CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Computer Science

Bachelor's Thesis

System architecture for sensor data stream processing using machine learning capabilities

Damián Filo
Software engineering and technologies

May 2021
Supervisor: doc. Ing. Leoš Boháč, Ph.D.

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Filo Damián**	Personal ID number:	**483510**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Software Engineering and Technology**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

System architecture for sensor data stream processing using machine learning capabilities

Bachelor’s thesis title in Czech:

Architektura systému zpracování senzorických proudových dat s využitím strojového učení

Guidelines:

Identify requirements for the ICT system architecture allowing smooth development and deployment of machine learning models. This system including ML models is intended for sensor data collection and analysis with support for real-time and historical data.
The concept of system focuses on these characteristics:
• scalability (parallel learning of models as well as data classification)
• efficient use of resources (model learning planning, automatic model validation)
• simplicity of development and deployment (agnosticism of prog. languages)
• perf. (a measurement of ref. metrics - data throughput, streams count)
• cloud deployment
• openness (primarily open technology, licensing)
• cost (infrastructure, maintenance)
Find and analyse tools to build required architecture. Based on the analysis, design suitable architecture prototypes and evaluate their characteristics on a theoretical (if possible, also practically) basis. Then select the most appropriate architecture for required purposes.
Also, implement the selected preferred architecture in testing mode and focus on validations of machine learning functionality. Measure aforementioned characteristics using data for testing purposes and compare them with the estimates.

Bibliography / sources:

1. GRATTAN, L.S. a B.T. MEGGITT. Optical Fiber Sensor Technology. 2010. ISBN 1441949836.
2. CHOLLET, François. Deep learning v jazyku Python: knihovny Keras, Tensorflow. Přeložil Rudolf PECINOVSKÝ. Praha: Grada Publishing, 2019. Knihovna programátora (Grada). ISBN 978-80-247-3100-1.
3. BISHOP, Christopher M. Pattern recognition and machine learning. [New York]: Springer, c2006. Information science and statistics. ISBN 0-387-31073-8.
4. SMITH, Michael D. a Rahul TELANG. Streaming, sharing, stealing: big data and the future of entertainment. Cambridge, Massachusetts: MIT Press, 2017. ISBN 978-0-262-53452-9.

Name and workplace of bachelor’s thesis supervisor:

doc. Ing. Leoš Boháč, Ph.D., Department of Telecommunications Engineering, FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **15.02.2021** Deadline for bachelor thesis submission: **21.05.2021**

Assignment valid until: **19.02.2023**

doc. Ing. Leoš Boháč, Ph.D. Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

. Date of assignment receipt	Student’s signature

Acknowledgement / Declaration

I would like to heartily thank doc. Ing. Leoš Boháč, Ph.D. for exceptional guidance, patience and support.

I hereby declare that this thesis is a product of my own independent work and that I have listed all of the information sources used in the thesis according to the Methodological Guideline No. 1/2009 – "On the respect of ethical principles when working on a final university thesis, CTU in Prague"

In Prague on 21. May 2021

....

Abstrakt / Abstract

Proces výběru vhodných nástrojů pro architekturu našeho IT systému je stále obtížnější. V moderních technologických trendech, jako je strojové učení a velká data, je snadné ztratit se v bezpočtu nástrojů použitených k našim účelům. Tato práce je příkladem toho, jak dlouhá může být analýza návrhu relativně jednoduchého, ale zároveň zatěžujících případů užití.

Naším hlavním cílem je seznámit čtenáře s technologiemi běžně používanými pro budování špičkových systémů zpracování proudů dat. Tato práce vysvětluje obecné pojmy a trendy návrhu takové architektury a dále se specializuje na případy užití v reálném světě. Tyto případy užití zahrnují sběr, zpracování, ukládání a analýzu údajů ze senzorů.

Po návrhu potenciální architektury a výběru vhodných nástrojů teoreticky vyhodnotíme a vybereme vhodné nástroje pro naše účely. Tyto nástroje jsou poté využity v návrzích různých konfiguracích systému. Z těchto možných konfigurací je jedna vybrána a dále vyhodnocena na reálných datech a scénáři.

Klíčová slova: bakalářská práce; strojové učení; zpracování proudových dat; distribuce dat; časosběrné data; IT architektura; zpracování senzorových dat;

Překlad titulu: Architektura systému zpracování senzorických proudových dat s využitím strojového učení

The process of selecting proper tools for our IT system architecture becomes increasingly difficult. With modern technology trends, such as Machine Learning and Big Data, it is easy to get lost in the myriad of tools capable of serving our purposes. This thesis exemplifies how thorough analysis for designing a relatively simple yet demanding use case can be.

Our main goal is to familiarise the reader with technologies commonly used for building cutting-edge stream processing system. This thesis explains general concepts and trends of designing such architecture and is further specialised for real-world use cases. These use cases involve sensor data collection, processing, storage and analysis.

After drafting potential architecture and selecting appropriate tools, we theoretically evaluate and select well-suited tools for our purposes. These tools are then suggested for use in various system configurations. Out of these possible configurations, one is selected and evaluated further on real-life data and scenario.

Keywords: bachelor thesis; machine learning; stream processing; data distribution; time-series data; IT architecture; sensor data processing;

Contents /

1 Introduction 1
1.1 The Abstracted Problem Statement 1
1.2 The General Problem Statement 1
1.3 The Scope of This Thesis 2
1.4 Outline of This Thesis 2
2 Analysis 4
2.1 System Characteristics of Interest 4
2.1.1 Scalability 4
2.1.2 Efficient Use of Resources 5
2.1.3 The Simplicity of Development and Deployment 5
2.1.4 Performance 6
2.1.5 Cloud Deployment 6
2.1.6 Openness 6
2.1.7 Cost 6
2.2 Requirements 7
2.2.1 Functional Requirements (FR) 7
2.2.2 Non-Functional Requirements (NFR) 7
2.2.3 Architectural Requirements 7
2.3 Current System Architecture 9
2.4 Prospective Architecture Proposal 10
2.4.1 L1 10
2.4.2 Data Distribution 11
2.4.3 Data Processing 12
2.4.4 Database 13
2.4.5 Analytics 13
2.5 Tool Selection 14
2.5.1 Data Distribution 14
2.5.2 Processing Engine 20
2.5.3 Data Storage 24
2.5.4 Process Automation 25
2.6 Analysis Conclusion 26
3 Theoretical Evaluation of Selected Tools 27
3.1 Metrics and Methodology 27
3.2 Tool Evaluation 28
3.2.1 Results 28
3.3 System Configurations 28
3.3.1 First Approach 28
3.3.2 Second Approach 29
3.3.3 Third Approach 29
3.4 Evaluation Conclusion 29
4 Testing 30
5 Conclusion 32
References 33
A Detailed Evaluation 39
A.1 Data Distribution 39
A.2 Data Processing 40
A.3 Data Storage 41
B Attached Files 42

Chapter 1
Introduction

1.1 The Abstracted Problem Statement

In the physical world, we can build a "building" for various purposes. The building can become a house, a factory, an office, a hospital, etc. Different requirements dictate the optimisation of the external and internal layout of a building. These optimisations also influence the equipment inside of such a building. A similar story also emerges in the field of information system design. As we do not build houses as one large multi-purpose room, likewise we do not design information systems unsuitable for our intentions and purposes.

This analogy might seem a bit silly it might be though helpful for people unfamiliar with topics discussed within this thesis. To a degree, we can compare building a house with designing an information system. Imagine that the bricks used to build a house are invisible, but you can perform various measurements on them. Upon these measurements, you can make an informed decision and select the most suitable types of bricks for your house. It is though time-consuming to measure each type of brick. Our thesis aims to gather these measurements and compile them in an insightful manner. Eventually, we would like to perform a few key measurements on our own to confirm our theoretical analysis. Even down the line, having an additional room for an upcoming child, having a garage for potential car purchase might be a worthwhile investment for the future. Since we also do not have unlimited time to study every type of brick, we apply strict pre-analysis to select the most suitable types for further analysis.

We sincerely hope this brick & building abstraction helps the reader to understand the potential hurdles of designing an information system suitable for our use cases.

1.2 The General Problem Statement

Most technology progressive companies are implementing or considering to implement machine learning (ML) capabilities [1] [2]. Mainly with intent to further enhance their operation. Existing ML tool-kits have a great reputation for building complex prediction systems quickly. However, at the same time, there are many hurdles in order to integrate machine learning into large-scale production systems [3] [4]. The underlying system architecture design is not oftentimes ready for ML-specific issues. According to Venturebeat [5], 87% of ML projects fail. This extremely high failure rate can be attributed to overlooking the crucial differences between AI/ML and conventional software development. Sure, it is possible to train and test ML model in just minutes these days. But eventually, without proper platform providing continuous maintenance for ML models, the cost of maintenance will be much higher than anticipated. There needs to be extensive coordination between various teams and roles from data scientists to DevOps. Responsibilities and ownership of models must be clear to prevent low quality algorithms remain in production. The more dynamic the real-world use-case is, the more important is clear stable ML model development and maintenance workflow.

A *technical debt* is a phenomena quite fitting for ML issues description. Technopedia [6] defines it as: “Technical debt is a concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution” But why is machine learning so difficult to maintain though? According to [7], on top of the typical code maintenance cost there are also ML-specific issues. Those issues stem from the fact that with ML, the data influences the system behaviour. That is the reason why common methods of technical debt mitigation fail. It is preferable to validate our systems readiness for ML workflows *before* we start pushing ML models into production. If the validation uncovers some either common anti-patterns used (glue code, code smells, pipeline jungle etc.) or ML-specific anti-patterns (as specified in [7]), you should view it as a warning sign. Do not blame just the developers though. Your systems might not be ready for ML. One can build a ’house’ without proper tools, but it will take much more time, and in the long run, the costs will stack. This aspect is the main focus of this thesis, to build a proper set of tools before we start building ’houses’.

Our general approach is not unique, and many case studies are adopting similar thought processes and patterns, as can be seen throughout this thesis. Some of the examples include: [8], [9], [10], [11].

1.3 The Scope of This Thesis

As in the case of many companies implementing ML systems, a company called Safibra also experiments with these technologies. Cooperation with this company allows us to specify our goal further and introduce real-world examples. This specification is also important because general studies and analyses are readily available but do not provide significant beneficial insights into our particular goals. System architect would be still required to assess the situation, compile insightful information and perform custom measurements. Our focused assessment allows companies and individuals in similar fields of interest to apply our analysis without much additional research into production directly.

Safibra specialises in optical sensor development and would like to use machine learning capabilities for sensor data evaluation. Recently there have been attempts at deploying ML on top of their existing systems. These attempts were developed using a top-down approach, i.e. data science first. Much valuable work has been done, and it provides reference points to this thesis. But unfortunately, those results were difficult to use in the production environment. This might be considered a real example of why the bottom-up approach with ML is more suitable than a top-down approach.

1.4 Outline of This Thesis

Characteristics, as defined in the assignment, are subject for further explanation in the chapter 2.1. Upon these characteristics we define specific constraints called requirements, bounding our goals within Safibra use case environment. With the bottom-up approach, in chapter 2.3, we evaluate the current state of the system architecture of Safibra, and in chapter 2.4, we also discuss a resulting prospective architecture design. Then in chapter 2.5, we take a look at which tools are available for our purposes and approaches we may implement within our resulting design.

In chapter 3.1 we further define evaluation methodology of tools discussed within chapter 2.5. This methodology is heavily reliant upon requirements defined in chapter

2.2. Preference is then determined upon general evaluation best-practices from various trusted sources. Upon evaluation in chapter 3.2, in chapter 3.3 we define multiple approaches upon which we select and configure tools within the resulting architecture variants. In chapter 4 one of the prospective variants is then selected and further evaluated on real-world examples.

Chapter 2
Analysis

Based on the thesis assignment, we would like first to explain the idea behind designing a system according to the defined characteristics and requirements. Second, the current state of the system brings our attention to a few key areas of interest. Upon these areas, we elaborate on how to implement them. The implementation comprises of correct tool selection and connecting them according to architecture we devise.

2.1 System Characteristics of Interest

Initially, to further develop characteristics defined within our assignment, we would like to offer supporting explanations. Upon these characteristics, we define requirements in a standardised manner. Metrics upon which we evaluate selected tools are defined in chapter 3.

2.1.1 Scalability

This article [12] offers an elegant guide to how the scalability of cloud systems can be defined and measured. In short, it is the ability to "sustain increasing workloads by making use of additional resources". They identified two distinct approaches on how to view scalability.

- "Resource Demand Metric" looks at how much resources are required for a particular workload size. An exponential curve would mean that with the increasing workload, we would need exponentially more resources.
- "Load Capacity Metric" looks at how our processing capacity evolves with additional resources. Here a logarithmic curve would be analogous to the exponential curve of demand metric. That is, how our processing capacity changes (increases) with additional resources. Indeed, a linear curve would be ideal scalability, but it is impossible to achieve in the real world.

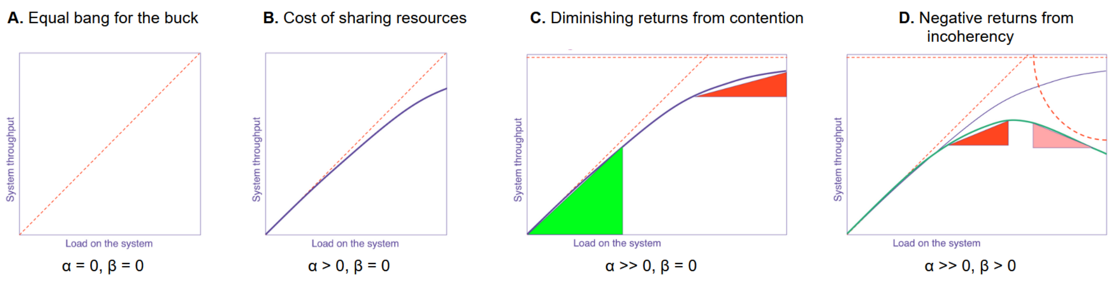

Figure 2.1. Load capacity metric of different scalability scenarios. (Source: [13])

According to [13] universal scalability law can be represented as:

$$C(N) = \frac{N}{1 + \alpha(N-1) + \beta N(N-1)}$$

Where $X(N)$ is a throughput at a given load N, and $C(N) = X(N)/X(1)$ is normalised throughput.

There are also different methods of scaling up our operation. As discussed in article [14]. Vertical scalability corresponds to expanding our singular instance to handle larger workloads. In contrast, horizontal scalability utilises more instances to handle larger workloads. While vertical scalability is usually more efficient, it has obvious technological limits. That way for large scale operations, we are usually forced to work with horizontal scalability. The limitation of vertical scalability is also one of the reasons why cloud technology development accelerated during the last decade. As workloads increased, companies needed easy and affordable solutions.

2.1.2 Efficient Use of Resources

We internally define the efficient use of resources as utilising on-premises infrastructure as close to 100% as viable. The details differ from operation to operation. For our use cases, efficiency would be at near-optimal levels if we deployed tools allowing for:

- Machine-learning model training automation
- A high degree of pipeline automation (model validation, continuous training and tuning, serving)
- Automatic data cleanup

Our general goal is to lower the idle time of our infrastructure as low as possible. Of course, this does not mean utilising the infrastructure with bogus tasks, but it is better to run even low importance tasks rather than letting the infrastructure idle. This paradigm also applies to human resources. If we have hired a data scientist to work on machine learning models, we would like to set up our system so he won't be spending much time on overhead tasks. But the situation changes if we would like to deploy our application onto public cloud service. The goal becomes the opposite. We want to utilise the infrastructure as little as possible but still provide fully for our use cases.

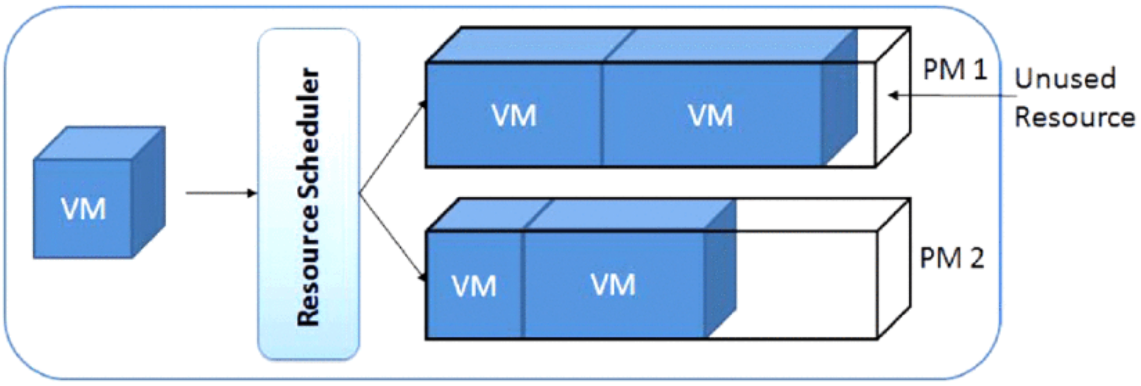

Figure 2.2. An example of inefficient resource use with static VM[1] resource allocation. (Source: [15])

2.1.3 The Simplicity of Development and Deployment

Suppose a company uses a system for which there are not many people who know how to operate it. The company may fall into the trap of unavailability of additional human resources to operate the system. Or the required qualification would cost much time and many resources. This situation can cause devastating problems for smaller companies. Easy migration support between programming languages or system modules can be considered crucial for smaller companies utilising advanced technologies. This way, the technical debt could be ameliorated by the ability to phase out deprecated code gradually

The interfacing capabilities between programming languages are becoming increasingly important as the software industry gradually shifts from monolithic to modular approaches. An article [16] illustrates the importance of cross-language interoperability in modern IT systems development.

2.1.4 Performance

Internally we define performance as a metric of how much can you get from a system that costs a certain amount of money and labour. This metric is especially useful if your finances are constricted, and you can't afford to experiment with such technologies wildly. Together with scalability metrics, you can calculate unit costs of your desired workload. This knowledge can be advantageous to planning (business plan, project plan, research plan). Some of the metrics relevant for this thesis are:

- Stream processing capability (count of streams)
- Data I/O (Database throughput, data distribution throughput, network bandwidth, Technology limits)
- Latency (offload latency-sensitive workloads to L1)

As performance metrics are generally difficult to grasp and require purpose-build definition for each type of tool, we define these metrics further in chapter 3.

2.1.5 Cloud Deployment

For small vendors the flexibility is crucial. Sometimes it is desirable to test new technology on leased infrastructure first. That is why we would want to keep an eye open on easy deployability on public cloud services of our selected tools. As we are considering cloud, cluster-based architecture, there shouldn't be a problem to achieve this characteristic. One of the criteria for selecting the right tools for our architecture is their popularity and user base size. Popular tools are usually natively supported by large cloud service providers.

2.1.6 Openness

One way to save the cost of technology is to use open-source community-developed software. This paradigm shouldn't be pursuit at all cost. Sometimes it restricts us from pursuing more reliable or faster software alternatives. Usually, proprietary software comes with perks such as extended support from vendors or higher platform stability. The detriment usually comes at the cost of fees and usually an inability to customise the software. Open-source software, on the contrary, is preferred for experimental usage. Lately, even big players such as Google, Microsoft or Amazon are supporting open-source projects and sometimes replacing their proprietary software with them. The popularity of for example Linux and Apache software is only expected to grow in the future. Usually, this phenomenon does not compromise big software companies because there is increasingly more money providing support for these products or customised versions of those products than in retail sales.

2.1.7 Cost

We can identify two types of costs: one time costs and recurring costs. In the realm of our project, let's focus on our infrastructure and software tools. With infrastructure, the difference boils down to whether or not we want to pay an upfront cost and own the infrastructure or we would like to deploy our solution onto public cloud service. Within this context, we would like to allow for both. Open source tools are usually available for free upfront, but the costs stack up with more difficult maintainability. We want to consider the advantages and disadvantages of each approach within this thesis.

2.2 Requirements

Requirements are usually categorised into two groups. Functional requirements are definitions of what the system should do. In contrast, non-functional requirements define how the system will perform defined functions. Based on the characteristics explained above, we can identify these requirements.

2.2.1 Functional Requirements (FR)

The system must:

1) Allow for the processing of streaming data.
2) Be compatible with AS-IS custom data sources (read/write abilities).
3) Support various machine learning libraries, including TensorFlow.
4) Be primarily developed in Python.
5) Offer programming language flexibility.
6) Be highly scalable through using, for example, parallel cluster computation and data lake technologies.
7) Have work planning and scheduling.
8) Allow for machine learning model development automation for more efficient infrastructure usage.
9) Support in-memory data processing to allow for accelerating data-intensive scenarios.
10) Support resiliency to be used, at least when needed.
11) Not lose data, at least when needed.
12) Be able to visualise data.

2.2.2 Non-Functional Requirements (NFR)

1) Commercial usage of the system must not require special licensing.
2) The cost of running the system must be approved by the client.
3) The system architecture must be built with cloud deployment in mind to either public or private cloud infrastructure.
4) The system should be built upon open and widely available tools.
5) Tight integration of existing proprietary software should be possible.

2.2.3 Architectural Requirements

The two most popular approaches to designing a big data analytics system architecture are Lambda and Kappa architectures [17] [18]. Lambda architecture is older and has been developed with legacy batch processing in mind. In addition to batch processing, there is also a modern stream processing layer. The newer streaming approach is usually used for workloads that require fast responses to events.

The idea behind Kappa architecture is that you can utilise the stream processing layer also for batch processing. Furthermore, the ability to apply the same algorithms and approaches used for stream data also on batch data significantly simplifies the code

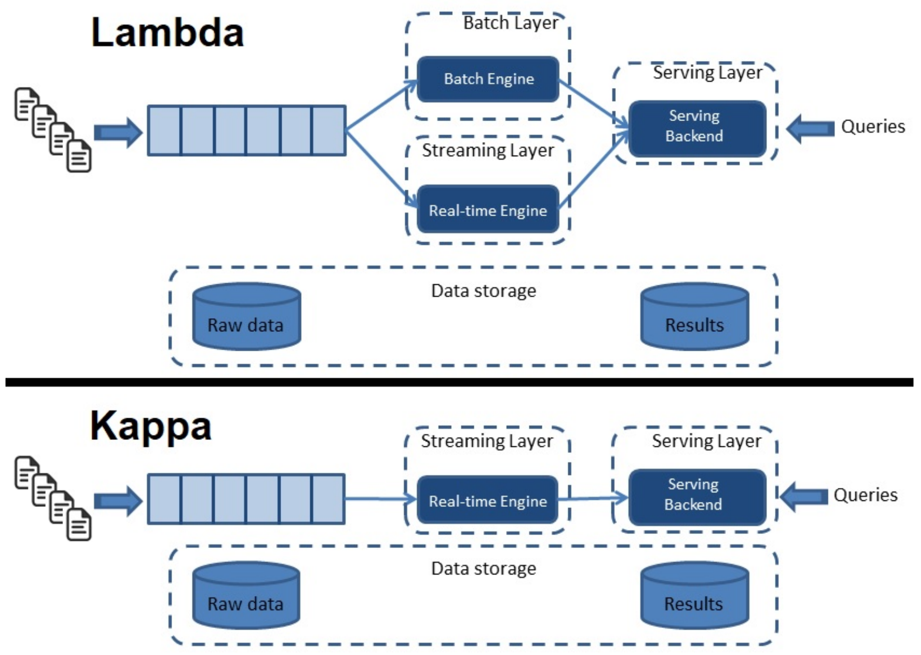

Figure 2.3. Overview of Lambda and Kappa architectures. (Source: [19])

maintainability. With Lambda architecture, the batch and stream code-bases would be usually separated as our use cases are usually focused on processing stream data. As batch and streaming data ought to be using the same channels, there are usually concerns about the data distribution layer becoming the bottleneck of our system. But as in the article at Uber Engineering [20], this deficiency can be resolved by utilising native database connectors, separate from common channels.

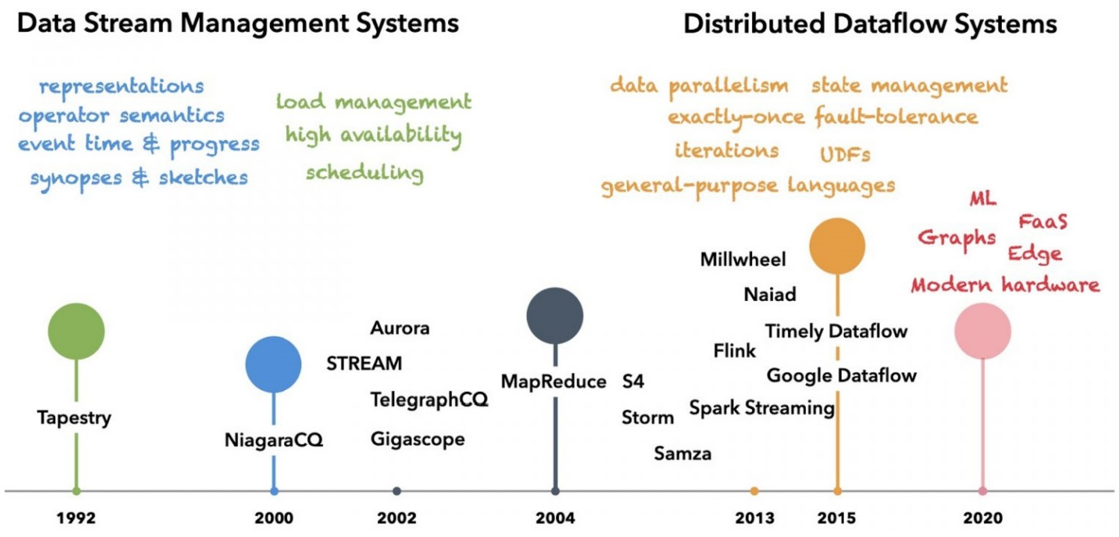

Figure 2.4. Timeline of various stream processing tools and indication of the ideology shift. (Source: [21])

Although the preference of Kappa architecture throughout this thesis is based on the intended usage of the system, each approach is valid. Our preference is aligned with the historical shift of trends in data processing approach [21]. Primary data units are streams of sensor data and events. Batch data could still be processed as a particular type of bounded stream. This approach allows us to further focus on stream processing optimisation, as we experiment with various configurations.

2.3 Current System Architecture

Let us begin with an overview of the current system architecture.

The current architecture comprises of two processing components (levels) serving different roles. This role separation is crucial to the cost-efficiency and stability of the platform. Each level is independent of one another. The communication is based on a traditional TCP/IP network with VPN[1] tunnelling.

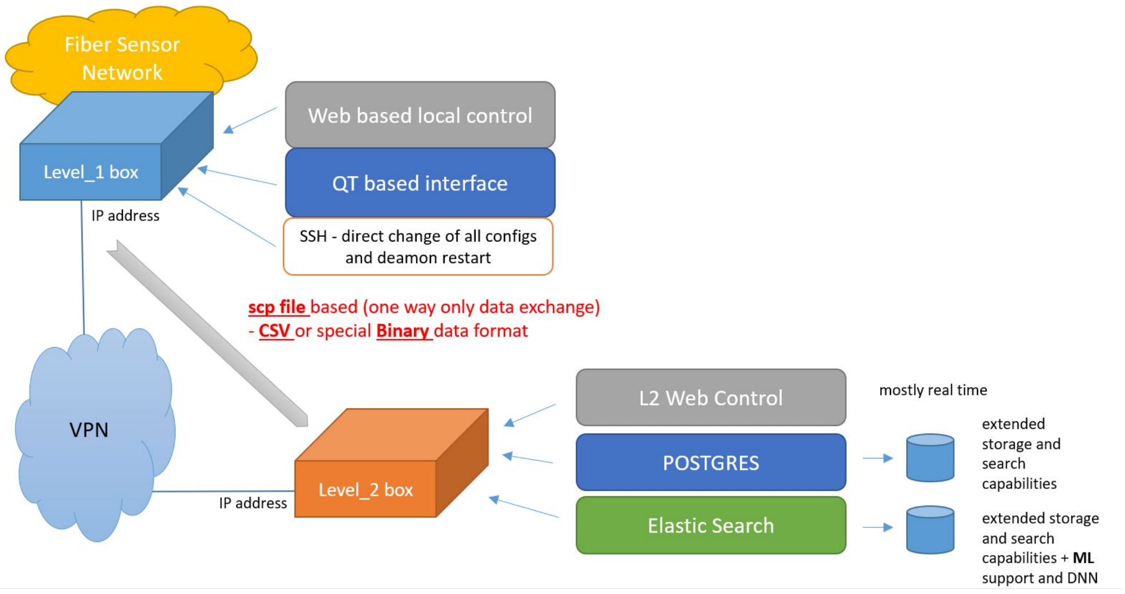
Fiber Sensor Network
Level_1 box
IP address
Web based local control
QT based interface
SSH - direct change of all configs and deamon restart
scp file based (one way only data exchange)
- CSV or special Binary data format
VPN
IP address
Level_2 box
L2 Web Control
POSTGRES
Elastic Search
mostly real time
extended storage and search capabilities
extended storage and search capabilities + ML support and DNN

Figure 2.5. High-level view of the current system architecture. (Source: own data)

The ”Level1 box” is responsible for the direct collection of data from sensors and sending them out for further processing. This component is low performance, I/O focused platform running a distribution of Linux-based OS. It can be configured and monitored via a local web-based app, QT interface or SSH. The idea behind this box is to have a lightweight mobile data collection point that doesn't require additional equipment to operate. But for advanced capabilities, a network connection is required.

The ”Level2 box” is responsible for data processing, storing and evaluation. This component accepts and aggregates data from ”Level1 boxes”. The hardware is much more powerful than ”Level1 box”, but the component is still severely limited by how much data it can process at a time. Services running on this level are database server, web control and elastic search. The ML does not have proper integration, and there are no automated pipelines available. Users need to manually select and prepare data, train and validate models and deploy them.

The communication between levels is usually performed via the internet. That is why there is a need for a secure connection. Usually, and in this case, VPN provides end-to-end encryption of data travelling across unsecured networks. This way, the communication cannot be wiretapped somewhere between our networks. The security of the data transfer may be crucial if different sensor technologies such as cameras are

[1] Virtual Private Network

used. With cameras, there would be high privacy concern also with how and where the camera feed is stored. VPN also allows for the convenience of virtually being directly connected into our internal network, allowing for better and easier device management.

Overall this architecture has many bottlenecks barring us from scaling it up meaningfully. It is certainly not ready for optimal machine learning deployment and development pipelines. But the architecture is quite simple and doesn't require specialised knowledge to operate it.

2.4 Prospective Architecture Proposal

We would like to begin with an exploration of how it is possible to improve the AS-IS state. The design below can be considered as a general overview of areas of interest. With a strict following of requirements defined above, we can identify 5 different areas to enhance this operation further. In this proposal, we incorporate ”Level2 boxes” into either private or public cloud infrastructure. This way, we don't need to consider them as an independent entities, but as mere resource (infrastructure) providers.

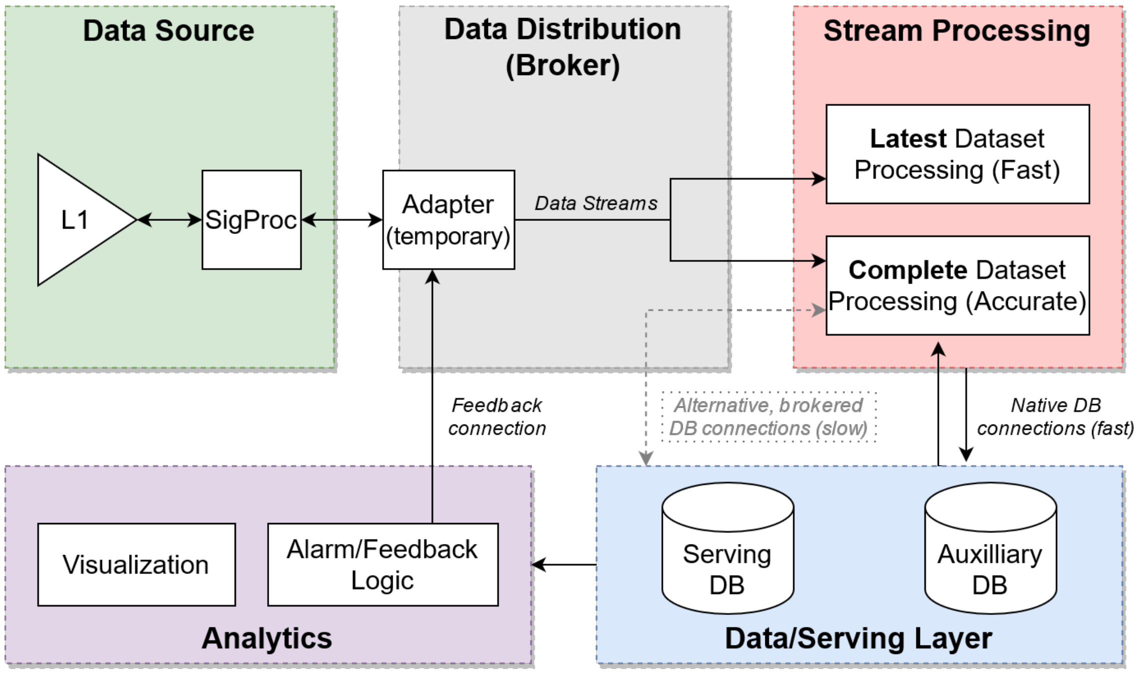

Figure 2.6. High-level view of kappa-inspired system architecture. (Source: own data)

2.4.1 L1

With regards to aforementioned requirements, we can now consider L1 boxes connected to SigProc[1] software as generalised data sources that can provide readings of various types of sensors. SigProc, together with L1, is responsible for initial data preparation, or raw data can be forwarded directly from L1. However, the raw data streaming possibility is not a viable option for universal usage due to the high throughput requirement for each data stream. Our goal is to test this option, but it is not a requirement. Optionally L1 can be equipped with an ML accelerator used for low-latency offline data evaluation/classification with ML models provided by higher layers. This would greatly expand the possibilities of L1 data preprocessing. Microsoft, for example, is quite active in the field of so-called Edge Machine Learning [22].

[1] Signal Processor

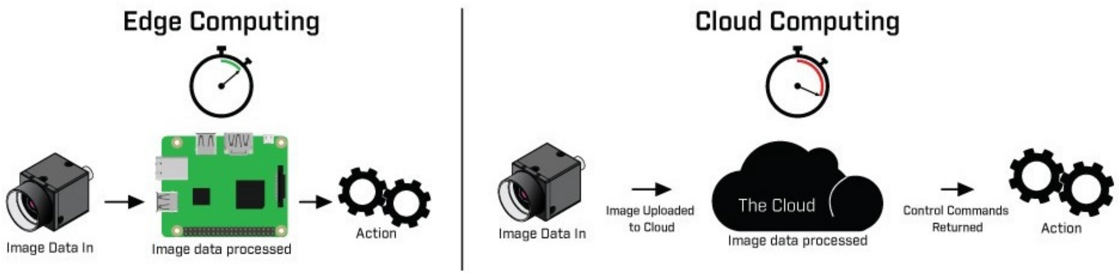

Figure 2.7. Simple comparison of the Edge and Cloud computing. (Source: [23])

2.4.2 Data Distribution

Depending on our requirements, we may use two distinct approaches for the concept of data distribution [24]. First, there are P2P[1] utilities capable of high throughput and low latency but are more challenging to set up and are prone to errors. If either side of the delivery process is disrupted, the whole pipeline might fail, and data might be lost. Second, broker-based data distribution systems are rising in popularity, mainly due to their reliability. Although we can see substantial performance degradation, the ability to configure our data paths to a much larger extent is often welcome. Also, usually brokered systems come with the ability of data retention in the case of operation disruptions. These systems can also have the ability to be hosted across multiple servers or clouds to provide high resiliency from service disruptions.

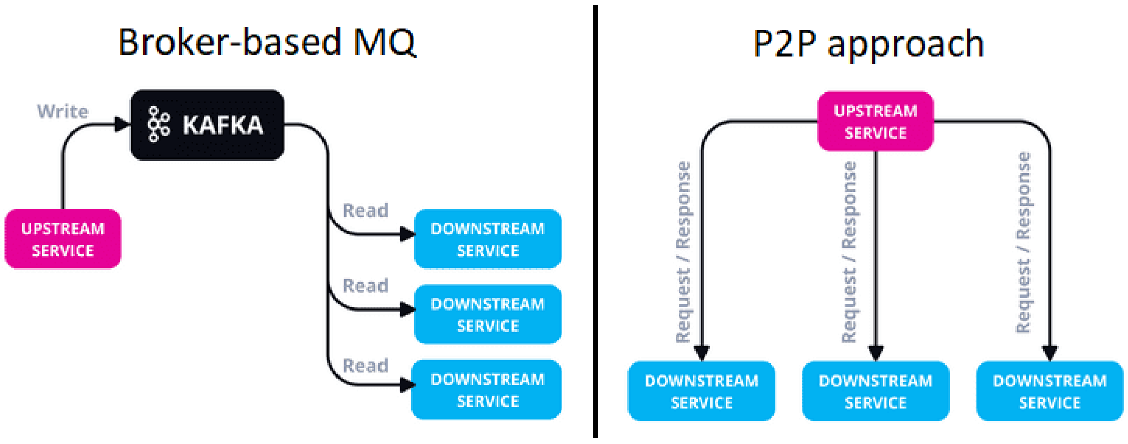

Figure 2.8. Simple comparison of the Broker-based and P2P communication. (Source: [25])

To allow for better scalability and stability, we would prefer to unify the data distribution layer under brokered solution. Together with the Adapter module, it creates universal data hub for even proprietary software and communication protocols. This allows for the connection of legacy L1 + SigProc systems to modern brokered data distribution environments (such as: Kafka, Flume, Kinesis, MQTT, etc.). The primary function of this layer is to distribute and route data where needed. The presence of a QoS[2] equivalent is welcome. This would allow for flagging and executing selected jobs as high-priority. The platform needs to be highly universal, so we would be able to

[1] Point-to-Point
[2] Quality of Service

swap connected tools easily when required. This is the reason why we focus primarily on industry-standard tools.

Even though these systems are usually quite well optimised, there are still concerns about whether brokered solutions are usable for high throughput scenarios. The historical batch data processing use cases sometimes require GB or even TB of data loaded. Brokered tools applicable for our data distribution requirements, such as Apache Kafka, may have problems with sudden loads like these. This incentivizes the use of hybrid solutions.

To prevent congestion, on the path DB→Processing, native database connectors are preferable. Uber Engineering tested this approach and showcased remarkable performance improvement within their experiment [20]. But the compute layer must support this approach. If not, then for compatibility reasons, we also depicted the less preferable variant in Fig 2.3.

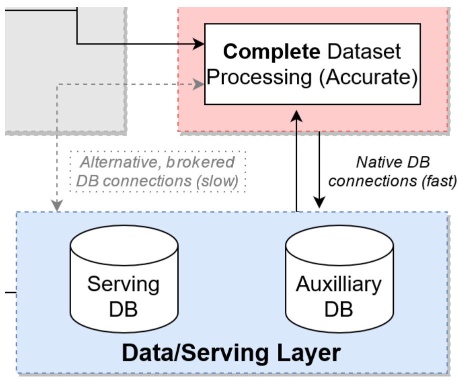
Complete Dataset Processing (Accurate)
Alternative, brokered DB connections (slow)
Native DB connections (fast)
Serving DB
Auxilliary DB
Data/Serving Layer

Figure 2.9. Potential congestion point. (Source: own data)

On the path of L1→Processing, the expected data throughput required for streaming raw data is in the neighbourhood of 1.8MB/s per stream. This may also be quite troublesome for brokered systems. Offloading to P2P connections might provide attractive solution to such workloads. This could be realised by the adapter instead of streaming the data into brokered system, streaming the data directly to processing layer. This approach could be more difficult to set up, but can significantly improve the performance for high throughput scenarios.

In the future, after validation of this approach in production, it is desirable to move data sources closer to the data distribution layer. This can be achieved by integration of Adapter into SigProc software and maybe integrating SigProc functions directly into L1 devices. This way the data would have shorter path to processing layer.

2.4.3 Data Processing

As Lambda-type architectures utilise separate batch and stream processing layer, with Kappa architecture we can emulate batch processing within stream processing. Batch processing usually loads data in large chunks and performs desired operations all at once. This approach leads to usually highly accurate results. With modern tools we can be emulate batch processing as bounded data stream processing in larger chunks, which will lead to similar or even identical results. Its behaviour also becomes more predictable. On the contrary, stream processing does not have definite result and produces results gradually. The accuracy is traded for speed upon which we arrive to results. The usual usage is in conjunction with real-time analytics or, for example, continuous machine learning. The data flows in small chunks and can be considered unbounded (does not have defined end). Support for this approach is essential for the use-cases of signal processing. These layers also should be optimised for high parallelisation. The division of Kappa stream processing layer into two distinct approaches is defined in [17] as ”Accuracy: Only when required.” approach.

The compute resources are structured into clusters, upon which workload can be efficiently distributed. This approach is in accordance to flexible cloud infrastructure. The ability to allocate resources as needed is crucial for maintaining the ”efficient use of resources”. Batch jobs generally require larger allocations of resources over short period of time and stream jobs smaller allocations for longer periods. Each job is defined usually as Python script defining where to get data and what to do with it.

2.4.4 Database

Raw data, results, models need to be stored and handled somewhere. This is the job for data layer. As the data layer is usually the bottleneck in big data processing architectures, we would like to diversify. The idea is to utilise various database technologies, each for its optimised purpose. Let’s say for raw sensor data storage our requirements for database are quite different than storing large chunks of static data. For example, for our data collection purposes we would like to use NoSQL columnar time-series databases, optimised even for larger quantities of timestamped streaming data. But let us say, for big data storage, we would like to use disk-optimised clustered database solution.

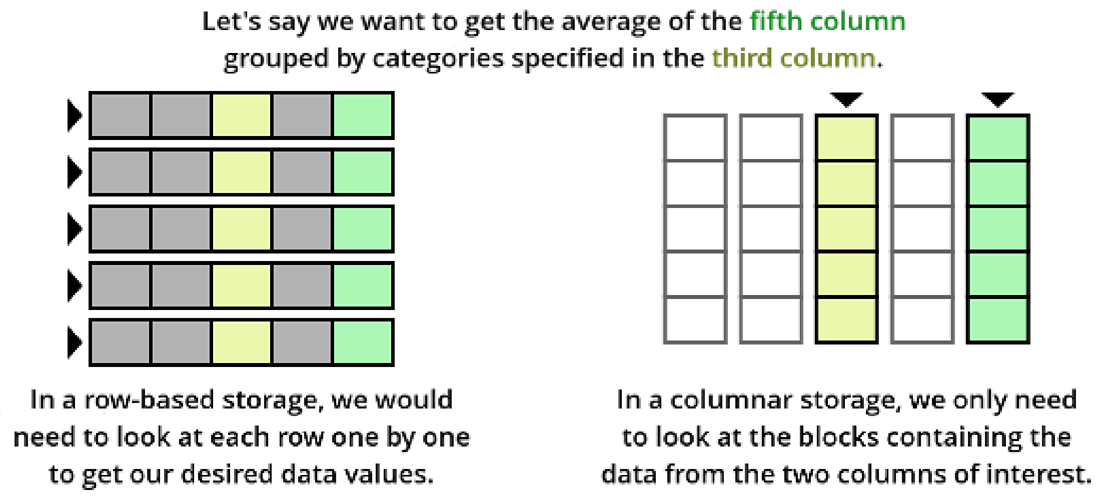
Let's say we want to get the average of the fifth column grouped by categories specified in the third column.
In a row-based storage, we would need to look at each row one by one to get our desired data values.
In a columnar storage, we only need to look at the blocks containing the data from the two columns of interest.

Figure 2.10. Comparison between row-based relational databases (such as PostgreSQL), and column-based databases (such as Cassandra). (Source: [26])

The general trend [27] is to gradually phase out do-it-all databases if we can be certain of our use cases . Within this thesis we would like to mainly focus on the specialised streaming data destination databases. General purpose databases are well documented and we feel that the knowledge about time-series specialised databases would be more beneficial.

2.4.5 Analytics

Last distinct layer is Analytics layer. This is the place, where data scientists spend most of their time. Data visualisation is important part of any data-centric operation. It is especially crucial, when you need to showcase your results to investors or potential clients. From available data, you should be able to improve your machine learning models, reaction responses, or find new use cases for your technologies. In this case we would like to include also feedback connection to lower level devices, so we can for example sound alarms or provide on-site real-time advanced analytics. As these alarm and feedback applications are in this case quite custom-made, our goal is to prepare suitable environment allowing for straightforward development.

2.5 Tool Selection

For the purposes of concrete approach definition and tool selection we would like to narrow down the areas to focus on. These areas correspond with layers as in the general to-be architecture proposal.

- Data distribution
- Processing engine
- Database

On top of the main tool selection, we can extend the architecture's functionality even further using additional tools. Additions are supposed to be swapped easily, as requirements may change. These tools can be utilised for various purposes, such as:

- Process automation
- Visualisation
- Additional Databases

Deployment environment solves the problem of where to run our code and in which way the hardware resources would be accessible for our system. As we are exploring both possibilities of deployment on private and public cloud, we need to account for the differences of such approaches. For better portability, we can consider also packaging our application into containers with a platform such as Kubernetes.

2.5.1 Data Distribution

2.5.1.1 P2P Solutions

gRPC

The gRPC is a modern RPC[1] tool with bi-directional streaming support. gRPC is also an open-source platform backed by Google and is rapidly gaining popularity in the industry. It is optimised to offer high performance and low latency transport. Using HTTP/2 and Protocol buffers serialiser makes it much faster and efficient than JSON or XML based RPCs. Though the benefits come at decreased compatibility, which would not be a problem for our purposes.

Solution based on gRPC would be among the fastest, but it would come at a cost of point-to-point only connections. The reason is that gRPC does not use middleware to broker messages. Only two nodes can communicate with each other in a traditional client-server set-up. By using HTTP/2 gRPC is not restricted to request-response synchronous communication, but can offer continuous asynchronous communication (streaming). Eventually it would require a little bit more work to set-up than other tools, but the additional performance may be worth it. Especially considering high throughput required for raw sensor data ingestion. gRPC uses Protocol buffers for data

[1] Remote Procedure Call

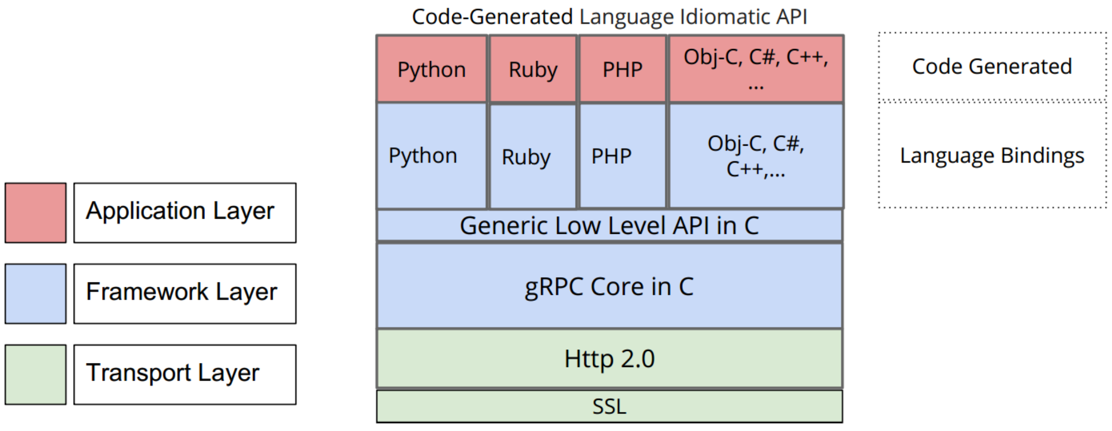

Figure 2.11. Overview of gRPC stack. (Source: [28])

serialisation, which is proven [29] to be more efficient with respect to data throughput and lower resource usage in contrast to JSON [30]. The main downside is fixed schema, which needs to be maintained and synchronised across applications. The more difficult maintainability should not be a problem for our purposes of sensor data ingestion.

ZeroMQ

Another great pick for data distribution might be ZeroMQ library, an open-source message queue API. The name represents message queue (MQ) and brokerless design (Zero). This means it can be described as a RPC enhanced with MQ capabilities. It also offers wide variety of message schemas and transport capabilities. This is possible mainly because of its low-level nature. The ZeroMQ is schema agnostic, it accepts any binary data. The serialisation must be handled independently and on top of that you need to partition the message according to maximum ZeroMQ message size. Alternatively, you can send multipart[1] messages, but the total size must fit into system memory, otherwise you need to split it into chunks.

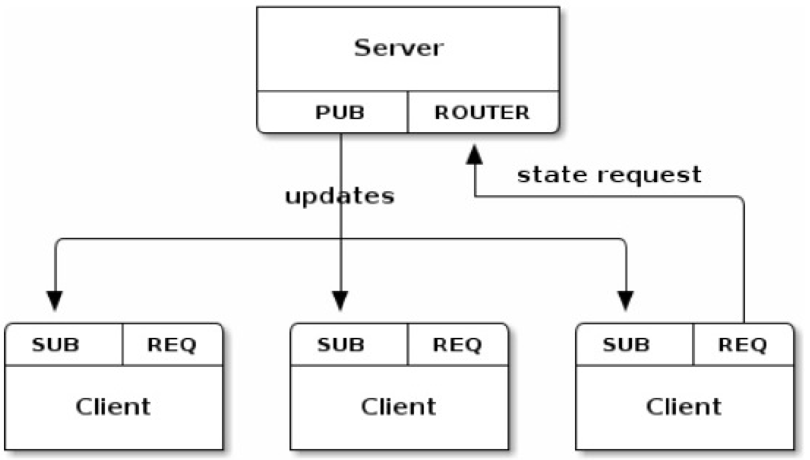

Figure 2.12. Typical usage of ZeroMQ library. (Source: [31])

ZeroMQ offers high-performance, low-latency communication with MQ capabilities, but with high implementation cost. The high implementation cost can be explained, for

[1] Multiple messages sent as a single group.

example, by our need for streaming support. ZeroMQ does not implicitly define it, so all the data partitioning and preparing would be in our hands. This may be actually an advantage for some specific projects, but for us it is probably just a hassle. The lack of any guarantee of message delivery should not be that much of a trouble for our purposes, but it may be nice-to-have. The redeeming property of ZeroMQ is its customisability, it would add much work on our side, but it could be crafted exactly according to our needs. ZeroMQ also offers flexibility of using it in broker mode, although it is not simple to setup as in Apache Kafka.

Strengths of its brokerless mode against broker-based RabbitMQ are detailed in [32].

2.5.1.2 Brokered Solutions

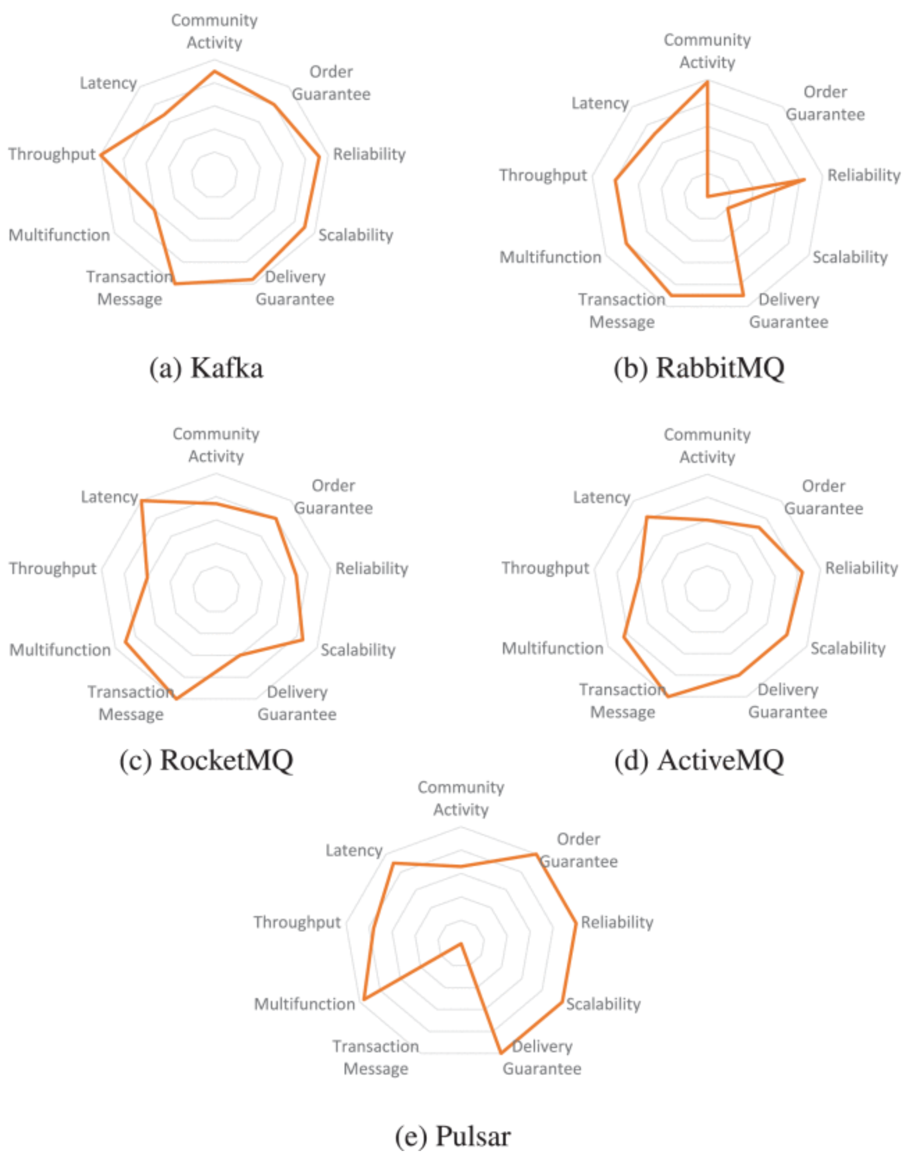

Figure 2.13. Comparison of popular message queuing tools. (Source: [33])

There are many compelling brokered data distribution tools, but selecting few worthy candidates is a daunting task. Thankfully a detailed article [33] comparing popular message queuing tools provides an invaluable insights into inner workings and traits of each tested tool.

Apache Kafka

Kafka is a very popular message broker/stream processing platform. It is open-source and widely adopted for similar use-cases as ours. The seamless compatibility not only with other Apache tools makes it a good choice for most architectural designs. It is proven to be one of the fastest message broker tools [34]. Its wide usage also in the context of machine learning applications is indicative of the advantages it has in contrast to other message brokers. It works rather differently from RPC-based tools. The client-server roles are replaced by publisher-subscriber ones. Publishers send data, subscribers filter and receive data. Kafka requires middleware (hence the broker) to accomplish its goal. This approach allows it to manage data in a reliable way and run separately of other applications. But the downside is added latency and also higher overall complexity of our system. Kafka can use different message schema such as JSON, Protocol buffers, Avro. This enables us to choose the best schema for our purposes, whereas with gRPC we are restricted to Protocol buffers.

Apache Kafka may not be the most modern tool for this job, but it has a large community supporting it and a few performance enhancements, beneficial for our purposes. To explain these benefits we need to first define two fundamental approaches of message handling as defined in [35].

- Shared message queue
- Publish-subscribe model

The main advantage of a shared message queue is its easy scalability. The idea is that one producer delivers through the message queue a message to one consumer, which redistributes it further. The message is deleted right after consuming it from the message queue. Thus if we would like to multicast or broadcast our message streams, we would need to use multiple instances of this type of queue. The scalability is not affected, however. This approach is better suited for sending commands to consumers as we can address each consumer individually.

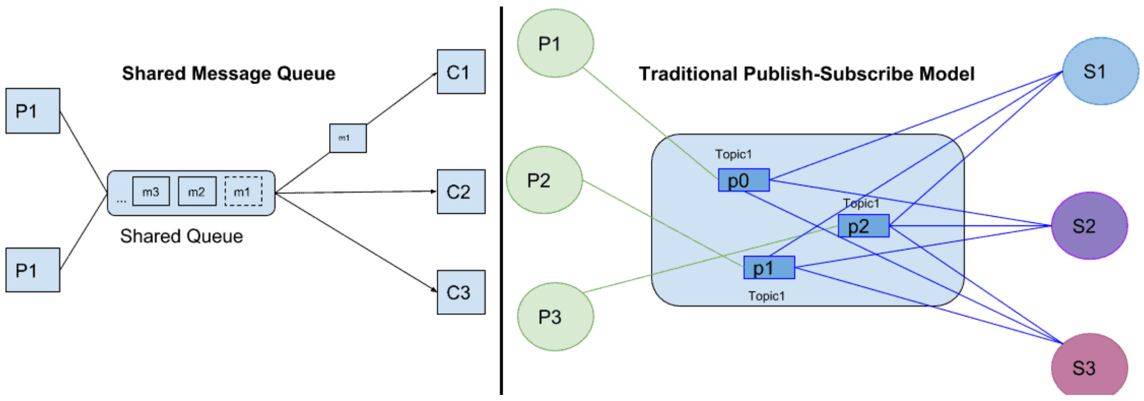

Figure 2.14. Comparison of shared MQ / Publish-Subscribe. (Source: [35])

The publish-subscribe model allows multiple consumers to consume messages. Topics are created as needed, each providing a ”channel”, where you are able to subscribe or publish updates. Scalability is limited, mainly because each subscriber needs to subscribe to each instance of the topic (so called partitions). Within small scale operation, this would not be a big problem. But as we also focus on the system's scalability, we cannot overlook the possibility of medium-large scale deployment.

In their pure form, these approaches are not usually implemented in real-world. Apache Kafka offers a solution to these deficiencies of a publish-subscribe model by creating "consumer groups" and utilising message retention by brokers. When consumers join a group and subscribe to a topic, only one consumer within group consumes the message from the broker. Then the message gets distributed within the group. This approach lowers the count of subscriptions to topics by consumers dramatically. Furthermore, Kafka prefers fair distribution of subscription by consumers within a group. Each consumer should have the same amount of different topic subscriptions if possible. Flexible data distribution without a significant performance penalty is then possible. Indeed, it requires more configuration but the advantages outweigh the disadvantages. These definitions and claims are supported by [35]. For our purposes, benefits of consumer groups would be well utilised. If we would like to run multiple compute pipelines on the same stream of data, our data distribution performance would not suffer much, if we bond these pipelines within consumer groups.

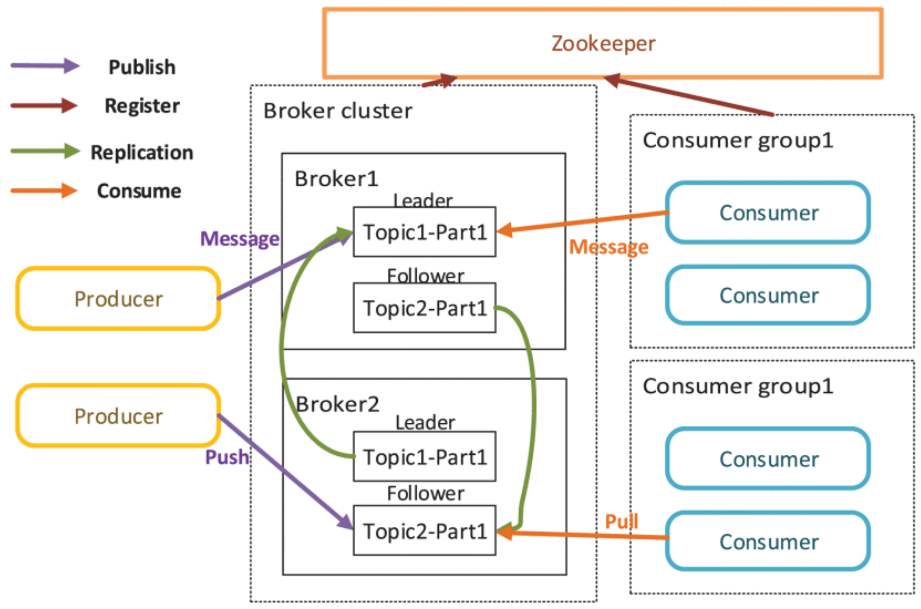

Figure 2.15. Diagram of Apache Kafka architecture and consumer grouping. (Source: [33])

Furthermore, Kafka outperforms most broker message queue systems in pure throughput metrics [36]. This fact indicates its usability also for potential raw data distribution. Its deficiencies are visible in the latency metrics, which may hamper our ability to provide low latency responses. For these kinds of purposes, different methods of data distribution may be preferable. Faster alternative to Kafka would be for example broker-less message queue ZeroMQ. It could also be used in conjunction with Kafka. As there might be throughput intensive workload, which would unnecessarily strain Kafka broker. Workloads like these could be offloaded to a faster alternative, even at the cost of more difficult manageability and loss of broker advantages.

Amazon Kinesis

With AWS[1] suite Amazon offers its own proprietary data ingestion service with native support for streaming and real-time data analysis. The proprietary Amazon-maintained nature of it offers high-stability, easy set-up and high-performance platform. The downside is that you *must* use AWS for entirety of your operation. You are therefore locked in a one vendor ecosystem which has its own advantages and disadvantages. The upside is that you would not need to troubleshoot the system on your own and you

[1] Amazon Web Services

would pay just for the services you use (amount of data transferred). Choice of this tool would largely depend on the willingness to use AWS in (almost) full stack of your operation.

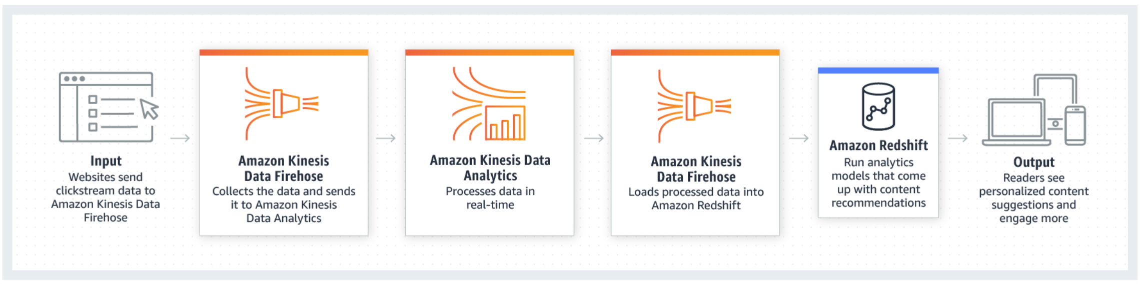
Input
Websites send clickstream data to Amazon Kinesis Data Firehose
Amazon Kinesis Data Firehose
Collects the data and sends it to Amazon Kinesis Data Analytics
Amazon Kinesis Data Analytics
Processes data in real-time
Amazon Kinesis Data Firehose
Loads processed data into Amazon Redshift
Amazon Redshift
Run analytics models that come up with content recommendations
Output
Readers see personalized content suggestions and engage more

Figure 2.16. Overview of data ingestion and processing with Amazon Kinesis pipeline. (Source: [37])

Apache Pulsar

Apache Pulsar, relatively fresh addition to the Apache ecosystem, aims to create a highly scalable message queue system. Although architecturally different from Apache Kafka, it offers similar capabilities. The main difference from Apache Kafka is that data distribution and data storage layers are separated. The data distribution is still handled by so-called broker, but the data storage is delegated to so-called bookkeeper.

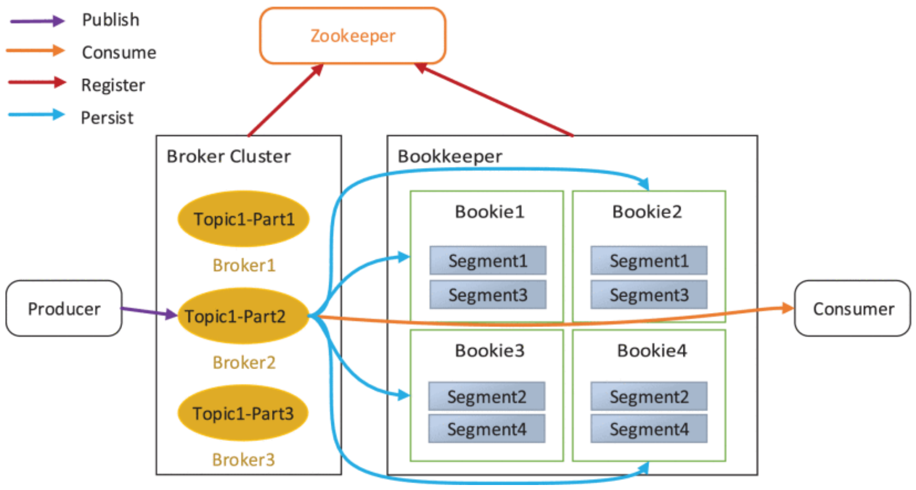

Figure 2.17. Diagram of Apache Pulsar architecture. In contrast to Kafka, the messages are pushed to consumers. (Source: [33])

This approach, although introduces more complexity, Is much more robust in terms of scalability, efficiency and performance. The creation of topics and message distribution is not dependent on data storage within topics. This approach allows us to scale up the topic count independently on the storage subsystem and vice-versa. Pulsar also supports multiple QoS guarantees, similar to Kafka but offering extended support for delayed queuing and priority queuing. Even though it is a relatively young platform, it has gained significant popularity among open source community.

Even through the numerous advantages of this architectural approach are significant, there are also some drawbacks. First, the separation of data distribution and data storage introduces additional latency and network traffic while reading or writing data into the data storage system. Due to this limitation it might not be wise to use this approach for every use case. Second we might run into some compatibility problems due to the immaturity of this system, Kafka has much larger industry support. Third, the increased complexity of this approach objectively complicates deployment of such system.

Arguments and claims concerning Apache Pulsar are supported by [33].

2.5.2 Processing Engine

As we already researched multiple data distribution methods, now we need to think about how the data will be processed. With requirements for high scalability, our compute platform also needs to be robust and scalable. As we discussed in the scalability definition above, we cannot only rely purely on vertical scaling. Then, with horizontal scalability, there are a few obstacles we need to address. The first is how well will we be able to parallelise our workload. Second, with parallelisation comes higher programming difficulty. Thus, the pipeline and job creation tools should be reasonably well abstracted from low-level operations. A significant benefit would also be easy interoperability and compatibility with tools we select. As the computer engine connects to data distribution and serving layers, it is crucial to look out for any possible bottlenecks within the compute engine.

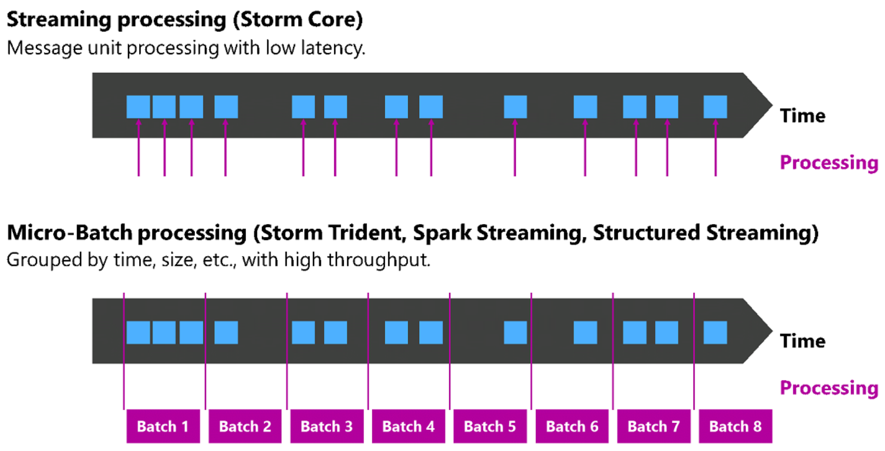
Streaming processing (Storm Core)
Message unit processing with low latency.
Time
Processing
Micro-Batch processing (Storm Trident, Spark Streaming, Structured Streaming)
Grouped by time, size, etc., with high throughput.
Time
Processing
Batch 1
Batch 2
Batch 3
Batch 4
Batch 5
Batch 6
Batch 7
Batch 8

Figure 2.18. Simple comparison of Native/Microbatch stream processing. (Source: [38])

Historically stream processing started as an extension to DBMS[1] systems as an execution environment for standard database operations (joins, aggregations, filtering, grouping). Later, with increasing data volume, these tools secluded and evolved capabilities for parallel processing. Usually these tools adopted unified single-pass data processing. Within this thesis we call this approach "Native Stream Processing". The other, different processing model, which evolved alongside Native Stream Processing is Micro-Batch Processing. As we would like look and compare these tools objectively, we selected few relevant and processing tools for further inspection.

2.5.2.1 Native Stream Processing

The tools within this group share common processing model, but their inner working may differ significantly. The processing is generally designed as data-flow graph with nodes serving as transforming operations or general processing elements. This meant

[1] DataBase Management System

that input stream is forwarded to processing nodes, which perform various operations on the data stream, leading to either transforming elements within the stream or creating a new stream. The scalability might be limited, because these continuous stream operations might not scale well on multiple threads (or across cluster). But this approach allows better predictability of delays, which may occur during processing (also latency).

Apache Storm / Twitter Heron

Apache Storm and Twitter Heron (incubating Apache) are popular stream processing tools based on almost equivalent approach. Apache Storm was introduced in 2011 and later was acquired by Twitter, upon which it was open-sourced and licensed under Apache license. As Twitter grew, the requirements for faster stream processing did as well. In 2015 they introduced Heron. With the experience from the Storm, they strived to improve upon the time proven stream processing tool. Heron improves on many deficiencies of Apache Storm architecture. It should be faster, better scalable and more efficient overall. Twitter detailed the deficiencies of Storm architecture and described improvements made in the Heron architecture within this paper [39]. The API remained compatible with Apache storm applications, thus it might be an attractive replacement for Storm-based systems.

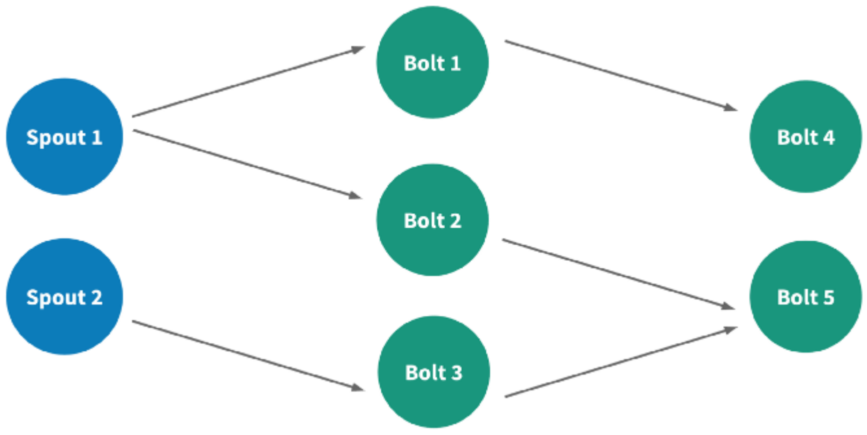

Figure 2.19. Diagram of Apache Heron/Storm acyclic graph structure of Spouts and Bolts. (Source: [40])

Recently there have been some voices, advising to not jump the ship to Twitter Heron just yet [41]. On top of that, the popularity and usage of Heron compared to Storm is not great.

Storm contains two types of elements which are connected in uni-directional graph. Data sources are called Spouts and data manipulators are called Bolts. Data is processed by Tuples, as soon as they arrive on Spouts. This approach is preserved in Heron. The dataflow can be optimised by configuration for either latency or throughput. There is also wide support for API languages (Java, Python, C++, Scala) and numerous data sink connectors.

Apache Flink

Apache Flink has many similarities to Apache Storm. Within both, Flink and Storm, the dataflow is represented as directed graph. Each vertex is intended for data manipulation and each edge represents a flow of data. As Flink is based on Java, instead of using Spouts and Bolts (custom functions), its functions and data structures are

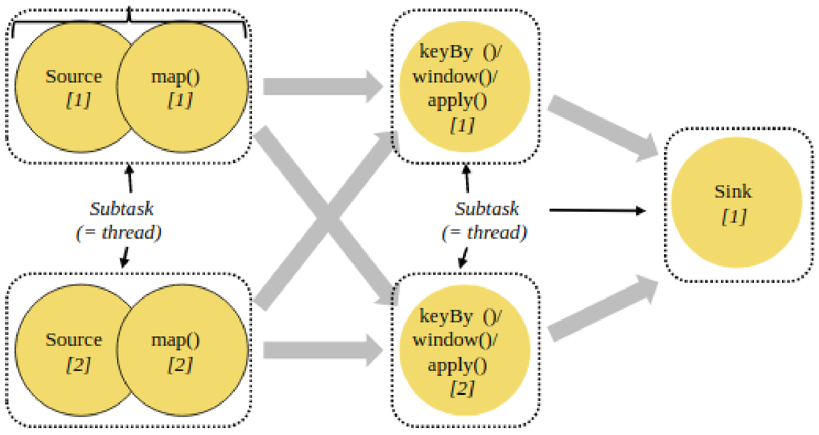

Figure 2.20. Diagram of Apache Flink acyclic graph structure. (Source: [42])

defined natively within Java (map, flatMap, filter, project, reduce). This approach does not restrict us to only embedded functions, but it also allows us to write custom operations in a standardised manner.

The Flink architecture is divided into 2 modules in a master-slave relationship. The master module is called Job Manager, which controls jobs, task scheduling and coordination with slave nodes. The slave module is called Task Manager, these modules have configurable number of task slots to allow for defined maximum number of running tasks on each node. Flink also features wide connection capabilities either with data sources and sink connectors (Kafka, Cassandra, Elasticsearch, HBase, Hive, Kinesis, etc.).

2.5.2.2 Micro-Batch Execution

Even as there is a quite simple possibility of emulating streaming processing capabilities on top of batch processing framework, there are not many success stories among tools utilising this principle. Utilising so-called micro-batching at core of stream processing emulation means that stream of data is packaged into chunks of data, which are then distributed among workers and processed similarly as a large batch of data. This approach still allows for virtually all stream processing use cases, but the latency and unpredictability in delays may pose a problem. The Micro-Batch execution usually scales better, as data can be distributed across multiple workers easily. Batch processing tools also pre-date stream processing and historically were much more popular due to their relative simplicity. As different processing approaches evolved, we could see the rise of machine learning and operational analytics in big data systems. Stream processing became highly desired model [21]. One distinct tool made its name among users by providing flexible distributed batch data processing environment and built its streaming framework on top of it: Apache Spark.

Apache Spark

Apache Spark is an open-source tool with a large community supporting it. First released in 2014, it has undergone significant changes since then. As one of the first big data analytics frameworks, it is known as one of the best traditional batch data processing tool. Deficiencies in streaming performance lead to an emergence of multiple diverging platforms, even within the Apache ecosystem, such as Apache Flink or Apache Storm. But relatively recent updates reworked the streaming system, optimising it for low latency processing. This way, Apache Spark became relevant also in the streaming processing area. With the updates, Spark became the go-to multipurpose analytical-compute framework.

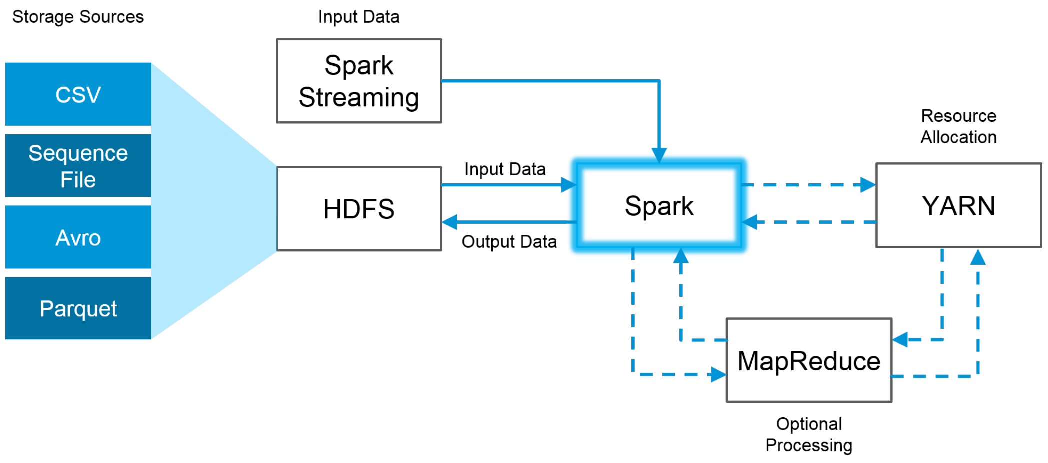

Figure 2.21. Typical usage and positioning of Apache Spark within architecture. (Source: [43])

Spark is written in Scala and is available together with multi-language APIs (Java, Scala, R, Python, SQL, C#, F#). The well supported Python API is also quite beneficial for our requirements. Our goal to utilise machine learning is also well supported within Apache Spark. The performance shouldn't also be a problem due to Apache Sparks in-memory processing capabilities and a large scale of configuration options. These information are readily available in Spark documentation [44] and in a wide variety of different real-world and scientific use cases.

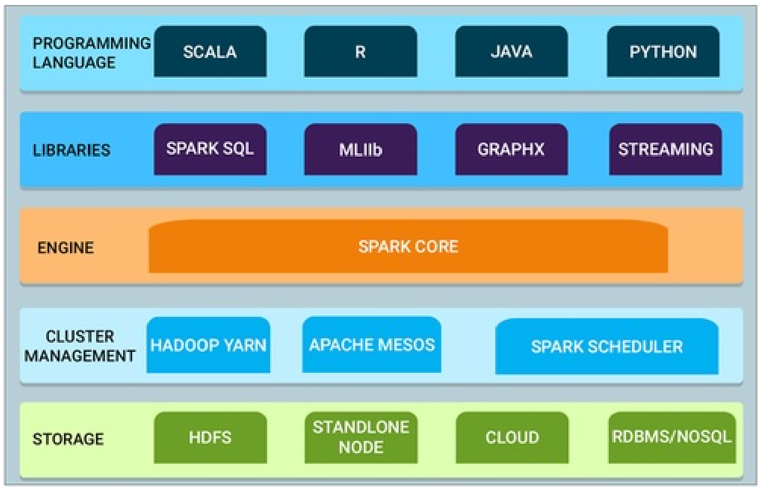

Figure 2.22. High-level view of Apache Spark components and capabilities. (Source: [45])

Apache spark is perfectly capable of running in the cloud, and many public cloud providers also offer it as a default compute platform (e.g. Cloudera [46]). For cluster resource management, we can utilise YARN or Apache Mesos. These resources (CPU cores, memory, storage space) are assigned as needed to each compute job. The tasks defined inside a job are usually run in parallel, spanning multiple threads. There are also three different ways, how to organise resource allocation with Spark. [47]

- Default: The job is executed on all of the resources. That means the whole cluster can only execute one job at a time.
- Static: The user needs to allocate the resources for each job manually. This way, it is possible to run multiple jobs on a single cluster.
- Dynamic: Instead of allocating a fixed amount of resources to each job, we can define the expected amount of resources the job would take. If any of these resources (CPU cores, chunk of memory) are not utilised by the job, they can be released and allocated to a different application. These resources can also be claimed back as needed.

Overall, Spark is a well-rounded compute engine and shouldn't be overlooked by anyone whose use cases are similar.

2.5.2.3 Amazon EMR

Public cloud platforms gained much popularity over recent years. Amazon as one of the leading public cloud infrastructure providers also offers the space and processing power for tools we mentioned above. Using Amazon EMR [48] service companies are able to run petabyte scale analytics without the need to own such large infrastructure. These services are billed on per second And are ideal for testing purposes or for non-repeating jobs. Amazon EMR is just an example of service, which variants can be found among multiple [49] [46] cloud service providers.

2.5.3 Data Storage

Per our use cases, our system needs to be able to handle large amounts of streaming timestamped data. Due to specific nature of such data-flow, there has been a push for specialised data stores suitable for such use cases. The requirement to allow fast decision-making based on a real-time data is something traditional relational transaction-based data stores struggle with. These claims are supported by [50]. Similarly we can not overlook the query performance optimisations of using columnar stores as depicted in Figure 2.10.

Apache Druid

Working with large event data sets or high throughput event streams puts significant strain on our data storage subsystem. This strain is the reason why we should prefer specialised data stores, optimal for our purposes. One of these tools is Apache Druid, an open-source, column-based, distributed data store based on Java. But why Druid is not called a 'database'? Although it can store data persistently, its primary use cases require mainly fast query operations over timestamped data. The temporal configuration capabilities are the centre-point of this tool, as you can, for example, set up Druid to release data after a particular time, either into permanent storage or dump. For our purposes, temporal query capabilities and quick data manipulation are the primary reasons we pre-selected this tool.

Other key-value stores such as Cassandra or HBase are not quite optimised for similar larger-scale operations. They usually need to load relevant data into separate space before starting any query operations on the data itself. This approach may introduce severe bottlenecks. Similarly, SQL-on-Hadoop systems, although highly flexible, compromise performance for flexibility. Druid was created to solve these inefficiencies of run-in big data storage systems.

Druid offers unique tiering functionality among data store tools we discuss within this analysis. This means, 'colder', older, less-used data could be automatically offloaded

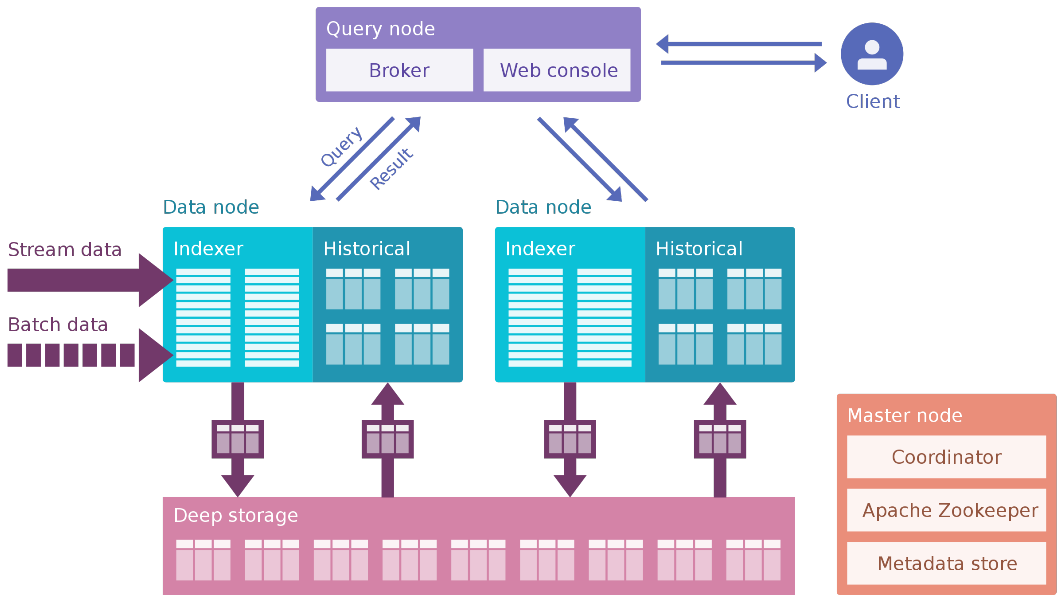

Figure 2.23. Overview of separate Apache Druid components and its architecture. (Source: [51])

to slower type of storage. This may preserve high-performance storage space (memory, NVMe) for 'hot' data. [52]

This video [53] perfectly summarises and compares Druid and other alternatives. And these case studies [54] [55] provides further insights into why and when should we choose Druid.

ClickHouse

ClickHouse is a relative newcomer into the field of OLAP[1]. Its development originates from Russian company Yandex. The main advantage of using ClickHouse instead of the tools explained above is its generally faster engine built on C++ instead of Java. But as explained in this comparison [52], ClickHouse might not be the best tool for our purposes. This concern stems from the way of how it scales.

It might be superior tool for smaller-scale deployments, due to its efficiency, but for large-scale deployments companies should prefer alternatives from Apache. There is a possibility though, if your company might have skilled C++ developers, but not much Java developers, you should be able to customise ClickHouse for your purposes. As Yandex also further customises ClickHouse's architecture, it could be sub-segmented into isolated groups and this way perform similarly or even faster than Java-based Apache tools.

2.5.4 Process Automation

Apache Airflow

As industrial revolution linked better production efficiency to higher production process automation, similar story can be seen in the IT industry. As computer technology evolved, the tools became increasingly complicated and difficult to regularly interact with. No company wanted to hire workers to just operate the information systems, as

[1] OnLine Analytical Processing

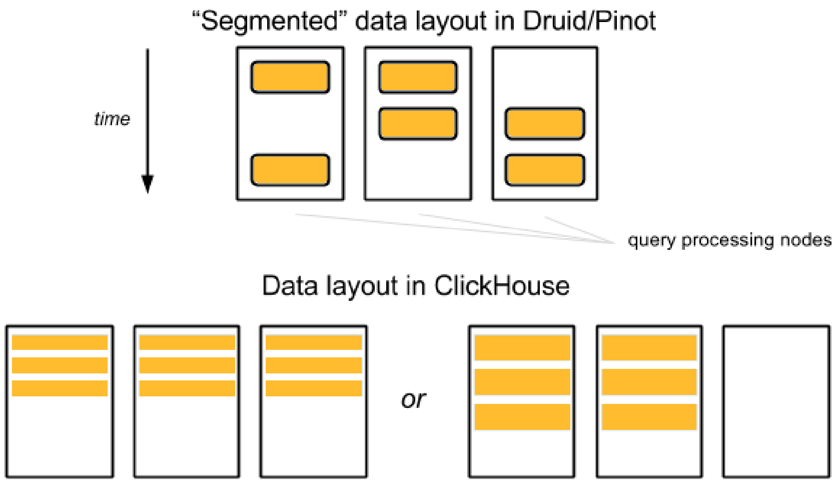
"Segmented" data layout in Druid/Pinot
time
query processing nodes
Data layout in ClickHouse
or

Figure 2.24. Different ways of handling scalability. (Source: [52])

knowledgeable employees are expensive. This pushed for development of multitude of automation systems and frameworks. Initially these tasks were handled on operating system level, such as 'cron jobs'. These days these functions are usually separated from the operating system, due to better maintainability, compatibility and resiliency.

Apache Airflow, an open-source automation framework offers extended automation functionality based on graph schemas. It offers user-friendly interface and plethora of integrations with various tools and systems. The automation jobs are organised as DAG[1].

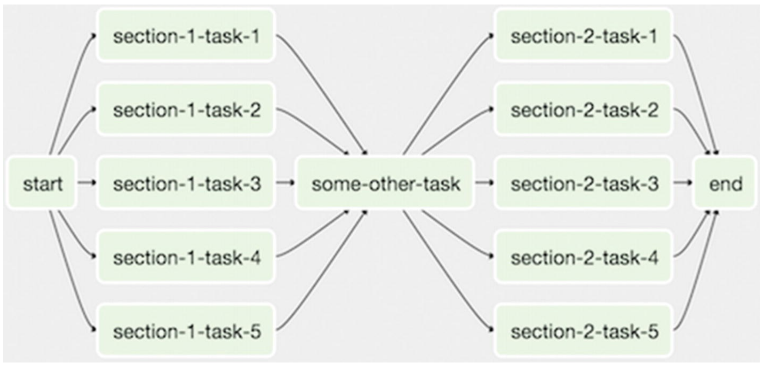

Figure 2.25. Airflow service as DAG. (Source: [56])

2.6 Analysis Conclusion

As we can see, there is a wide variety of tools available, and the open-source community is growing at an unprecedented rate. This allows us to choose and eventually even customise the tools according to our exact requirements.

We hope this analysis provided you with valuable insight of what we would like to achieve within this thesis. Next, we further analyse the quality of each selected tools on purpose-built measurement system and scale.

[1] Directed Acyclic Graph

Chapter 3
Theoretical Evaluation of Selected Tools

Within this chapter we focus on developing and compiling benchmarking methods to objectively evaluate suitability of selected tools from chapter 2. Then, we comment, evaluate and rank each tool according to established metrics. At last, we select the tools based on our evaluation and design three possible configurations applicable for our purposes. From these three configurations we will then select one for further testing and evaluation in next chapter.

3.1 Metrics and Methodology

As our goal is primarily to find out which tool can better suit our purposes, we decided to split the overall evaluation factors into two groups.

Weight	Evaluation group
3/4	Rate of Requirement Compliance
1/4	Preferential Ranking

Table 3.1. Composite evaluation factors with weights assigned.

- *Rate of Requirement Compliance*, as the name suggests, is dependent exclusively on the compliance with relevant requirements specified in chapter 2.
- *Preferential Ranking*, on the other hand, represents various factors not explicitly specified as requirements, but nonetheless may be important in final ranking. These factors might include nice-to-have features, detailed performance comparisons, popularity rankings or compatibility ratings.

Evaluation of each of these factors is done on the interval $[0, 1]$. But there are two distinct groups of factors, which we need to evaluate differently. Similar rating system is proposed in [57].

- Feature support factors
  - $1 =$ Complete feature support, or with minor restrictions
  - $0,5 =$ The cost of feature implementation is high, or some significant restrictions are present
  - $0 =$ The feature is unsupported
- Performance/popularity metrics
  - $1 =$ Best
  - $0,5 =$ Slight disadvantage
  - $0 =$ Worst

The weight system always equals to sum of 1. At this section 3.2, evaluation overview is provided and each tool has its final result represented with % points. Detailed overview of the evaluation process is located in attachment section.

As each category of tools requires different approach to its evaluation, the evaluation metrics can be divided into respective categories as in chapter 2.

3.2 Tool Evaluation

Our knowledge from the analysis chapter of this thesis enables us to make an informed evaluation of each tool discussed. To prevent confusion, we enumerated the references to sources we used for evaluation.

- Data Distribution
  - [57] [34] [36]
- Data Processing
  - [58] [59] [60]
- Data Storage
  - [52]

3.2.1 Results

Complete detailed rating categories and results can be found within **Appendix A**.

Tool	Requirement rating	Preferential rating	Total
gRPC	87,50%	80,00%	**85,63%**
ZeroMQ	75,00%	66,67%	**72,92%**
Kafka	93,75%	81,58%	**90,71%**
Pulsar	93,75%	57,89%	**84,79%**
Storm	95,00%	52,63%	**84,41%**
Flink	100,00%	92,11%	**98,03%**
Spark	100,00%	71,05%	**92,76%**
Druid	100,00%	82,14%	**95,54%**
ClickHouse	81,25%	46,43%	**72,55%**

Table 3.2. Complete evaluation results.

3.3 System Configurations

3.3.1 First Approach

Within this approach we would like to explore custom solution based on Apache ecosystem. Apache tools are generally open-source and can be used in commercial applications without licensing fees. Many of these tools are utilised by big players in the market, so they usually have large contributor base and tightly follow industry trends.

The main goal is to focus on the post popular and well performing tools.

Category	Selected tool
Data Distribution	Apache Kafka
Data Processing	Apache Spark
Data Storage	Apache Druid

3.3.2 Second Approach

As our requirements can enter easily into the big data territory, focusing on the best scalability and performance might be a valid tactic. We can expect even less popular tools to evolve over time and become as well community-supported as their mature counterparts.

Category	Selected tool
Data Distribution	gRPC
Data Processing	Apache Flink
Data Storage	Apache Druid

3.3.3 Third Approach

Sometimes companies are reluctant to adopt progressive technologies, their preference might be to develop small-scale operation first and if successful, then even rework their infrastructure to accommodate more scalable system. Within this approach our main goal is to consider effectiveness of smaller scale deployment.

Category	Selected tool
Data Distribution	gRPC
Data Processing	Apache Flink
Data Storage	ClickHouse

3.4 Evaluation Conclusion

As we can see, it is not always straightforward to select the right tools for the job. Even we were quite surprised of the plethora of approaches we can consider.

There is also a lesson within this evaluation. We believed, Apache Storm/Heron would get much better rating. Eventually Apache Flink, similarly built tool, surpassed it in every metric. Initially we did not even considered the Flink being a worthy competitor, but our opinions significantly shifted during researching this thesis. It is easy to judge something at first sight, it is difficult to objectively evaluate.

For our purposes we would like to select the first approach, because these most popular tools seemingly provide a middle ground between performance of second approach and simplicity of third.

Chapter 4
Testing

For our purposes we utilise rack-mounted server with following hardware specifications:

Specification	Value
Model	HPE ProLiant DL20 Gen10
System Memory	16GiB ECC
CPU	Intel(R) Pentium(R) Gold G5420
Storage	1TiB HDD ST1000DM010

Due to the difficulty of extending communication protocol capabilities suitable for our purposes, we needed to develop an Adapter program. This Adapter program listens to a port into which the proprietary software SigProc can send streaming data. The Adapter interprets the data and packages it into efficient binary schema. This schema is based on Protocol Buffers (protobuf) which is an open-source schema developed and maintained by Google. It is also the primary encoding schema of gRPC. And is widely supported in the IT industry. These messages are then serialised and forwarded into designated Kafka topic.

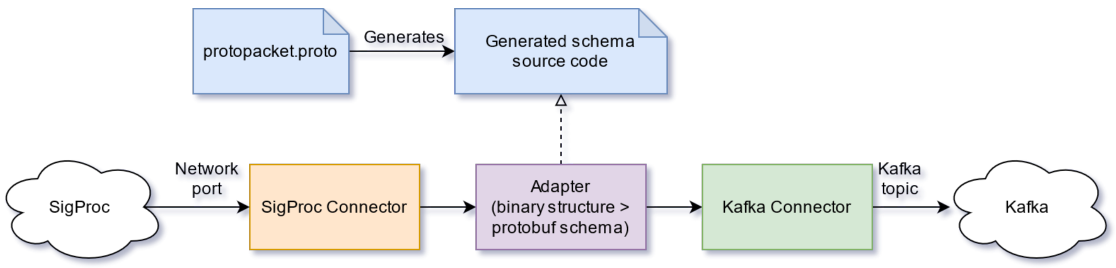

Figure 4.1. Adapter high-level design.

We have available data with various sound anomalies and we would like to utilise machine learning, namely anomaly detection.

This data flows through Kafka into defined Spark Job where micro-batch sampling is taking place and the batches are then fed into machine learning algorithm. This process is called model training. After we determine, either by condition or manually, that we would like the model to stop learning, we can then start evaluating the incoming data. As we are working with rows of data, we add a evaluation variable column, where the model will save its decision. As we work with anomaly detection, the result is binary, anomaly or not. Then the resulting data is streamed back into Kafka, although into different topic. This topic is consumed by Druid data store. Within Druid we may execute queries over incoming data, or we can connect visualisation engine such as Grafana. Grafana would then make queries on our behalf and may for example highlight anomalous readings. But we are free to design our own analytical tools and

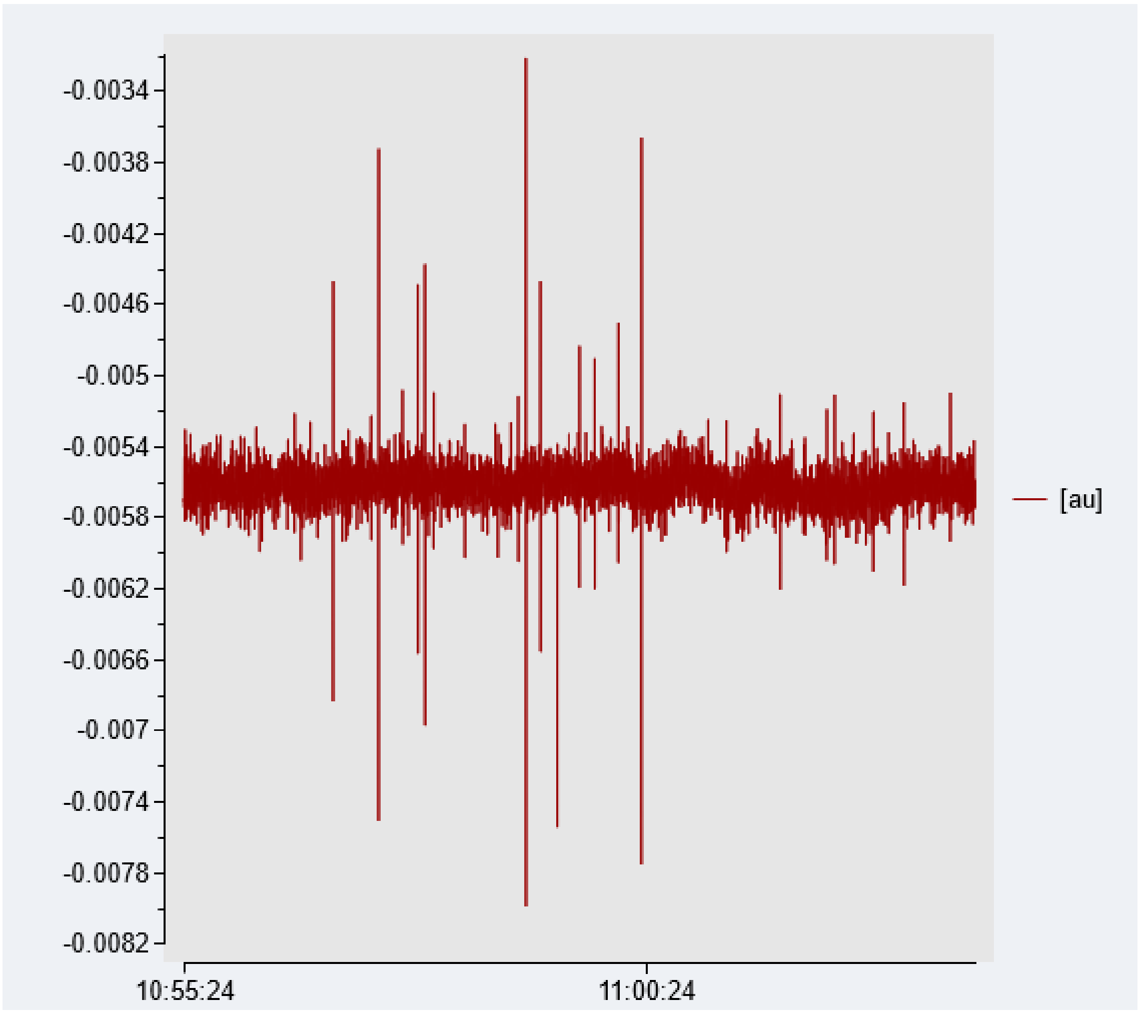

Figure 4.2. Graph of provided data.

for example produce alarms based on the classification results. These alarms would be sent to either SigProc or L1 device, to produce a desired action.

In the future, the models can be serialised and distributed to relevant L1 devices supporting Edge machine learning classification [22].

Chapter 5
Conclusion

We believe our thesis will serve primarily as a guide for building modern stream-oriented systems. Our goal was to gather information about tools suitable for our model use cases. Upon which we may build our system scalable, easier to work with and potentially future-proof.

We defined our goals and key points to focus on in the introduction. Then we elaborated upon the characteristics of the system we tried to build. Then formally defined requirements which we significantly accentuated in the evaluation chapter. We followed the requirement definition with architectural requirements and various standard approaches we deliberated upon. Continuing by introducing the reader to the current state of the model environment, we tried to ameliorate. Upon this state, we proposed an architectural design for consideration. We thoroughly explained each proposed component and their connections. Upon which, we followed with selecting and further analysing prospective tools selected for further evaluation.

We seemingly succeeded at our goal to gather the information and evaluate suitable tools. Upon these evaluations and the analysis, we created more specific model scenarios upon which we selected the most suitable and tested it on a real-world use case.

This topic is vast, and we may try to fully understand it for years before we might be able to say: "We are proficient." That is why we might not be able to develop a fully informed opinion about these systems, because in It world, the development of new technologies moves forward rapidly. If it takes years to develop complete understanding, this knowledge may become obsolete quickly. We hope this thesis shows a way forward and may represent a starting point for everyone seeking to process streaming data.

References

[1] soshace.com [online]. *The Era of Change – IoT and Machine Learning — Trends in Industry for 2020*. https://soshace.com/the-era-of-change-iot-and-machine-learning-trends-in-industry/. 2020. Accessed: 2021-05-11.

[2] bloomberg.com [online]. *Machine Learning Market Size To Reach $96.7 Billion By 2025, Based on Rising Usage of Data Science & AI Technologies For Driving*. https://www.bloomberg.com/press-releases/2021-04-06/machine-learning-market-size-to-reach-96-7-billion-by-2025-based-on-rising-usage-of-data-science-ai-technologies-for-driving. 2021. Accessed: 2021-05-11.

[3] Alexandra L'Heureux, Katarina Grolinger, Hany El Yamany, and Miriam Capretz. Machine Learning With Big Data: Challenges and Approaches. *IEEE Access*. 2017, PP 1-1. DOI 10.1109/ACCESS.2017.2696365.

[4] blogs.oracle.com [online]. *Machine Learning Challenges: What to Know Before Getting Started*. https://blogs.oracle.com/bigdata/post/machine-learning-challenges-what-to-know-before-getting-started. 2018. Accessed: 2021-05-11.

[5] Venturebeat.com [online]. *Why do 87% of data science projects never make it into production?* https://web.archive.org/web/20201201151405/https://venturebeat.com/2019/07/19/why-do-87-of-data-science-projects-never-make-it-into-production/. 2019. Accessed: 2020-01-06.

[6] Technopedia.com [online]. *Definition - What does Technical Debt mean*. https://web.archive.org/web/20201101050026/https://www.techopedia.com/definition/27913/technical-debt/. 2017. Accessed: 2020-01-07.

[7] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. *Hidden Technical Debt in Machine Learning Systems*. In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*. Cambridge, MA, USA: MIT Press, 2015. 2503–2511.

[8] Yuanjun Guo, Zhile Yang, Shengzhong Feng, and Jinxing Hu. Complex Power System Status Monitoring and Evaluation Using Big Data Platform and Machine Learning Algorithms: A Review and a Case Study. *Complexity*. 2018, 2018 8496187. DOI 10.1155/2018/8496187.

[9] Ryan H.L. Ip, Li-Minn Ang, Kah Phooi Seng, J.C. Broster, and J.E. Pratley. Big data and machine learning for crop protection. *Computers and Electronics in Agriculture*. 2018, 151 376-383. DOI https://doi.org/10.1016/j.compag.2018.06.008.

[10] Mohammad Adibuzzaman, Poching DeLaurentis, Jennifer Hill, and Brian D. Benneyworth. Big data in healthcare - the promises, challenges and opportunities from a research perspective: A case study with a model database. *AMIA ... Annual Symposium proceedings. AMIA Symposium*. 2018, 2017 384-392. PMC5977694[pmcid.

[11] Huang Xiangdong Wang Ruotong, Wang Jianmin. *The Architecture Design of MICAPS4 Server System.*
http://html.rhhz.net/yyqxxb/html/20180101.htm. 2018. Accessed: 2021-05-20.

[12] Sören Henning, and Wilhelm Hasselbring. *How to Measure Scalability of Distributed Stream Processing Engines?.* In: New York, NY, USA: Association for Computing Machinery, 2021. 85–88. ISBN 9781450383318.
https://doi.org/10.1145/3447545.3451190.

[13] perfdynamics.com [online]. *How to Quantify Scalability.*
http://www.perfdynamics.com/Manifesto/USLscalability.html. 2007. Accessed: 2021-05-11.

[14] rootstrap.com [online]. *Horizontal vs Vertical Scaling: A Primer for Managers.*
https://www.rootstrap.com/blog/horizontal-vs-vertical-scaling/. 2021. Accessed: 2021-05-11.

[15] Sijin He, Li Guo, Moustafa Ghanem, and Yike Guo. *Improving Resource Utilisation in the Cloud Environment Using Multivariate Probabilistic Models.* In: *2012 IEEE Fifth International Conference on Cloud Computing.* 2012. 574-581.

[16] queue.acm.org [online]. *The Challenge of Cross-language Interoperability.*
https://queue.acm.org/detail.cfm?id=2543971. 2013. Accessed: 2021-05-11.

[17] peerislands.io [online]. *Data Processing – Lambda vs Kappa Architectures and Apache Beam.*
https://web.archive.org/web/20210501144637/https://www.peerislands.io/data-processing-lambda-vs-kappa-architectures-and-apache-beam/. 2021. Accessed: 2021-05-01.

[18] Keh Kok Yong, Mohamad Syazwan Shafei, Pek Yin Sian, and Meng Wei Chua. *Review of Big Data Analytics (BDA) Architecture: Trends and Analysis.* In: *2019 IEEE Conference on Open Systems (ICOS).* 2019. 34-39.

[19] Ignacio Mulas Viela [online] Nicolas Seyvet. *Applying the Kappa architecture in the telco industry.*
https://web.archive.org/web/20210308131450/https://www.oreilly.com/content/applying-the-kappa-architecture-in-the-telco-industry/. 2016. Accessed: 2021-04-13.

[20] eng.uber.com [online]. *Designing a Production-Ready Kappa Architecture for Timely Data Stream Processing.*
https://web.archive.org/web/20210302095928/https://eng.uber.com/kappa-architecture-data-stream-processing/. 2020. Accessed: 2021-05-01.

[21] sigops.org [online]. *Why stream processing systems are now more relevant than ever.*
https://www.sigops.org/2020/streams/. 2020. Accessed: 2021-05-16.

[22] blogs.microsoft.com [online]. *AI's big leap to tiny devices opens world of possibilities.*
https://blogs.microsoft.com/ai/ais-big-leap-tiny-devices-opens-world-possibilities/. 2017. Accessed: 2021-03-09.

[23] kdnuggets.com [online]. *Deep Learning on the Edge.*
https://www.kdnuggets.com/2018/09/deep-learning-edge.html. 2018. Accessed: 2021-03-09.

[24] dev.to [online]. *POINT-TO-POINT AND PUBLISH/SUBSCRIBE MESSAGING MODEL.*

https://dev.to/tranthanhdeveloper/point-to-point-and-publish-subscribe-messaging-model-41j0. 2020. Accessed: 2021-04-16.

[25] up9.com [online]. *What Is Contract Testing.* https://up9.com/what-is-contract-testing. 2021. Accessed: 2021-04-16.

[26] medium.com [online]. *Why do we need NoSQL? Is SQL enough, Isn't it?* https://medium.com/@ibeam.info/why-do-we-need-nosql-is-sql-enough-isnt-it-b6b27cd86af8. 2020. Accessed: 2021-05-16.

[27] Database architectures: Current trends and their relationships to environmental data management. *Environmental Modelling & Software.* 2006, 21 (11), 1579-1586. DOI https://doi.org/10.1016/j.envsoft.2006.05.004. Environmental Informatics.

[28] devopedia.org [online]. *gRPC.* https://web.archive.org/web/20201109043500/https://devopedia.org/grpc. 2020. Accessed: 2021-05-11.

[29] medium.com [online]. *Protocol Buffers vs JSON.* https://medium.com/@Keyology/protocol-buffers-vs-json-a6661a2e7716. 2019. Accessed: 2021-05-11.

[30] Octo.com [online]. *Protocol Buffers: Benchmark and mobile.* https://web.archive.org/web/20170611202233/http://blog.octo.com/en/protocol-buffers-benchmark-and-mobile/. 2016. Accessed: 2020-01-08.

[31] slideshare.net [online]. *Overview of ZeroMQ.* https://www.slideshare.net/pieterh/overview-of-zeromq. 2011. Accessed: 2021-05-11.

[32] Nicolas Estrada, and Hernán Astudillo. *Comparing scalability of message queue system: ZeroMQ vs RabbitMQ.* In: *2015 Latin American Computing Conference (CLEI).* 2015. 1-6.

[33] Guo Fu, Yanfeng Zhang, and Ge Yu. A Fair Comparison of Message Queuing Systems. *IEEE Access.* 2021, 9 421-432. DOI 10.1109/ACCESS.2020.3046503.

[34] Confluent.com [online]. *Benchmarking Apache Kafka, Apache Pulsar, and RabbitMQ: Which is the Fastest?* https://web.archive.org/web/20201128120010/https://www.confluent.io/blog/kafka-fastest-messaging-system/. 2020. Accessed: 2020-01-08.

[35] blog.cloudera.com [online]. *Scalability of Kafka Messaging using Consumer Groups.* https://web.archive.org/web/20210409140157/https://blog.cloudera.com/scalability-of-kafka-messaging-using-consumer-groups/. 2018. Accessed: 2021-05-01.

[36] Guo Fu, Yanfeng Zhang, and Ge Yu. A Fair Comparison of Message Queuing Systems. *IEEE Access.* 2021, 9 421-432. DOI 10.1109/ACCESS.2020.3046503.

[37] aws.amazon.com [online]. *Amazon Kinesis.* https://web.archive.org/web/20201231094400/https://aws.amazon.com/kinesis/. 2020. Accessed: 2021-05-11.

[38] docs.microsoft.com [online]. *Migrate Azure HDInsight 3.6 Apache Storm to HDInsight 4.0 Apache Spark.* https://docs.microsoft.com/en-us/azure/hdinsight/storm/migrate-storm-to-spark/. 2019. Accessed: 2021-05-16.

[39] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth

Taneja. *Twitter Heron: Stream Processing at Scale.* In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data.* New York, NY, USA: Association for Computing Machinery, 2015. 239–250. ISBN 9781450327589. `https://doi.org/10.1145/2723372.2742788`.

[40] medium.com [online]. *Introduction to Apache Heron.* `https://medium.com/@kramasamy/introduction-to-apache-heron-c64f8c7c0956`. 2019. Accessed: 2021-05-11.

[41] roshannaik @ github.com [online]. *Twitter Heron concern.* `https://github.com/apache/storm/pull/2241#issuecomment-317879907`. 2017. Accessed: 2021-05-21.

[42] towardsdatascience.com [online]. *An Introduction to Stream Processing with Apache Flink.* `https://towardsdatascience.com/an-introduction-to-stream-processing-with-apache-flink-b4acfa58f14d`. 2020. Accessed: 2021-05-11.

[43] towardsdatascience.com [online]. *A Beginner's Guide to Apache Spark.* `https://towardsdatascience.com/a-beginners-guide-to-apache-spark-ff301cb4cd92`. 2019. Accessed: 2021-05-13.

[44] spark.apache.org [online]. *Apache Spark official documentation.* `https://spark.apache.org/docs/latest/quick-start.html`. 2021. Accessed: 2021-05-01.

[45] towardsdatascience.com [online]. *A Beginner's Guide to Apache Spark.* `https://web.archive.org/web/20200501214345/https://towardsdatascience.com/a-beginners-guide-to-apache-spark-ff301cb4cd92`. 2019. Accessed: 2021-05-01.

[46] cloudera.com [online]. *Cloudera Data Engineering.* `https://www.cloudera.com/products/data-engineering.html`. 2021. Accessed: 2021-05-08.

[47] Muhammed Tawfiqul Islam, Satish Narayana Srirama, Shanika Karunasekera, and Rajkumar Buyya. Cost-efficient dynamic scheduling of big data applications in apache spark on cloud. *Journal of Systems and Software.* 2020, 162 110515. DOI https://doi.org/10.1016/j.jss.2019.110515.

[48] aws.amazon.com [online]. *Amazon EMR.* `https://aws.amazon.com/emr/`. 2021. Accessed: 2021-05-08.

[49] cloud.google.com [online]. *Google Dataproc.* `https://cloud.google.com/dataproc/`. 2021. Accessed: 2021-05-08.

[50] hazelcast.com [online]. *Time Series Database.* `https://hazelcast.com/glossary/time-series-database/`. 2021. Accessed: 2021-05-11.

[51] wikimedia.org [online]. *Apache Druid Architecture.* `https://upload.wikimedia.org/wikipedia/commons/d/df/Apache_Druid_Architecture.svg`. 2019. Accessed: 2021-05-20.

[52] leventov.medium.com [online]. *Comparison of the Open Source OLAP Systems for Big Data: ClickHouse, Druid, and Pinot.* `https://leventov.medium.com/comparison-of-the-open-source-olap-systems-for-big-data-clickhouse-druid-and-pinot-8e042a5ed1c7`. 2018. Accessed: 2021-05-20.

[53] scalac.io [online]. *Interactive Exploratory Analytics with Druid.* `https://imply.io/videos/interactive-exploratory-analytics-with-druid`. 2017. Accessed: 2021-05-20.

[54] scalac.io [online]. *How We Reduced Costs and Simplified Solution by Using Apache Druid.* `https://scalac.io/blog/how-we-reduced-costs-and-simplified-solution-by-using-apache-druid/`. 2020. Accessed: 2021-05-20.

[55] blog.netsil.com [online]. *A Comparison of Time Series Databases and Netsil's Use of Druid.* `https://blog.netsil.com/a-comparison-of-time-series-databases-and-netsils-use-of-druid-db805d471206`. 2016. Accessed: 2021-05-20.

[56] Pramod Singh. *Airflow.* In: *Learn PySpark: Build Python-based Machine Learning and Deep Learning Models.* Berkeley, CA: Apress, 2019. 67–84. ISBN 978-1-4842-4961-1. `https://doi.org/10.1007/978-1-4842-4961-1_4`.

[57] ALEKSANDER Bondarenko, and KONSTANTIN Zaytsev. Studying systems of open source messaging. *Journal of Theoretical and Applied Information Technology.* 2019, 97 (19), 5115–5125.

[58] Supun Kamburugamuve, and Geoffrey Fox. Survey of distributed stream processing. *Bloomington: Indiana University.* 2016,

[59] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, and Paul Poulosky. *Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming.* In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* 2016. 1789-1792.

[60] Huafeng Wang Wei Mao. *Streaming Report.* `https://files.meetup.com/16395762/Streaming%20Report_Meetup_Intel-Maowei-Wanghuafeng-LegalReviewed.pdf`. 2016. Accessed: 2021-05-13.

Appendix A
Detailed Evaluation

A.1 Data Distribution

Requirement	Weight	gRPC	ZeroMQ	Kafka	Pulsar
FR1 - Streaming support	1/8	1	0	1	1
FR4 - Python support	1/8	1	1	1	1
FR5 - Additional prog. language support	1/8	1	1	1	1
FR6 - Reasonable scalability	1/8	1	1	0,5	1
FR10 - Configurable resiliency	1/8	0,5	0,5	1	1
FR11 - On-demand delivery guarantee	1/8	0,5	0,5	1	1
NFR1 - No special licensing	1/8	1	1	1	1
NFR3 - Cloud deployability	1/8	1	1	1	0,5

Table A.1. Data distribution *requirement* evaluation.

Characteristic	Eval. Method	Weight	Kafka	Pulsar
Scalability	Scaling efficiency	2/19	0,5	1
Efficient res. use	Idempotent messaging	1/19	1	0
Simplicity of Dev.	Ordering guarantee	1/19	0,5	1
	Transactions	1/19	1	0
	Popularity	1/19	1	0
	Delivery guarantee	1/19	1	1
Performance	Throughput	2/19	1	0
	Latency	2/19	1	0
	Persistence	1/19	1	1
	Priority queuing	2/19	0	1
Cloud deployment	Compatibility	2/19	1	0,5
Openness	Open-Source	1/19	1	1
Cost	Free to use	2/19	1	1

Table A.2. Data distribution tools *with broker*, preferential evaluation.

Characteristic	Eval. Method	Weight	gRPC	ZeroMQ
Efficient res. use	Binary schema support	2/15	1	1
Simplicity of Dev.	Ordering guarantee	1/15	1	0
	Popularity	1/15	1	0
Performance	Throughput	3/15	0	1
	Latency	3/15	1	0
Cloud deployment	Compatibility	2/15	1	1
Openness	Open-Source	1/15	1	1
Cost	Free to use	2/15	1	1

Table A.3. Data distribution tools *without broker*, preferential evaluation.

A.2 Data Processing

Requirement	Weight	Storm	Flink	Spark
FR1 - Streaming support	1/10	1	1	1
FR3 - Machine Learning support	1/10	0,5	1	1
FR4 - Python support	1/10	1	1	1
FR5 - Additional programming language support	1/10	1	1	1
FR6 - Reasonable scalability	1/10	1	1	1
FR8 - Automation support	1/10	1	1	1
FR9 - In-Memory processing support	1/10	1	1	1
FR10 - Configurable resiliency	1/10	1	1	1
NFR1 - No special licensing	1/10	1	1	1
NFR3 - Cloud deployability	1/10	1	1	1

Table A.4. Data Processing *requirement* evaluation.

Characteristic	Eval. Method	Weight	Storm	Flink	Spark
Scalability	Scaling capabilities	2/19	0,5	1	1
Efficient res. use	Direct DB connection	2/19	0,5	0,5	1
Simplicity of Dev.	Flow control	1/19	0	1	1
	Popularity	1/19	0	0,5	1
	Delivery guarantee	2/19	0,5	1	1
Performance	Throughput	2/19	0,5	1	0
	Latency	2/19	0,5	1	0
	Del. guarantee perf. impact	1/19	0	1	0
	Time window capability	1/19	0	1	0,5
Cloud deployment	Compatibility	2/19	1	1	1
Openness	Open-Source	1/19	1	1	1
Cost	Free to use	2/19	1	1	1

Table A.5. Data processing tools, preferential evaluation.

A.3 Data Storage

Requirement	Weight	Druid	ClickHouse
FR1 - Streaming support	1/8	1	1
FR4 - Python support	1/8	1	1
FR5 - Additional programming language support	1/8	1	1
FR6 - Reasonable scalability	1/8	1	0,5
FR9 - In-Memory processing support	1/8	1	1
FR10 - Configurable resiliency	1/8	1	1
NFR1 - No special licensing	1/8	1	1
NFR3 - Cloud deployability	1/8	1	0

Table A.6. Data storage *requirement* evaluation proposal.

Characteristic	Eval. Method	Weight	Druid	ClickHouse
Scalability	Scaling capabilities	2/14	1	0
Efficient res. use	Core engine efficiency	2/14	0	1
	Tiering	1/14	1	0
Simplicity of Dev.	User interface	1/14	1	0,5
	Popularity	1/14	0,5	1
Performance	Temporal data optimisations	2/14	1	0
Cloud deployment	Compatibility	2/14	1	0
Openness	Open-Source	1/14	1	1
Cost	Free to use	2/14	1	1

Table A.7. Data storage tools, preferential evaluation.

Appendix B
Attached Files

`SigProcAdapter.zip` : SigProcKafkaAdapter C++ source code.