Understanding and Building Fault-Tolerant, Scalable and Low-Latency Event Stream Analytics Solutions With Correctness Guarantees

Bachelor's Thesis / T3300

for the study program
Computer Science

at the
Baden-Wuerttemberg Cooperative State University Stuttgart

by
Dominik Stiller

Submission Date	September 7, 2020
Project Period	12 Weeks
Company	DXC Technology
Corporate Supervisor	Dipl.-Ing. Bernd Gloss
University Reviewer	Dipl.-Inf. Daniel Amor
Matriculation Number, Course	4369179, TINF17A

Declaration of Authorship

Ich versichere hiermit, dass ich meine Bachelorarbeit mit dem Thema "*Understanding and Building Fault-Tolerant, Scalable and Low-Latency Event Stream Analytics Solutions With Correctness Guarantees*" selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Sindelfingen	September 7, 2020	
Place	Date	Dominik Stiller

Confidentiality Clause

This thesis contains confidential data of *DXC Technology*. This work may only be made available to the first and second reviewers and authorized members of the board of examiners. Any publication and duplication of this thesis–even in part–is prohibited.

An inspection of this work by third parties requires the expressed permission of the author, the project supervisor, and *DXC Technology*.

Abstract

Batch data processing has become crucial and powers business-critical operations across industries. However, this style of processing is ill-suited for the continuous streams arising from the digitalization or processes and increasing number of sensors to capture events in a connected world. Streams are inherently unbounded and often unordered, which requires a fundamental shift of approach to guarantee correctness while paving the way for new real-time insight techniques.

The goal of this thesis is to build the understanding required for the design of correct, fault-tolerant, low-latency and scalable event stream analytics solutions whose key challenges are time and state. Then we demonstrate the concepts and design considerations through the implementation of a real-time public transportation analytics solution, which we evaluate to further highlight the practical challenges of building such a solution. The findings and considerations can be applied to a wide range of stream analytics use cases and can help developers and architects to build a stream-native mindset.

Contents

Abbreviations **V**

List of Figures **VI**

List of Tables **VII**

List of Code Listings **VIII**

1 Introduction **1**
1.1 Problem 1
1.2 Scope 2

2 Background **3**
2.1 Batch Processing 3
2.2 Stream Processing 5
2.3 Stream Transport 19
2.4 Event Processing 22

3 Streaming Platforms **24**
3.1 Stream Transport Platforms 24
3.2 Stream Processing Platforms 27

4 Solution Design **36**
4.1 Requirements 36
4.2 Architecture 37

5 Analytics Usecase **45**
5.1 HSL Public Transportation APIs 45
5.2 Analytics 48
5.3 Data Flow Example 50

6 Evaluation **53**
6.1 Methodology 54
6.2 Results 58
6.3 Discussion 60

7 Conclusion **62**

Bibliography **63**

Glossary **68**

Abbreviations

HDFS Hadoop Distributed File System

HFP High-Frequency Positioning

HSL Helsinki Regional Transport Authority

protobuf Protocol Buffers

List of Figures

2.1 Relationship between event time and processing time 7
2.2 Lambda architecture 8
2.3 Kappa architecture 8
2.4 Relationship between bounded and unbounded datasets 9
2.5 Example of a dataflow graph 10
2.6 Common window types 12
2.7 Windowing in different time domains 14
2.8 Example of different processing semantics after failure recovery 18
2.9 Message distribution patterns with multiple consumers 20
2.10 Log partitioning 21

3.1 Structure of a Kafka cluster 26
3.2 Components of Flink stack 31
3.3 Cluster architecture of Flink 33
3.4 Logical and physical view of parallel dataflow graph 34
3.5 State checkpointing mechanism of Flink 35

4.1 Solution architecture 37

5.1 Daily event volume of the HSL HFP API 47
5.2 Ingestion lag of the HSL HFP API 47
5.3 Analytics for public transportation 49
5.4 Visualization on map with geocells 52

6.1 In-band latency tracking system 54
6.2 Volume scaling system 57
6.3 End-to-end latency for vehicle distribution job at normal volume 59
6.4 End-to-end latency distribution for vehicle distribution job at different scaling factors 60
6.5 Ingestion-to-processing latency distribution for all jobs at different scaling factors 60

List of Tables

2.1 Example of windowing output modes 15

List of Code Listings

4.1 Common Protocol Buffers schema for all events 39
4.2 Interaction between connectors and processors 40

5.1 Job definition for vehicle distribution analytics 52
5.2 Vehicle distribution result as protobuf message 52

1 Introduction

Data processing and data analytics in particular have become essential tools across industries. Being able to automate business processes and manage large amounts of data can increase efficiency and decrease cost, but also enables completely new insight and business models. The demand for ever better and more scalable data processing gave rise to platforms like Hadoop and Spark which are widely used for many purposes. They process transactions to create bills and reports for accounting, they analyze purchase activities to reveal customer behavior to retailers, they provide the foundation for decision-making and planning in sectors like manufacturing or oil and gas, and they help governments combat climat change and improve citizen's quality of life.

1.1 Problem

Traditionally, data was processed in batches. In many cases, the dataset was imported as bulk from transactional systems or other sources. Then, the processing job was executed once per day. There are two main problems with this approach. First, most data are actually produced as a continuous stream. Processing such an infinite stream in batches cannot unlock the data's full potential. Correctness might suffer if the dataset is incomplete, and some stream-specific operations like data-based windowing with variable size cannot easily be accomplished. Secondly, batch processing inherently has a high latency between data being created and results becoming available. This makes it unsuitable for applications where low-latency results are required in real-time without sacrificing the capabilities of mature batch systems. Additionally, insights decrease in value over time.

Lately, more data sources provide data as continuous stream as digitalization of historically manual processes progresses and ubiquitous sensors capture events in the real world. Instead of processing these data with the traditional platforms, new stream processing frameworks which embrace the nature of stream data have been developed. The true extent of the paradigm change needed for this transition was aptly put into words by Google researchers Tyler Akidau, Robert Bradshaw, Craig Chambers, *et al.*:

> We propose that a fundamental shift of approach is necessary to deal with [the] evolved requirements in modern data processing. We as a field must stop trying to groom unbounded datasets into finite pools of information that eventually become complete, and instead live and breathe under the assumption that we will never know if or when we have seen all of our data, only that new data will arrive [and] old data may be retracted. [1, p. 1792].

There are two key challenges that need to be solved by such stream processing platforms, especially for more complex jobs like stream analytics which correlates events over time for aggregation or pattern recognition. State in form of intermediate results needs to be maintained in a fault-tolerant manner for correct and consistent results even in the face of software or hardware failures. Additionally, the processing framework needs to be aware of the time at which events occurred since they might arrive delayed and out-of-order. Some stream processing frameworks that support both aspects have been developed recently to enable stream analytics at large scale. However, many software engineers, solution architects and technical managers are only beginning to understand the power of this new type of processing.

1.2 Scope

We believe that a foundational understanding of stream processing concepts as well as a new mindset compared to batch processing is necessary to build successful stream analytics solutions. Therefore, we have a twofold goal for this thesis:

- Build an understanding for designing correct, fault-tolerant, low-latency and scalable processing systems for stream datasets
- Demonstrate important concepts and design considerations through the implementation of an event stream analytics solution for a public transportation use case

The four characteristics of correctness, fault tolerance, low latency and scalability serve as guide to connect the theoretical and practical aspects.

We start by presenting the fundamental challenges of and common concepts behind stream processing (Chapter 2). Then we present existing platforms for stream transport and processing (Chapter 3). We select one of each to design a solution that can be used in many scenarios (Chapter 4). Next, we implement real-time analytics for a public transportation system on top of the solution (Chapter 5). Finally, we evaluate the latency of analytics results under different levels of load (Chapter 6).

2 Background

Understanding the challenges that are inherent to the building blocks of stream analytics is key to building a good solution. Therefore, this chapter provides background on characteristics and processing of event streams.

2.1 Batch Processing

For the better part of history, data was processed in form of *batch datasets*. An early analog example of batch data processing is the United States census: when the census was initiated in 1790, horseback riders recorded citizen counts per area and then transported their records to a central location for aggregation. While this is an extreme example, the principle still holds for digital data like periodic database dumps or bulk log transfers found in many batch processing systems today, where the the whole dataset is processed at once after arrival [2, p. 28].

A batch processing system takes a large amount of input data and runs a *job* to process it. The produced result are often analytics, but arbitrary applications like search index building and machine learning feature extraction can be built with this method. Since batch jobs usually take a while to execute, they are not interactive but scheduled to run periodically. For example, web server logs can be imported once per day from the web server nodes and then user behavior analytics are available on the next morning. While latency is high, throughput, i.e. the amount of data processed per second, is a key performance metric since data volume is usually very large [3, p. 390].

As the volume of data grew, dataset became too large to be handled by a single *node*. This sparked the development of distributed processing engines[1] like Hadoop[2] (based on the MapReduce [4] programming model) and Spark[3]. These frameworks tackle two common challenges of large-scale batch processing [3, p. 429], [5, pp. 362–373]:

[1] We use the terms *processing engine*, *processing framework* and *processing platform* interchangeably since their semantic difference is not relevant for our purpose.
[2] `https://hadoop.apache.org/`
[3] `https://spark.apache.org/`

- Scalability: support for distributed processing across nodes requires orchestration and *partitioning*, i.e. the division of the dataset into subsets that can be processed in parallel, possibly on different nodes
- Fault tolerance: guarantee of consistent and correct results even in case of job failures caused, for example, by hardware failure or scheduler-induced preemption

Having a framework to handle these issues makes focusing on the actual problem much easier.

Distributed batch processing engines assume that all functions applied to the data are stateless (no intermediate results are stored) and have no externally visible side effects (e.g., database updates) [3, p. 430]. While these assumptions result in a deliberately restricted programming model, they facilitate distributed execution. Since no state needs to be shared between nodes, partition-based scalability is simple. In case of faults, the job can be restarted using the same input data, and the final output will be the same as if no faults had occurred, assuming deterministic operations. This is possible because input data are stored in a distributed and fault-tolerant file system like Hadoop Distributed File System (HDFS). Therefore, the underlying file system facilitates processing across multiple nodes. Some processing engines store intermediate results to speed up re-computations after failures, but this often requires tracking of data ancestry or checkpointing [3, p. 430].

Batch processing has been successfully applied at massive scales, with Hadoop clusters at Yahoo of 35,000 nodes being used to store 600 PB of data and run 34 million jobs every month [6]. However, it is only suitable for applications where low latencies are not required. Batch engines fall short when real-time processing is required, since they only process data once all input data are available. In practice, most data arrive as a continuous stream but are be divided into batches of a certain size for batch processing [3, p. 439]. An obvious solution might be to decrease the batch size and run the job at a higher frequency, a technique known as *micro-batching*. This can decrease the latency to less than a second, but ultra-low latency applications are still infeasible with micro-batch processing [7]. This is especially true when considering that data might arrive with a delay, which usually requires deferred processing or re-proccessing when late data arrive. Also, jobs that might span batch bounds, such as user session analysis in web applications, are inherently complex to implement [5, pp. 34–35].

Apart from the technical shortcomings, processing a continous stream of data in batches seems wrong from a philosohphical point of view. Batch processing frameworks are fundamentally ill-suited for this type of data. Why not build processing engines specifically designed with continuous streaming data in mind, that can overcome and embrace stream characteristics to enable new types of applications?

2.2 Stream Processing

Stream-native processing, as opposed to batch processing on streams, comes with many challenges, but is ultimately the more powerful approach when dealing with streaming data. It can enable many applications like real-time analysis of IoT sensor data, continuous credit card fraud and anomaly detection, live business process and quality monitoring, among many others. This section is an introduction to streams and stream processing, showing the fundamental characteristics and challenges.

2.2.1 Streaming Data Properties

The terms "stream processing" has been assigned a variety of meanings. Many associate low-latency, approximate, or speculative results with stream processing systems, especially in comparison to batch processing systems [5, pp. 23–24]. While many historic systems had these properties, they are not inherent and should therefore not be used for definitions. Well-designed stream processing systems are perfectly capable of producing correct results. Therefore we use the definition of Akidau, Chernyak, and Lax:

> [A stream processing system is] a type of data processing engine that is designed with infinite datasets in mind [5, p. 24].

Accordingly, a *stream* is an *unbounded* dataset that is infinite in size. Unboundedness means that a stream does not terminate and new data will arrive continuously. Therefore the dataset will never be complete. Many data sources found in the real world produce data naturally as unbounded stream: sensors measurements, stock updates, user activities, credit card transactions, retail purchases, public transportation updates and business activities come from processes that are theoretically infinite (or at least very long-running), so we have to assume that they do not end. This is in contrast to *bounded* datasets found in batch processing, which are regarded as complete.[4]

The reason for the prevalence of batch processing despite the stream nature of most data stems from historical technical limitations of data collection [2, p. 29]. Batch collection was the norm, be it for early census calculations or digital bulk dumps. Now we see a shift to more continuous data processing thanks to automation and digitalization in the data collection process, which reduces latency but also requires new processing techniques. For the census example, this could mean to record births and deaths to produce continuous calculation counts.

[4] This assumption can be made because there usually is a delay between data collection and data processing. Correct results can only be produced if this assumption holds and no data is late.

Streams can also be regarded as *data in motion*. Scanning through the stream, it is possible to observe the evolution of data over time and build a view of the data at a single point in time. Such view are also called tables, which are *data at rest*. Relational databases have traditionally dealt with tables. Capturing the changes of a table in turn yields a stream. Therefore, streams and tables are really just two representations of the same data, a philosophy that many stream processing systems build upon [5, pp. 174–212]. This concept is also known as stream–table duality [8].

Time Domains

A stream consist of *records* that usually contain information about events. Events might, for example, be purchases, website views, temperature changes or the arrival of a bus at a stop. A stream emanates from a *producer* and can be received by multiple *consumers*. When processing an event stream, two time domains are involved [5, p. 29]:

- *Event time*: the time at which the event actually occured in the real world
- *Processing time*: the time at which events are observed at a given processing stage

These two time domains often do not coincide. The processing time can never be before the event time. However, the delay between the occurance and processing of an event can be arbitrarily large. Usually, there is some small base delay due to, for example, network latencies and resource limitations. Other events might occasionally arrive later than expected, for example, when a vehicle broadcasting its position enters a tunnel or people using their phone sit in an airplane. In case historic data are processed, there might even be years of delay between event and processing time. Note that processing time is the natural order in which events arrive and are processed, processing by event time order requires additional effort.

The relationship of the two time domains can be visualized by plotting the progress of processing time over event time as shown in figure 2.1. Events (denoted by the diamonds) occur in event time and arrive at the system in processing time. The delay between these two is also known as *event-time skew* or *processing-time lag* (both terms are two perspectives on the same issue) [5, p. 30–31]. The event-time skew for the green event is shown by the arrow. Events on the diagonal line would have no event-time skew. This would mean that data are processed instantly after occurring, which simplifies processing because events would arrive at the system in event time order. In reality, events are always above this line due to the base delay. However, the delay is not constant. While events occur every 30 s as shown in the top margin, some are observed much faster than others as shown in the right margin. In case of the green, blue and orange events, this even changes their order. This makes the stream (partially) unordered with respect to event

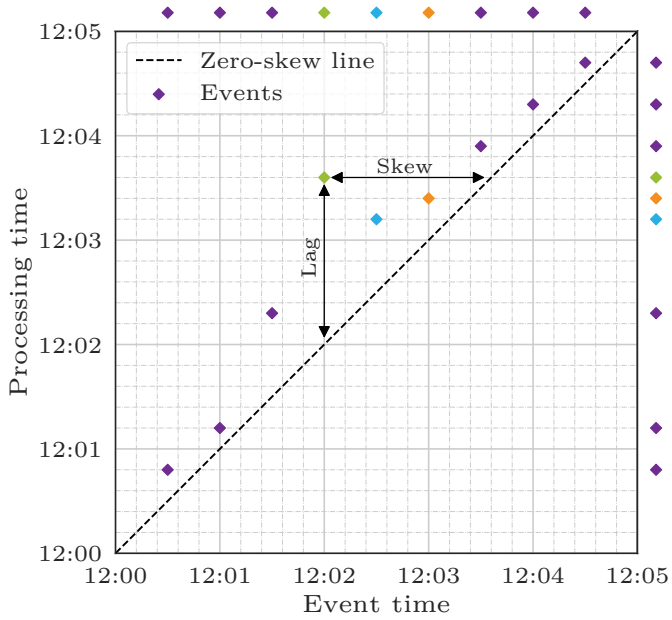


Figure 2.1: Relationship between event time and processing time: time skew varies a lot and leads to out-of-order arrival

time. Handling this skew and unorder is a key challenge stream processing frameworks have to solve [9, p. 3].

2.2.2 Stream Processing Architectures

The unbounded nature of streams, requiring continuous processing, cannot be handled by batch processing engines like Hadoop. While academic and commercial stream processing engines like SEEP, Naiad, Microsoft StreamInsight and IBM Streams have existed before [10, p. 37], Storm[5] was the first one to find widespread adoption when it was released in 2011 [5, p. 375]. Like MapReduce, it solves many of the common challenges like fault tolerance, networking and serialization and allows developers to focus on solving the actual problem [11].

While Storm excelled at providing low-latency results, it did so by sacrificing features like exactly-once processing required for guaranteed correctness. This sparked the development of the Lambda architecture [12], shown in figure 2.2. The batch layer produces correct results and handles fault tolerance and scalability through the underlying processing engine, often Hadoop. Jobs are expressed in the MapReduce framework and store their results in a database optimized for batch writes and random reads for serving. The batch layer naturally lags behind real-time, therefore data is simultaneously processed in a real-time/speed layer, often implemented using Storm. The speed layer provides low-latency

[5] `https://storm.apache.org/`

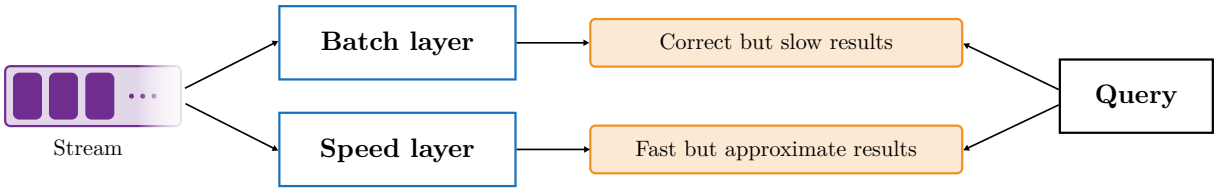


Figure 2.2: Lambda architecture: the batch layer provides correct results, the speed layer provides low-latency results

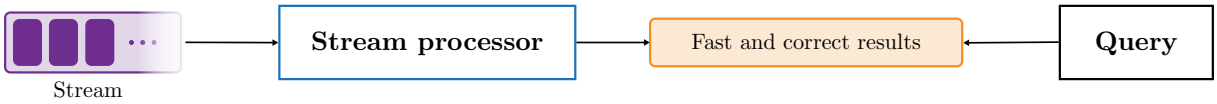


Figure 2.3: Kappa architecture: a single processing engine provides correct, low-latency results

results but lacks in the correctness department due to approximative algorithms or possible system faults. This is acceptable, however, since speed layer results are overwritten by correct batch layer results once available. Even if a speed layer job fails, batch layer results will be available at a later point. This requires the batch layer to store incoming data in an immutable and fault-tolerant way, also enabling recomputation in case processing code changes. By leveraging the two layers, the Lambda architecture provides low-latency, eventually-correct results [13, pp. 14–20, pp. 27–28].

While the Lambda architecture has been used to build many successful systems, it is inherently complex. The processing logic needs to be implemented twice and in both cases specifically engineered towards the processing engine. Even if the logic is implemented in a higher-level API that can be compiled to MapReduce and stream processing jobs, the twofold operational effort remains [14].

The Lambda architecture was born out of necessity since no framework could guarantee both low latency and correctness. However, more and more modern stream processing frameworks are able to provide the batch layer's correctness and the speed layer's correctness in a single system, much simplifying development and operations. This is called the Kappa architecture, shown in figure 2.3. Instead of a storing data on a distributed file system, the stream is often stored in a stream transport platform like Kafka[6] that allows *replay*, i.e. rewinding to an earlier point in time and start reading from there instead of the latest record. This enables fault tolerance and recomputation in case of processing logic changes [14].

Spark Streaming[7] was the first large-scale stream processing engine being suited for use in a Kappa architecture. While not a true streaming but rather micro-batch processing engine, the latency was low enough for most applications. Since micro-batching uses batch

[6] `https://kafka.apache.org/`
[7] `https://spark.apache.org/streaming/`

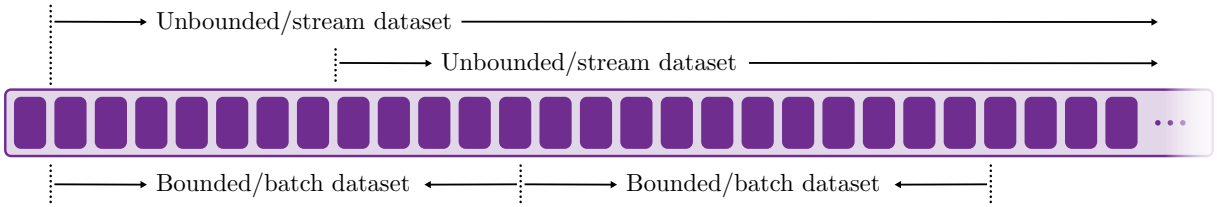


Figure 2.4: Relationship between bounded and unbounded datasets: bounded datasets are fixed-size sections of unbounded datasets

processing under the hood, consistency and correct results were guaranteed. However, Spark Streaming lacked support for processing in event-time order, therefore producing correct results only in case of in-order data or event-time-agnostic computations. Correctness is absolutely required for stream processing engines to achieve parity with batch processing engines. Tools for reasoning about time, and especially event time, are essential for dealing with unbounded streams [5, pp. 27–28]. Sophisticated time handling with high flexibility was explored in Google's company-internal MillWheel [15] framework and Dataflow [1] processing models. Flink[8] was the first open-source framework to incorporate the ideas into a high-throughput, low-latency stream processing engine that supports event-time processing and guarantees correctness.

Another contribution of Dataflow and Flink is the realization that batch and stream processing can be unified. Bounded batch datasets are effectively a section of an unbounded stream dataset, as illustrated in figure 2.4, and jobs can be specified using the same API and be executed on the same engine. However, bounded datasets are amenable to additional optimizations towards throughput at the cost of latency by increasing bundling sizes and computing processing stages successively instead of continuously [10, p. 35], [5, pp. 198–199]. Such a unified processing engine decreases development and operations cost since code and infrastructure can be shared, and allows to balance latency and throughput based on the use case.

Stream processing jobs often consist of multiple stages that are connected into a directed *dataflow graph*[9] [2, p. 30]. The graph starts at one or more stream sources and ends at one or more stream sinks. *Operators* in between can filter, aggregate, join and split streams or transform them otherwise. Operators are essentially the graph's vertices, with streams between sources, operators and sinks forming the edges. A vertex further to the start of the graph (i.e. the sources) is referred to as *upstream*, and a vertex further to the end of the graph (i.e. the sinks) is referred to as *downstream*. An example graph is shown in figure 2.5. The graph can contain also contain loops to enable iterative algorithms like incremental machine learning and graph processing [10, p. 33]. Dataflow graphs provide

[8] `https://flink.apache.org/`
[9] Note that dataflow graphs are a general concept also found in batch processing, and are different from the Dataflow stream processing model.

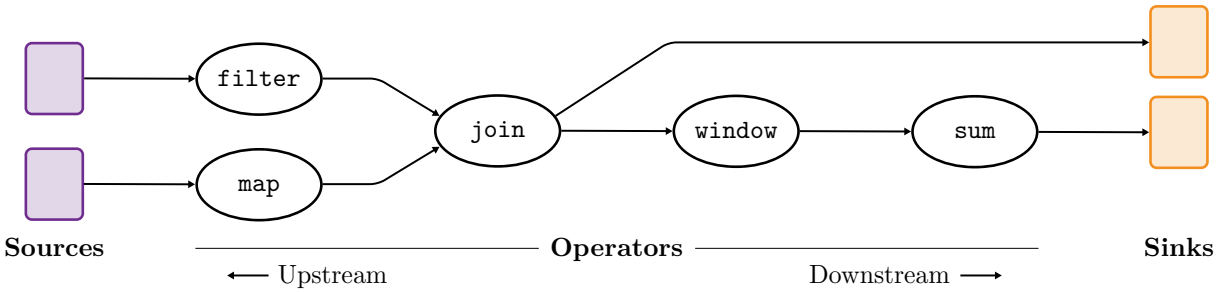


Figure 2.5: Example of a dataflow graph: two streams are transformed, joined, aggregated over windows and emitted

a very flexible programming model for building stream processing jobs, but also allow the processing framework to apply optimizations like operator fusion and fission [16]. Stream processing jobs can also be written in a higher-level abstraction like streaming SQL and pattern matching specifications [17], which are eventually compiled down to dataflow graphs [10, p.30].

2.2.3 Processing Patterns

Streams require processing patterns that support their unbounded nature. There are three major categories: [5, p. 35]:

Time-agnostic When the processing logic is purely data-driven, ordering by time is irrelevant. This makes processing very simple because out-of-order records need not be accounted for, therefore this pattern is supported by even the most basic streaming systems. This includes record-by-record processing like filtering based on a record attribute but also inner joins, where a joined record is produced once the respective records from all input streams have been observed.[10]

Approximation Approximation algorithms like sketches for frequency distribution or distinct-value queries [18] are optimized to handle large quantities of data by trading exact correctness for computational feasability, albeit usually within some error bounds. However, they are often complicated which makes it difficult to invent new ones. Furthermore, they usually work only in processing time, limiting their applicability.

Windowing To handle the unboundedness and lack of completeness of streams, they can be chopped into bounded datasets known as *windows*, which can be processed independently. For example, a stream can be divided into contiguous sections of 1 min, and results like aggregations are computed per section. More complex, even

[10] If many uncompleted joins are to be expected, a timeout-based garbage collection becomes necessary to limit memory requirements, introducing a time component.

arbitrary, windows are also possible. Note that this pattern includes many more time-based processing types that are not immediately obvious. For example, pattern recognition effectively builds a window for each stream record ending with the final record of the pattern, resulting in variable-length windows [19, p. 350]. Additionally, regular windows can be used to limit the records regarded for pattern matching and expire partial patterns to keep state size in check [19, p. 354]. Another example are outer joins, where a joined record can also be produced when the respective records have only been observed from some of the input streams. Outer joins on streams require a timeout after which a partial join should be produced, which effectively determines the window length, where the window contains all records that were regarded for a record's join.

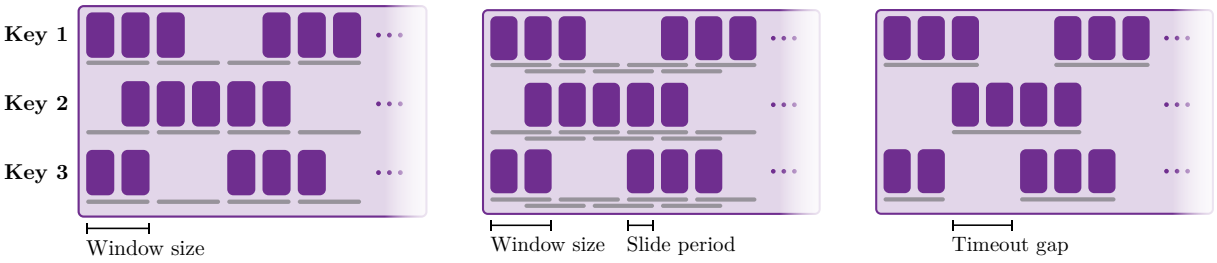
The focus of the rest of this thesis will be on the windowing pattern, since it has unique challenges compared to time-agnostic and approximative processing. Specifically, correct windowing of out-of-order streams requires event-time awareness, and relating data within those window requires keeping consistent and fault-tolerant state. We will refer to this type of stream processing as *stream analytics*, since time and state are required for producing sophisticated and valuable insights. Note that the term "stream analytics" is not well-defined in the literature and the boundary to other techniques like Complex Event Processing, Event Stream Processing, Distributed Stream Computing and Information Flow Processing is blurry [20], [3, p. 466]. For our purpose, we consider any kind of sophisticated stream processing as stream analytics. We will now regard the challenges of time and state from a stream processing job developer's perspective. For an elaborate overview of specific implementations of out-of-order-data management and fault-tolerant state management in early and modern stream processing frameworks, refer to [9, chapters 3–5].

2.2.4 Windowing

Windowing is a key technique for enabling processing of unbounded datasets which inherently lack completeness. Each window of a stream is a finite chunk that is a complete dataset in itself. In this section, we will look at window types and how latency and correctness can be balanced for the use case at hand.

Windows can either be non-keyed (windows apply to the stream as a whole) or keyed (the stream is divided into subsets by key, e.g., per user, to which windows are applied individually). Three commonly found window types are shown in figure 2.6 [1, p. 1794]:

Tumbling/fixed Tumbling windows are defined by a fixed-length temporal window size. For example, a tumbling window of 10 min divides the stream into subsets of data from 12:00 to 12:10, then 12:10 to 12:20, continuing that way until the processing



(a) Tumbling (b) Sliding (c) Session

Figure 2.6: Common window types: aligned tumbling and sliding windows, and unaligned session windows. The elements contained in each window are denoted by the grey bars.

is stopped. Windows may either be aligned or unaligned across keys, depending on if the windows of different keys start at the same time or are staggered by an offset, which spreads window completion load more evenly across time.

Sliding/hopping Sliding windows are defined by a fixed window size and a fixed period. For example a sliding window of 10 min starting every 1 min divides the stream into subsets from 12:00 to 12:10, 12:01 to 12:11, so every record ends up in 10 windows. The window size is often an integer multiple of the period, and sliding windows can also be aligned or unaligned. Note that fixed windows are a a special case of sliding windows where size equals period.

Session Session windows are defined by a timeout gap to capture periods of activity. For example, user activity analysis on a website during one sitting is a common use case for session windows. Session windows are defined per key and the length depends on the data involved, therefore they are inherently unaligned. Because the window length cannot be defined in advance, they are one area where the stream processing excels compared to batch processing. Since sessions may span multiple bounded batch datasets, the dataset must be treated as unbounded. Otherwise, complex stitching is required [5, p. 35].

Apart from these time-based window types, there are also tuple-based windows that contain a fixed number of records. However, they are essentially a form of time-based windows with incrementing logical timestamps [21, p. 47] and will therefore not be regarded further here.

All windows can be defined in both time domains. When windowing by processing time, incoming data are buffered for the specified period and then processed, as shown in figure 2.7a. This is straigtforward because windows are complete as soon as the window time has passed, therefore there are no late data to handle. It works well for many monitoring scenarios, where insights about data as they are observed is desired.

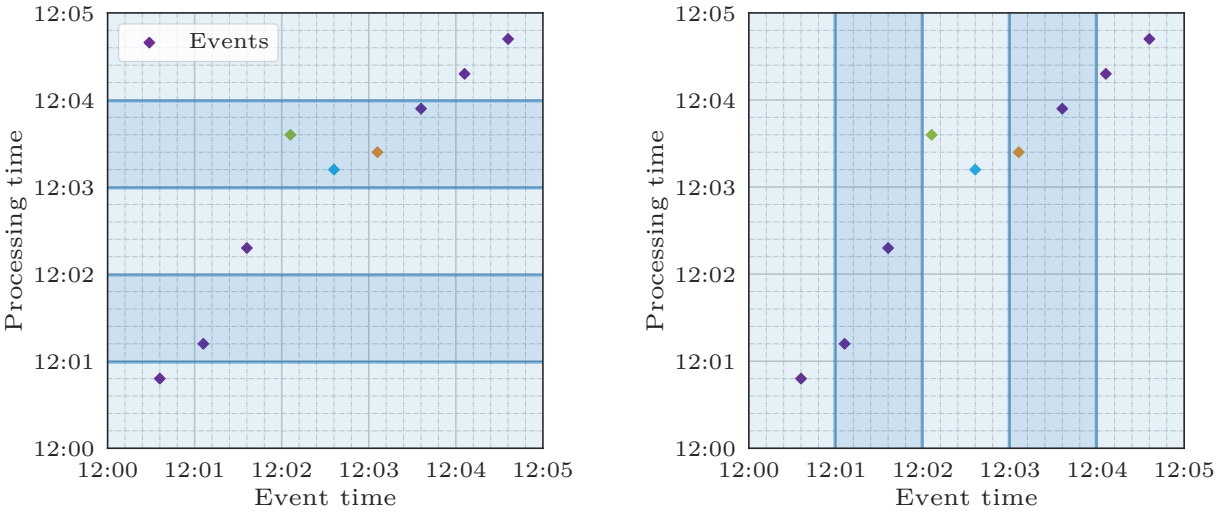
However, most use cases require processing of data in event-time order, but there might be an arbitrarily long delay between an event occuring and the event being processed, which changes the order of events. This can lead to incorrect results if not handled appropriately [5, p. 41]. For example, when recognizing patterns, out-of-order data can result in matches that do not actually exist, and other matches might be missed. In billing applications, where correctness is paramount, quarterly reports might contain incorrect numbers if records end up in the wrong windows. Therefore, windowing by processing time is not sufficient in many cases.

Windowing in the event-time domain, as shown in figure 2.7b requires ordering the out-of-order data to assign them to the correct window. This requires extra effort because event time is not the natural time domain of processing. On the one hand, data need to be buffered longer until the window is closed, therefore windows of the same size are open much longer in event time than in processing time. This demands more resources, but optimizations can be made to, for example, store aggregates incrementally. What is more challenging, however, is judging the completeness of a window. If the event-time skew can be arbitrarily long, it is non-trivial to judge when all data for a specific event-time window have been observed. This simplest approach is to delay processing for a fixed amount of time. For example, if data is usually not delayed for more than 30 s, we can reasonably assumed that all data for this window has arrived when we close the window 35 s after a record with the window end timestamp has been observed. However, this is essentially a tradeoff between latency and completeness (and by extension, correctness), since waiting longer necessarily increases latency but also increases the probability that no data is missed. This black-and-white tradeoff is far from satisfactory for many use cases. Therefore, the Dataflow [1] model introduced fine-grained control over window semantics to balance correctness, latency and cost.

Windowing in Dataflow

In the Dataflow, windowing is strictly event-time-based. However, processing-time windows are possible when assigning the arrival time as the event time. We will now look at the four aspects that enable a clear and flexible definition of windows. For a more detailed description, refer to [5, chapter 2].

Transformations Transformations define what results are produced from the records in a window. This includes aggregations like summing and counting, training machine learning models or detecting anomalies. Depending on the specific transformation, individual records can either be accumulated and processed all at once when window results are *materialized* (i.e. emitted and sent downstream for storage or further



(a) Processing time

(b) Event time

Figure 2.7: Windowing in different time domains: only event-time windowing assigns events correctly despite delayed and out-of-order arrival

processing), or records can be aggregated eagerly to spread computation load more evenly and minimize state size. There are various choices for which event timestamp to assign the result, but often the end of the window is used [5, p. 101].

Windowing Windowing determines which records are grouped together based on some strategy. This includes fixed, sliding and session windows, but custom strategies are supported as well. Custom strategies (but also the built-in ones) consist of window assignment, which assigns records to one or more windows, and optional window merging, which allows merging of windows for window evolution as more data arrive. For example, window merging is required for session windows when a record arrives that connects two sessions which were before separated by the timeout gap [5, pp. 136–146].

Triggers Where windowing determines the location of windows in event time, triggers specify when transformation results are materialized in processing time. This allows windows to be evaluated more than once, where each specific result of the window's transformation is referred to as *pane*. There are two general types of triggers [5, p. 60]:

- Repeated update triggers: these trigger window evaluation periodically, either after a specific count of records or at some processing-time frequency, such as every minute. The choice of period is primarily a tradeoff between latency and computation requirements.
- Completeness triggers: these trigger window evaluation when they believe that all data for the window has been observed, and therefore the window is complete.

	Delta	Value	Value and Retracting
Pane 1: inputs=[3]	3	3	3
Pane 2: inputs=[6, 1]	7	10	10, -3
Value of final pane	7	10	10
Sum of all panes	10	13	10

Table 2.1: Example of windowing output modes

Repeated update triggers show evolving results over time that converge towards correctness, but they do not indicate when correctness is achieved [5, p. 63]. Therefore, completeness triggers may be more appropriate for use cases where correctness is important.

Output Mode The output mode describes how different panes, i.e. subsequent evaluation results of a window, are related and refine previous results. Therefore, the choice is only relevant if windows are triggered multiple times. We will use the naming proposed in [5, p. 94] instead of the original naming from the Dataflow paper for clarity. There are three types of output modes, with an example of two panes shown in table 2.1:

- Delta: upon triggering, the result is materialized and any stored state is discarded. Therefore, successive panes are independent of each other. For example, when summing input records, only the sum of all panes will yield the total sum for the window.
- Value: upon triggering, the result is materialized but stored state is retained. Therefore, successive panes build on each other's results. For example, when summing input records, each pane contains the total sum for the window so far.
- Value and retracting: upon triggering, the result is materialized and any stored state is discarded. Additionally, previous panes are explicitly retracted. For example, when summing input records, each pane contains two parts: the total sum for the window so far, and a retraction for the old sum.

The choice of output mode usually depends on the input expected by downstream consumers. Aggregating consumers might expect deltas, while databases that are updated with new data require values.

These four composable pieces provide flexible tools to balance correctness and latency by adjusting trigger frequencies and output modes, but also cost by affecting compute and memore requirements. Completeness triggers play an important role for correctness, but can be hard to implement, especially when event-time skew is highly variant. *Watermarks*

are an approach to indicating input completeness in the even-time domain [5, pp. 64–66]. The watermark denotes the point in event time up to which the system believes all inputs with lower event timestamps have been observed. In other words, the watermark is an assertion that no more data with event timestamps earlier than the watermark will arrive. Completeness triggers can trigger window materialization once the watermark passes the window end in the belief that no more records will be assigned to that window. Note that watermarks must be monotonically increasing [5, p. 88].

Watermarks can be a strict guarantee or an educated guess of completeness. Perfect watermarks are possible when the system has full knowledge of all input data, for example, when assigning arrival timestamps as event timestamps. In some cases, the data source itself might produce watermarks. *Late data*, i.e. data with an event timestamp earlier than the watermark that arrive past the watermark, will never occur. In most practical applications, only heuristic watermarks that approximate a perfect watermark based on the available information are possible. Heuristic watermarks can be generated by incorporating knowledge of the sources [5, p. 66], but also in form of percentile watermarks [5, pp. 106–108] based on the event-time skew distribution, if known. This would, for example, enable watermarking after 99% of all data are believed to have been observed, decreasing latency by ignoring stragglers. Another common strategy is by specifying a fixed bound for event-time skew, limiting the expected out-of-orderness. For example, the watermark could always lag 10 s behind the latest known timestamp if we know that the event-time skew will never exceed 10 s.

While watermarks are very useful to judge window completeness, they have two shortcomings [5, pp. 68–69]. Watermarks may sometimes be too slow, which increases latency. This might either be the case because the data really have a high delay, or because the watermark generation overestimates the delay. On the other hand, heuristic watermarks might be too fast due to their approximate nature, in which case late data might arrive after the watermark. Therefore, watermark-based completeness triggers alone cannot provide both low-latency and correctness.

This motivates the use of multiple triggers per window. Early repeated update triggers compensate for watermarks being too slow by periodically providing early results which are incomplete. A single on-time trigger based on the watermark materializes results which the system believes to be correct. In case the heuristic watermark was too fast, late repeated update triggers refine results when late data arrive. Often, the late trigger fires for every late data record, which can drastically increase the number of computations in case of watermarks that are too fast by a large margin. Note that the output mode needs to be set appropriately when windows might be triggered more than once. This ensures that downstream consumers process multiple panes per window correctly.

Window state needs to be retained after the watermark when late triggers are enabled. Due to practical resource limitations, a maximum *allowed lateness* in processing time must be specified. After a window is completed by a watermark, state is expired after the maximum allowed lateness. Any record that arrives later will be discarded. Since the value of data diminishes with time, trading off resource cost for data value is usually sensible

2.2.5 State Consistency and Persistence

Any stream analytics solution that does not process streams record-by-record but correlates multiple records is stateful. For windowing, state consists of intermediate aggregation results. For pattern recognition, state consists of partial matches. For online machine learning training, the state consists of the current model parameters. Since stream processing jobs are effectively intended to run forever, interruptions like node failures, infrastructure maintenance or code changes are inevitable. To ensure correct results, the state needs to be persisted in a fault-tolerant way. This is especially important, since unbounded datasets usually cannot be replayed in their entirety, either because they are not retained forever, or because it is computationally infeasible [5, pp. 216–218]. Simply storing state externally in a database can become a bottleneck [22, pp. 1718–1719]. Compare this with batch processing, where it is often assumed that the dataset can be reprocessed in its entirety until the job succeeded.

Therefore, correct and efficient fault tolerance in stream processing requires persistent state that can be checkpointed. *Checkpointing* is the process of persisting in-memory state to a durable storage medium. This state needs to be exposed to the stream processing framework for management. To expose state, frameworks usually provide a flexible API with support for a variety of data structures [5, p. 228], often with efficient implementations of lists and maps [22, p. 1721]. Apart from checkpointing for fault tolerance, this allows state redistribution during cluster scaling and alleviates the developer of needing to implement efficient persistence [22, pp. 1718–1719].

After fault recovery, the processing framework needs to guarantee that any materialized results are identical to the results if no fault occured. This is required for *consistency* [9, p. 15]. The key to consistent and correct results is *exactly-once processing*. This means that every stream record is guaranteed to be processed exactly once even in case of failures. For example, assume that a job counts the number of records in a stream, as shown in figure 2.8. After having processed record 5, the intermediate count is checkpointed but the job fails and needs to be restarted on another node. If the restarted job starts earlier than record 5, the total count will be higher than the actual count. This is referred to as

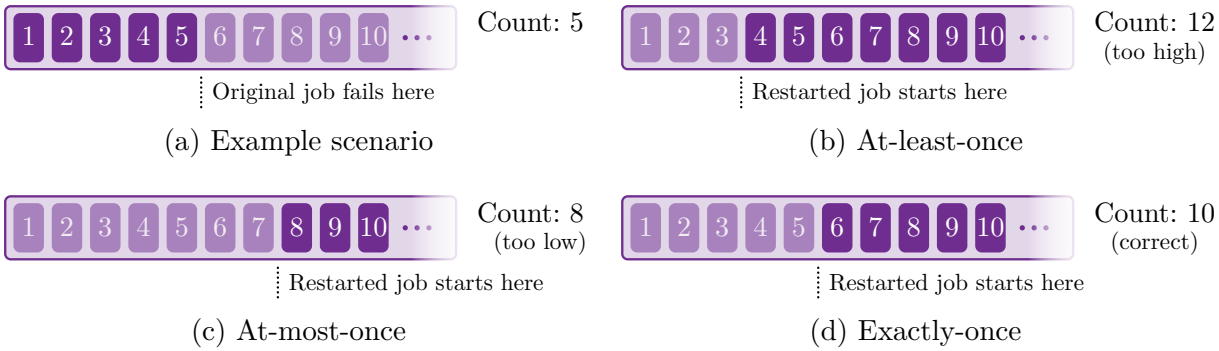


Figure 2.8: Example of different processing semantics after failure recovery: only exactly-once processing can guarantee correct results

at-least-once processing, since each record is guaranteed to be processed at least once. On the other hand, of the restarted job starts after record 5, the total count will be lower than the actual count. This is called *at-most-once processing*. Only if the job is guaranteed to be restarted from record 5 for exactly-once processing, the total count will be correct.

Exactly-once processing needs to be end-to-end, which means that checkpointing not only needs to consider application state but also sources and sinks. A persistent and immutable source is required to be able to replay the stream from the last checkpointed position [22, p. 1722]. If the stream is ephemeral, only at-most-once processing might be possible since the stream cannot be restarted from the correct position. The sink needs to be either idempotent (like most databases) or support transactions (like using two-phase commits) [22, pp. 1725–1726]. To enable end-to-end exactly-once processing, the persistent state consists of the actual application state, but also needs to include the position in the stream on which that application state is based. Based on these information, a recovered job can provide consistent and correct results. However, exactly-once processing is not possible due to the nature of some sources and sink, and consistent and correct results cannot be guaranteed in those cases.

Most stream processing frameworks do not offer true exactly-once processing due to performance reasons, but rather *effectively-once processing*. This means that each record will only affect the results once, but might actually be processed multiple times in case of a job restart due to failure. This raises consistency issues, since non-deterministic operations might produce different results for the same inputs. For example, a database lookup for stream enrichment might return different data of a table has been updated in the meantime. Frameworks cope with non-determinism by checkpointing results from such transformations [5, pp. 155–156] or simply assuming deterministic transformations [22, p. 1722].

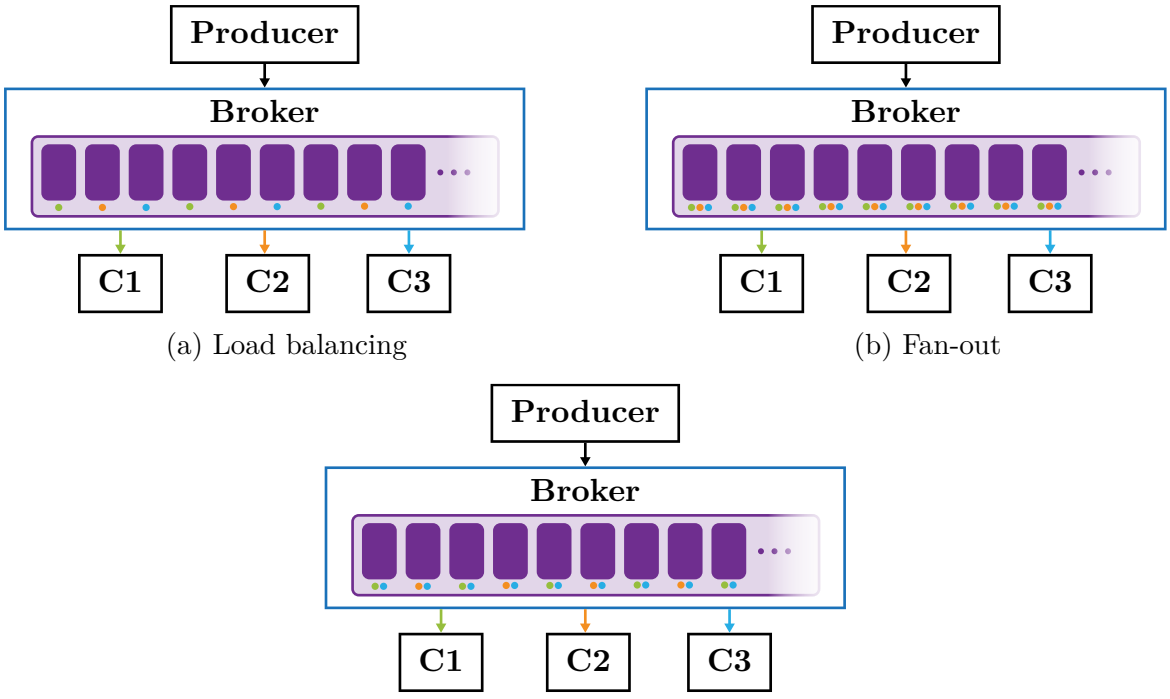
2.3 Stream Transport

Having regarded streams as a concept, we now show how streams are physically manifested, stored and transported around. A *stream transport* system is responsible for moving streams between producers, processing systems and consumers. In its simplest form, a direct TCP connection can transport records from upstream producers to downstream consumers. For low-latency applications, UDP multicasting can be used. Brokerless messaging libraries like ZeroMQ[11] implement publish/subscribe messaging on top of these network protocols. Even HTTP and RPC requests can be used to push records from producers to consumers. However, such direct messaging methods fall short in terms of delivery guarantees and fault tolerance [3, pp. 441–441].

Message brokers like ActiveMQ[12], Pulsar[13] or RabbitMQ[14] provide a way to decouple producers and consumers using common protocols such as AMQP or JMS. The broker can store records durably to cope with offline or slow consumers by persisting them on disk until delivery is acknowledged. Many are also distributed for fault tolerance in case of broker node crashes. The decoupling makes record transport asynchronous, since producers do not know when a message is delivered to consumers [3, p. 443]. Broker-centric instead of direct messaging furthermore allows multiple consumers to receive a stream without needing to change the producer, enabling organizational scalability and flexibility [2, pp. 20–22] as well as debugging and monitoring by peeking into the stream [2, pp. 31–32]. There are two patterns when multiple consumers receive the same stream, as illustrated in figure 2.9 [3, pp. 444–445]. Load balancing means that each record is delivered to only a single consumer, so processing load is spread over all consumers. Fan-out means that each record is delivered to all consumers, which allows for independent consumers to receive the full stream. Combinations of these two patterns are possible as well, where records are load-balanced between some consumers but others receive all records.

However, when using message brokers as transport platform in streaming systems, they have a number of shortcomings. First, they might deliver records more than once in case of lost acknowledgements, which might break exactly-once semantics or require deduplication in the processing framework. Secondly, most brokers do not make guarantees about record order, which requires special consideration in the processing framework. Thirdly, message brokers are ephemeral by design, which means that messages are discarded in the broker once delivered [3, pp. 445–446]. This prevents stream replay for fault recovery or reprocessing after processing code changes.

[11] `https://zeromq.org/`
[12] `http://activemq.apache.org/`
[13] `https://pulsar.apache.org/`
[14] `https://www.rabbitmq.com/`



(a) Load balancing

(b) Fan-out

(c) Mixed

Figure 2.9: Message distribution patterns with multiple consumers: the colored dots denote which consumer receives a record

2.3.1 Immutable Logs

Message brokers that use a log as data structure do not have the shortcomings of regular message brokers. A *log* is an append-only sequence where each record is assigned a monotonically increasing number called *offset* [2, pp. 1–3]. Records are stored in the order they are appended and cannot be updated or deleted, therefore logs are immutable and are totally ordered. Log data structures are commonly found in databases for in form of write-ahead logs or replication logs, but also in many distributed consensus systems [23, pp. 54–66]. When used for streaming, producers append new records to the end of the log, and consumers receive records by reading the log sequentially. Once a consumer reaches the end of the log, it waits for new records to be appended. Fan-out for multiple consumers is trivial since records are retained.

The total ordering solves the at-least-once delivery and out-of-orderness problems of traditional message brokers, since the offset enables consumers to know exactly which records they have or have not read yet. This offset can be part of state checkpointing as described when talking about state consistency in subsection 2.2.5. A configurable, possibly infinite, retention period instead of ephemeral records is feasible since performance does not degrade with increasing data size [24, p. 3]. This enables stream replay which is required for exactly-once processing in case of faults. In fact, fault tolerance in many modern stream processing systems was only made possible by fundamentally relying on

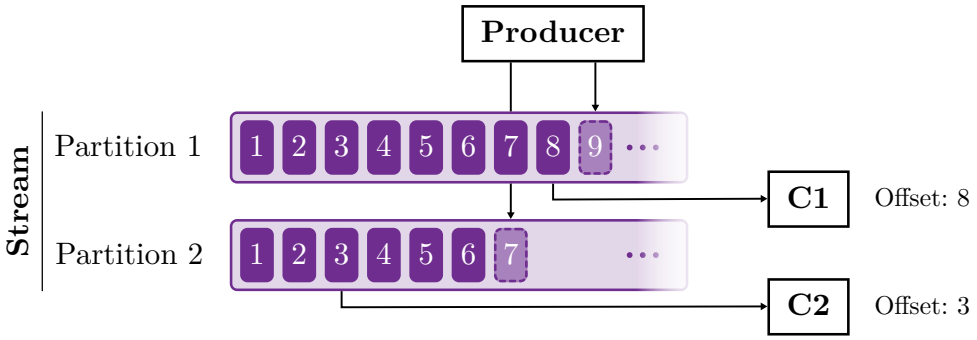


Figure 2.10: Log partitioning: each partition is an independent log with internal total order. Load balancing is used as multi-consumer pattern in this example, but multiple consumers could also read the same partition for fan-out.

this ability of log-based sources [5, pp. 390–391]. The ability to replay old data is also useful for reprocessing after bugfixes and for development and regression testing. While this capability is very common in batch processing, durable logs offer a robust and reliable streaming alternative [5, p. 390].

Durable log transportation platforms also provide an isolation layer between producers and consumers [3, pp. 450–450], [2, p. 32]. The log acts as large buffer that prevents consumers from being overwhelmed by a high volume of data by allowing consumers to read records at their own pace. Once the produced volume decreases, the consumer can catch up without needing to worry about data loss. This moves backpressure handling from the consumer to the transport layer.

A stream can be divided into *partitions* that contain the same record types but are otherwise separate logs that can be read and written to independently from other partitions [2, pp. 24–26]. This is shown in figure 2.10. Partitions enable efficient distributed logs, since each partition can be hosted on a different node for horizontal scalability without synchronization overhead. Additionally, partitions can be replicated to other nodes for fault tolerance. Partitions also enable load balancing, since each partition can be assigned to a single consumer. In that respect, partitions are the smallest unit of parallelism [24, p. 4]. While each partition is totally ordered within itself, there is no global order between partitions. In practice, this is only a minor limitation since processing jobs often handle partitions independently (e.g., in keyed windows). In that case, it makes sense to use the same key for determining the log partition, since a random partition is selected by default.

2.4 Event Processing

So far, we have regarded streams of arbitrary records that have a timestamp. Usually, each record represents an event in the real-world. Many authors [9], [25] go as far as treating stream records and events synonymously. Events are usually generated continously and therefore lend themselves well to be treated as unbounded stream, leveraging the processing and transport methods mentioned before. Thus, event processing can also be implemented as operator in a general-purpose stream processing framework instead of using a standalone event processing engine. However, the notion of events has been used long before stream processing frameworks gained popularity, for example in active database systems arising around 1990 [26, p. 5]. Therefore, we will regard events and event processing detached from stream processing in this section.

An *event* is an occurence, something that has happened, within a particular system [27, p. 4]. That system is often the real world, but may also be artificial like a simulation. Change events refer to significant changes of an environment, while status events refer to the observation of some value, even if it remained the same [28, p. 1]. Events are always time-related and are therefore assigned a timestamp. Events can also have attributes that provide more detailed information like the value of a sensor readings. The attributes are defined in the *event type* which determines the event's semantic intent, and individual instances of an event type are called *event occurences* [27, pp. 62–63]. For example, events of type "environment measurement" might always have the attributes "temperature" and "humidity", and instances of the event happening at different times have different values for these attributes. Events originating in the real world might have a geographical location as attribute, making the event a *geo-event*. *Complex events*, also called composite events, emerge as result of relating multiple events in event processing, for example through windowed aggregation or pattern matching. In this context, regular events are also referred to as *primitive event*.

Processing of events comes in two flavors [27, pp. 10–11]. *Event-based programming*, sometimes called *event-driven architecture*, uses events for interaction between components of a system. This asynchronous model provides better decoupling than synchronous request–response communication, since event producers have no expectation of event consumers' actions [27, pp. 33–34]. This event-based communication, can be as simple as a notification, in which case the attribute size is rather small, often just an identifier which consumers can use to lookup further information. In other patterns like event-carried state transfer or event sourcing, that information is itself stored in an event attribute [29]. Event-driven architectures are described in-depth in [27].

In the other flavor, events are the subject of filtering, transformations and pattern recognition instead of a means of communication [27, pp. 121–127]. This type of event processing is very similar, if not identical, to stream processing, and many concepts like event-time ordering and windowing apply to both in the same way. Therefore, treating stream records specifically as events instead of arbitrary objects that happen to be events is rather a semantical than technical difference. This thesis will focus on the second processing type, and we will approach event stream processing from the streaming side rather than the event side.

Pattern recognition over event streams emanates from the event domain, therefore it makes sense to regard stream records as events. *Pattern recognition* is the process of matching a stream of events against some pattern, and detecting instances of the pattern in the stream. This technique can be used to detect anomalies, monitor patients or stock tickers, or do predictive analytics. Pattern recognition is sometimes referred to as Complex Event Processing, but we will avoid this term due to its ambiguity in the literature. Processing of events in the order that they happened in is often crucial for correct pattern recognition. While standalone pattern recognition engines have existed for a long time, libraries built on top of stream processing frameworks can leverage their event-time ordering guarantees and allow pattern recognition to be easily integrated into stream processing jobs.

The *event pattern* is the template specifying one or more combinations of events to match the stream against. There is a wide range of pattern types that can be combined, but the most common include [27, pp. 219–236]:

- Logical expressions: “events of type A, B and C occured”
- Comparisons: “an event of type A with attribute x larger than 10 occured”
- Temporally ordered sequences: “first, an event of type A, then multiple events of type B, and finally an event of type C occured”
- Trends: “multiple events of type A occured with attribute x increasing monotonically”
- Spatial distance: “events of type A and B occured within 5 km of each other”
- Spatial direction: “events of type A and B occured with A moving towards B”

The pattern recognition process is governed by policies that, for example, determine which events are selected for matching, whether events can participate in multiple matches and whether matches can overlap [27, pp. 237–242]. The combination of patterns and policies provides rich tools to express a variety of situations and can therefore be applied widely.

3 Streaming Platforms

After having regarded the theoretical foundations of streaming, we now look at existing implementations of stream transport and processing platforms that are available to the public (some companies have custom platforms for internal use only). We will present them under the aspects of correctness, fault tolerance, latency and scalability.

3.1 Stream Transport Platforms

Stream transport platforms are responsible for moving streams between producers and consumers, but may also store them for replay. Theoretically, any message queue or publish–subscribe message broker like ActiveMQ, Pulsar or RabbitMQ but also hosted services like Google Cloud Pub/Sub[1] could be used for this task. However, these do not provide the end-to-end exactly-once[2] and ordering guarantees as well as stream retention required for correct and fault-tolerant stream processing. Log-based messaging systems, on the other hand, store stream records in order and can be used as exactly-once sources by including the record offset in checkpointed state. Kafka and Kinesis Data Streams[3] are two publicly available log-based stream transport platforms[4].

Kinesis Data Streams is a fully managed and reliable service hosted in the AWS cloud [32] and allows easy integration with other AWS offerings. The number of partitions per log, retention period and record encryption are the only configuration options. Up to 1 MB or 1000 records can be produced to each partition per second, and up to 2 MB can be consumed from each partition per second. Records can be retained for up to 7 days, which is sufficient to allow replay for fault tolerance in stream processing, but generally not enough for long-term retention to support data reprocessing in retrospect. While Kinesis

[1] `https://cloud.google.com/pubsub`

[2] RabbitMQ has limited support for being used as exactly-once source but not as exactly-once sink since transactions are not supported [30]. Pulsar can be used as exactly-once source using the same mechanism and as exactly-once sinks once transactions are implemented in Pulsar 2.7.0, therefore Pulsar could become a viable Kafka alternative for some applications in the future [31].

[3] `https://aws.amazon.com/kinesis/data-streams/`

[4] The actual implementation of Kinesis Data Streams is unknown since it is a managed service, but because only appending is supported we regard it as log for all intents and purposes.

Data Stream's operational cost is about four times lower than Kafka's, it lacks flexibility and throttles producers quickly [33].

3.1.1 Apache Kafka

Apache Kafka is a widely used [34] open-source stream transport platform first developed at LinkedIn. Kafka was the first system to leverage logs for high-throughput, low-latency messaging [24] and has been very influential on the design of modern stream processing platform by enabling fault-tolerance through replay [5, pp. 390–391]. Records can be retained based on a time-based or size-based policy, with support for infinite retention which makes Kafka not only suitable for stream transport but also for persistent storage, effectively making it a file system for streams. This enables streams which show the evolution of data as primary data source, instead of traditional tables which represent data at a specific point in time[5]. For example, the New York Times stores all historic and present articles and assets like images and tags in a single Kafka partition for total order, and every update to content does not overwrite old data but is appended to the log [36]. For serving, the log is replayed from the beginning and an index is built for efficient lookup of content. However, Kafka is usually used as continuous event streaming platform. For in-depth information on development with and operation of Kafka as well as internals, refer to [37].

Nodes in a Kafka cluster are called *brokers* and serve as distributed storage of *topics*. Topics are Kafka's notion of streams that contain a certain event type, and each topic consists of one or more *partition*. The logical position of a record in a partition is called the offset. Records are guaranteed to be stored in the order they are appended to the log within a single partition, but not across partition of a topic. The retention period can be configured per-topic, with records exceeding the retention period being removed from the beginning of the log. Logs can also be compacted, which means that only the most recent record for a key is retained. This can make sense to keep log size small if applications are only interested in the most recent record for a key. Partitions can be replicated across the cluster for fault tolerance. Each partition has a broker acting as partition leader, which clients connect to to consume existing records or append new records to a partition log, as shown in figure 3.1. Follower replicas do not serve client requests but replicate records from the leader, and can be promoted to become the new partition leader in case the current leader crashes. Only in-sync replicas, i.e. follower replicas that have caught up with the current leader, are eligible for leader election. Cluster metadata like broker membership,

[5]This approach leverages the stream–table duality to "turn the database inside out" [35], [3, pp. 459–462], [5, pp. 174–212]

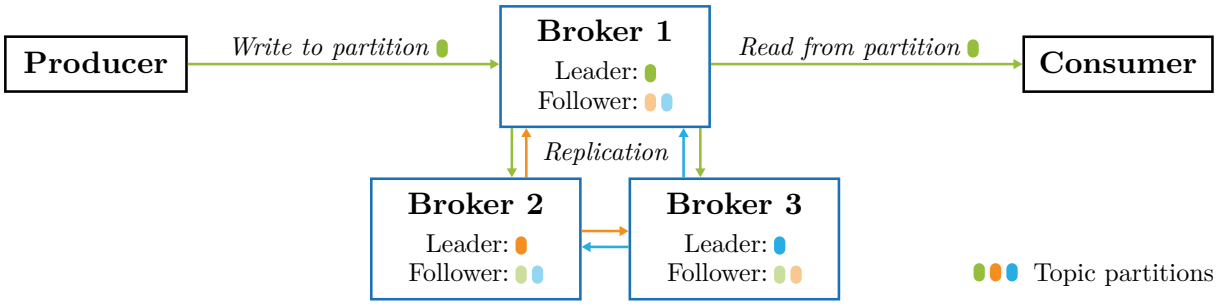


Figure 3.1: Structure of a Kafka cluster: partitions of a topic are spread across brokers acting as leader and follower replicas

partition locations and topic configurations are stored in Zookeeper[6], a distributed and highly reliable configuration service required to run Kafka. However, this dependency will be removed in future releases to simplify deployment.

Kafka supports flexible consumption patterns as previously shown in 2.9 using *consumer groups*. Each partition of a topic is assigned to one consumer in a consumer group. Therefore, load balancing can be achieved when every consumer belongs to the same consumer group. On the other hand, if all consumers are part of different consumer groups, every consumer receives all records for fan-out. Mixed patterns are also possible by having multiple consumer groups with multiple consumers. When using the load balancing pattern, Kafka effectively acts like a traditional message queue, whereas fan-out is more similar to publish–subscribe semantics. Offsets between consumers of a consumer group are coordinated via special topics, which becomes necessary when consumers within a group change and partitions need to be rebalanced between consumers. Consumers commit their offsets regularly to avoid duplicate consumption. An important design decision is the use of consumer polling instead of broker pushing of records. This allows consumers to control the pace of consumption and not be overwhelmed, but also enables simple seeking to read from an earlier or later offset. Kafka brokers use a zero-copy method when serving consumers, which means that Kafka sends records directly from disk to network without intermediate buffers, which is enabled by using the wire format for storage. Multiple brokers called bootstrap servers can be specified in the consumer for the initial connection to the cluster, after which the consumer can ask for the desired partition leader.

Producers also use bootstrap servers to connect to the partition leader for appending records to the log. Producers can control how many replicas must have received the record before the write can be considered successful. A producer may not wait for a reply from the partition leader at all to achieve very high throughput. A producer may wait for a success reply from the leader to be able to retry in case of leader failure. At the highest

[6] `https://zookeeper.apache.org/`

level of write safety but also the highest latency, a producer may wait for all in-sync replicas to have received the message. Producers can also batch multiple records in one write request to improve throughput at the cost of latency. The partition a new record will be assigned to is determined by a partitioner. The default partitioner assigns partitions based on a key like a user identifier or at random if none is given, but custom partitioning strategies are supported. Random partitioning balances partitions evenly while key-based partitioning might lead to high imbalances with hot keys. Depending on the configuration, records sent from a single producer may or may not be appended to the partition in the same order. Events must be serialized before being stored as binary payload in a record. Kafka provides basic serializers, but often custom serializers based on JSON or a generic serialization library like Avro[7], Protocol Buffers[8] or Thrift[9] is used. Kafka also supports transactions for atomic writes of multiple records across multiple partitions, which enables producers to implement exactly-once stream sinks. The prevent consumption of records of uncommitted transactions, consumers must support the isolation level directive.

Due to Kafka's popularity, a rich ecosystem of commercial offerings and libraries has emerged. Confluent, a company founded by one of the inventors of Kafka, provides a commercial Kafka platform with features like schema registries (centralized store for serialization schemas to be shared between consumers and producers) and tiered storage (offloading of old retained data from broker nodes to cheaper and scalable store like S3, also accelerating failure recovery and cluster rebalancing). Cloud platform providers like Amazon and Google also have fully managed offerings for Kafka, combining the flexibility and configurability of Kafka with the ease of operation of Kinesis Data Streams. Other tools like Kafka Connect allow the ingestion of streams from other systems to integrate Kafka into existing technology landscapes, and even more tools exist to mirror whole clusters, monitor performance and availability or run queries on streams.

3.2 Stream Processing Platforms

Stream processing platforms are also part of the stream transport platform ecosystem, consuming and producing streams while doing processing in between. They can also act as bridge between different transport platforms. The platform facilitates the development of processing jobs by taking care of aspects like event-time ordering, fault tolerance of state and scalability, and providing APIs to perform transformations on streams like windowing or pattern recognition. There is a number of frameworks with different approaches and paradigms that have been tried over time, refer to [9] for a comprehensive overview of the

[7] `https://avro.apache.org/`
[8] `https://developers.google.com/protocol-buffers`
[9] `https://thrift.apache.org/`

evolution and features of stream processing systems. We will focus on modern frameworks that can guarantee correct and consistent results through exactly-once processing. Therefore, we will not regard Samza[10] (which only supports at-least-once processing), Spark Streaming (which has no support for event-time ordering and uses micro-batching) and Storm (although rudimentary stateful processing is supported with the Trident extension). However, they might still be suitable for certain applications. For example, Spark Streaming can leverage the extensive machine learning features of the Spark ecosystem. Kafka Streams[11], Spark Structured Streaming[12], Hazelcast Jet[13] and Flink are open-source stream processing frameworks which are suitable for stream analytics. All frameworks support high throughput and low latency, but exact numbers depend on the workload and configuration and will therefore not be compared here. Some performance results are presented in [38]. Note that all frameworks are under active development and new features are developed constantly, therefore existing limitations might be overcome.

Kafka Streams is a lightweight processing framework designed to run alongside the transport cluster, simplifying deployment because no separate cluster is needed but limiting it to Kafka as stream transport platform. Low-latency exactly-once processing with event-time ordering is supported. State can be stored in special compacted changelog topics for fault tolerance and queried from external applications. Kafka Streams also transparently handles rebalancing of partitions when adding and removing processing instances through consumer groups. Kafka Streams applications perform stateless or stateful transformations on stream and table abstractions as well as joining and windowing. While not having the flexibility of the Dataflow mode, windowing in Kafka streams supports all three common window types and early/late triggers. In general, Kafka Streams is suitable for many processing types but is limited to Kafka sources and sinks, and the simple deployment lacks flexibility.

Spark Structured Streaming is a stream processing API built on top of the Spark batch processing framework. Jobs can be submitted to regular Spark clusters and run there in a scalable and fault-tolerant fashion. Kafka and Kinesis Data Streams can be used as stream sources and sinks. Jobs can be run in two modes. Exactly-once processing can be guaranteeed in micro-batching mode, while only at-least-once processing is possible in continuous processing mode. However, latencies can only be as low as 100 ms in micro-batching mode whereas latencies of 1 ms can be achieved with continuous processing. Spark Structured Streaming supports all common stream operations, but uses the latest event timestamp as watermark, which can lead to incorrectly discarded data in case of varying event-time skew [5, p. 386]. Spark Structured Streaming does not use a dataflow graph

[10] `http://samza.apache.org/`
[11] `https://kafka.apache.org/documentation/streams/`
[12] `https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html`
[13] `https://jet-start.sh/`

processing model but uses an incremental version of the static queries on an unbounded table from the batch API [39, p. 601]. Compared to other frameworks, Spark Structured Streaming has not many unique advantages. However, stream processing jobs can be run on an existing Spark cluster and existing developer know-how from Spark batch jobs can be applied to Spark Structured Streaming.

Hazelcast Jet is the the newest of the four frameworks, having been released in 2018 with focus on high performance and easy maintenance [40]. Hazelcast Jet is built on top of the Hazelcast In-Memory Data Grid to support distributed in-memory data structures and lookups for stream enrichment. Jobs can be run on a a cluster for scalability and fault tolerance. The framework comes with many connectors for sources and sinks, and supports exactly-once processing with Kafka. While it has no SQL-like API, common stream operations with event-time ordering are supported, including early triggers. It also integration with change data capture software for relational databases, and supports batch processing with a similar API to stream processing. Coroutines with shared threads instead of individual threads are used for executing tasks, maximizing CPU utilization and reducing context switching overhead. Hazelcast Jet is a promising stream processing engine, but still immature due to its young age.

While Beam[14] is not a stream processing engine, it is still worth mentioning in this context. Beam is an abstraction layer that allows to write stream and batch processing jobs which can be executed on a number of stream execution frameworks, including Flink, Hazelcast Jet, Spark and Samza but also Google's fully managed Cloud Dataflow[15] platform. Beam is one implementation of the Dataflow paper [1] and therefore provides an API for all windowing concepts presented earlier. While Beam was designed as portability layer for jobs, the exact features like exactly-once processing and window output modes are dependent on the underlying stream execution framework.

3.2.1 Apache Flink

Flink is one of the most popular distributed processing framework [38, p. 54] for stream and batch jobs. It was the first open-source platform to support event-time ordering and guarantee exactly-once processing with consistent managed state while achieving high throughput and low latency [10, p. 37]. Flink is successfully being used in production for search index building, online machine learning, real-time activity monitoring and alerting, dynamic ride pricing and continuous ETL [41]. There is also a number of fully managed and commercial offerings for Flink. The framework can connect to a number of sources and

[14] `https://beam.apache.org/`
[15] `https://cloud.google.com/dataflow/`

sinks, including Kafka, Kinesis Data Streams, RabbitMQ, ActiveMQ, Cassandra, Redis and HDFS. Refer to [10], [42] and the official documentation for more detailed information on job development for Flink as well as its internals.

The Flink stack is shown in figure 3.2. Flink can run on a single node, for example during development, but is usually distributed across nodes as a cluster, possibly on top of a resource manager like YARN or Mesos, or containerized on Kubernetes. The runtime which executes dataflow graphs sits at the core of Flink. Dataflow graphs are specified using the DataSet API for batch processing or DataStream API for stream processing. Using the same engine for both bounded and unbounded data is possible since batch datasets are effectively special streams that are complete at the time of processing. however, the created dataflow graphs are optimized for either type. For example, operators in the graph are blocking when processing bounded datasets. The entire dataset is processed by the operator before forwarding the intermediate stream to the next operator. This stages scheduling increases latency but is more efficient than continuous processing which is necessary for streams. On top of these two APIs, Flink comes with domain-specific libraries for graph processing and pattern recognition, with a machine learning library being under active development. Additionally, operations known from relational databases on streams and bounded datasets are possible using the table API or SQL. These levels of abstraction allow the job developer to trade off between conciseness and expressiveness of the programming model, and open up stream processing to more users, possibly even non-experts who are only familiar with SQL.

API and Time Handling

We will focus on operations on streams using the DataStream API. Common stream operations like `filter` and `map` but also splitting of a stream into multiple streams as well as joining of records of multiple streams based on some key are supported. Streams can be logically partitioned into keyed streams using the `keyBy` operation based on a key selector. All supported operators can be found in [43]. Arbitrary operations are possible using low-level process functions, which have access to stream records, can store fault-tolerant state, and can set timers in processing time and event time. This allows to implement transformations which are hard to implement using other operations. Streams can also be enriched with external data using asynchronous functions to, for example, make database requests, query external APIs, run machine learning inference or execute other operations that may take long. Asynchronous functions handle timeouts and stream ordering while keeping latency low.

Windowing is a common operation to divide the unbounded stream into bounded datasets. Flink's windowing is inspired by the Dataflow model. Records are assigned to windows

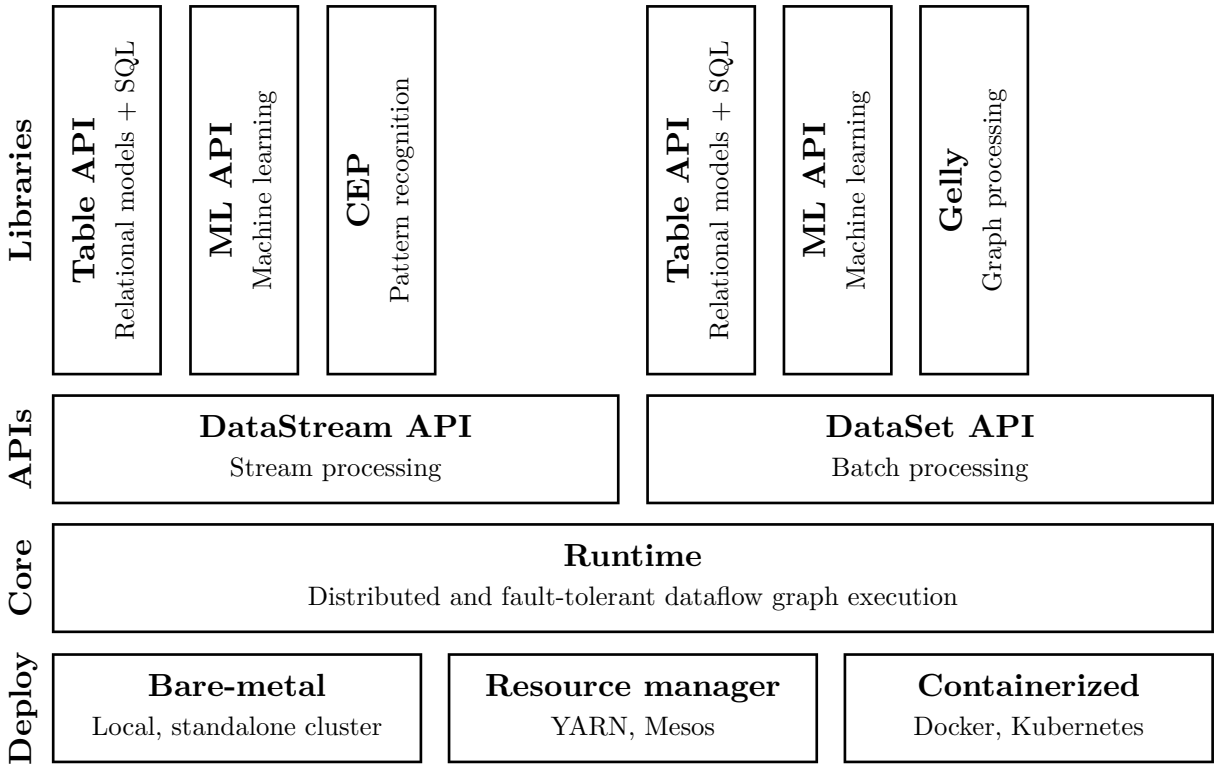


Figure 3.2: Components of Flink stack: multiple layers of abstraction run on top of diverse deployment environments

using a window assigner. Out of the box, tumbling, sliding and session windows are supported, but implementing custom window assigners is possible. Every window is aligned with epoch by default, which means that, for example, hourly windows start on the hour instead of relative to the job start time. Records emitted from a window carry the window end timestamp, which allows windows to be chained. For example, chaining a 10 min and a 30 min introduces only 30 min instead of 40 min of latency. window Flink provides two types of transformations on the records of a window. Reducing and folding transformations like summing or averaging aggregate records eagerly, which reduces state size, especially for large windows. Arbitrary transformations are supported using process window functions, but require Flink to buffer all records for a window. Since a copy of each record is created for every window it belongs to, this can quickly lead to large states, for example when using a sliding window of 1 day with an evaluation period of 1 second. Flink supports completeness triggers, but only count-based triggers are supported for repeated updates. Custom triggers are supported to implement custom early/on-time/late triggers for balancing latency and correctness. Flink supports the delta and value output modes, but retractions must be implemented manually. Window evictors provide a way to remove certain records from the window before and after evaluation, which can for example be used for deduplication. However, the use of evictors prevents eager aggregation.

Processing data in the right time domain is crucial for correct results. Flink supports ordering of records by event time and processing time. The event timestamp can be extracted from records using a timestamp assigner, which also allows to use the ingestion timestamp (the time at which a record enters the Flink cluster) as event timestamp. Watermarks can be generated based on the event timestamps and emitted periodically (every 200 ms by default) or punctuated (when special records are observed). Flink ships with two periodic watermark generators. Watermarking for monotonously increasing timestamps assumes that records arrive in order with increasing timestamps. Watermarking with bounded out-of-orderness makes the watermark lag a fixed amount of event time behind the latest event timestamp. It is not recommended to emit watermarks for every record, since watermarks usually trigger downstream computations and would therefore degrade performance. Watermarks originate at sources in the dataflow graph, and are then propagated downstream. Simple operations like `map` simply forward watermark, while more complex operators like windows only forward watermarks after emitting the results of the window computation. Therefore, windowing adds to the latency of the watermark like it does to regular records. Join operators, which have multiple inputs, use the minimum of all incoming watermarks. To improve performance, event-time ordering is not guaranteed on a global level but only in operators that actually require the correct ordering.

Architecture and Execution Model

A Flink cluster comprises a client, a job manager and one or more task managers, as shown in figure 3.3. In most production cases, a separate *per-job cluster* for each job is recommended for resource isolation and limited impact of failures. However, shared *session clusters* are also possible for better resource utilization. The client creates a dataflow graph from the job specification and submits it to the job manager after optimization. The job manager coordinates the distributed execution of the dataflow on the task managers. This includes tracking the state and progress of each operator and intermediate stream between operators, but also coordinating checkpoints and recovery. While there is only one active job manager at any given time, failover is supported by promoting a standby job manager using a Zookeeper-based election. This eliminates the job manager as single point of failure for high availability. Task managers are JVM processed that execute operators and report their status to the job manager. Every task manager offers a certain number of task slots (usually equal to the number of CPU cores) which determines how many tasks can run in parallel on the task manager. Equal portions of the task manager's memory are allocated for each task slot. Inter-task manager communication to exchange intermediate streams is handled through direct network connections, which contain mechanisms for flow control to propagate backpressure from operators to sources. The network stack is detailed in [44] with considerations for trading off latency and throughput. Any object

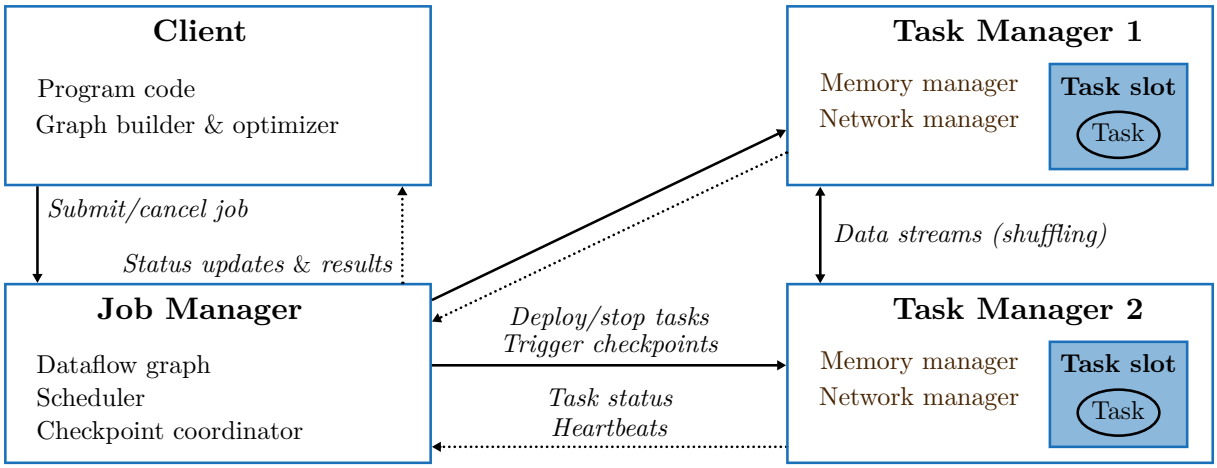


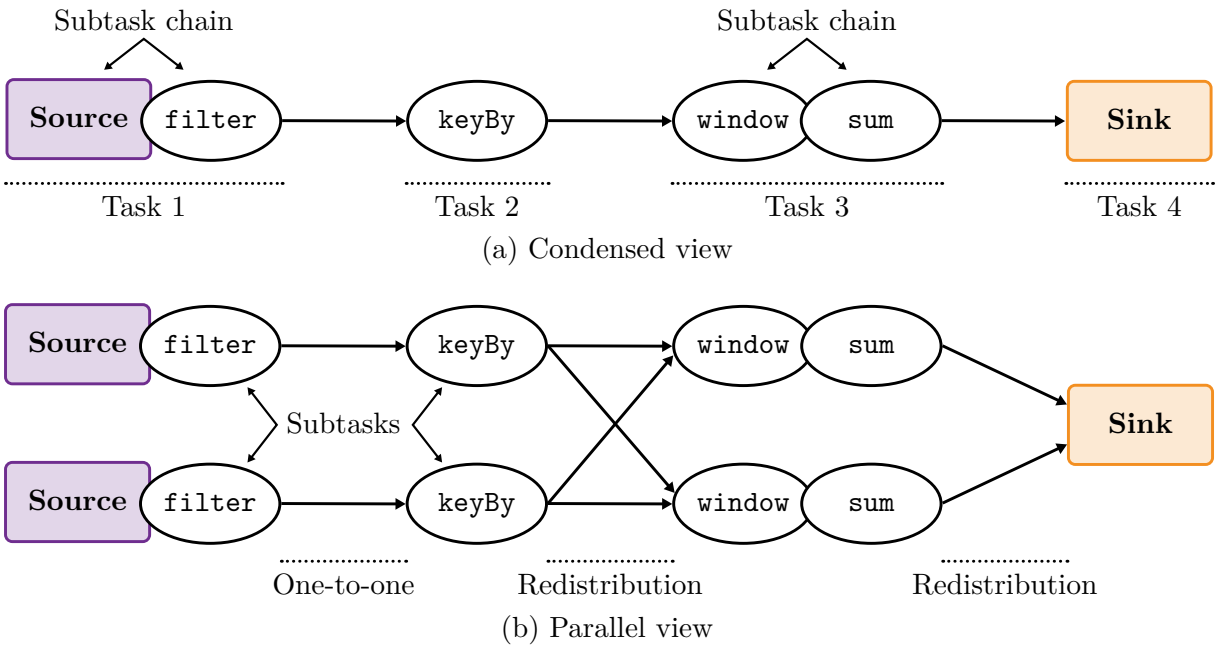
Figure 3.3: Cluster architecture of Flink: jobs running on task managers are coordinated by a job manager

needs to be serialized before transfer. Flink uses type inference to find the appropriate serializer/deserializer for a specific object, and provides optimized implementations for different data types. Custom serializers can be provided in for unsupported types.

While the dataflow graph shows the logical structure of the job, it is parallelized for physical execution. This is shown in figure 3.4. Operators are split into one or more parallel instances called *subtasks*, and streams are split into one or more partitions. The degree of parallelism can be defined for each operator. Parallelization results in two types of inter-operator stream distribution patterns. Redistribution is necessary in case two subsequent operators do not have the same number of subtask, or when streams need to be shuffled after a `keyBy` operation. *Network shuffling* denotes the routing of records to a different task slot (possibly on a different node) since records with the same key need to be processed by the same subtask. Subsequent operators with the same number of subtasks and without shuffling result in one-to-one streams and the stream partitioning can be preserved. Some subsequent operators like two `map` transformations can also be chained to run in the same thread, which reduces the overhead of inter-thread buffering to improve throughput and latency. A *task* comprises either a subtask chain or an individual unchained subtask. Multiple successive tasks can occupy the same task slot through *slot sharing*, which allows to run one parallel instance of the whole pipeline in one task slot. This yields better resource utilization and limits the number of required task slots to the maximum degree of parallism of a job.

State and Checkpointing

Many operations like windowing and pattern recognition require state. Instead of storing the state in memory, state can be stored in the cluster so it can participate in checkpointing



(a) Condensed view

(b) Parallel view

Figure 3.4: Logical and physical view of parallel dataflow graph: subtasks are parallel operator instances with one-to-one or redistribution streams on between. Each parallel pipeline of tasks can occupy the same task slot, with each subtask (chain) running in a separate thread.

for fault tolerance. Flink provides interfaces to declare value, list and map states with optional time-to-live. State in operators can either be scoped to a key or a whole subtask. Broadcast state is available to all subtasks and is often used to update rules for alerting or parameters for machine learning. State can also be queried from the outside. Flink supports three state backends, which determine where state is stored. The memory state backend stores state in the job manager's memory and is only recommended during development. The file system and RocksDB backends write state to disk. Since this state is stored locally and maintained in memory as well if possible, performance is usually not impacted much. RocksDB allows keeping very large state and is recommended for production.

State can be regularly checkpointed for fault tolerance by taking a state snapshot (possibly incremental in case of RocksDB) and storing it to a durable medium like HDFS or S3, as shown in figure 3.5. Flink uses barriers inserted into the regular stream to notify operators of checkpointing and dividing the stream into a part which will be included in the current snapshot and a part which will be snapshotted later. An efficient algorithm is used to align barriers in case of `join` operators with multiple input streams. Snapshots also include input offsets of sources like Kafka and scheduled timers to guarantee end-to-end exactly-once processing even after failure recovery. The mechanism is detailed in [22]. *Savepoints* are a special kind of manually triggered checkpoint that can be used to stop

FIGURE OF CHECKPOINTING

Figure 3.5: State checkpointing mechanism of Flink: barriers trigger snapshots which are moved to a durable storage

and resume Flink jobs to perform code updates or maintenance, rescale by changing the parallelism, or fork the job for A/B testing or red/blue deployment.

Flink provides a comprehensive set of features for correct, fault-tolerant, low-latency and scalable stream and batch processing. In combination with a distributed and durable log transport platform like Kafka, exactly-once processing and backpressure handling are guaranteed even at large scales. Together with the powerful DataStream and DataSet APIs, this makes Flink and Kafka suitable for many use cases.

4 Solution Design

In this chapter, we regard an example solution for event stream analytics. While this is only one specific incarnation and many different designs are possible, it highlights important aspects that every solution needs to consider. The solution is independent of the use case presented in the next chapter and therefore applicable to many scenarios. In any case, [3] is a good foundation for designing distributed data processing solutions.

4.1 Requirements

The goal of the solution is to ingest data from an external stream source, process the stream to produce insights, and visualize those insights in real time. Arriving at those insights requires stateful processing in the correct order that goes beyond record-by-record transformations. Additionally, we want to see results in real time regardless of the data volume. This leads us to the four requirements that have guided us through the past two chapters:

- Correctness: results should be guaranteed to be correct through exactly-once event-time-order processing even if events arrive out of order and faults occur
- Fault tolerance: the solution should guarantee consistent results and preserve state during faults without heavy recomputations
- Low latency: results based on an ingested event should become available for visualization in (near) real time (a latency of a few seconds is acceptable/inevitable, depending on the job)
- Scalability: the solution should be able handle large volumes of data without performance degradation

Correctness and fault tolerance go hand in hand, but are in tension with low latency and scalability. Providing real-time results of computations at high volume requires a distributed stream processing platform that spans across nodes, but keeping consistent distributed state without impacting performance severely is challenging. The challenge

FIGURE OF ARCHITECTURE

Figure 4.1: Solution architecture: streams flow from ingestion through processing to visualization

becomes easier when using fast solid state drives, large memory pools and nodes with many CPU cores. We will disregard the cost factor in this design since we approach the problem from a technical perspective. However, the combination of Kafka and Flink for stream transport and stream processing can fulfill all of the four requirements even on inexpensive commodity hardware through efficient designs and implementations, and allows us to build jobs on top while taking care of the execution.

4.2 Architecture

The components and stream flow of our architecture are shown in figure 4.1. All components run on separate clusters of nodes but shared nodes are possible as well. Kafka sits at the core to provide stream transport and storage while decoupling producers and consumers. Having Kafka act as a durable buffer enables replay for fault tolerance, but also prevents consumers from being overwhelmed by faster producers and allows producers not to be held up by failing consumers. Additionally, consumers can flexibly be added or start reading at different offsets without affecting the producer. While every stream only has a single consumer in our architecture, more consumers could be added, for example for monitoring, data warehousing or event-driven applications. This kind of centralized platform can democratize data and allow seamless integration and streamlined dataflow between different teams in an organization. The Zookeeper cluster required for Kafka broker coordination and Flink high availability runs on the same nodes as the Kafka cluster, but can easily be extracted into a separate cluster for isolation in case of faults.

We define all infrastructure (nodes, network, storage) as code using Terraform[1], which provisions the resources on AWS. We then import the data into Ansible[2] using a custom dynamic inventory script to deploy the platforms and components. This fully automated deployment allows us to quickly set up and tear down everything and treat our infrastructure as immutable. This means that we do not change existing infrastructure but always make a clean and consistent deployment. Any data is stored externally so no data is lost in this process. However, we have not done extensive online updates yet, so the deployment approach might need to be adjusted for large-scale production situations.

[1] `https://www.terraform.io/`
[2] `https://www.ansible.com/`

Streams are not initially created within our solution but ingested from an external source. This source can be anything from database change captures over message queues to arbitrary event sources. The ingestion component is responsible for capturing these external events, filter and transform them, and write them to the appropriate Kafka topic. Therefore it basically acts as connector to the outside world. Analytics jobs running in the Flink cluster of the processing component consume these input streams and write their analytics results back into Kafka. The visualization component consumes the results streams in the backend and forwards them to the frontend, where they are displayed in real time in a format appropriate for the use case. This could, for example, be a dashboard for monitoring or a map for geospatial data. Having the visualization consume a stream is different from most traditional user interfaces which query a database or web service. We are deliberately using an end-to-end streaming design to embrace the continuous nature of our data.

4.2.1 Transport and Storage

With the current set of components, there are two types of Kafka topics. Topics prefixed with `input` originate in the ingestion component, and topics starting with `analytics` are produced by the processing component. A clear and consistent naming scheme helps when adding more topics and producers but can be chosen freely. The number of partitions needs to be chosen based on the expected data volume, with random partitioning for even distribution if possible. This does not retain the order of incoming events but we rely on Flink for processing in the correct order. We use a replication factor of 3 for each topic since this provides a good balance between fault tolerance and performance. The number of total partitions determines the required cluster size. We use a short retention period which is higher than the Flink checkpointing period since we only use retention for fault tolerance, not replays from a much earlier point in time. Note that the disk size requirements increase proportionally to the data volume and retention period. We configure our cluster to be accessible from the outside from certain address ranges. While this is not recommended for production setups, it allows developers to run certain components on their local computer during development or peek into topics for debugging.

We have one topic per event type, with all records having a common schema. We use Protocol Buffers (protobuf) for schema specification since it has support for many languages, flexible records and schema evolution. There are two serialization modes. The binary mode creates compact messages, while the JSON mode creates human-readable messages suitable for debugging. Switching between these modes is simple and allows, for example, to use JSON during development but binary for production. Every record, whether input or analytics result, has the schema shown in listing 4.1. We refer to the common schema

```
message Event {
   google.protobuf.Timestamp event_timestamp = 1;
   google.protobuf.Timestamp ingestion_timestamp = 2;
   google.protobuf.Any details = 3;
}
```

Listing 4.1: Common Protocol Buffers schema for all events

defined in `Event` as base event. The `event_timestamp` may contain the actual time the event occurred if the external source provides such a timestamp, or the end of the event time window for analytics results. The `ingestion_timestamp` is the local system time in the ingestion and processing components before handing over the record to Kafka. The `details` are an arbitrary nested protobuf message which is specific to the event type. The `Any` attribute type stores the nested type name and actual object to be able to deserialize the event with type safety. The structure of the type can be looked up in a type registry that is compiled into the components, but a central schema registry such as the one bundled with Confluent becomes indispensable when the number of producers and consumers grows.

4.2.2 Ingestion

All input streams originate at an external source and therefore need to be integrated into the solution first. This is the task of the ingestion component. Incoming events are first filtered so only relevant ones are ingested into the system. Then they are converted to the specific protobuf schema of the event type by selecting attributes and casting them to the right data types. Together with an extracted event timestamp, they are wrapped in the common event schema and produced to the appropriate Kafka topic prefixed with `input`. The current implementation of the ingestion component only supports a single node and no fault tolerance, but it can be extended in the future to run in a distributed fashion by dividing the input streams between multiple nodes. For example, each node could be assigned a mutually exclusive subset of source partitions for ingestion. Note that multiple nodes might require distributed coordination in case shared state is required, in which case the implementation of the partitioning is not as straigtforward anymore. Another alternative is to use an existing tool like Kafka Connect instead of a custom implementation if only simple transformations need to be done.

Our implementation provides flexibility through two base abstractions that work together as shown in listing 4.2[3]. A `Connector` connects to the external source to receive the raw event stream. Every raw event is processed by the `Processor`, which determines if an

[3]We use Python-inspired pseudocode for all code listings in this thesis for brevity.

```python
class Connector:
    def __init__(self, config, processor):
        # Set up connector

    def start(self):
        # Start the ingestion of events
        # Example:
        for raw_event in external_source:
            if topic, event_timestamp, details
                  := processor.process(raw_event):
                self._ingest(topic, build_event(event_timestamp, details)

    @final
    def _ingest(self, topic, event):
        produce_to_kafka(topic, serialize(event))


class Processor:
    def __init__(self, config):
        # Set up processor

    def process(self, raw_event) -> Optional[topic, event_timestamp, details]:
        # Extract timestamp and event details
        # Return None if event should be filtered out
```

Listing 4.2: Interaction between connectors and processors

event should be filtered out and therefore not be ingested, or extracts the timestamp and event details with the correct protobuf schema. These are then wrapped in a base event where the ingestion timestamp is added, serialized to the binary or JSON protobuf wire format, and ingested into the appropriate Kafka topic. Multiple connector can be run in parallel.

The `Connector` and `Processor` abstractions based on a common interface allow easy extension of the ingestion component for a wide range of different source types. We provide connectors for two types. The first connector subscribes to topics on an MQTT[4] broker. The topics to subscribe to are determined by the processor. The second connector replays a recording of MQTT events captured with a custom tool. The recording is streamed from S3 while preserving the original timing between events. The processor can also modify the event, for example to adjust the event timestamp to the current time. Replaying previous recordings is useful for repeatability of evaluations. Decoupling the event source and pre-processing using the two abstractions allows us to use the same processor for multiple connectors.

[4]MQTT is a publish–subscribe protocol commonly found in IoT applications. A central broker routes events from clients who publish them to clients which have subscribed certain topics of events. Topics are often structured hierarchically, but the topic format can be chosen at will.

4.2.3 Processing

The actual event stream analytics happen in the processing component, which is a Flink session cluster on which all jobs run. All jobs consume the input topics for the events they require, and produce their results to a job-specific topic prefixed with `analytics`. This allows downstream consumers to receive only the analytics they are interested in. The cluster consists of 3 job managers for high availability, but the number of task managers is determined by the number of jobs, their parallelism (the number of task slots required can be equal to the highest parallelism with slot sharing), and how many task slots each task manager offers (usually equal to the number of CPU cores). For example, 5 jobs where the operator with the higest parallelism has a parallelism of 20 requires 25 task manager nodes when each node has 4 cores and slot sharing is enabled. The parallelism needs to be adjusted to the expected throughput and the number of Kafka partitions. If there is only a single partition, the whole topic is consumed from and redistributed by a single task, even if the configured parallelism is higher. Note that increasing the number of nodes increases the overhead of shuffling after `keyBy` operations. The capacity planning depends on many factors like state size and backends bit also chaining and slow sharing, therefore we cannot give recommendations for the actual cluster size. [45] shows rough calculations for the required network capacity.

All jobs are written in Java and extended a common base class. This base class handles the setup of checkpointing as well as event-time processing and exactly-once processing by default. However, at-least-once processing might be required for some ultra-low latency applications. The checkpointing frequency affects the required topic retention period and the recomputation effort in case of failures. However, if not many failures are expected, it can make sense to increase the delay between checkpoints. The base class also configures Kafka bootstrap servers and consumer groups, and provides methods for working with Kafka topics. Dataflow graph sources can be defined by the topic and expected event type, while the base class sets up the serialization and correct watermarking and event timestamp extraction from the base event. While we normally use the actual event timestamp, it would also be possible to assign the ingestion timestamp as event timestamp in this step to order events by the time they arrive at the ingestion component. Dataflow graph sinks can be defined by the job-specific topic, and another method handles the creation of the base event with the correct timestamps and details. Creating accurate heuristic watermarks is very hard. Therefore we use the bounded-out-of-orderness strategy based on the expected delay, which can for example be determined using offline analysis of the distribution of event-time skew. Once the distribution is known, a percentile watermark can be created with the threshold set to balance latency and completeness. If the watermark underestimates the event-time skew, many records are marked as late which increases computation effort when late triggers are enabled. Watermarks are emitted periodically every 200 ms, which

is a good tradeoff between latency and computation effort, since most watermarks trigger downstream computations like window evaluations. In general, watermark boundedness and allowedness need to be configured to balance correctness/completeness, latency and cost/resource requirements. We do not guarantee end-to-end exactly-once processing in our solution, since that would require an idempotent or transactional sink. While Kafka supports transactions, the Kafka client used in the visualization component does not, therefore results might be produced with at-least-once semantics. However, exactly-once can be easily enabled once the client adds transaction support. Then the checkpointing frequency needs to be increased since transactions are only committed once the checkpoint was completed successfully.

We use RocksDB as state backend, which is recommended for production environments and supports large state. However, state size should be kept reasonably small since it influences checkpointing time and resource requirements. This especially applies to operations which must retain many records like windows or pattern matching. These are some optimizations to minimize state size:

- Since a record is copied for every window it is assigned to, use longer evaluation periods when using large sliding windows. For example, an hourly sliding window evaluated every second would result in 3600 copies of every record. If the evaluation period is decreased to every 10 s, the number of copies is reduced by a factor of 10.
- If the window transformation supports eager aggregation (like summing or averaging), state size can be reduced significantly because records are collapsed into a single intermediate result instead of having to retain all records until evaluation. Therefore, use `AggregationFunction` and `FoldFunction` instead of `ProcessWindowFunction` if possible.
- While there is only one active tumbling window at any point when using processing time, multiple tumbling windows can be active simultaneously when defined in event time. With a high bounded-out-of-orderness and allowed lateness, windows and records need to be retained much longer. While changing the watermark is usually not sensible since it affects correctness, keeping the allowed lateness low allows records to be garbage-collected earlier.
- Only send relevant parts of the record to downstream tasks to reduce serialization and network overhead as well as state size. For example, if only two attributes of a protobuf event are needed, map the incoming event to a two-tuple early in the dataflow.

4.2.4 Visualization

The visualization component is the only one with which end users interact. This component consists of two parts. The backend consumes the analytics result topics and forwards forwards them to the frontend using Socket.IO (a library for client–server communication transparently using WebSockets or HTTP), where the results are displayed in real-time in a format appropriate for the analytics. There are two technical reasons for not connecting the frontend directly to Kafka. First, the frontend needs to be written in JavaScript, and there is currently no JavaScript library to connect to Kafka from the browser. Secondly, the protobuf type registry would need to be part of the frontend for deserialization. Therefore, we deserialize the analytics result protobuf records in the backend and send plain JSON strings to the frontend. While this does not maintain the order of the ordered topic partitions, the order usually does not matter for display. Currently, the frontend receives all analytics results but only displays those that the user has selected, so there is no communication from the frontend to the backend. Moving this responsibility of determining the data display to the frontend allows us to support multiple frontend users without extra effort.

While the backend is very simple and effectively only forwards events, the frontend has the challenge of invalidating events. When aggregating over keyed windows, windows are only created when there are actually events for that key. If a key creates windows for multiple subsequent windows, the frontend can simply display those results as updates to the previous results. However, once a key disappears (for example a user ends his session), no updated results are produced for that key. In these situations, the frontend needs to decide when to stop displaying the last results, since there is no explicit invalidation event (a so-called tombstone record, where the value for a key is null) for those results. The invalidation timeout/time-to-live of analytics results should be longer than the window period, since results might simply be a bit delayed (for example, due to network congestion, failover, or long Java garbage collection pauses). This leads us to using two policies for displaying analytics in the frontend:

- Only results with the same or updated event timestamp per key are accepted in the frontend, previous results are simply discarded. Multiple results per key with the same event timestamp (for example, produced when having multiple triggers for a window) might occur when late data are processed in the processing component.
- Expire window results after the window period plus 5 s, since we can be relatively certain that after this point data are not recent anymore and no more updates are going to arrive. For non-window results (for example, pattern recognition matches) and other punctuated events (occurring at a single point in time as opposed to being

the result of correlating multiple events over time), the time-to-live should be set to allow the user to observe the result while preventing the display of outdated results.

5 Analytics Usecase

We implemented the event stream analytics solution presented in the previous chapter for an example use case. We wanted an interesting use case with diverse analytics possibilities that can be visualized well. Additionally, we wanted to use real data instead of mock data we generated ourselves, since real data are less predictable and can therefore better reveal real-world challenges of stream analytics. Smart and connected cities with public APIs provide an ideal source of streams since many different events occur in a system as complex as a city, and IoT facilitates digital access to these continuous streams in real time. Especially for public transportation many cities offer developers access to these data to integrate real-time information into, for example, route planning apps. Helsinki has one of the most mature public transportation API worldwide in terms of technology and documentation [46], therefore we will use its data as foundation for our use case.

5.1 HSL Public Transportation APIs

Helsinki Regional Transport Authority (HSL) is the transportation authority for Helsinki and the surrounding municipalities. It oversees the operation of buses, trams, metros and overvehicles as well as ticket sale and inspection, but relies on third-party contractors for vehicle operation since it does not own any vehicles itself. HSL offers multiple publicly available APIs [47]:

- High-frequency positioning (MQTT): vehicle positions every second, as well as events like arrivals at and departures from stops
- Routing (GraphQL): itinerary planning and information about stops and timetables
- Geocoding (HTTP): conversion between coordinates and places
- Map (HTTP): map images with points of interests
- Sales (HTTP): ticket sales and pricing information

We focus on the High-Frequency Positioning (HFP) API since it provides us with a stream of real-time events, but we use other APIs to enrich events with more information.

The HFP API is accessible through a single MQTT broker [48]. The topic structure allows subscription to specific event types, vehicle types and routes, but multiple/all can be selected by using wildcards. Following event types are available, among others:

- Vehicle position (every second): geographical coordinates
- Vehicle has arrived at stop or departed from stop
- Vehicle's doors are being opened or closed
- Traffic light priority requests and responses

The JSON payload differs slightly between events, but usually contains an event timestamp, geographical coordinates (latitude and longitude), heading, speed, acceleration, as well as route and vehicle information. Some examples are shown in appendix A Because vehicles are operated by different contractors, vehicles can only be uniquely identified by the combination of operator and vehicle number.

The volume of events over the course of a day is shown in figure 5.1. The volume is very low during the night and starts to increase after 05:00 with a first peak at 08:05 (1081 events per second). The volume remains steady until the second peak at 16:08 (1123 events per second), after which the volume slowly abates during the evening. This amounts to 58.4 million events per day. The delay between events occuring in the real world (event timestamp) and the timestamp of arrival at the ingestion component over the course of a minute is shown in figure 5.2. Note that this is only partially the event-time skew/processing-time lag, since these metrics are defined using the timestamp of arrival at the *processing* component. Most of the events actually have a negative delay, which is implausible since it would mean that events are ingested before they happen. This is probably due to clock synchronization issues between the vehicles and our system (we synchronize the system time of our EC2 instances using NTP as recommended by AWS). However, this does not affect correctnes or latency, since all processing is done in event time. Of more significance is the high tail delay, with some events only arriving after more than 30 s. This confirms the need for explicit event-time processing, where we can explicitly treat these events as late. With processing-time ordering, they would end up in the wrong window.

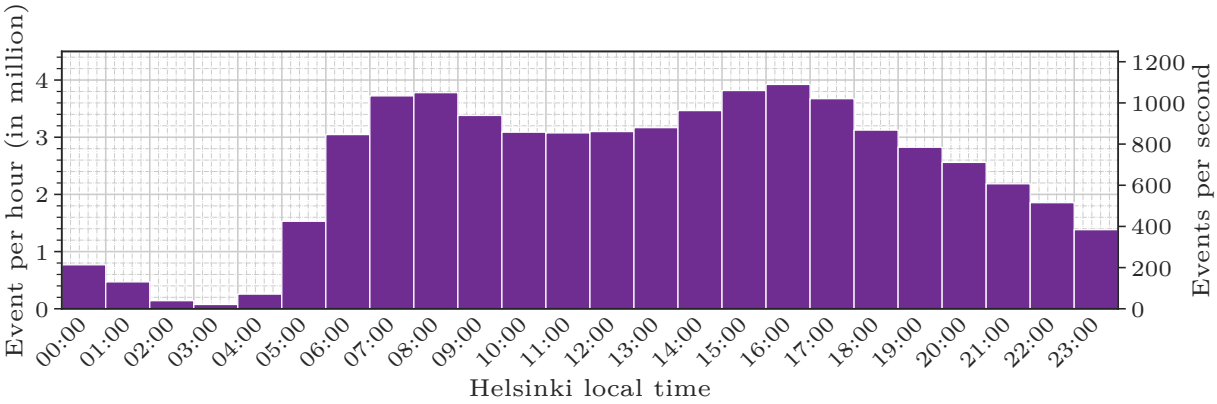


Figure 5.1: Daily event volume of the HSL HFP API: the volume is the highest during rush hour

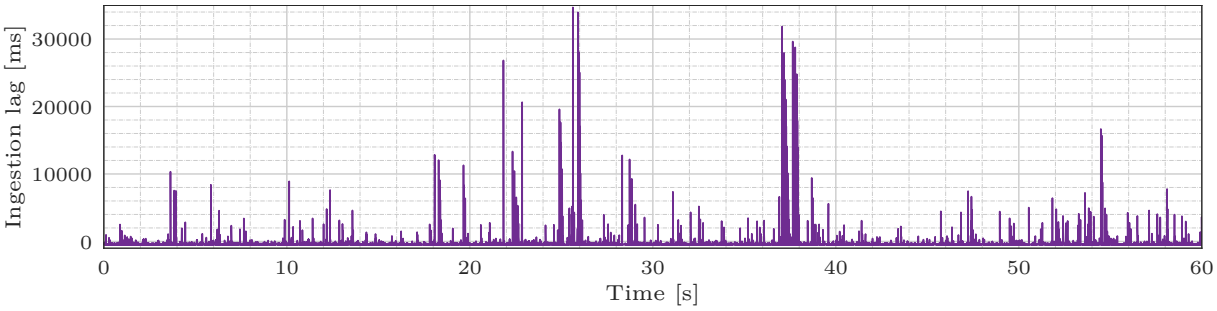


Figure 5.2: Delay between the occurrence of events from the HSL HFP API and their arrival at the ingestion component over time (min: −542 ms, median: −465 ms, 90th percentile: −120 ms, 99th percentile: 9279 ms, max: 34 684 ms)

5.2 Analytics

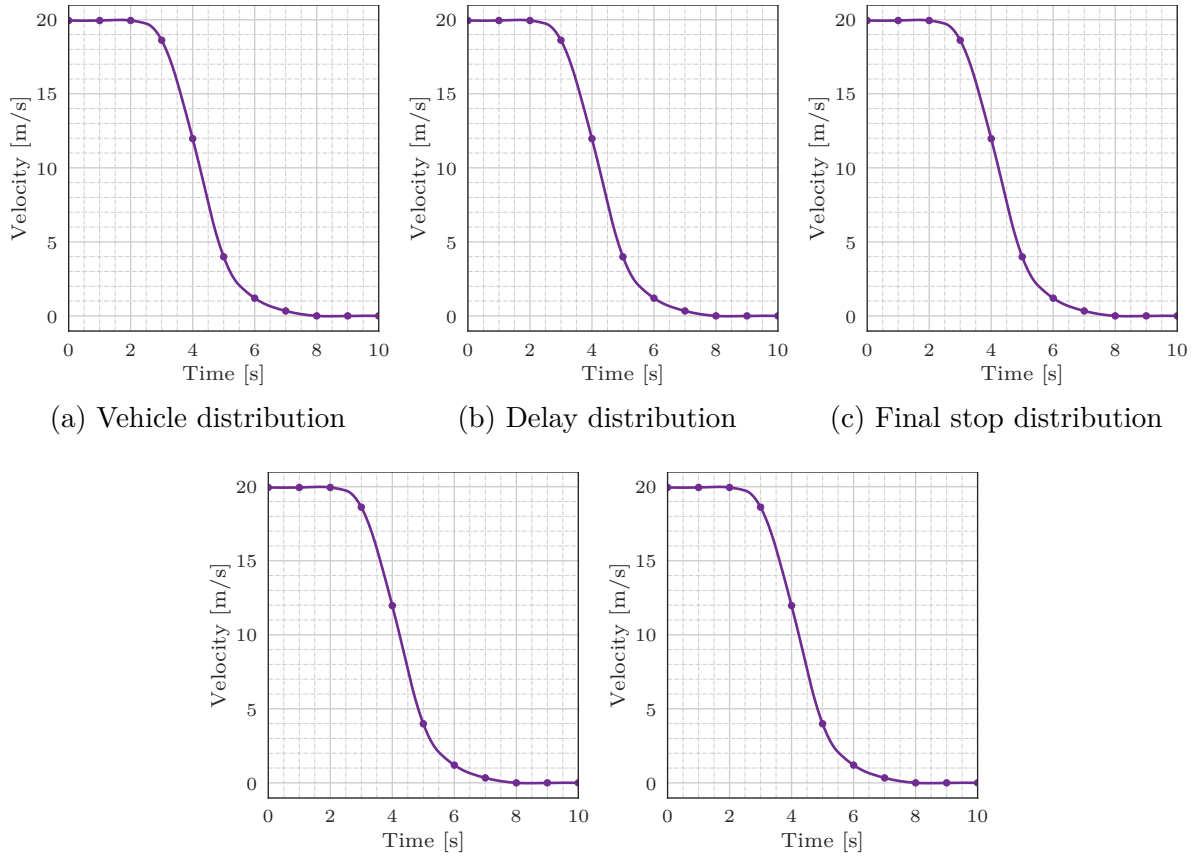
We decided on five analytics based on the HSL HFP data, visualized in figure 5.3. They cover different aspects of stream analytics and therefore allow us to evaluate multiple scenarios. All analytics jobs produce a stream that is produced to a separate Kafka topic. Most analytics divide the city of Helsinki into a grid of *geocells* of equal size and calculate aggregates for each cell, resulting in a geographical distribution for the analytics when observing the results for all geocells.

Vehicle distribution Calculates the number of vehicles per geocell in the last 30 s, evaluated every 5 s. First, key the vehicle position stream by vehicle and apply a sliding window of 30 s evaluated every 5 s with a window evictor for duplicate events. This deduplication is necessary because each vehicle sends their position every second but we are only interested in the last position within the window, otherwise we would count each vehicle 30 times. Then, key the deduplicated stream by geocell and count the number of vehicles per geocell.

Delay distribution Calculates delay statistics (median, 90th and 99th percentile) per geocell in the last 5 min, evaluated every 5 s. First, calculate the difference between the scheduled and actual arrival on the arrival event stream in minutes (scheduled arrivals are only provided with minute resolution). Then, apply a sliding window of 5 min evaluated every 5 s and calculate the statistics for all arrivals within the window. No deduplication is necessary.

Final stop distribution Calculates the number of vehicles on a route with a final stop within a certain geocell per geocell in the last 5 min, evaluated every 5 s. First, key the vehicle position stream by vehicle and apply a sliding window of 5 min evaluated every 5 s with vehicle deduplication. Then, query the final stop of the route for each vehicle from the routing API using Flink asynchronous functions, since the final stop is not part of the event itself. All query responses are cached which drastically improves throughput and latency. The order of events is not maintained but can still be processed in event time by subsequent operators. Finally, key the stream of final stops by geocell and count the number of final stops per geocell.

Flow direction Find the neighbor cell that the majority of vehicles move to per geocell in the last 5 min, evaluated every 5 s. First, key the vehicle position stream by vehicle and apply a process function that only emits events when a vehicle changes the geocell. Then, key by origin cell and apply a sliding window of 5 min evaluated every 5 s and find the neighbor cell which the most vehicles moved to. The result is a directed edge between two cells and the count of vehicles that traversed the edge within the window.



(a) Vehicle distribution (b) Delay distribution (c) Final stop distribution

(d) Flow direction (e) Emergency stop detection

Figure 5.3: Analytics for public transportation

Emergency stop detection (stream) Detect when vehicles brake quickly from cruising speed to a stop within 10 s. First, key the vehicle position stream by vehicle and use Flink's pattern recognition library to detect stops. The pattern requires vehicle to cruise at more than 10 m/s, then decelerate and finally slow down to less than 1 m/s within 10 s. For every detected stop, calculate the difference between the cruise speed and the speed after breaking as well as the maximum deceleration.

Emergency stop detection (table) Does the same as the analytics above, but appends all events to a table using Flink's table API. The pattern is specified in SQL using the standardized `MATCH_RECOGNIZE` clause, which uses Flink's streaming pattern recognition library under the hood. Finally, convert the detected stops and their speed difference and maximum deceleration back to a stream.

All analytics jobs use watermark-based completeness triggers and value output modes. Jobs that do not use deduplication have an allowed lateness equal to the window evaluation period (i.e. until the next window is complete), since later data would not be displayed in the visualization component according to the second policy. Jobs with deduplication do not have late triggers since they do not work well with eviction and multiple subsequent windows.

5.2.1 Geocells

To support analytics that calculate aggregations (like vehicle distributions) over discrete geographical regions, a discrete global grid system is required which can partition the continuous space of geographical coordinates into geocells. While this grid can be defined based on data like postal codes, regular grid systems with even distribution are desirable for most applications [49, p. 121]. There are many approaches to reduce distortion introduced by applying a regular two-dimensional grid onto the three-dimensional sphere of the earth. We use Uber's H3[1] library, which first projects the earth's surface onto an icosahedron and then overlays a grid of 122 hexagonal base cells (pentagons need to be used at vertices, but these are deliberately located in the water to reduce error). Each base cell has an average edge length of 1108 km, but can be recursively sub-divided into seven smaller cells. At the finest resolution, the average edge length is only 50 cm. The choice of hexagons over triangles and squares (the only other two regular polygons that tile perfectly) simplifies motion analysis, since the distance between the center of a hexagon and the center of is hexagons is the same on all sides, compared to two different distances for squares and three for triangles. This enables easy approximation of distances and radii through the number of cells between two points. For a more detailed description of H3, refer to [50].

H3 has bindings for all languages that we use in our solution (Java, Python and JavaScript). This enables us to use geocells in analytics jobs and then draw the according cells on a map for visualization. At the resolution that we use, the globe is divided into 4,800,000,000 geocells with an average edge length of 174 m and an average outer diameter of 348 m.

5.3 Data Flow Example

To better understand the data flow through and interaction of components of our solution, we now look at an end-to-end example and trace the route of a vehicle position event from the HFP API until its visualization as part of the vehicle distribution job. Use the architecture from figure 4.1 as reference.

The ingestion component is the component interfacing with the HSL MQTT broker. When it is first started, the component creates an MQTT connector which subscribes to the `/hfp/v2/journey/ongoing/vp/#` topic to receive vehicle positions (the `#` is a wildcard supported by the HSL broker since the actual topic is much more elaborate). An example of a vehicle position event as it is produced by the MQTT API is shown in listing A.1. This event is processed by the respective processor which extracts the event timestamp and all other relevant information to build a protobuf message. This message is then

[1] `https://h3geo.org/`

wrapped in the common event schema and serialized. While we normally use the much more compact binary format (150 B on average), the JSON variant shown in listing A.2 (450 B on average using UTF-8 without indentation) is very useful for debugging and can be used interchangeably (compare with the schema definition from listing 4.1). Note how the message contains the nested type, which enables type-safe deserialization, and the ingestion timestamp is added. The event leaves the ingestion component when it is written to the Kafka topic `input.vehicle-position` for all vehicle position events. Examples for all three input event types that we use (vehicle position, arrival, departure) can be found in appendix A.

The processing component runs all stream analytics jobs. The Flink definition for the vehicle distribution job is shown in listing 5.1 (most referenced methods and classes are custom but not shown for clarity). The vehicle position Kafka topic as well as the expected class for protobuf messages is declared in line 1. Then the stream is keyed by vehicle, which partitions the stream and results in shuffling to bring the events to the appropriate task slot (in case of a parallelism larger than 1). This might add network latency if the task slot is located in a task manager on another node. The keyed stream is then windowed into sliding 30 s windows which are evaluated every 5 s. This means that every vehicle position event will be part of 6 window instances. The window is only triggered once on-time when the watermark for the window event passes. We use bounded-out-of-orderness watermarking of 1 s. Since the vehicle position event happens at 08:24:36, the window is evaluated when the first event with an event timestamp of 08:24:37 is observed (all timestamps are in UTC, which is 3 hours ahead of Helsinki time). We apply deduplication at evaluation to only retain the most recent vehicle position event for each vehicle within the window. The transformation of the window is an identity function that emits all events as-is (apart from adjusting the event timestamp to the end of the window). This is required so the stream can be re-shuffled in the next step, where we key the stream by geocell.
This results in partitions where all events within a partition occurred in the same geocell. Since we already applied a sliding window which emits its contents every 5 s with adjusted event timestamps, we only need a tumbling window the length of the slide period. Note we still calculate the distribution over the last 30 s. The second window is aggregated eagerly using an `AggregationFunction`. Finally, the result shown in listing 5.2 is emitted to the Kafka topic `analytics.vehicle-distribution`. The result shows the number of vehicles in the same geocell as our example vehicle for the window from 08:24:20 to 08:24:50. When executing this job on a Flink cluster, the framework automatically handles the distributed and fault-tolerant execution with event-time ordering.

The visualization component shows the vehicle distribution to the user. The backend consumes the vehicle distribution results Kafka topic and converts all protobuf messages

```
source("input.vehicle-position", VehiclePosition.class)
   // First, key and deduplicate by vehicle
   // This selects only the most recent position
   .keyBy(new UniqueVehicleIdKeySelector())
      .window(SlidingEventTimeWindows.of(Time.seconds(30), Time.seconds(5)))
      .evictor(new MostRecentDeduplicationEvictor())
      .process(new IdentityProcessFunction<>())
   // Then, key and aggregate by geocell to count vehicles
   .keyBy(GeocellKeySelector.ofVehiclePosition())
      .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
      .aggregate(new CountVehiclesPerGeocellFunction<>(),
                 new CreateOutputMessageFunction())
      .addSink(sink("analytics.vehicle-distribution"));
```

Listing 5.1: Job definition for vehicle distribution analytics

```
{
   "event_timestamp": "2020-07-27T08:24:50Z",
   "ingestion_timestamp": "2020-07-27T08:24:51.852916Z",
   "details": {
      "@type": "type.googleapis.com/com.dxc.ptinsight.analytics.VehicleCount",
      "geocell": "617769472917241855",
      "count": 14
   }
}
```

Listing 5.2: Vehicle distribution result as protobuf message

to a JSON string and forwards them to the frontend using Socket.IO. The conversion eliminates the need for integrating the protobuf definitions into the frontend. When the user loads the frontend website, it establishes a connection to the Socket.IO server using HTTP or WebSockets. Once the analytics results arrive, the frontend uses H3 to calculate the hexagon outline of the hexagon and draws it on the map as shown in figure 5.4. Because each hexagon is shaded according to the vehicle count, looking at all geocells gives an impression of the vehicle distribution in real time. All results are invalidated after 10 s and the associated hexagon is removed from the map if no updates for that geocell arrived within that time. This can be the case for regions that are not very busy, since results for a geocell are only produced if there are vehicles within it. The total time is takes from the position measurement of a vehicle until a result based on that measurement is shown in the visualization component can be as low as 3 s.

FIGURE OF VISUALIZATION MAP

Figure 5.4: Visualization on map with geocells: each geocell is shaded according to its vehicle count in real-time

6 Evaluation

Finally, we want to evaluate our solution with the public transportation use case. Specifically, we want to observe the relationship of the latency and scalability aspects. We assume that results are correct and consistent even in case of failures according to Flink's guarantees (apart from at-least-once results due to the lack of transaction support in the visualization Kafka client). Latency effectively describes the real-time capabilities of our solution at a certain data volume. We define two types of end-to-end latencies:

- Processing latency: the delay between an event being observed by the ingestion component and results based on that event arriving at the visualization component, can be divided into functional latency (e.g., due to buffering for windowing or joining, especially when using completeness triggers with watermarks) and non-functional latency (e.g., network shuffling and operator computations)
- Event latency: the delay between an event occurring and results based on that event arriving at the visualization component, consists of the processing latency plus the delay between the event's occurrence and ingestion

Note that the accuracy of the event latency is subject to time synchronization errors since we do not control the timestamping at the event source (i.e. the vehicles within the Helsinki public transportation system). This is not as much of an issue for the processing latency since we can synchronize the system clocks of all solution components using the same process. We will only regard the end-to-end processing latency, which comprises a major part of the event latency, and will therefore refer to it simply as *end-to-end latency*. The other part was already described in 5.2 but is not under our control.

In this chapter, we only evaluate the performance of our public transportation analytics solution. For a more general performance evaluation and comparison of Flink with other stream processing platforms, refer to [51], [38].

FIGURE OF LATENCY TRACKING

Figure 6.1: In-band latency tracking system: the latency tracker tracks and timestamps latency markers

6.1 Methodology

To achieve the goal of analyzing the relationship between latency and data volume, we need to add two extra components to our solution. First, we need a means of measuring the processing and event latencies. Secondly, we need a way to increase volume without interfering with or distorting our results. Therefore, all data must be plausible and the number of keys must increase with the volume scaling factor to affect the number of partitions.

6.1.1 Latency Tracking

The job of the latency tracker is to track events within the solution and measure timestamps at different points. Based on the differences of these timestamps, the latencies can be calculated. Flink comes with an in-built latency tracker. However, it only measures non-functional latency and therefore does not reflect the true latency. Additionally, it only works within the Flink cluster, but our the latency we want to observe is end-to-end to include the ingestion and visualization components as well.

Therefore, we use our custom in-band latency tracking system. The ingestion component periodically injects special records called *latency markers* into the regular stream which are received and evaluated by an additional latency tracker component. Latency markers are mock events with plausible data but special values to distinguish them from real events without needing to treat them separately in processing job. Therefore they experience the same latency as real events, including windowing, network shuffling and checkpointing latency, which can be significant if the state is large. The frequency of latency marker generation determines the upper bound for the resolution of the measurement.

The flow of latency markers is shown in figure 6.1, with all timestamps marked t_1–t_4. The black arrows show the regular event flow which we presented in section 5.3. The blue arrow is the end-to-end latency. It comprises several partial latencies between the individual components denoted by the green, red, and blue arrows. These can also be captured with our system, which allows us to better analyze where latencies come from. For example, when Flink slows down consumption due to backpressure, only the processing latency (the time between an event entering Flink and results based on it being produced by the job) increases. However, we cannot observe these partial latencies directly. While the ingestion

and processing components add the ingestion timestamps just before producing the record to Kafka, we cannot measure the arrival time (i.e. the arrow tip) directly, since this would require modifying the existing components. Instead, we approximate these arrows by measuring the arrival time at the latency tracker. This assumes that the transport latency from Kafka to all downstream components is identical, i.e. records take the same time from Kafka to the processing and latency tracker components.

Each latency marker appears twice in the system, once as input record and once as analytics result record. The key challenge is to make latency markers unique identifiable between these two instances even when they are subject to windowing, which collapses multiple events into a single one. This is required to be able to track them end-to-end. To solve this problem, we must retain a one-to-one relationship between latency marker records and window results. Every window we use is keyed by either geocell or vehicle. Therefore, every latency marker has a unique geocell and vehicle number, so the latency marker will always be the only record in a window. Unique geocells are generated by spiraling outwards from an origin geocell. The k-th batch of geocells is the ring with distance k from the origin cell. For example, the first batch of generated geocells are the direct neighbors of the origin cell, while the second batch are the neighbors of the first ring. H3 provides functions to easily generate these rings. We use "Point Nemo" as origin, the place on earth which is the farthest from any land. We can be relatively certain that no real events will originate from this point in the South Pacific. The vehicle number is generated randomly with a range outside of real numbers. Additionally, each latency marker uses a special value for the vehicle operator field. The event timestamp of latency markers is equal to the highest real event timestamp at any time. This ensures that latency markers experience the full functional latency introduced by watermarking.

To track latency markers generated every second in the ingestion component, the latency tracker component consumes all input and analytics topics. For every record, it checks if the record is a latency marker. For input records, checking if the event has the special operator is sufficient. Since the operator is not part of the analytics results records, checking if the geocell is within a certain distance of the generation origin cell is required for analytics result records. When a latency marker is first observed in an input topic, it is added to an internal cache with the geocell as key. The marker is retained for 5 min, which sets an upper bound for the latency that can be measured. The latency tracker then stores two timestamps for the newly observed latency marker. The first is the ingestion timestamp (t_1) added by the ingestion component. The second one is the arrival timestamp at the latency tracker (t_2). Once the matching analytics result arrives, identified by the geocell, two more timestamps are added. After the ingestion timestamp (t_3) added by the processing component and the arrival timestamp (t_4) are recorded, all four timestamps are written to a CSV file for later analysis.

The latency tracker consumes all Kafka topics and partitions in a single thread. This means that it does not scale well and may measure wrong results under too much load. While we tried to parallelize the latency tracker by runnning multiple processes which each consume only one partition, the already observed latency markers must be shared between processes. Python does not have high-performance shared dictionaries, therefore this approach failed. The load on the latency tracker should be monitored to see if it may distort results.

Note that this approach is highly specific for our use case and does not generalize to all situations. We generate special markers for each input topic and also need to generate pseudo-patterns to produce latency marker results for the emergency stop detection jobs. Also, we do not support latency tracking for the final stop distribution job (since latency markers cannot be used for querying the routing API) and the flow direction job (since latency markers are limited to a single geocell). While these limitations can be overcome, applying our approach to other use cases must guarantee that unique end-to-end tracking of latency markers is possible.

6.1.2 Volume Scaling

The second component required for our evaluation can increase the volume of data, or throughput, that is being ingested into our system and needs to be processed. However, it should not distort the evaluation results, i.e. the results should show the same behavior as if the scaled data volume was ingested naturally from the HFP API. For example, the number of keys influences the number of partitions and network shuffling overhead, therefore there must be 10 times as many keys if the volume is scaled by a factor of 10. Since this scaling while maintaining the distribution of data is use case specific, we are using a custome volume scaler.

The volume scaler is integrated in the ingestion component. Instead of using the real-time HFP stream in the MQTT connector, we use the replay connector which replays a recording of MQTT messages from the actual API. Using the predictability of a recording allows us to repeat and compare results. It also allows us to replay the recording multiple times simultaneously to increase the data volume. To scale the volume by a factor of n, the ingestion component starts n processes of the replay connector, as shown in figure 6.2. Each process consists of one thread. The first thread reads and decompresses the recording file stored in S3, and parses it to retrieve the original MQTT messages. It then schedules the processing of the message in the second thread with the same original timing. For example, if two messages were received 10 ms apart during recording, these messages will be processed with the same time offset. The recorded MQTT message is then processed

FIGURE OF VOLUME SCALING

Figure 6.2: Volume scaling system: each process replays the recording with an offset

using the same processor as regular messages to extract the event timestamp and details into the appropriate protobuf message.

Before producing the event to Kafka, the details need to be adjusted to prevent distorted results. Therefore, we change the unique identifier of each vehicle to increase the number of keys. We do not adjust the geocells, since these would stay the same if the higher volume occurred naturally. We also offset each individual replay to naturally diversify the ingested events. For example, the first replay starts from the very beginning of the recording, while the second replay starts 1000 records later, and the third replay 2000 records later. We then need to adjust the event timestamp so simultaneously ingested events have approximately the same event timestamp. We do so by setting the event timestamp of all replays to be within 0.1 s of the latest event timestamp in the first replay.

This latest timestamp is the only shared state between replay processes. However, this makes it hard to distribute the ingestion component across different nodes. Usually, every process runs on a single CPU core. Therefore, the possible scaling factor scales linearly with the number of cores but is also limited by it. If there are too many processes running, the desired volume is not achieved, records are not produced as quickly as they should be, and therefore the observed event time is slower than it actually was. Then, the ingestion component instead of the processing component becomes the bottleneck of the solution. This needs to be prevented during evaluation by ensuring that the ingestion rate is higher than the consumption rate of the Flink job [52, p. 73]. This can be achieved by monitoring the volume within the Kafka topics for plausibility.

6.1.3 Setup

Due to the limitations of the latency tracker, we only test the vehicle distribution, delay distribution and both emergency stop detection jobs. We run all jobs at once on the same cluster, and measure the latencies with different scaling factors between 1 and 64. The normal volume is about 700 events per second. While we only run the experiment once for each factor, we have observed the same behaviors in previous experiments with different setups, therefore the results can be assumed to be repeatable and not coincidencental.

We are using the following setup on AWS for evaluation designed to make the task managers the bottleneck:

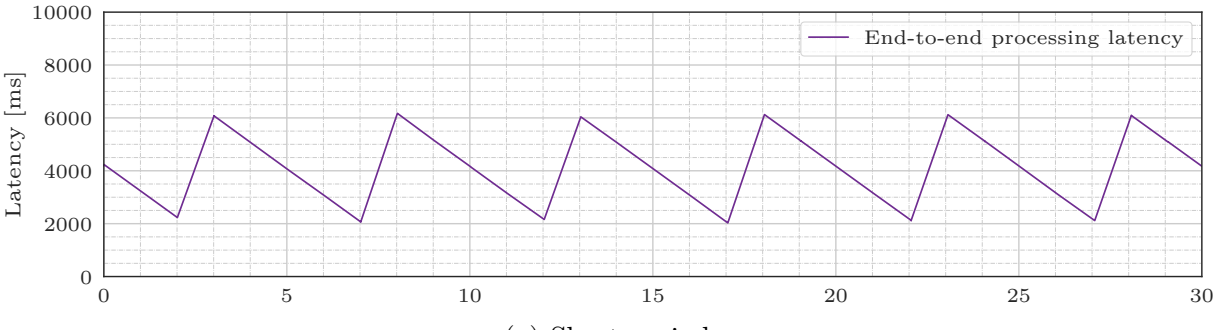
- Kafka: 4×m5.xlarge with Kafka 2.5.0, 15 min retention, 4 partitions per topic with replication factor 3
- Flink Job Manager: 1×t3.medium with Flink 1.11.1, 2 GB memory per job manager
- Flink Task Manager: 4×t3.medium with Flink 1.11.1, 2 GB memory per task manager, memory state backend
- Ingestion: 1×c5.24xlarge
- Latency tracker: 1×m5.xlarge

All nodes use CentOS 7.6 with Linux kernel 3.10.0-957, Python 3.8.3 and OpenJDK 11.0.8. We deliberately choose small instances for Flink task managers to see the effects of scaling quicker. The ingestion instance has many cores to support large scaling factors. We run all jobs with a parallelism of 2, so each source task consumes two Kafka partitions. Every task manager has two task slots (one per vCPU), but still one job is usually executed on two different nodes and therefore requires network shuffling. We do not use checkpointing since the effect on performance strongly depends on its configuration. All jobs use event-time processing with 1 s bounded out-of-orderness watermarking. Note that this setup is for evaluation only. During regular operation, there would be no latency tracker, better task manager instances and a less powerful ingestion instance.

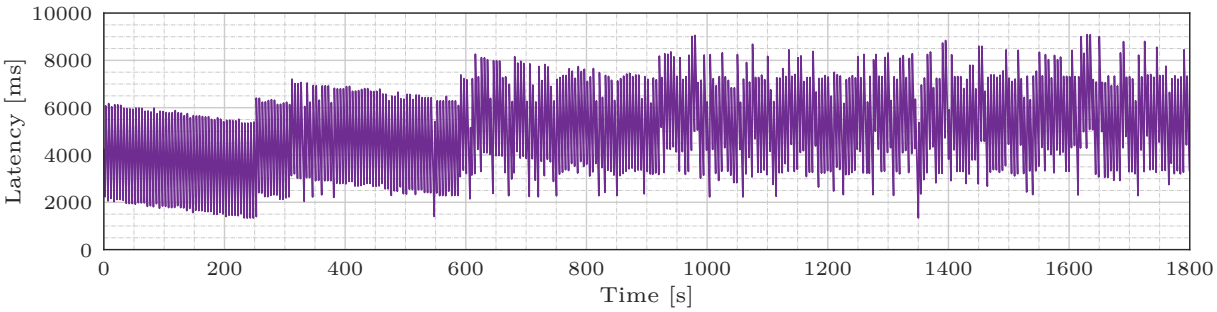
6.2 Results

The end-to-end latency of the vehicle distribution job over a span of 30 s without volume scaling is shown in figure 6.3a. The buffering of records for windowing manifests itself in the sawtooth pattern with a period of 5 s, equal to the period of the sliding window. This adds functional latency for most records, additionally to the baseline latency of 1 s due to bounded-out-of-orderness watermarking (latency markers always have the event timestamp of the latest ingested event and therefore are buffered until an event with a timestamp one second in the future is observed). Most of the latency therefore comes from the processing latency within Flink (about 99.7% of the total end-to-end latency), and the other partial latencies consisting mostly of transport within Kafka topics is negligible. Note that the emergency stop detection jobs are only affected by the watermarking, therefore they do not show the same windowing effects, as can be seen in figure B.1. However, the results of these jobs are implausible when looking at higher scaling factors since the latency falls below 1 s, which should not happen with event-time processing.

The end-to-end latency of the vehicle distribution job over a span of 30 min is shown in figure 6.3b. While the regular sawtooth pattern from the shorter excerpt can be observed



(a) Short period



(b) Long period

Figure 6.3: End-to-end latency for vehicle distribution job at normal volume

the whole time, its center varies more when looking when regarding this longer span. A periodic pattern with a period of several minutes emerges in all jobs when looking at higher scaling factors up to 16×. This can be seen in the first 600 s in figure 6.3b and even more in figures B.1–B.5. The periods are aligned across jobs that run simultaneously, but are not constant over time within measurements of a single scaling factor, and the period length generally increases with higher scaling factors. Still, the median end-to-end latency decreases, as shown in figure 6.4 (see figure C.1 for all jobs). On the other hand, the tail of the ingestion-to-processing latency (effectively the Kafka transport latency) increases but still remains negligibly small. The irregularity in the periodic pattern also decreases, which can probably attributed to the decrease in variance of event time since event timestamps are adjusted to be within 0.1 s of the latest known event timestamp for volume scaling. This is less variance than occurs naturally, as can be inferred from figure 5.2, but is hard to accurately replicate for replays.

When increasing the scaling factor beyond 16×, the measurement loses the regular pattern becomes more and more unpredictable, as can be seen in figures B.6–B.8. At 32×the normal volume, the CPU utilization on Flink task managers is between 20% and 70%, while the latency tracker is at over 90%. At 64×, the task manager load is always below 30%, and the latency tracker below 10%. Additionally, the Kafka load is low and throughput is much less than expected. Therefore, the latency tracker and later the ingestion component become bottlenecks at higher scaling factors. Therefore, we regard only the measurements for 1×–16×as reliable.

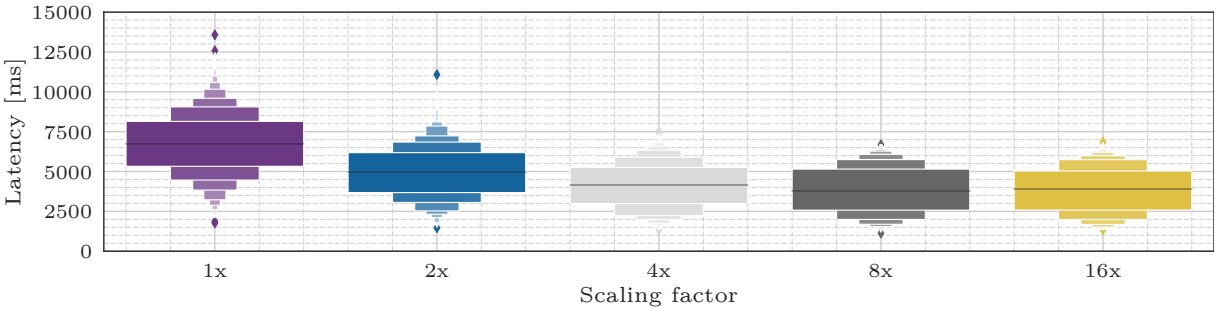


Figure 6.4: End-to-end latency distribution for vehicle distribution job at different scaling factors

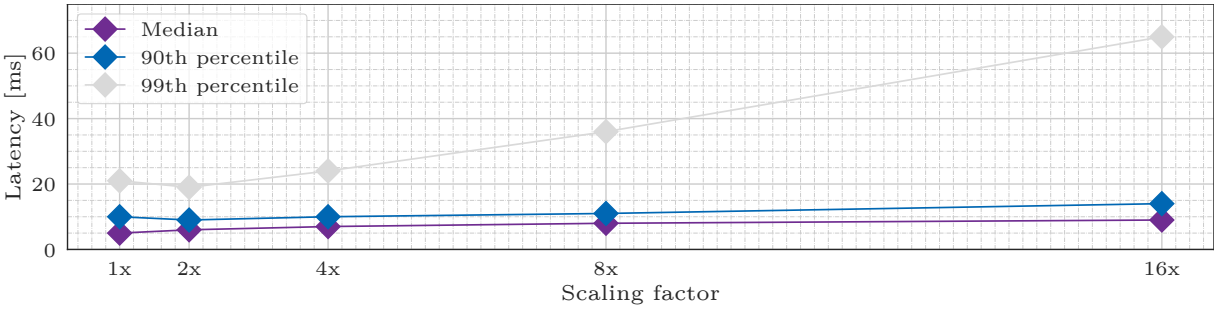


Figure 6.5: Ingestion-to-processing latency distribution for all jobs at different scaling factors

6.3 Discussion

At the normal volume that can be expected from the HSL HFP API during the day, our solution can produce results in near real-time. An end-to-end latency under 10 s (and in some cases as low as 2 s) is sufficient for many applications, especially since the functional parts of the processing latency cannot be overcome without increased resource requirements (due to more frequent window evaluations or tighter watermarks resulting in more recomputations due to late data). Also, ultra-low latencies can be achieved if desirable through optimization and appropriate configuration, as has been demonstrated by others.

The results also show that the system is perfectly capable of handling up to 16×the normal volume. However, we cannot make statements about the performance at larger volumes with certainty. During all experiments, the CPU utilization of the Flink task manager nodes remained reasonably low despite their low computational power and low parallelism. Rather, we are pushing the limits of the ingestion component because it does not parallelize well, resulting in slow ingestion and therefore the event time experienced at the processing component is slowed down (i.e. the event-time skew increases). Additionally, the latency tracker cannot handle the large volume and therefore produces unreliable results. But even at low volumes, we are not certain of the accuracy of our results, since there are many factors that can distort the measurements such as the approximation of latencies.

In many real-world scenarios, however, the ingestion can be scaled much better and therefore our solution can still successfully be applied. Operator throughput and Kafka consumer lag are important metrics when monitoring the real-time capability of a stream processing system. These can be captured through Flink's metric system, which can replace our custom latency tracker in production scenarios (see [53]). This also enables monitoring of other features like checkpointing and memory used for keeping state. We found that the recommended state backend RocksDB severely slows down our jobs, and prevents the emergency detection jobs from making progress when checkpointing is disabled (pattern recognition has large state). Since RocksDB is recommended for production because it writes state to disk, this needs to be investigated.

7 Conclusion

Stream processing constitutes a fundamental shift of approach of dealing with data. Embracing the unbounded and unordered nature of streams enables low-latency applications and stream-specific processing techniques like session windowing and continuous pattern recognition. Event stream analytics solutions that natively deal with such data needs to solve the key challenges of time and state. Furthermore, software developers and and architects need to adopt a new mindset when making the switch from bounded batch datasets to incomplete and continuously changing streams.

This thesis highlighted the theoretical and practical aspects of designing and building such event stream analytics solutions that produce correct result even in case of faults while being able to deliver them with low latency at large scales. We successfully designed and implemented an analytics solution for the Helsinki public transportation system which allows us to monitor the current situation in real time. This is just one use case for event stream analytics and we believe that many more areas will profit from these real-time insights.

Bibliography

[1] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle, "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing," *Proceedings of the VLDB Endowment*, vol. 8, pp. 1792–1803, 2015.

[2] J. Kreps, *I Heart Logs*, First edition. Sebastopol CA: O'Reilly Media, 2014, ISBN: 978-1-491-90938-6.

[3] M. Kleppmann, *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*, First edition. Boston: O'Reilly Media, 2017, ISBN: 9781449373320.

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008, ISSN: 0001-0782. DOI: 10.1145/1327452.1327492.

[5] T. Akidau, S. Chernyak, and R. Lax, *Streaming systems: The what, where, when, and how of large-scale data processing*, First edition. Sebastopol CA: O'Reilly, 2018, ISBN: 1491983876.

[6] Yahoo Developer Network, *Hadoop Turns 10*, 2016. [Online]. Available: https://developer.yahoo.com/blogs/138739227316 (visited on 08/11/2020).

[7] Hazelcast, *Micro-Batch Processing vs Stream Processing*. [Online]. Available: https://hazelcast.com/glossary/micro-batch-processing/ (visited on 08/11/2020).

[8] M. J. Sax, G. Wang, M. Weidlich, and J.-C. Freytag, "Streams and Tables," in *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*, D. Chatziantoniou, Ed., ser. ACM Digital Library, New York, NY: ACM, 2018, pp. 1–10, ISBN: 9781450366076. DOI: 10.1145/3242153.3242155.

[9] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos, *A Survey on the Evolution of Stream Processing Systems*, 2020.

[10] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and Batch Processing in a Single Engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[11] N. Marz, *History of Apache Storm and lessons learned*, 2014. [Online]. Available: `http://nathanmarz.com/blog/history-of-apache-storm-and-lessons-learned.html` (visited on 08/14/2020).

[12] ——, *How to beat the CAP theorem*, 2011. [Online]. Available: `http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html` (visited on 08/14/2020).

[13] N. Marz and J. Warren, *Big Data: Principles and best practices of scalable real-time data systems*. Shelter Island NY: Manning, 2015, ISBN: 9781617290343.

[14] J. Kreps, *Questioning the Lambda Architecture*, 2014. [Online]. Available: `https://www.oreilly.com/radar/questioning-the-lambda-architecture/` (visited on 08/14/2020).

[15] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle, "MillWheel: Fault-Tolerant Stream Processing at Internet Scale," in *Very Large Data Bases*, 2013, pp. 734–746. DOI: `10.14778/2536222.2536229`.

[16] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Computing Surveys*, vol. 46, no. 4, pp. 1–34, 2014, ISSN: 0360-0300. DOI: `10.1145/2528412`.

[17] M. Hirzel, G. Baudart, A. Bonifati, E. Della Valle, S. Sakr, and A. Akrivi Vlachou, "Stream Processing Languages in the Big Data Era," *ACM SIGMOD Record*, vol. 47, no. 2, pp. 29–40, 2018, ISSN: 0163-5808. DOI: `10.1145/3299887.3299892`.

[18] G. Cormode, "Sketch Techniques for Massive Data," in *Synposes for Massive Data: Samples, Histograms, Wavelets and Sketches*, ser. Foundations and Trends in Databases, G. Cormode, M. Garofalakis, P. Haas, and C. Jermaine, Eds., NOW publishers, 2011.

[19] R. Adaikkalavan and S. Chakravarthy, "Seamless Event and Data Stream Processing: Reconciling Windows and Consumption Modes," in *Database Systems for Advanced Applications*, ser. Lecture Notes in Computer Science, J. X. Yu, M. H. Kim, and R. Unland, Eds., vol. 6587, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 341–356, ISBN: 978-3-642-20148-6. DOI: `10.1007/978-3-642-20149-3{\textunderscore}26`.

[20] M. Dayarathna and S. Perera, "Recent Advancements in Event Processing," *ACM Computing Surveys*, vol. 51, no. 2, pp. 1–36, 2018, ISSN: 0360-0300. DOI: `10.1145/3170432`.

[21] S. H. Ahmed and S. Rani, "A hybrid approach, Smart Street use case and future aspects for Internet of Things in smart cities," *Future Generation Computer Systems*, vol. 79, pp. 941–951, 2018, ISSN: 0167739X. DOI: 10.1016/j.future.2017.08.054.

[22] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in Apache Flink," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1718–1729, 2017, ISSN: 21508097. DOI: 10.14778/3137765.3137777.

[23] M. Kleppmann, *Making Sense of Stream Processing*, 1st edition. O'Reilly Media, Inc, 2016, ISBN: 9781491937280.

[24] J. Kreps, N. Narkhede, J. Rao, *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, vol. 11, 2011, pp. 1–7.

[25] M. Kleppmann, A. R. Beresford, and B. Svingen, "Online Event Processing: Achieving Consistency Where Distributed Transactions Have Failed," *Communications of the ACM*, vol. 62, no. 5, pp. 43–49, 2019, ISSN: 0001-0782. DOI: 10.1145/3312527.

[26] G. Cugola and A. Margara, "Processing flows of information," *ACM Computing Surveys*, vol. 44, no. 3, pp. 1–62, 2012, ISSN: 0360-0300. DOI: 10.1145/2187671.2187677.

[27] O. Etzion and P. Niblett, *Event processing in action*. Greenwich 74° w. long.: Manning, 2011, ISBN: 9781935182214.

[28] A. Hinze, K. Sachs, and A. Buchmann, "Event-based applications and enabling technologies," in *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, A. Gokhale and D. C. Schmidt, Eds., New York, N.Y.: ACM Press, 2009, p. 1, ISBN: 9781605586656. DOI: 10.1145/1619258.1619260.

[29] M. Fowler, *What do you mean by "Event-Driven"?* 2017. [Online]. Available: https://martinfowler.com/articles/201701-event-driven.html (visited on 08/17/2020).

[30] Apache, *Apache Flink 1.11 Documentation: RabbitMQ Connector*. [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.11/dev/connectors/rabbitmq.html#rabbitmq-source (visited on 08/18/2020).

[31] StreamNative, *Pulsar vs Kafka: From Use Cases and Community to Features and Performance*, StreamNative, Ed., 2020.

[32] Amazon Web Services, *What Is Amazon Kinesis Data Streams?* 2020. [Online]. Available: https://docs.aws.amazon.com/streams/latest/dev/introduction.html (visited on 08/18/2020).

[33] D. Nguyen, A. Luckow, E. Duffy, K. Kennedy, and A. Apon, "Evaluation of Highly Available Cloud Streaming Systems for Performance and Price," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, IEEE, 2018, pp. 360–363, ISBN: 978-1-5386-5815-4. DOI: 10.1109/CCGRID.2018.00056.

[34] Apache, *Powered By Apache Kafka*. [Online]. Available: https://kafka.apache.org/powered-by (visited on 08/18/2020).

[35] M. Kleppmann, *Turning the database inside-out with Apache Samza*, 2015. [Online]. Available: https://martin.kleppmann.com/2015/03/04/turning-the-database-inside-out.html (visited on 08/18/2020).

[36] B. Svingen, *Publishing with Apache Kafka at The New York Times*, 2017. [Online]. Available: https://open.nytimes.com/publishing-with-apache-kafka-at-the-new-york-times-7f0e3b7d2077 (visited on 08/18/2020).

[37] N. Narkhede, G. Shapira, and T. Palino, *Kafka: The Definitive Guide: Real-time data and stream processing at scale*, First edition. Sebastopol CA: O'Reilly Media, 2017, ISBN: 9781491936160.

[38] E. Shahverdi, A. Awad, and S. Sakr, "Big Stream Processing Systems: An Experimental Evaluation," in *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*, IEEE, 2019, pp. 53–60, ISBN: 978-1-7281-0890-2. DOI: 10.1109/ICDEW.2019.00-35.

[39] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia, "Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark," in *Proceedings of the 2018 International Conference on Management of Data*, G. Das, Ed., ser. ACM Digital Library, New York, NY: ACM, 2018, pp. 601–613, ISBN: 9781450347037. DOI: 10.1145/3183713.3190664.

[40] Hazelcast, "Hazelcast Jet Datasheet," [Online]. Available: https://www2.hazelcast.com/l/30822/2017-02-05/bpdslz/30822/106931/Hazelcast_Jet_Datasheet.pdf.

[41] Apache, *Apache Flink: Powered by Flink*. [Online]. Available: https://flink.apache.org/poweredby.html (visited on 08/19/2020).

[42] E. Friedman and K. Tzoumas, *Introduction to Apache Flink: Stream processing for real time and beyond*, First edition. Sebastopol CA: O'Reilly Media Inc, 2016, ISBN: 9781491976586.

[43] Apache, *Apache Flink 1.11 Documentation: Operators*. [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.11/dev/stream/operators/ (visited on 08/19/2020).

[44] ——, *Apache Flink: A Deep-Dive into Flink's Network Stack*, 2019. [Online]. Available: https://flink.apache.org/2019/06/05/flink-network-stack.html (visited on 08/19/2020).

[45] R. Metzger, *How To Size Your Apache Flink Cluster: A Back-of-the-Envelope Calculation*, 2018. [Online]. Available: https://www.ververica.com/blog/how-to-size-your-apache-flink-cluster-general-guidelines (visited on 08/21/2020).

[46] Ably Realtime, *The maturity of public transport APIs 2019*, 2019. [Online]. Available: https://files.ably.io/research/whitepapers/the-maturity-of-public-transport-apis-2019-ably-realtime.pdf.

[47] Helsinki Regional Transport Authority, *Open data*. [Online]. Available: https://www.hsl.fi/en/opendata (visited on 08/22/2020).

[48] ——, *High-frequency positioning*. [Online]. Available: https://digitransit.fi/en/developers/apis/4-realtime-api/vehicle-positions/ (visited on 08/22/2020).

[49] K. Sahr, D. White, and A. J. Kimerling, "Geodesic Discrete Global Grid Systems," *Cartography and Geographic Information Science*, vol. 30, no. 2, pp. 121–134, 2003, ISSN: 1523-0406. DOI: 10.1559/152304003100011090.

[50] I. Brodsky, *H3: Uber's Hexagonal Hierarchical Spatial Index*, 2018. [Online]. Available: https://eng.uber.com/h3/ (visited on 08/23/2020).

[51] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking Distributed Stream Data Processing Systems," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, IEEE, 2018, pp. 1507–1518, ISBN: 978-1-5386-5520-7. DOI: 10.1109/ICDE.2018.00169.

[52] R. Lu, G. Wu, B. Xie, and J. Hu, "Stream Bench: Towards Benchmarking Modern Distributed Stream Computing Frameworks," in *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, IEEE, 2014, pp. 69–78, ISBN: 978-1-4799-7881-6. DOI: 10.1109/UCC.2014.15.

[53] Apache, *Apache Flink: Monitoring Apache Flink Applications 101*, 2019. [Online]. Available: https://flink.apache.org/news/2019/02/25/monitoring-best-practices.html#monitoring-latency (visited on 08/26/2020).

Glossary

allowed lateness

the maximum delay in processing time after the watermark that a late record may have without being discarded

at-least-once processing

the guarantee that every record of a dataset will be processed at least once, but possibly more often

at-most-once processing

the guarantee that every record of a dataset will be processed at most once, but possibly not at all

batch dataset

a bounded dataset that is complete at the time of processing

checkpointing

a technique for persisting in-memory state to a durable storage medium for fault tolerance

complex event

an event resulting from the relation of multiple primitive events

consistency

a property of data processing meaning that a job will produce the correct results, even in case of failures

data at rest

a table that shows data at a specific point in time

data in motion

a stream that shows the evolution of data over time

dataflow graph

a model for the directed data flow between sources, operators and sinks in a job with streams moving along the edges from upstream to downstream

dataset partition

a subset of a dataset, usually determined by a key

effectively-once processing

the guarantee that every record of a dataset will affect the result as if it were processed exactly once, even though it might actually be processed more often for failure recovery

event occurence

a single event instance of a specific event type

event pattern

a template that specifies one or more combinations of events to for pattern recognition

event time

the time domain in which events occur

event type

a specification of attributes for event occurences that share the same semantic intent and structure

event-based programming

also known as *event-driven architecture*, an interaction model between system components that uses events for communication

event-time skew

a measure of the skew of event time perception in the processing system, identical with processing-time lag

exactly-once processing

the guarantee that every record of a dataset will only be processed once, even in case of failures

geo-event

a primitive event with an associated geographical location

job

a specific processing task

late data

data with an event timestamp earlier than the watermark that arrive past the watermark

log

an immutable and durable append-only data structure with total order of records

log partition

a method for parallelizing logs where each partition itself is a log, multiple partitions of a log share no global order

materialized result

the result of a window evaluation that is emitted as stream record and sent to downstream consumers

micro-batching

a technique which decreases batch processing latency by decreasing batch size

node

an individual server in a cluster

offset

the position of a record in a log, monotonically increasing due to total order

operator

a transformation of a stream in the dataflow graph of a job

pattern recognition

a method for detecting subsets of events that match an event pattern specification

primitive event

an occurrence within a particular system, often just called *event*, can be a significant change or observation of some value

processing time

the time domain in which events are observed at a given processing stage, never before event time

processing-time lag

a measure of the delay between processing time and event time, identical to event-time skew

record

an element of a bounded or unbounded dataset

replay

the process of consuming a stream from an earlier point in time instead of starting with the latest record

stream

an unbounded dataset that will never be completed

stream analytics

any type of sophisticated stream processing that requires correct time handling and intermediate state

stream consumer

a component of a streaming system that receives an unbounded dataset

stream producer

a component of a streaming system that creates an unbounded dataset

stream transport

the component in a streaming system responsible for storing and transporting streams between producers and consumers

watermark

the timestamp of the point in event time up to which the system believes all inputs with lower event timestamps have been observed, a perfect or heuristic measure of completeness

window

a bounded subset of another bounded or unbounded dataset, often grouping records within a certain time period

window pane

the result of a single window evaluation of the same window with multiple triggers

A Examples of Input Events

```
{
   "VP": {
      "desi": "550",
      "dir": "1",
      "oper": 12,
      "veh": 1327,
      "tst": "2020-08-24T08:24:36.501Z",
      "tsi": 1598257716,
      "spd": 13.63,
      "hdg": 173,
      "lat": 60.191254,
      "long": 24.806850,
      "acc": 0.46,
      "dl": -133,
      "odo": 26471,
      "drst": 0,
      "oday": "2020-08-24",
      "jrn": 317,
      "line": 261,
      "start": "10:30",
      "loc": "GPS",
      "stop": null,
      "route": "2550",
      "occu": 0
   }
}
```


Listing A.1: Vehicle position event as raw JSON from the HSL HFP API


```
{
   "event_timestamp": "2020-08-24T08:24:36.501Z",
   "ingestion_timestamp": "2020-08-24T08:24:37.148598Z",
   "details": {
      "@type": "type.googleapis.com/com.dxc.ptinsight.input.VehiclePosition",
      "vehicle": {
         "operator": 12,
         "number": 1327,
         "type": "BUS"
      },
      "route": {
         "id": "2550",
         "operating_day": "2020-08-24",
         "departure_time": "10:30",
         "direction": false
      },
      "latitude": 60.191254,
      "longitude": 24.80685,
      "heading": 173,
      "speed": 13.63,
      "acceleration": 0.46
   }
}
```


Listing A.2: Vehicle position event as protobuf message after ingestion

```
{
   "ARS": {
      "desi": "147",
      "dir": "2",
      "oper": 6,
      "veh": 738,
      "tst": "2020-08-24T08:25:07.115Z",
      "tsi": 1598257507,
      "spd": 3.00,
      "hdg": 132,
      "lat": 60.145833,
      "long": 24.654697,
      "acc": -0.05,
      "dl": -66,
      "odo": 2545,
      "drst": 0,
      "oday": "2020-08-24",
      "jrn": 693,
      "line": 233,
      "start": "11:18",
      "loc": "GPS",
      "stop": 2414219,
      "route": "2147",
      "occu": 0,
      "ttarr": "2020-08-24T08:24:00.000Z",
      "ttdep": "2020-08-24T08:24:00.000Z"
   }
}
```


Listing A.3: Arrival at stop event as raw JSON from the HSL HFP API


```
{
   "event_timestamp": "2020-08-24T08:25:07.115Z",
   "ingestion_timestamp": "2020-08-24T08:25:08.379103Z",
   "details": {
      "@type": "type.googleapis.com/com.dxc.ptinsight.input.Arrival",
      "vehicle": {
         "operator": 6,
         "number": 738,
         "type": "BUS"
      },
      "latitude": 60.145832,
      "longitude": 24.654697,
      "stop": 2414219,
      "scheduled_arrival": "2020-08-24T08:24:00Z",
      "scheduled_departure": "2020-08-24T08:24:00Z"
   }
}
```


Listing A.4: Arrival at stop event as protobuf message after ingestion

```
{
   "DEP": {
      "desi": "41",
      "dir": "2",
      "oper": 12,
      "veh": 726,
      "tst": "2020-08-24T08:25:39.551Z",
      "tsi": 1598257539,
      "spd": 12.00,
      "hdg": 87,
      "lat": 60.238311,
      "long": 24.884214,
      "acc": 0.10,
      "dl": 49,
      "odo": 27,
      "drst": 0,
      "oday": "2020-08-24",
      "jrn": 1243,
      "line": 63,
      "start": "11:24",
      "loc": "GPS",
      "stop": 1331105,
      "route": "1041",
      "occu": 0,
      "ttarr": "2020-08-24T08:26:00.000Z",
      "ttdep": "2020-08-24T08:26:00.000Z"
   }
}
```


Listing A.5: Departure from stop event as raw JSON from the HSL HFP API


```
{
   "event_timestamp": "2020-08-24T08:25:39.551Z",
   "ingestion_timestamp": "2020-08-24T08:25:40.238112Z",
   "details": {
      "@type": "type.googleapis.com/com.dxc.ptinsight.input.Departure",
      "vehicle": {
         "operator": 12,
         "number": 726,
         "type": "BUS"
      },
      "latitude": 60.23831,
      "longitude": 24.884214,
      "stop": 1331105,
      "scheduled_arrival": "2020-08-24T08:26:00Z",
      "scheduled_departure": "2020-08-24T08:26:00Z"
   }
}
```


Listing A.6: Departure from stop event as protobuf message after ingestion

B Latency Measurements

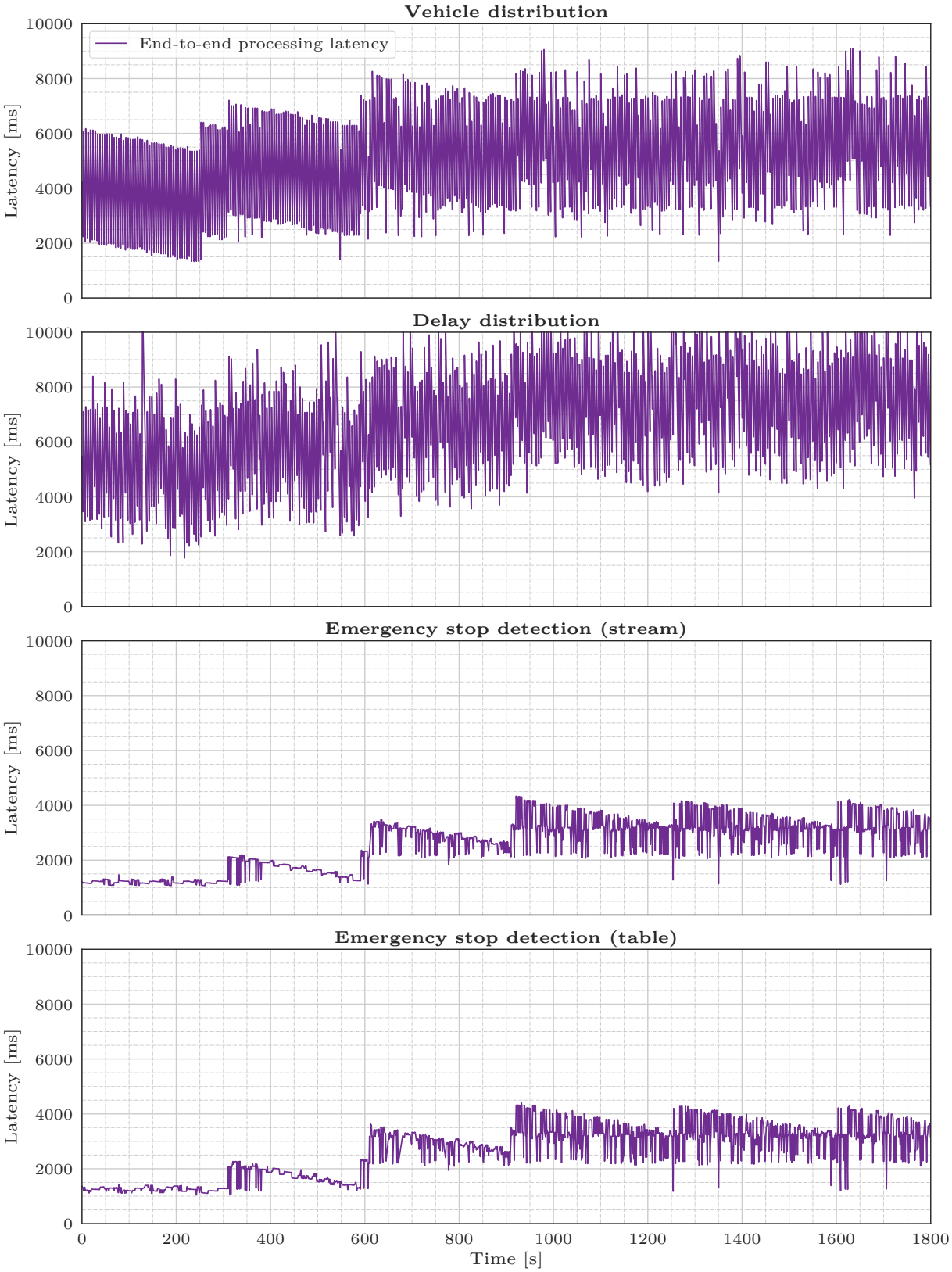


Figure B.1: End-to-end latency for all jobs at 1x volume

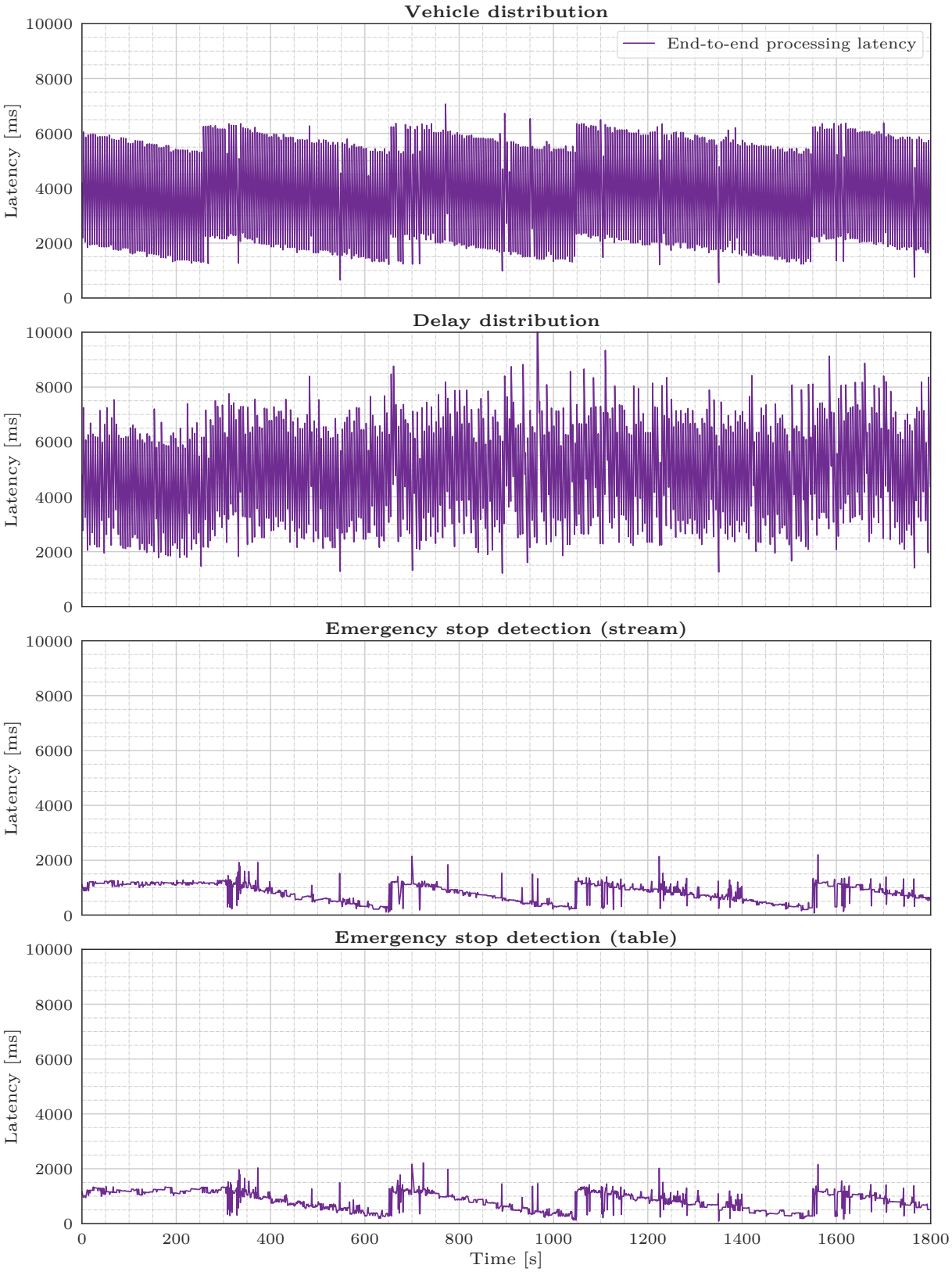


Figure B.2: End-to-end latency for all jobs at 2x volume

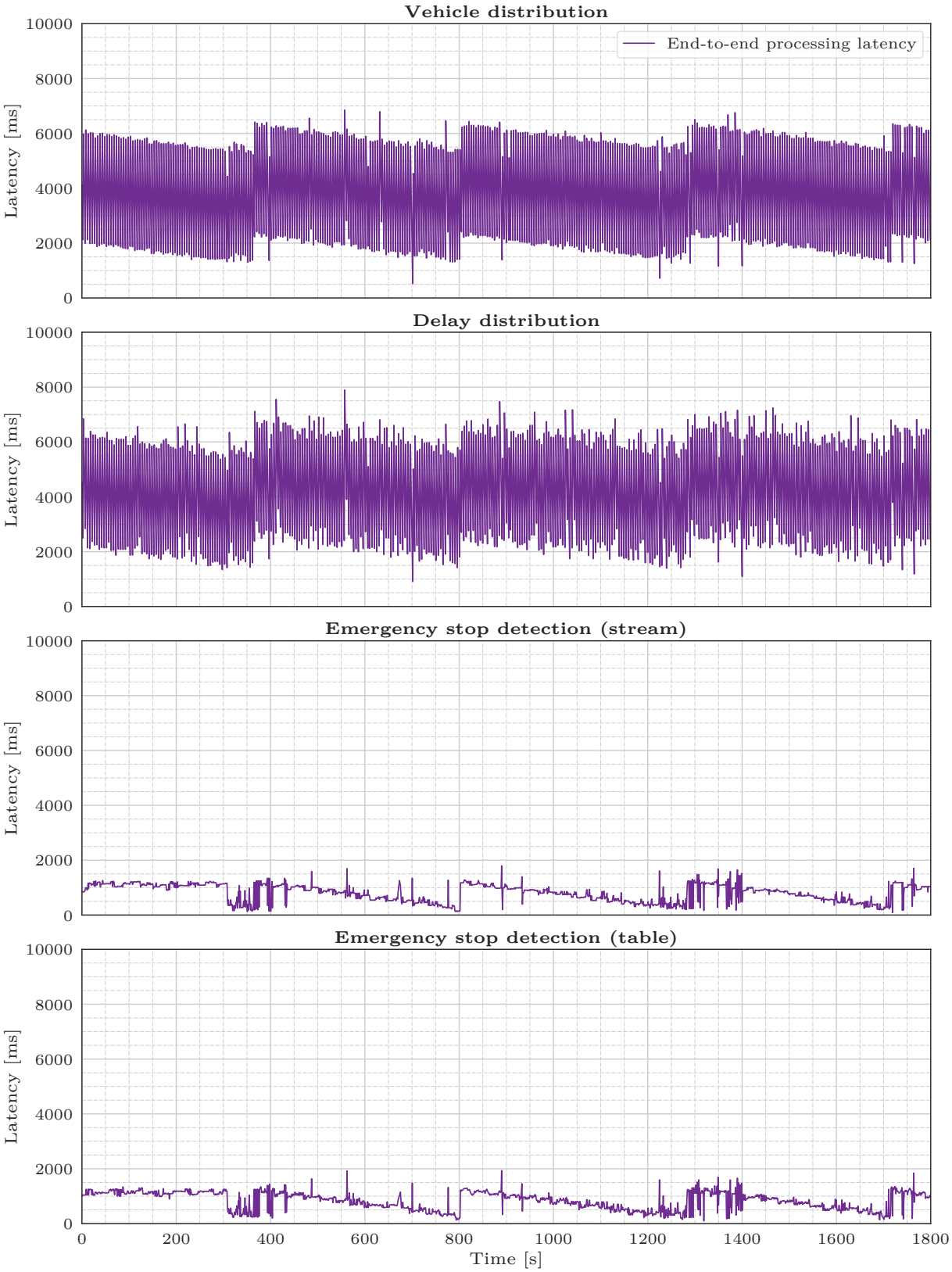


Figure B.3: End-to-end latency for all jobs at 4x volume

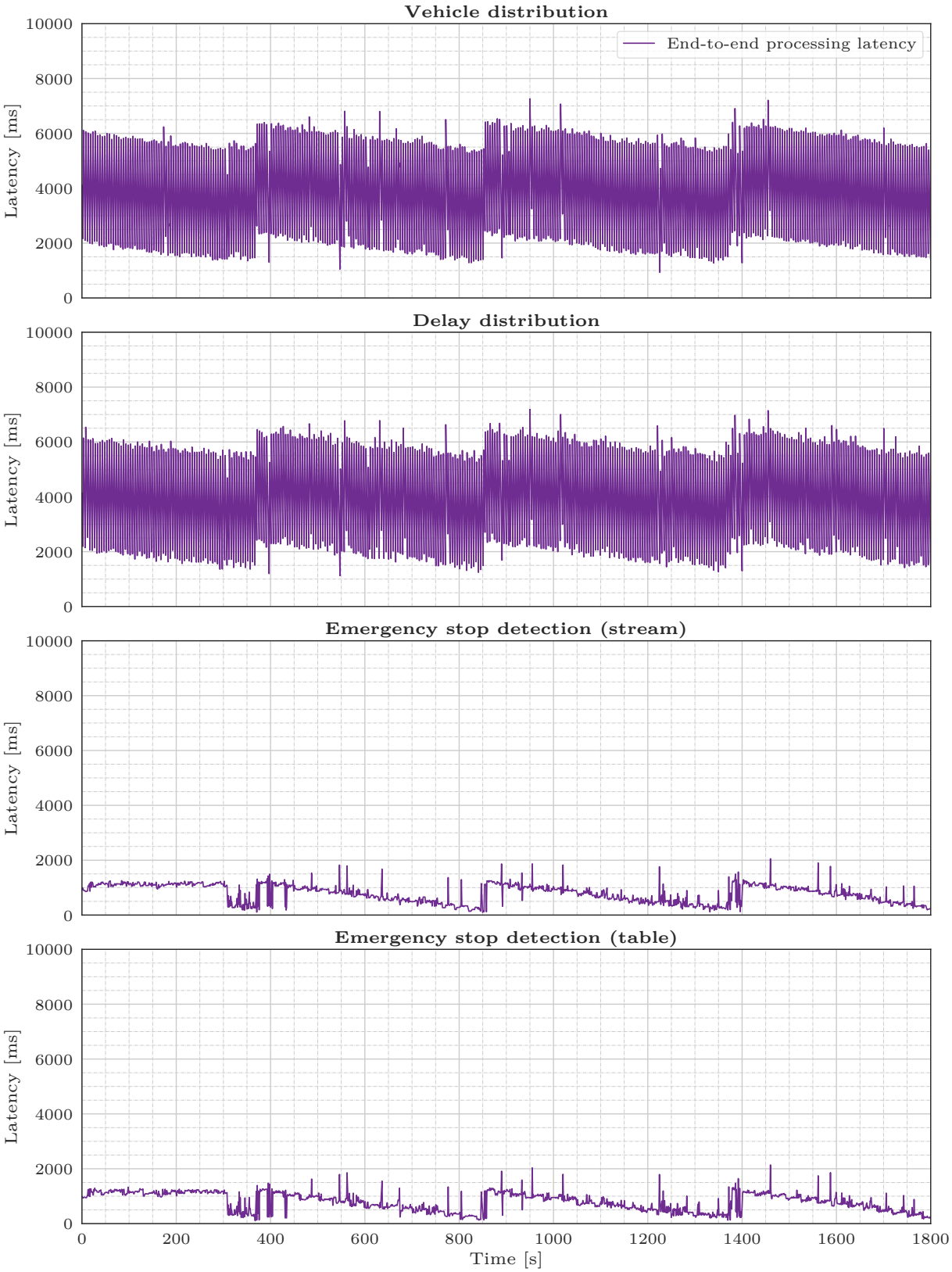


Figure B.4: End-to-end latency for all jobs at 8x volume

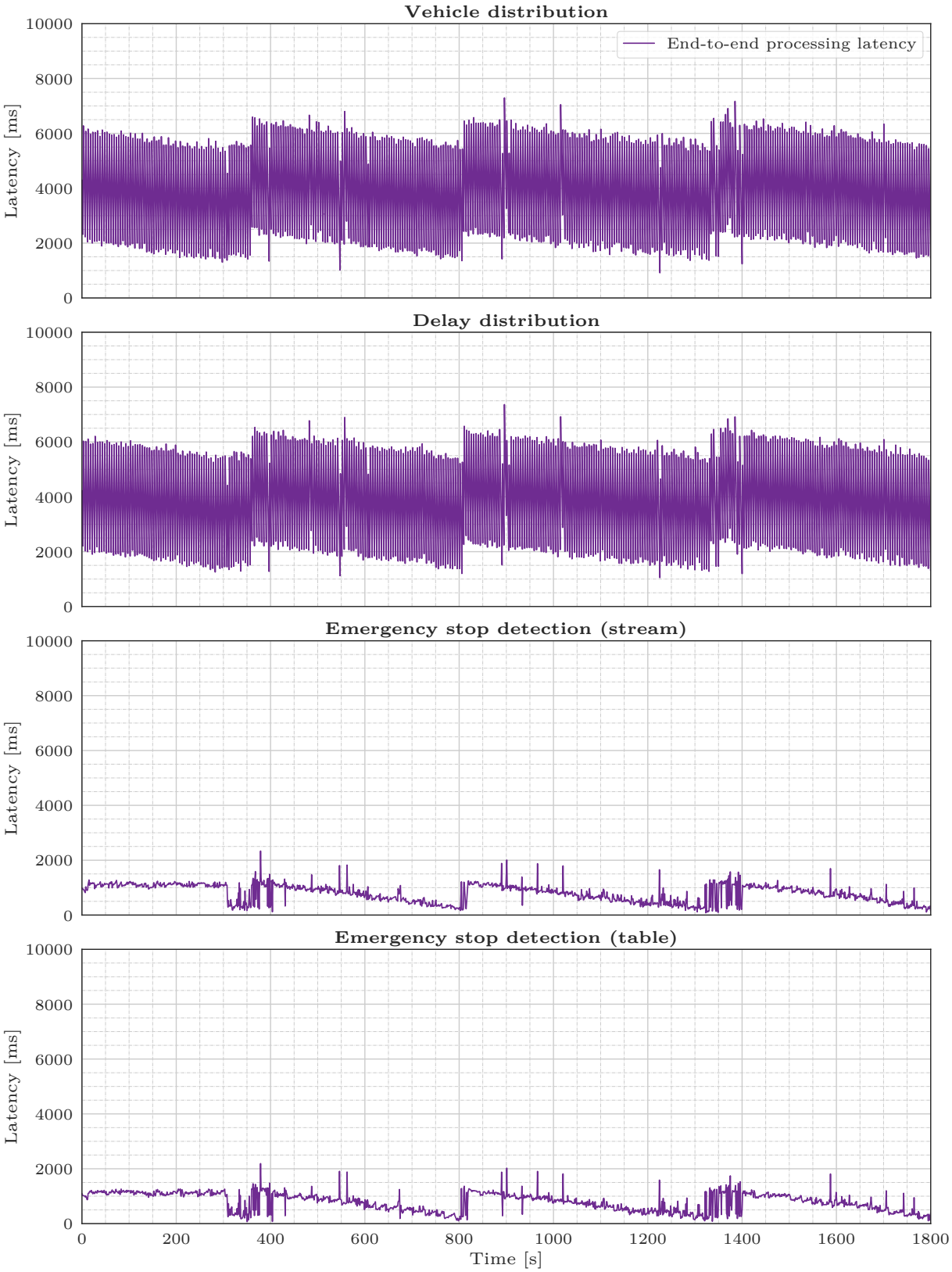


Figure B.5: End-to-end latency for all jobs at 16x volume

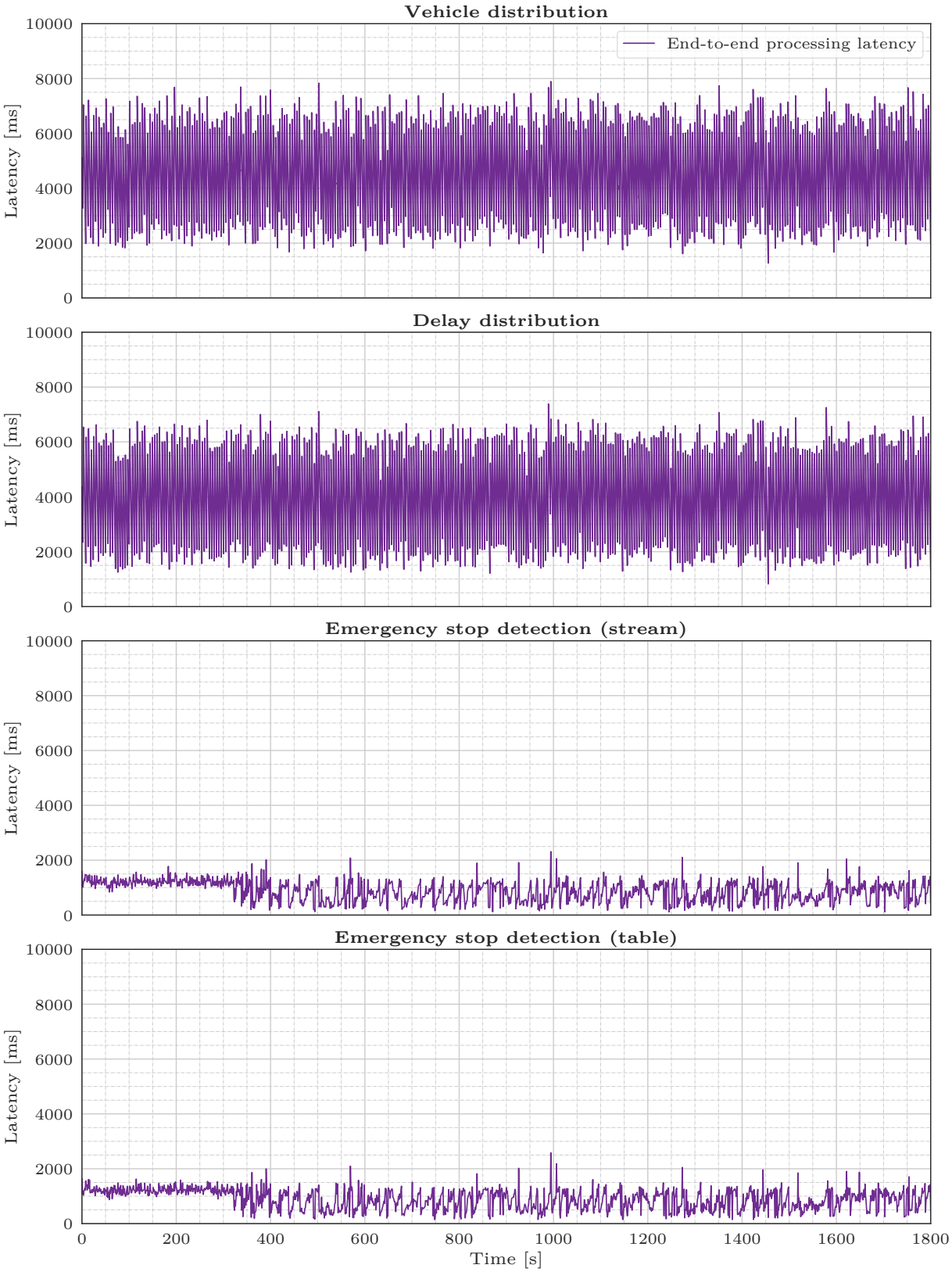


Figure B.6: End-to-end latency for all jobs at 32x volume

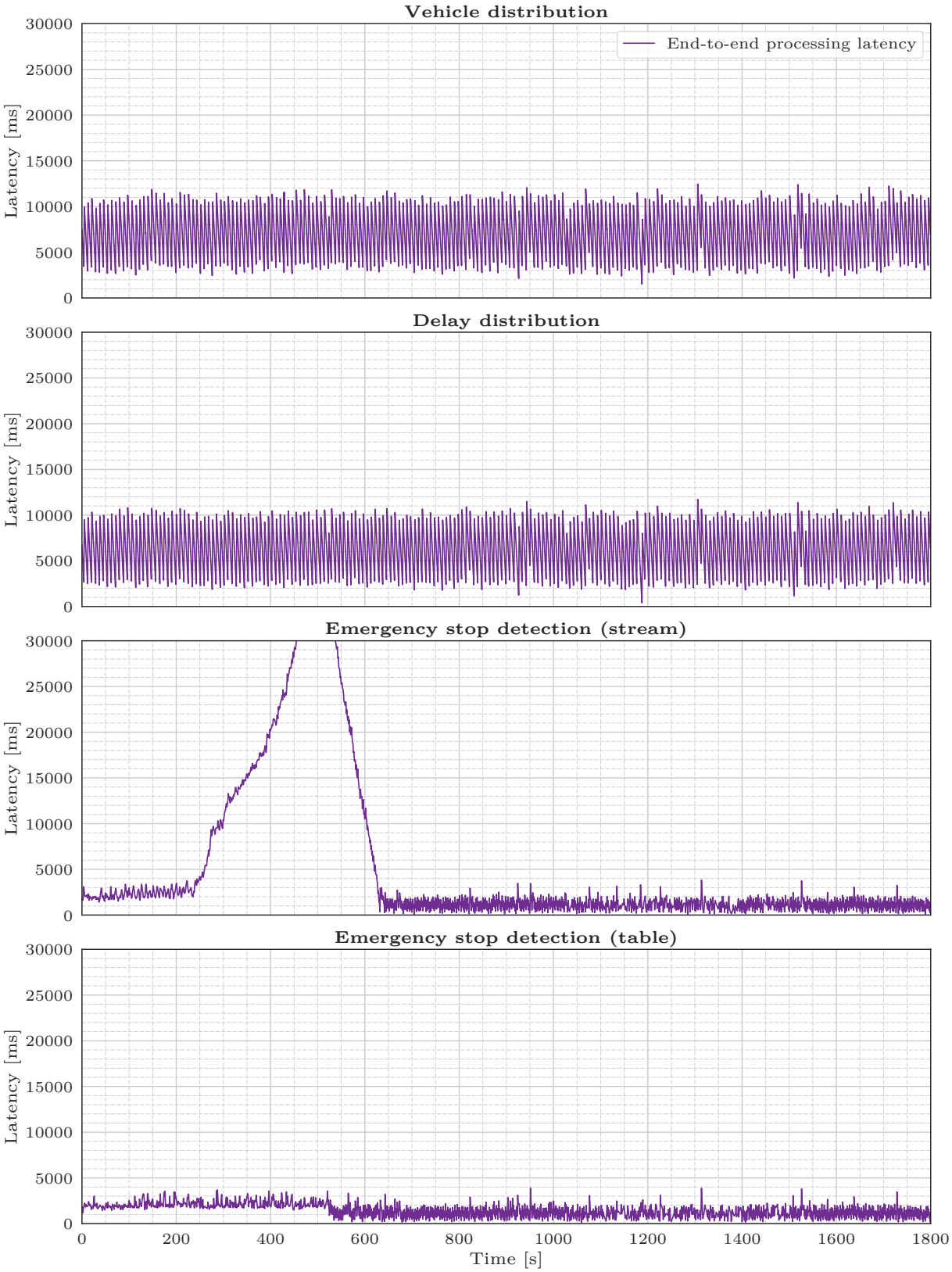


Figure B.7: End-to-end latency for all jobs at 48x volume

Figure B.8: End-to-end latency for all jobs at 64x volume

C Latency Distributions

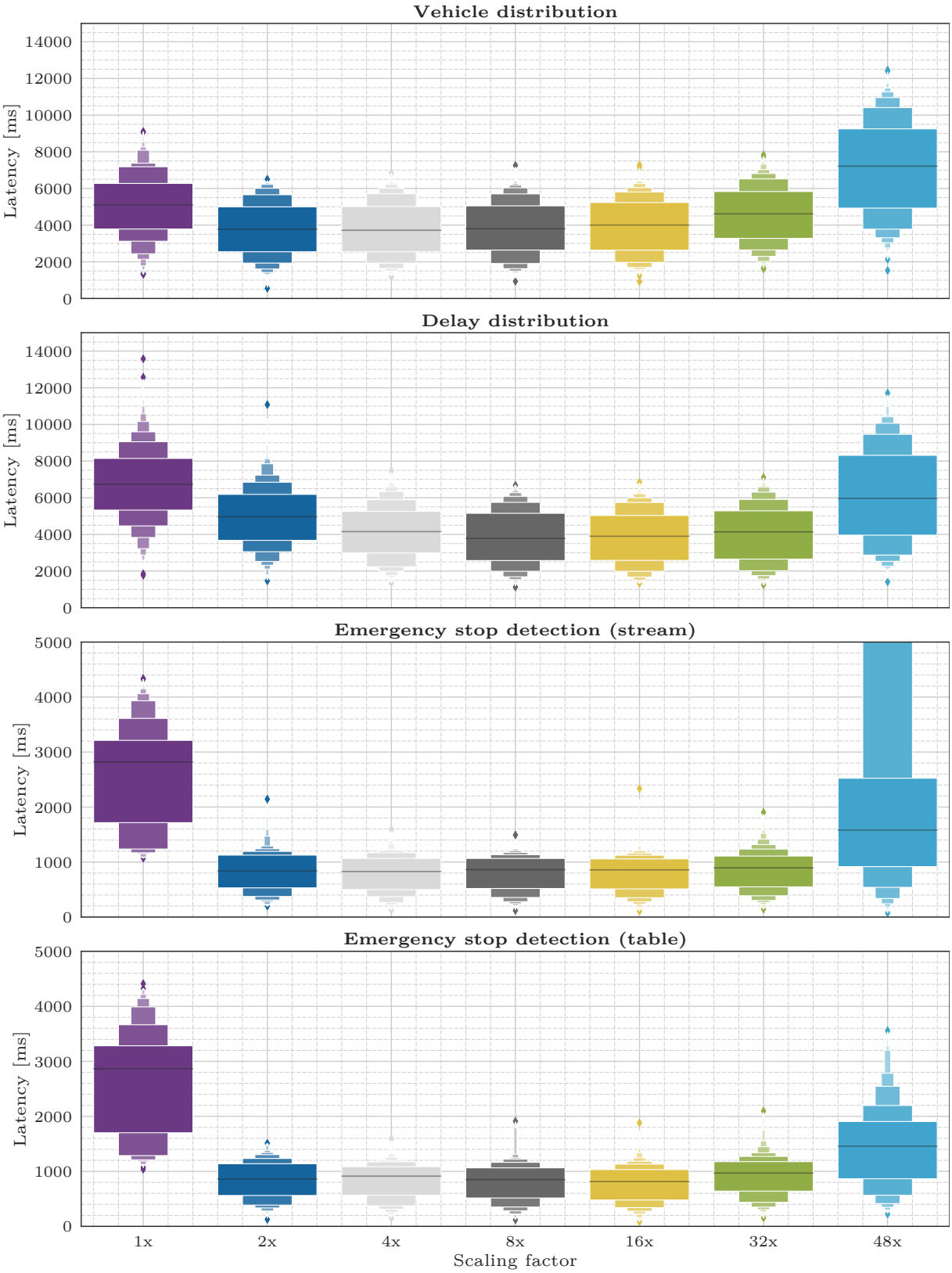


Figure C.1: End-to-end latency distribution for all jobs

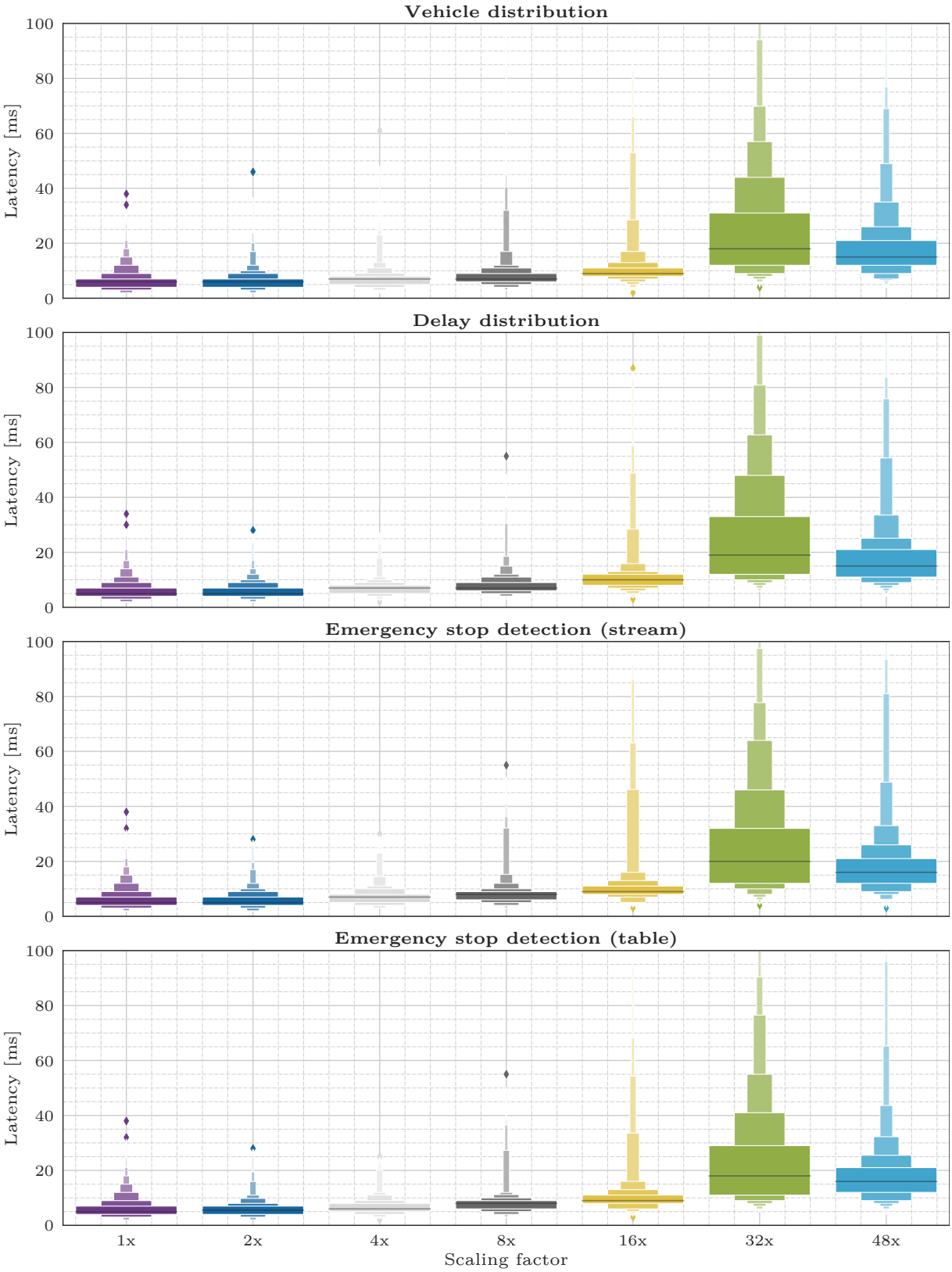


Figure C.2: Ingestion-to-processing latency distribution for all jobs

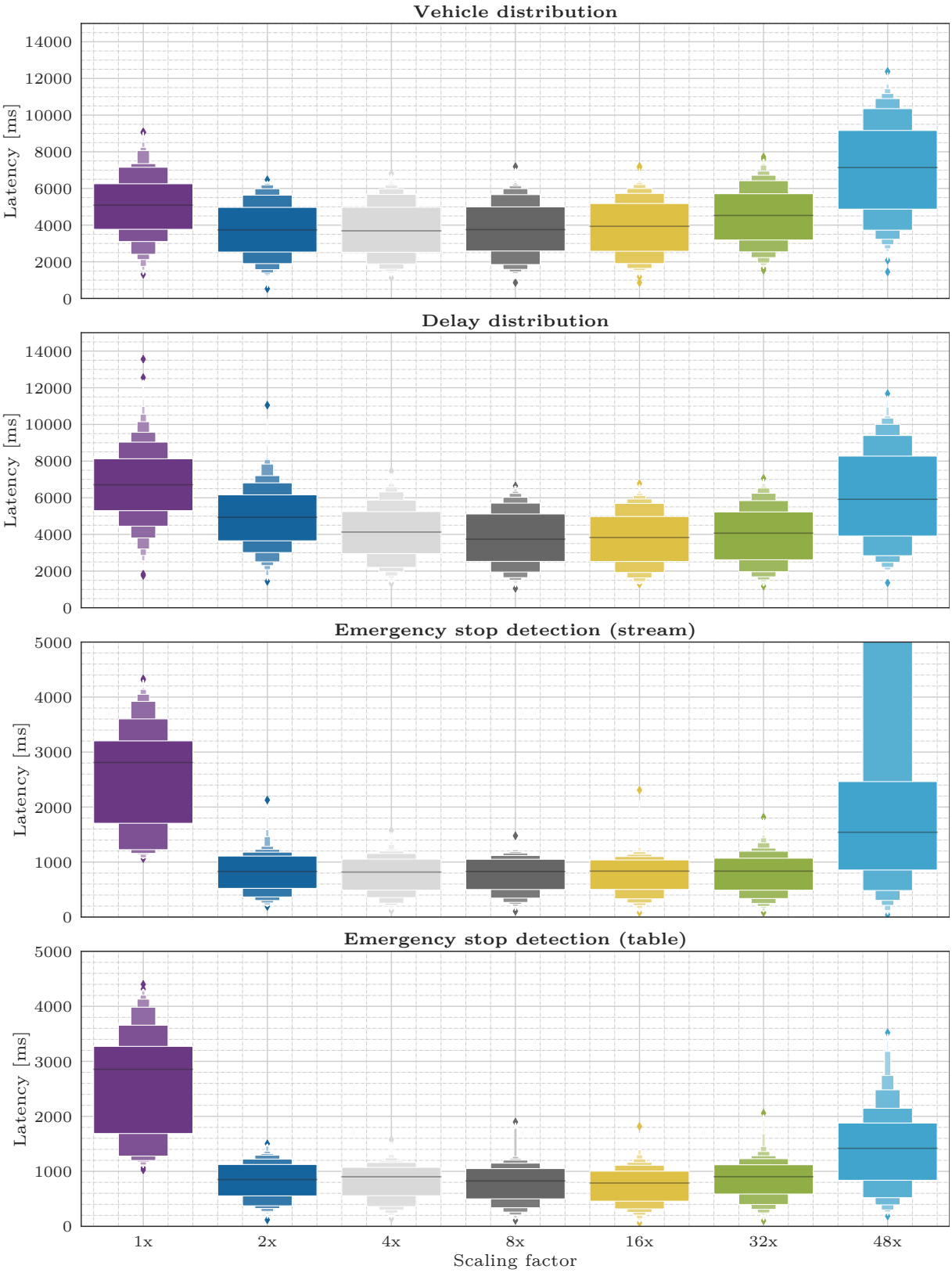


Figure C.3: Processing latency distribution for all jobs

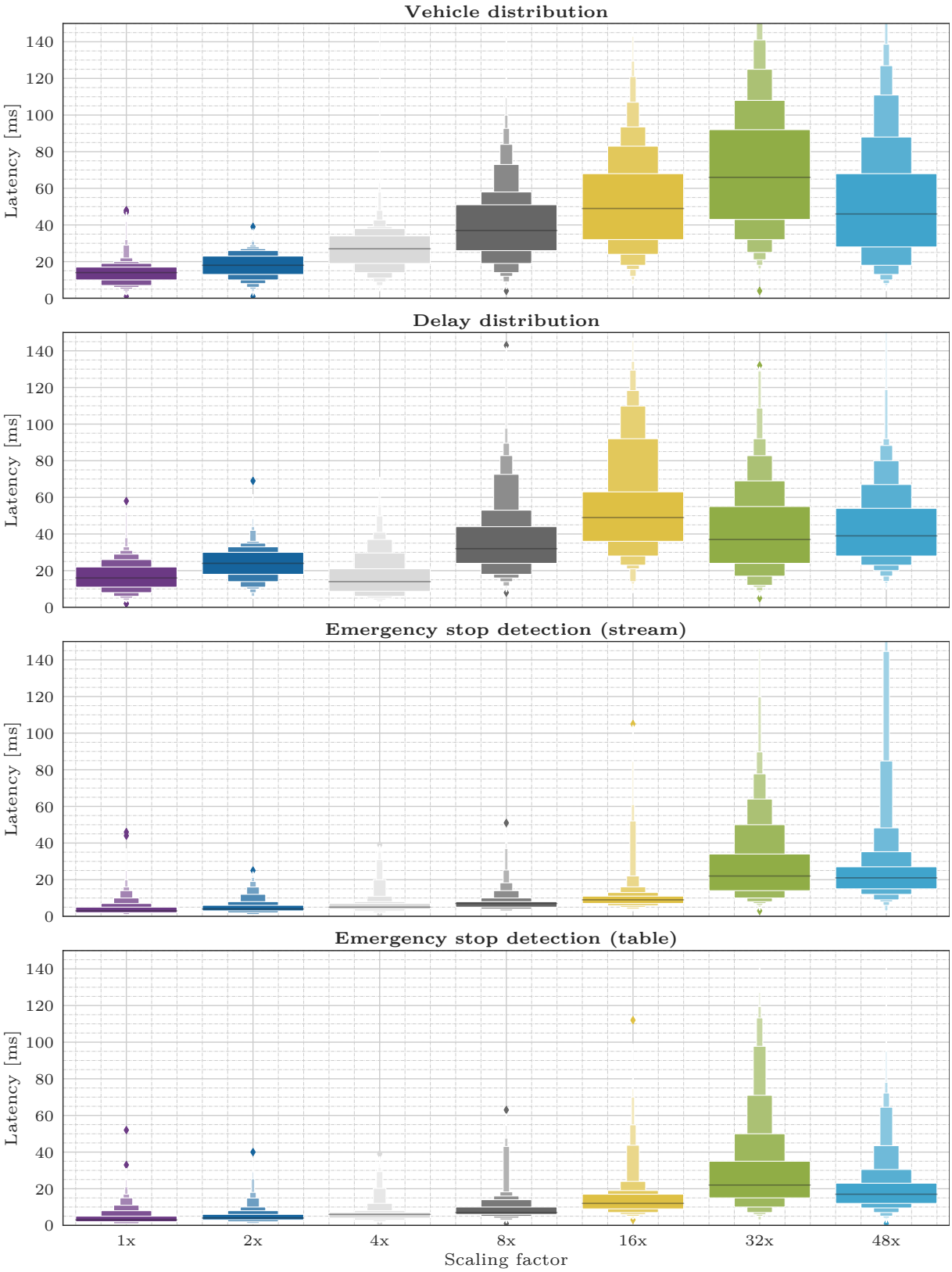


Figure C.4: Processing-to-visualization latency distribution for all jobs