input data cannot be fully reconstructed to guarantee data quality is maintained throughout the pipeline.

Empirical observation of pipeline operation in the production environment over a period of six weeks shows high robustness with no large scale outages. A single network outage caused a group of pipeline instances to fail, but due to the Apache Airflow-based implementation, these instances could quickly be restarted after email alerts were emitted.

Data governance Data governance and lineage tracking are implemented directly within every pipeline step. A centralised file catalogue database table is used to keep a record of every file and its progress through the pipeline. This allows the data management operation to know which files have been ingested into the environment, along with metadata about every file and its ingestion status. Additionally, this catalogue database can be used to construct user-facing dashboards to support users to quickly find data based on search criteria such as vehicle, date, or time, thereby increasing their productivity and efficiency when working with such data volumes.

The pipeline architecture is designed to be flexible and reusable for similar projects. As such, it serves as a performant reference design, and allows for simpler and faster deployment of proper data transfer pipelines in future projects, without the need for falling back to ad-hoc solutions.

Data security Compared to the baseline solution, the designed data transfer pipeline incorporates a number of improvements regarding security of the transferred data. As part of the pipeline deployment, access control groups and permissions are configured to protect data at rest in all steps of the pipeline. This prevents unauthorised access, modification, or deletion of data. Similarly, strong encryption techniques are applied to protect data while in transit from remote sites to the data centre. Thus, unauthorised reading of data is prevented. Further, by computing and storing checksums of all files, tampering or corruption can be detected efficiently.

6.5 Summary

The presented results show clear improvements to pipeline latency and throughput, as summarised in Table 6.6. By benchmarking a variety of compression strategies, a strategy which balances speed and effectiveness can be selected for use in the proposed pipeline for data compression. The developed pipelines show significant improvements in latency and throughput, providing more timely delivery as well as an increased daily data transfer capacity. By applying compression, the original network bandwidth bottleneck is circumvented by harnessing otherwise unused CPU resources on the remote side. Further, introducing in-memory processing to the data preparation adjusts the resource utilisation to be more efficient by reducing the volume of slow disk access operations. This synergy of compression and in-memory processing significantly improves resource efficiency and performance of the pipeline compared to the baseline solution.

Pipeline	**Latency** (s)	**Average throughput** (MB/s)	**Maximum daily volume** (TB)
Baseline	12 542	26.9	2.322
Advanced	9 002	37.4	3.234
In-memory	6 842	49.3	4.256

Table 6.6: Latency and throughput performance for transfer of 337 GB test data set using each pipeline. The average throughput is used to estimate a maximum daily volume of data which can be handled by each pipeline design.

7
Related Work

Efficient transfer of large data volumes in various applications is subject of a large body of research. Techniques are developed to improve performance by reducing data volumes via compression, either lossless or by approximation. Further works study integration of such techniques into full frameworks for managing, aggregating, and processing data in applications such as vehicles or Internet of Things (IoT). Often, these applications involve continuous transmission of data in a streaming fashion, whereas bulk transfer of raw data appears to become less prevalent as it severely limits the potential for live analysis.

As the theoretical limits of lossless compression become insufficient for the data volumes to be processed, stronger techniques for space saving become essential. By giving up losslessness to produce approximations of data, it can be compressed to significantly smaller volumes. However, as such approximation irreversibly removes detail, the original raw data cannot be recovered any longer. The loss of detail, or error, must be within an acceptable limit for the approximation to remain useful. As such, construction of efficient approximation techniques with bounded precision loss is a well-studied challenge. An example of such an approximation technique is Piecewise Linear Approximation (PLA), which represents a time series of data points as a sequence of segments, and achieves several times higher reduction in data volume than lossless techniques such as ZIP would [19]. In order to apply PLA in the growing domain of fog/edge distributed systems, where data streaming is common, efficient online generation of the segments is paramount alongside trade-offs between achieved compression, latency, and approximation error as studied by Duvignau *et al.* [20] While this thesis studies similar trade-offs and metrics to evaluate overall pipeline performance, data within the general purpose pipeline, and the case study in particular, is not immediately suited for streaming and the application requires lossless compression.

As the prevalence of IoT and connected devices increases, low-latency streaming of data has become an essential performance target to enable real-time decision making. Once the challenges of implementing a network transport with the required low-latency and reliability have been addressed by 5G technology [35], [46], opportunities for data-intensive applications have increased significantly. For example, the ERAIA framework [47] establishes a flexible and scalable basis for implementing data pipelines in the IoT domain. Latency and throughput are again key metrics, while tying each pipeline step to its performance cost allows for further optimisation in terms of resource allocation and orchestration, similar to how service rates of pipeline steps are isolated and used to make targetted performance improvements in this work.

Despite the improved latency and bandwidth provided by 5G, edge computing will remain as a technique to process data closer to the source and reducing incurred latency due to centralised solutions [48], particularly in autonomous driving applications. D3 is a framework for managing data processing on-board autonomous vehicles [49], where timeliness of task execution is critical for safe operation. Schedulability analysis of real-time systems studies this problem to provide rigorous guarantees regarding execution times of all tasks in the system. However, data processing times increase as the data volumes needed for decision-making in autonomous vehicles grow. Gog *et al.* [49] model the flow of data within the vehicle as a sequential pipeline of steps; sensing and data gathering, perception and localisation to extract information from sensor data, prediction, planning, and control. Execution of these steps is subject to deadlines, in order to enforce that decisions are made in a timely manner, which requires a balance between accurate processing and execution time. Although real-time constraints are not a part of the work at hand, utilising edge computing resources to preprocess data before transfer is a common technique.
Other efforts into tuning of high-volume data transfer also point out the size of the parameter space as a problem, and propose a variety of techniques to address it. Yildirim *et al.* [50] identifies pipelining, parallelism, and concurrency as important parameters to improve data transfer throughput by latency hiding. Models of the complete data transfer system, including dataset characteristics, are developed alongside optimisation algorithms to determine optimal parameter values. In a similar vein, Arslan *et al.* [51] studies parameters pertaining to I/O throughput and parallelism to improve transfer throughput, as well as adaptive tuning to adjust parameters in real-time. The parameters studied in this thesis intentionally do not include tuning of parallelism, although concurrent processing of data is already present due to the batched-streaming pipeline model. However, resource limitation at the edge (upload station) effectively prevent further parallelism, unless more efficient processing tools are used. Liu *et al.* [52] explore the possibility of more efficient integrity verification in high-volume transfers, which is achieved through parallelising and overlapping transfer and integrity verification in order to hide the introduced latency better. To address the movement cost associated with transfer of big data volumes, data compression can be introduced into such pipelines Zou *et al.* [53]. While this work manually determines the optimal placement and best compression algorithm for the data, Zou *et al.* [53] develop an adaptive model to determine this automatically.

8

Conclusion

This work studies the synergy of compression and in-memory processing techniques that can improve performance of data transfer pipelines. As industries embrace data as a valuable asset, big data volumes continue to grow. Tools and systems need to be able to scale accordingly to maintain necessary performance in throughput and latency in face of the increasing demand for constrained hardware resources. New paradigms such as edge computing shift data processing tasks from centralised computing infrastructures to thinner, less powerful devices. As such, efficiency and performance in these resource-constrained environments are important challenges that must be addressed to support growth and development in this field. This thesis shows how data compression and in-memory computation can be used within the context of big data transfer pipelines in order to achieve timely delivery of Big Data. A case study at Volvo Trucks is used to evaluate these techniques under real-world conditions, and demonstrates an average throughput uplift of 82.6 % over the existing baseline solution.

In future work, it would be interesting to explore how to further reduce latency and improve performance, particularly for important data. By identifying data of higher importance already at the edge and source of the pipeline, transfers can be prioritised and scheduled accordingly to yield lower latencies for such important data. For example, when transferring logged vehicle data from field tests, there is value in transferring data relating to faults and error conditions before the remaining bulk data. Additionally, the possibility to avoid transferring the raw data of a system in its entirety is a promising technique to cope with growing data volumes. By selectively transferring only data pertaining to relevant conditions and events, data volumes to be transferred could be reduced by several orders of magnitude. In other applications where continuous analysis of data is desired, summarisation techniques can be applied to compress data volumes with bounded accuracy losses due to approximation.

Furthermore, as the space of design choices and tunable parameters is extremely large for such distributed systems, manual tuning for performance optimisation is infeasible. Autonomous performance tuning of systems using machine learning techniques is a common topic of research, and can be performed while the system is running. This could even allow the system to identify and adapt to changing parameters of the workload over time.

In order to support adoption of automatic data transfer pipelines by industry, various aspects outside of pure performance are relevant. Various aspects of the data, such as security, veracity, governance, and more must be considered in the life cycle of a transfer pipeline process. To this end, implementation frameworks such as Apache

Airflow would benefit from further support for performance monitoring and pipeline metrics visualisation to better support deployment and operation of such systems in real-world applications.

9

Bibliography

[1] "Using transparent compression to improve SSD-based I/O caches | Proceedings of the 5th European conference on Computer systems," ACM Conferences. (), [Online]. Available: https://dl.acm.org/doi/10.1145/1755913.1755915 (visited on 09/22/2022).

[2] V. Young, S. Kariyappa, and M. K. Qureshi, "Enabling Transparent Memory-Compression for Commodity Memory Systems," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2019, pp. 570–581. DOI: 10.1109/HPCA.2019.00010.

[3] B. Welton, D. Kimpe, J. Cope, C. M. Patrick, K. Iskra, and R. Ross, "Improving I/O Forwarding Throughput with Data Compression," in *2011 IEEE International Conference on Cluster Computing*, Austin, TX, USA: IEEE, Sep. 2011, pp. 438–445, ISBN: 978-1-4577-1355-2. DOI: 10.1109/CLUSTER.2011.80. [Online]. Available: http://ieeexplore.ieee.org/document/6061075/ (visited on 08/31/2022).

[4] "H.264 : Advanced video coding for generic audiovisual services." (), [Online]. Available: https://www.itu.int/rec/T-REC-H.264 (visited on 09/18/2022).

[5] "H.265 : High efficiency video coding." (), [Online]. Available: https://www.itu.int/rec/T-REC-H.265 (visited on 09/18/2022).

[6] "VP9 Video Codec Summary," The WebM Project. (Mar. 27, 2017), [Online]. Available: https://www.webmproject.org/vp9/ (visited on 09/18/2022).

[7] P. de Rivaz and J. Haughton. "AV1 Bitstream & Decoding Process Specification," Alliance for Open Media. (Jan. 18, 2019), [Online]. Available: https://aomedia.org/av1/specification/ (visited on 09/18/2022).

[8] M. van Beurden and A. Weaver. "Free Lossless Audio Codec," IETF Datatracker. (Aug. 21, 2022), [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-cellar-flac/ (visited on 09/18/2022).

[9] "Apple Lossless Audio Codec." (2016), [Online]. Available: https://macosforge.github.io/alac/ (visited on 09/18/2022).

[10] L. P. Deutsch, "GZIP file format specification version 4.3," Internet Engineering Task Force, Request for Comments RFC 1952, May 1996, 12 pp. DOI: 10.17487/RFC1952. [Online]. Available: https://datatracker.ietf.org/doc/rfc1952 (visited on 08/29/2022).

[11] ——, "DEFLATE Compressed Data Format Specification version 1.3," Internet Engineering Task Force, Request for Comments RFC 1951, May 1996, 17 pp. DOI: 10.17487/RFC1951. [Online]. Available: https://datatracker.ietf.org/doc/rfc1951 (visited on 08/29/2022).

[12] "/c++/src/util/compress/bzip2/README," NCBI C++ Toolkit source browser. (Jul. 25, 2016), [Online]. Available: https://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/util/compress/bzip2/README (visited on 08/29/2022).

[13] J. Tsai, "Bzip2: Format Specification," Mar. 17, 2016. [Online]. Available: https://raw.githubusercontent.com/dsnet/compress/master/doc/bzip2-format.pdf (visited on 08/04/2022).

[14] Y. Collet. "RealTime Data Compression: LZ4 explained," RealTime Data Compression. (May 26, 2011), [Online]. Available: https://fastcompression.blogspot.com/2011/05/lz4-explained.html (visited on 08/04/2022).

[15] ——, "LZ4 Frame Format Description," Aug. 12, 2020. [Online]. Available: https://github.com/lz4/lz4/blob/f7b1f6b7420822255b09acdd1b82848b12e1cda1/doc/lz4_Frame_format.md (visited on 08/04/2022).

[16] Y. Collet and C. Turner. "Smaller and faster data compression with Zstandard," Engineering at Meta. (Aug. 31, 2016), [Online]. Available: https://engineering.fb.com/2016/08/31/core-data/smaller-and-faster-data-compression-with-zstandard/ (visited on 08/04/2022).

[17] Y. Collet and M. Kucherawy, "Zstandard Compression and the 'application/zstd' Media Type," Internet Engineering Task Force, Request for Comments RFC 8878, Feb. 2021, 45 pp. DOI: 10.17487/RFC8878. [Online]. Available: https://datatracker.ietf.org/doc/rfc8878 (visited on 05/12/2022).

[18] M. Joshi, "The Pigeonhole Principle," in *Proof Patterns*, M. Joshi, Ed., Cham: Springer International Publishing, 2015, pp. 19–23, ISBN: 978-3-319-16250-8. DOI: 10.1007/978-3-319-16250-8_3. [Online]. Available: https://doi.org/10.1007/978-3-319-16250-8_3 (visited on 08/04/2022).

[19] B. Havers, R. Duvignau, H. Najdataei, V. Gulisano, M. Papatriantafilou, and A. C. Koppisetty, "DRIVEN: A framework for efficient Data Retrieval and clustering in Vehicular Networks," *Future Generation Computer Systems*, vol. 107, pp. 1–17, Jun. 1, 2020, ISSN: 0167-739X. DOI: 10.1016/j.future.2020.01.050. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X19310507 (visited on 05/02/2022).

[20] R. Duvignau, V. Gulisano, M. Papatriantafilou, and V. Savic, "Streaming piecewise linear approximation for efficient data management in edge computing," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, New York, NY, USA: Association for Computing Machinery, Apr. 8, 2019, pp. 593–596, ISBN: 978-1-4503-5933-7. [Online]. Available: https://doi.org/10.1145/3297280.3297552 (visited on 05/10/2022).

[21] Apache Software Foundation, *Apache Airflow*, version 2.1.2, Jul. 7, 2021. [Online]. Available: https://airflow.apache.org/ (visited on 09/18/2022).

[22] *Kubernetes - Production-Grade Container Orchestration*. [Online]. Available: https://kubernetes.io/ (visited on 09/18/2022).

[23] ——, *Airflow Logging Architecture diagram*, Aug. 27, 2020. [Online]. Available: https://github.com/apache/airflow (visited on 05/20/2022).

[24] *Celery - Distributed Task Queue*, version 5.2.7. [Online]. Available: https://github.com/celery/celery (visited on 09/18/2022).

[25] *Redis*. [Online]. Available: https://redis.io/ (visited on 09/18/2022).

[26] *Slurm Workload Manager - Documentation*. [Online]. Available: https://slurm.schedmd.com/ (visited on 09/18/2022).

[27] *Apache Spark™ - Unified Engine for large-scale data analytics*. [Online]. Available: https://spark.apache.org/ (visited on 09/18/2022).

[28] CPAC Systems. "We are proud to be one of the main actors in Brønnøy Project. Taking autonomous transport solutions to the next level.," We are CPAC. (May 15, 2019), [Online]. Available: https://cpacsystems.se/2019/05/15/taking-autonomous-transport-solutions/ (visited on 12/05/2021).

[29] "Volvo Trucks provides autonomous transport solution to Brønnøy Kalk AS," Volvo Trucks Press Releases. (Nov. 20, 2018), [Online]. Available: https://www.volvotrucks.com/en-en/news-stories/press-releases/2018/nov/pressrelease-181120.html (visited on 12/11/2021).

[30] F. Al Machot, M. Ali, A. Haj Mosa, C. Schwarzlmüller, M. Gutmann, and K. Kyamakya, "Real-time raindrop detection based on cellular neural networks for ADAS," *Journal of Real-Time Image Processing*, vol. 16, no. 4, pp. 931–943, Aug. 1, 2019, ISSN: 1861-8219. DOI: 10.1007/s11554-016-0569-z. [Online]. Available: https://doi.org/10.1007/s11554-016-0569-z (visited on 09/22/2022).

[31] N. Capodieci, P. Burgio, R. Cavicchioli, I. S. Olmedo, M. Solieri, and M. Bertogna, "Real-Time Requirements for ADAS Platforms Featuring Shared Memory Hierarchies," *IEEE Design & Test*, vol. 39, no. 1, pp. 35–41, Feb. 2022, ISSN: 2168-2364. DOI: 10.1109/MDAT.2020.3013828.

[32] A. Kondoro, I. Ben Dhaou, H. Tenhunen, and N. Mvungi, "Real time performance analysis of secure IoT protocols for microgrid communication," *Future Generation Computer Systems*, vol. 116, pp. 1–12, Mar. 1, 2021, ISSN: 0167-739X. DOI: 10.1016/j.future.2020.09.031. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X20305707 (visited on 09/22/2022).

[33] S. D. McLean, S. S. Craciunas, E. Alexander Juul Hansen, and P. Pop, "Mapping and Scheduling Automotive Applications on ADAS Platforms using Metaheuristics," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, Sep. 2020, pp. 329–336. DOI: 10.1109/ETFA46521.2020.9212029.

[34] S. Shukla, M. F. Hassan, D. C. Tran, R. Akbar, I. V. Paputungan, and M. K. Khan, "Improving latency in Internet-of-Things and cloud computing for real-time data transmission: A systematic literature review (SLR)," *Cluster Computing*, Apr. 16, 2021, ISSN: 1573-7543. DOI: 10.1007/s10586-021-03279-3. [Online]. Available: https://doi.org/10.1007/s10586-021-03279-3 (visited on 09/22/2022).

[35] M. A. Siddiqi, H. Yu, and J. Joung, "5G Ultra-Reliable Low-Latency Communication Implementation Challenges and Operational Issues with IoT Devices," *Electronics*, vol. 8, no. 9, p. 981, 9 Sep. 2019, ISSN: 2079-9292. DOI: 10.3390/electronics8090981. [Online]. Available: https://www.mdpi.com/2079-9292/8/9/981 (visited on 09/22/2022).

[36] S. Verma, Y. Kawamoto, Z. M. Fadlullah, H. Nishiyama, and N. Kato, "A Survey on Network Methodologies for Real-Time Analytics of Massive IoT Data and Open Research Issues," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1457–1477, 2017, ISSN: 1553-877X. DOI: 10.1109/COMST.2017.2694469.

[37] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, "CPU Scheduling," in *Operating Systems: Three Easy Pieces*, 1.00, Arpaci-Dusseau Books, Aug. 2018. [Online]. Available: https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched.pdf.

[38] S. Gulley, V. Gopal, K. Yap, W. Feghali, J. Guilford, and G. Wolrich, *Intel® SHA Extensions: New Instructions Supporting the Secure Hash Algorithm on Intel® Architecture Processors*, Jul. 2013. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/intel-sha-extensions-white-paper.pdf (visited on 08/05/2022).

[39] The OpenSSL Project, *OpenSSL*. [Online]. Available: https://www.openssl.org/ (visited on 09/23/2022).

[40] The GnuPG Project, *GnuPG - The GNU Privacy Guard*, The GnuPG Project, Mar. 28, 2022. [Online]. Available: https://gnupg.org/ (visited on 09/23/2022).

[41] W. Davison, *Rsync*, 2022. [Online]. Available: https://rsync.samba.org/ (visited on 09/23/2022).

[42] OpenBSD Project, *OpenSSH*, 2022. [Online]. Available: https://www.openssh.com/ (visited on 09/23/2022).

[43] G. Sebastien, *Sysstat*, Sep. 20, 2022. [Online]. Available: https://github.com/sysstat/sysstat (visited on 09/20/2022).

[44] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," presented at the Proceedings of the 2008 USENIX Technical Conference (USENIX'08), Jun. 1, 2008. [Online]. Available: https://www.microsoft.com/en-us/research/publication/design-tradeoffs-for-ssd-performance/ (visited on 08/11/2022).

[45] O. Shadura and B. Bockelman, "Compression Update: ZSTD & ZLIB," presented at the ROOT I/O Meeting (CERN), Jan. 26, 2018. [Online]. Available: https://indico.cern.ch/event/695984/#2-zstd-and-cloudflarezlib-upda (visited on 09/07/2022).

[46] G. Velez, E. Bonetto, D. Brevi, *et al.*, *5G Features and Standards for Vehicle Data Exploitation*, Apr. 13, 2022. DOI: 10.48550/arXiv.2204.06211. arXiv: 2204.06211 [cs]. [Online]. Available: http://arxiv.org/abs/2204.06211 (visited on 09/25/2022).

[47] A. Hernandez, B. Xiao, and V. Tudor, "ERAIA - Enabling Intelligence Data Pipelines for IoT-based Application Systems," in *2020 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, Austin, TX, USA: IEEE, Mar. 2020, pp. 1–9, ISBN: 978-1-72814-657-7. DOI: 10.1109/PerCom45495.2020.9127385. [Online]. Available: https://ieeexplore.ieee.org/document/9127385/ (visited on 03/15/2022).

[48] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi, "Edge Computing for Autonomous Driving: Opportunities and Challenges," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1697–1716, Aug. 2019, ISSN: 1558-2256. DOI: 10.1109/JPROC.2019.2915983.

[49] I. Gog, S. Kalra, P. Schafhalter, J. E. Gonzalez, and I. Stoica, "D3: A dynamic deadline-driven approach for building autonomous vehicles," in *Proceedings of the Seventeenth European Conference on Computer Systems*, Rennes France: ACM, Mar. 28, 2022, pp. 453–471, ISBN: 978-1-4503-9162-7. DOI: 10.1145/3492321.3519576. [Online]. Available: https://dl.acm.org/doi/10.1145/3492321.3519576 (visited on 08/16/2022).

[50] E. Yildirim, E. Arslan, J. Kim, and T. Kosar, "Application-Level Optimization of Big Data Transfers through Pipelining, Parallelism and Concurrency," *IEEE Transactions on Cloud Computing*, vol. 4, no. 1, pp. 63–75, Jan. 2016, ISSN: 2168-7161. DOI: 10.1109/TCC.2015.2415804.

[51] E. Arslan, B. A. Pehlivan, and T. Kosar, "Big data transfer optimization through adaptive parameter tuning," *Journal of Parallel and Distributed Computing*, vol. 120, pp. 89–100, Oct. 1, 2018, ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2018.05.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731518303289 (visited on 09/25/2022).

[52] S. Liu, E.-S. Jung, R. Kettimuthu, X.-H. Sun, and M. Papka, "Towards optimizing large-scale data transfers with end-to-end integrity verification," in *2016 IEEE International Conference on Big Data (Big Data)*, Dec. 2016, pp. 3002–3007. DOI: 10.1109/BigData.2016.7840953.

[53] H. Zou, Y. Yu, W. Tang, and H. M. Chen, "Improving I/O Performance with Adaptive Data Compression for Big Data Applications," in *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, May 2014, pp. 1228–1237. DOI: 10.1109/IPDPSW.2014.138.