# Upravljanje, obrada, analiza i vizualiziranje toka podataka u stvarnom vremenu

Josip, Rosandić

**UNIVERSITY OF ZAGREB**
**FACULTY OF ORGANIZATION AND INFORMATICS**
**VARAŽDIN**

**Josip Rosandić**

# REAL-TIME STREAMING DATA MANAGEMENT, PROCESSING, ANALYSIS AND VISUALISATION

**MASTER'S THESIS**

**Varaždin, 2022**

**UNIVERSITY OF ZAGREB**

**FACULTY OF ORGANIZATION AND INFORMATICS**

**VARAŽDIN**

**Josip Rosandić**

**Student ID: 0016129731**

**Programme: Databases and Knowledge Bases**

# REAL-TIME STREAMING DATA MANAGEMENT, PROCESSING, ANALYSIS AND VISUALISATION

**MASTER'S THESIS**

Mentor:

Bogdan Okreša Đurić, PhD

**Varaždin, July 2022**

*Josip Rosandić*

## Statement of Authenticity

Hereby I state that this document, my Master's Thesis, is authentic, authored by me, and that, for the purposes of writing it, I have not used any sources other than those stated in this thesis. Ethically adequate and acceptable methods and techniques were used while preparing and writing this thesis.

*The author acknowledges the above by accepting the statement in FOI Radovi online system.*

## Abstract

This master thesis addresses the topic of data streams from the very beginning of some kind of dynamic data processing and storage all the way to modern streaming data architectural approach. Concepts that are going to be explained are nature of streaming data, streaming data architectures, streaming data sources, ingestion and transportation, streaming data time frames and event streams, processing, storage and visualisation. Architectural patterns are implemented in Microsoft Azure Streaming data stack. Data streams are continuous flows of data which are streaming from the data sources being ingested into the system. Main purpose of this thesis is to give an overview of the data streams, sources, connections, storage, processing possibilities and main properties.

**Keywords**: data;stream;architecture;processing;storage;azure;visualisation;

# Table of Contents

# 1. Introduction

If we want to describe today's data universe in general, it can be done using one sentence - vast amount of data. Every small IoT device generates a lot of data, every user's click on the web represents one record of data, every transaction made in any store, restaurant, cafe bar or any other selling spot, our personal data being stored, traffic data, medical, weather etc. It's indeed a vast amount of data being generated every second, minute, hourly, daily, monthly and so on. Data management systems should be robust enough to successfully respond to this challenge. Their nonfunctional requirements like low latency or scalability gain in importance especially in fields full of continuously growing volumes of data. In recent years, users expect to have extremely fast and reliable data management systems and for their data changes to be updated as soon as possible.

Nowadays, an approach to tackle these huge amounts of data includes very important conceptual, architectural and technical ideas and concepts like cloud computing, parallel processing, distributed systems, containerisation and similar very actual concepts. All these concepts connected together create today's state-of-the-art solutions in a field of data engineering, data science, data analysis, software engineering and so on. Without cloud scalability and much higher processing power, it would be almost impossible to tackle with these tasks and requirements.

One of the most trending topics in IT today, and in general is the topic of Artificial Intelligence (AI). In order to be successful, usable and representative, AI needs high quality data to process and to fill it's purpose using high quality data. That means that data engineering part of the process has to be performed very carefully and as good as possible because it has a direct impact on the outcome of what AI produces.

In this master's thesis I will give an overview of what real-time streaming data includes, real-time data management, real-time processing (including analysis) and visualisation. Starting from the basics concepts, comparison between classic (static, non-dynamic) data through the comparison of classic queries with real - time queries, every part is elaborated, explained and shown in a very clear and precise manner. Large scale data storage for streaming data is discussed and compared with classic data storage. Some of the state-of-the-art architectures are displayed and explained in detail. At the end, it is very important to connect technical part with the practical part in order to show what's the actual purpose of this kind of systems.

# 2. Traditional Database Systems

## 2.1. A Brief History Of Data Management

Roots of the traditional database systems are dating back to the late 60s and early 70s [1, p. 2]. First databases were hierarchical and network databases. Hierarchical database model [1, p. 2] is based on a tree of records in which every node can store a piece of data and has its child nodes (except leaves) and parent nodes (except root node). A network model [1, p. 2] is based on a graph of records in which nodes can be connected in a directed graph. These two models were acceptable solution until the moment when their cons outperformed their pros and need for a better solution emerged.

Brittish computer scientist Edgar Frank Codd presented his paper "*A Relational Model of Data for Large Shared Data Banks*" followed by development and release of the first two relational database systems [1, p. 3]. After that moment, it can be said that the age of relational databases started and will be more and more successful in the following years. In the years after, relational model needed to be formalised and standardised in order to be more usable and reliable for the community and business. That is where the famous Entity-Relation model (ER mode) came to light, as well as different ISO [1] and ANSI [2] standards.

By default, traditional data management systems and databases were static, which means that data can be stored and retrieved, but such a database cannot react or trigger some action based on previously defined conditions or rules. That was changed when first active database mechanisms were introduced around 1975 [1, p. 3].

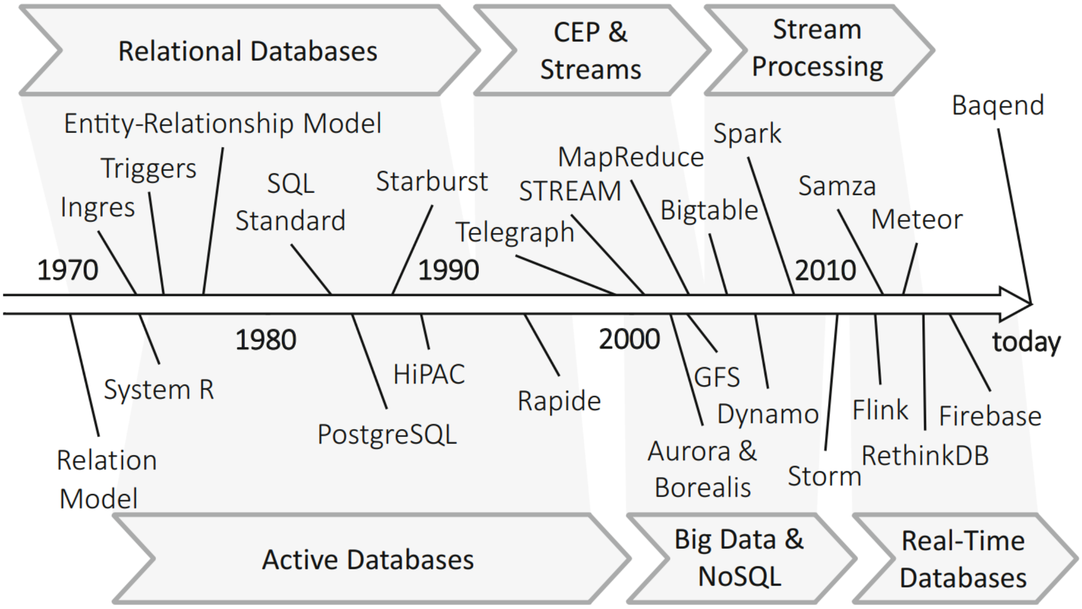


Figure 1: Data Management Systems timeline; from [1, p. 3]

Through the 90s, idea of managing "data in movement" arised with database systems

[1] International Organization for Standardization
[2] American National Standards Institute

like Rapide, Telegraph, STREAM and Aurora/Borealis. These technologies introduced more complex and new concepts like data streams and event sequences [1, p. 4], which takes the whole story to the next level, above the pure relational databases and static data collections. After the dawn of the new century, it was clear that the Internet, web, social networks, web-based business and related fields exploded in a way that attracted millions of users to be connected and to give their personal data to an internet service, social network, web shop etc. Because quantity of the data kept rising, there was a need for creating more "flexible" data storage. NoSQL data stores changed the way that data stores were designed to achieve high scalability and fault-tolerance. Data quantities were increasing and on-premise systems became very weak regarding vast and uncontrollable quantities of data in terms of scalability and storage. Latter problem is very successfully solved with help of cloud computing, servers and virtualisation, parallel and distributed processing and similar concepts. Google File System (GFS) and MapReduce took that story to the next level by successfully managing these quantities of data. That concept is known as big data - vast amounts of data being generated by various producers, stored as structured or semi-structured files in huge volumes [1, p. 4]. Workload was separated as OLTP [3] and OLAP [4] giving DWH [5] on more importance.

All of these concepts lead to the main topic of this thesis - streaming data. From the traditional relational databases all the way to the huge amount of data being streamed in real time from the producer, through the data processor to it's visual representation. In the following sections, path from the conventional databases and their active or reactional component will be analysed.

## 2.2. Active Databases

Initially, relational databases were designed as static repositories as a storage for structured data and it's retrieval as a response to an explicit request [1, p. 9]. There was a need for the database to be in some way active, i.e. responsive to events. Database triggers were the first active mechanism in traditional database systems [1, p. 10]. A database trigger is, in a nutshell, procedure that is implicitly invoked on some kind of a database CRUD [6] event [1, p. 10]. Triggers can also be invoked on system events like errors, logins or similar events. In addition to triggers, ECA [7] rules were introduced during 90s. They were capturing more complex events, including work with temporal components and complex compositions (e.g. disjunctions of sequences). Active databases' purpose reflects mostly on the user side during the interaction with the application.

One of the very important things to think about is that active databases, sometimes also called real-time databases, can quickly, if overloaded, become huge performance bottlenecks. Because of its compensation of execution time, these databases relax consistency guarantees, reduce concurrency or query and rule expensiveness. Because of their architecture and

[3] Online Transaction Processing

[4] Online Analytical Processing

[5] Data Warehouses - type of analytical data storage

[6] Create, Read, Update, Delete

[7] Event-condition rules

concepts, active databases were not supposed to be used as a scalable solution [1, p. 10].

## 2.3. Data Changes Notifications in Traditional Databases

Because of active databases' limitations, other forms of systems were proposed to track changes in the data. Systems that are supposed to be CDC [8] systems take data from the primary system (storage) sending them to the secondary system for processing. CDC can be used to cache data and send them to the main processing tasks [1, p. 11].

Timeseries databases [1, p. 11] store infinite sequences of events connected with their time occurrence with time as its main index. Their purpose is mainly analysis [1, p. 13].

One of the important concepts in context of the CDC are materialised views. Their purpose is to precompute the result of expensive queries in a way that they can be served immediately [1, p. 12]. In a contrast there are logical views that have to be evaluated at every request. Changes have to be detected and applied at write time in order to be up to date with new data in materialised views. One thing that is very helpful in these situations in incremental view maintenance, which avoids recomputations by detecting and applying changes directly. Idea of self-maintainable views postulates that a view can be kept up to date in some way using view contents and incoming modifications (database writes). Thinking of middle position between view recomputation and incremental maintenance, some of the methods only maintain auxiliary data by materialising their structures and incrementally maintaining it [1, p. 13].

In the end, there is a mechanism of tracking change notifications in the database. Some of the relational database systems have the ability on data changed. To compare change notifications with continuous query subscriptions in data streaming systems, change notifications do not carry changed data itself but only their identifiers (e.g. row ID, table identifier) and some basic info on change. Change notifications can be used as a mechanism for caching changes and speeding up data querying. Applications that want to track changes can subscribe to these notification changes and track changes that way [1, p. 14].

To summarize this section, traditional database systems don't have acceptable mechanisms to track data changes and, what is even more important, to push them to the user which can be called proactive data delivery, or push based data delivery [1, p. 14]. In the following chapters and sections, real time databases are discussed as a middle node on this path to the streaming data. Concept of the streaming data is elaborated starting from the traditional - static data storages, through their active components all the way to the streaming data , current architectures and technologies.

[8] Change Data Caputure

## 2.4. Real-Time Databases (RTDBs)

The main difference between traditional database systems and Real-Time Databases (RTDBs) is that traditional databases provide static, consistent snapshot of data, while RTDBs' data is dynamic and may evolve through the time. An RTDBs system has to be reactive, which means that it has to react to every data change and correctly issued information.

By the book, "real-time databases are systems that provide push-based access to database collections" [1, p. 21]. In addition to this, real-time queries are push-based queries. In their core, they are basically the same concept as a classic database queries but they return continuous stream of informational updates.

### 2.4.1. Real Time Queries

A real-time query delivers information that consists of two parts. The first one is an initial result and the second one a change events. The initial result represents data that would be returned by the common query and change events are sent whenever the result is changed (altered) - that process is called "evolution of dataset over time" [1, p. 22]. In a nutshell, RTDBs work with a kind of decision tree in order to decide in which way to act to maintain that collection of data up to date. There are two components of this approach: queries and writes [1, p. 23]. One is being determined relative to another by asking two decision questions [1, p. 23]:

- Does the item match the query now?
- Did the item match the query before?

That means that the database has to check every CRUD [9] operation in order to track changes, which can be an extremely expensive operation when huge quantity of concurrent queries is being started. Many concurrent real-time queries can make this process very expensive if the workload is large [1, p. 23].

[9] Create, Read, Update, Delete

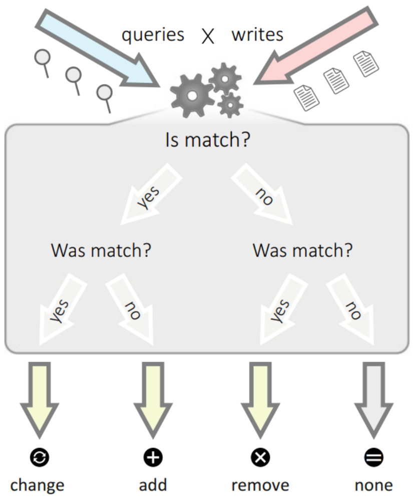


Figure 2: RTDBs Query Execution; from [1, p. 23]

## 2.5. Pull vs. Push Approach

Every data management system can be vaguely classified as a pull based system or a push based system [1, p. 4]. A pull-based system assembles data from a data repository and returns data at once, in a bulk. On the other hand, a push-based system generates incremental output over time and pushes it to the desired destination. Traditional database systems are mostly pull-based with the elements of push-based systems, e.g. through triggers.

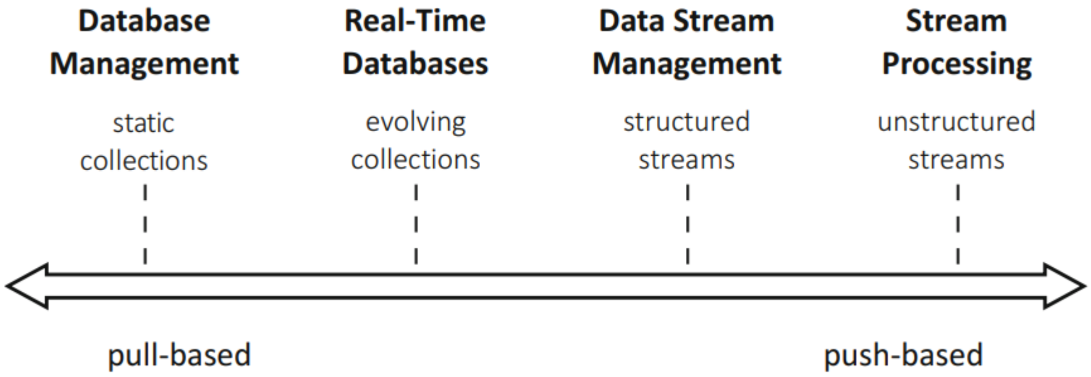


Figure 3: Data Management Systems timeline; from [1, p. 4]

Figure 3 shows the range of the database systems from push-based to pull-based [1, p. 4]. This is a very important relation to understand in order to connect traditional database systems with data streams, how they are similar and how do they differ. On the very left are static collections, so called traditional databases, which are enriched by some active components like triggers and stored procedures. Second type are real-time databases. They store data in a static collections, but they are constantly checking for a changes in the query or output in order to update the result. Next part is data stream management, which is a open gate to streaming data world. Data stream management systems provide APIs to query data streams which are more structured. At the very right are stream processing systems with the emphasis on unstructured streams. They are designed to generate output from unbounded or arbitrarily structured ephemeral data streams.

# 3. Streaming Data Fundamentals

After looking back to the roots of traditional database systems and many attempts to make statically stored data dynamic, now it's time to take a comprehensive look at so-called data in motion - streaming data. According to the Cambridge Dictionary [2], "*stream*,v. means to flow (...) quickly and in large amounts without stopping", or more IT related definition "*to listen or watch sound or video on a computer, mobile phone, etc. directly from the internet rather than downloading it and saving it first*". These definitions are very useful in understanding what streaming data implies. First feature, as these definitions define, is quick flow. That means that stream implies always open, continuous flow from point A to point B. Second very important feature of streaming data would be its large volume. Namely, streaming data is being generated constantly, without stopping that continuous flow, which means that pipelines for handling that amount of data in real time have to be extremely robust.

To explain the core of this concept, example from [3, p. 299] will be presented. Use case is related to real-time parking availability. For example, in one large garage in a big city in which a person wants to find a free parking place. First case is to imagine it without any sensors or technology. It would be very hard to find a free spot and in the worst case, you would need to go to the very last parking place of the garage to confirm that all of them are taken. Requirement from the client's perspective could be the following: *I would like to see where all free parking spaces are or which of the taken ones is going to be free very soon*. This use case uses IoT as a primary and probably the only data source and that is one of the possible types of data sources for streaming data - various IoT devices (mostly sensors) that are capturing some states of the environment sending data in a chunks to some central spot for further utilization.

## 3.1. Nature of Streaming Data

As Ellis [4, p. 7] states in his book, streaming data by it's nature differs from the another types of data based on the following three reasons:

1. always on type of data
2. loose and changing data structure
3. high cardinality and volume

**Reason No.1 - Streaming data is *always on*** This type of data is always available and new data is always being generated. That movement makes collecting, processing and analysis different and more complex than with static data. Data collection of the streaming data should be very robust. If primary collection system faces downtime, that usually means that data is permanently lost which can be very dangerous regarding critical (hard real-time) systems. Second thing is that system that processes the data has to be able to process given amount of data in a reasonable time frame. In simple words, if a system needs 2 minutes to process 1 minute time frame of data, it is very hard for that system to be a reliable real-time system. In practice, system has to process data faster than the data inflow is [4, p. 8].

Because of it's always on nature, additional care should be applied when working with statistical methods on the streaming data flows. As Ellis [4, p. 8] states in his book, most of the statistical methods that are developed in the last 100 years are focused on discrete collections or discrete experiments. This does not mean that these methods should not be used in practice, but they should be used with care.

### Reason No.2 - Streaming data is loosely structured

To disclaim one thought at the beginning – loose structure is not unique to the streaming data, nor can streaming data be structured only loosely. It is just more common for streaming data to be loosely structured than for the other, more static types of data [4, p. 8]. One of the reasons for that is that streaming data comes from a variety of sources some of which carry an arbitrary data payload. One of the examples for that can be social media content which can be very loosely structured or completely unstructured. Language processing is more difficult for computers when that data is represented in a human language. At that moment NLP [1] methods and techniques could be utilized in order to extract some insights from the data. This particular nature of the streaming data also means that not every dataset dimension is going to be available at every moment.

### Reason No.3 - Streaming data has a high-cardinality

As explained in [4, p. 9], cardinality is a number of unique values that a piece of data can take on. This feature can be common to both streaming and batch systems. It is harder to deal with high cardinality data in the streaming systems than in the non-streaming, static systems. Because of streaming data single-time processing possibilities, "long-tail" feature may occur. Long tail means small number of common states in the data which results in a large-dimension datasets or long tail. That means that many different occurrences for the same attribute can occur within the short time frame. Because of this problem, a lot of research addresses this problem of high-cardinality when speaking of streaming data.

[1] Natural Language Processing

## 3.2. Streaming Data System as a Real-Time System

In some domains, it is really important to have real-time insights into the domain data and possibility to monitor these domain systems while in others it is only very useful but not that important. There are many examples of data being used as it is being generated. Systems that work with this type of data are called real-time systems [5, p. 4].

### 3.2.1. Real-Time Data Systems

As [5, p. 4] Psaltis stated in his book, to better define real-time systems, we could classify them as following:

- hard real-time systems,
- soft real-time systems,
- near real-time systems.

In Table 1, an overview of that classification, some examples and features of each type of system as well as a tolerance for delay are presented.

Table 1: Real-Time Systems Classification; from [5, p. 5]

| Class | Example | Latency | Delay Tolerance |
|---|---|---|---|
| Hard | pacemaker, nuclear power plant data | $\mu\mathrm{s} - \mathrm{ms}$ | None - because of possible consequences (system failure or life at risk) |
| Soft | VoIP, stock/crypto prices | $\mathrm{ms} - \mathrm{s}$ | Low - no system failure, no life at risk |
| Near | Video call, home automation | $\mathrm{s} - \mathrm{min}$ | High - no system failure and no life at risk |

As shown in the table 1, hard real-time systems are almost always part of embedded systems and are one of the most critical types of systems. Some of the examples of hard real-time systems could be medical devices like pacemakers, heart monitors, devices in autonomous driving cars or even systems for monitoring nuclear power plants. System failure in any of these systems can result in very bad or even catastrophic consequences. Soft real-time systems and near-real time systems could sometimes be hard to differentiate. Very often, the line differentiating soft and near real-time systems can be really blurry because it depends on the given case. Based on the latter thoughts, it would be better for some systems to be soft real-time or for some of them near real-time.

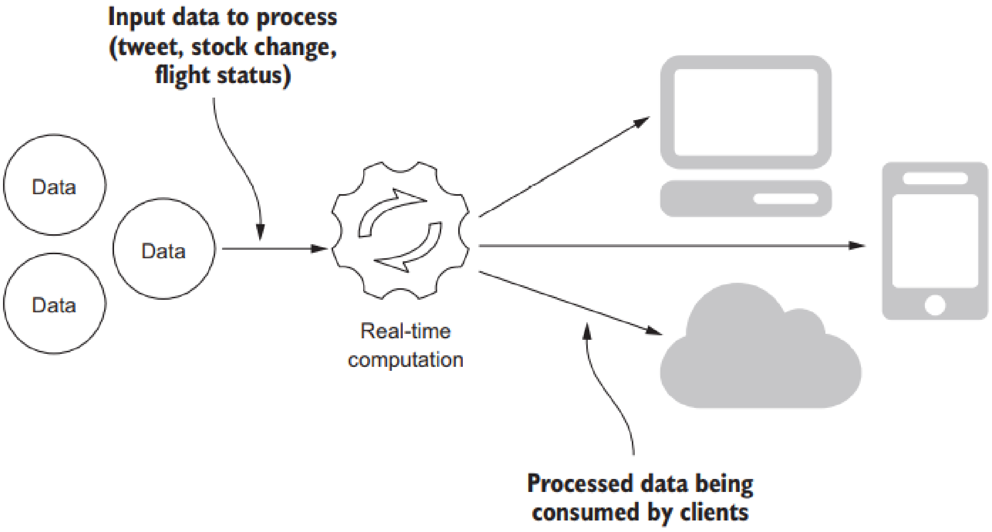

Input data to process (tweet, stock change, flight status)
Data
Data
Data
Real-time computation
Processed data being consumed by clients

Figure 4: Generic Real-Time System; from [5, p. 5]

Figure 4 shows a simplified, generic architecture of a real-time system. Main parts of the architecture are data producers, a real-time computation module for processing the produced data and consumers using that processed data.

### 3.2.2. Streaming Data Systems as Real-Time Systems

Psaltis [5, p. 7] defines a streaming data system as a "non-hard real-time service with clients that consume data when they need it". We can say that a streaming data system is a subset of the real-time system concept. Main point of this definition is that streaming data system provides data instantly as clients need it.

Figure 5 shows the data that is being processed in a streaming computation module and consumed by the clients when they need it. Later in the thesis, overview of the model is going to be presented with an explanation that these clients are "subscribed" to the data pipeline which is a sequence of activities or messaging middleware which is a kind of software for managing message flow, and that they are called consumers.

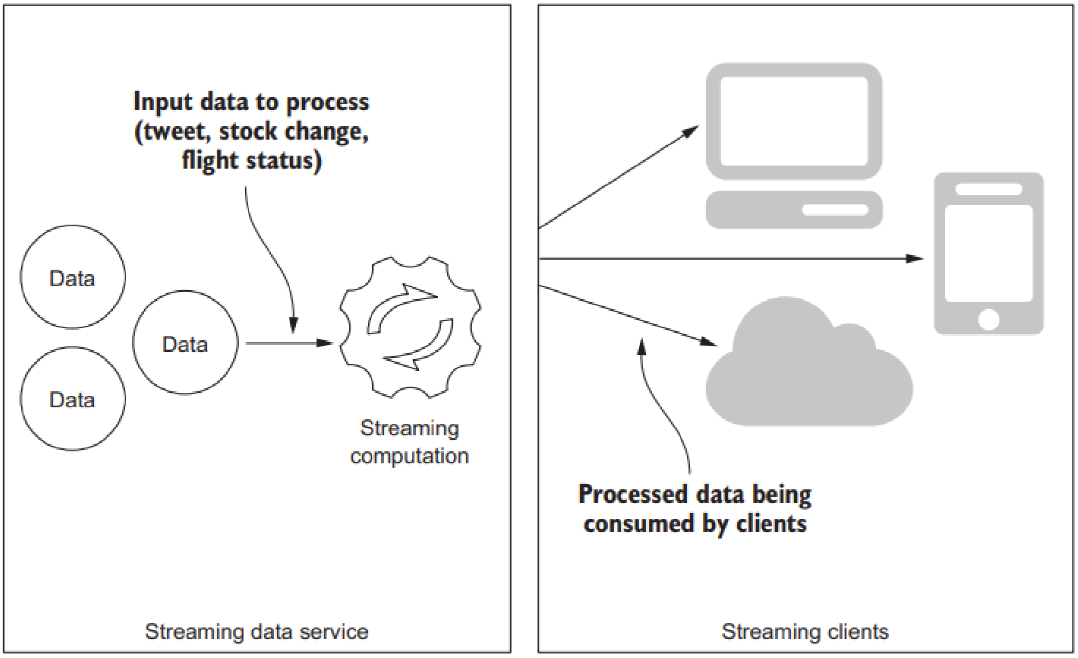

Input data to process (tweet, stock change, flight status)
Data
Data
Data
Streaming computation
Streaming data service
Processed data being consumed by clients
Streaming clients

Figure 5: Streaming Data System; from [5, p. 8]

### 3.2.3. Conceptual Architecture of Real-Time Systems

At the very abstract and high level, very basic functional parts of every real-time system, shown on a figure 6, can be recognized. At the beginning we have data producers or data sources that are actually generating data. First part of the system is a collection tier, which collects parts of data from the data source. Next component is a message queuing tier or messaging middleware, which queues incoming collected messages, and performs different actions in order to sort, order or label incoming chunks of data. Analysis tier is a part in which chunks of data are analysed in order to detect some important features of the data. If there is a need for it, data can go to the long term storage. In-memory data storage is a temporary storage being used by the data access tier, which handles access to the data from the consumers.

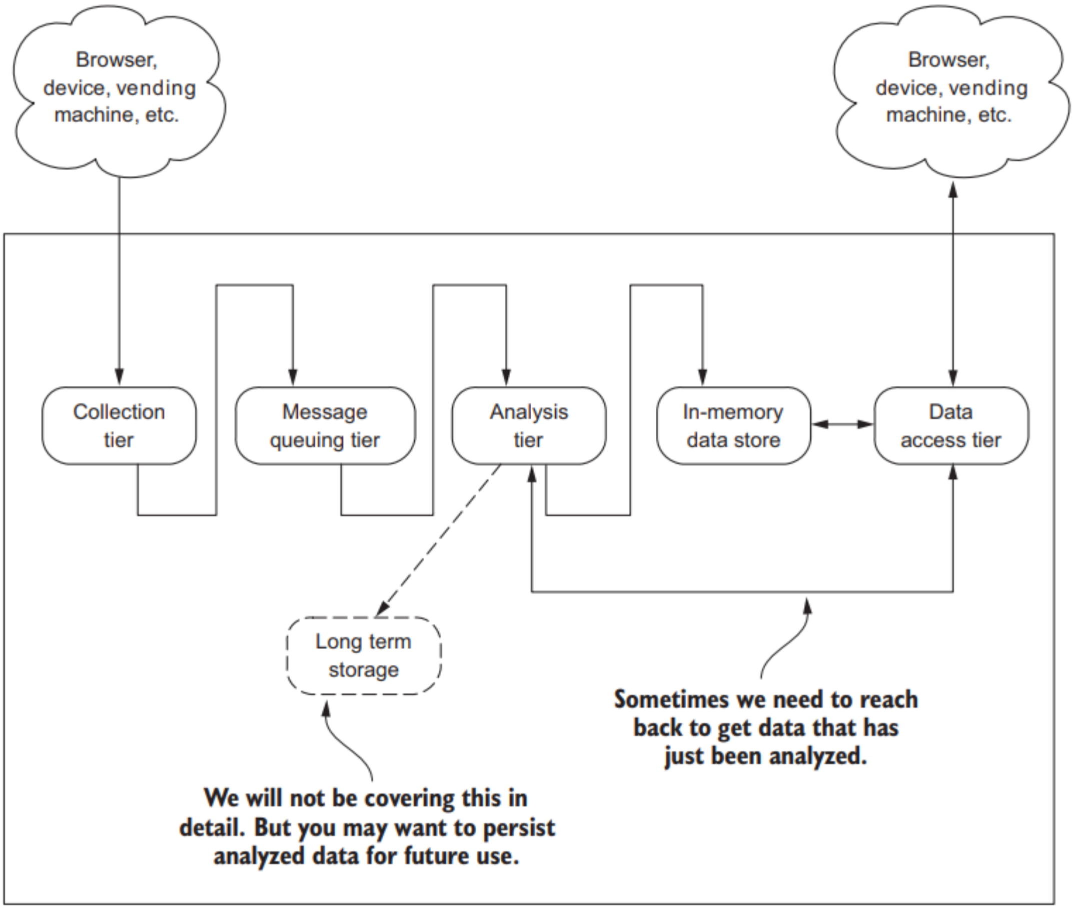

Browser, device, vending machine, etc.
Browser, device, vending machine, etc.
Collection tier
Message queuing tier
Analysis tier
In-memory data store
Data access tier
Long term storage
Sometimes we need to reach back to get data that has just been analyzed.
We will not be covering this in detail. But you may want to persist analyzed data for future use.

Figure 6: Generic Real-Time System Architecture; from [5, p. 9]

## 3.3. Streaming Data Architectures

Architectures of streaming data systems have common architectural elements that can be found in almost every streaming system architecture. Not every streaming data system will have every component but every system has components that are necessary for it's use case. Ellis [4, p. 15] and Wingerath et al. [1, p. 57] both recognised and documented in their papers four main architectural components (layers) of every streaming data system. They are:

1. collection component
2. stream (or flow) component
3. processing component
4. serving (or delivery) component

Streaming architectures, i.e. architectures of streaming data systems, in comparison to the traditional data management architectures, are focused on minimizing the time a single data chunk spends in the process pipeline [1, p. 57]. In addition, streaming systems achieve end-to-end latencies of several seconds or even less.

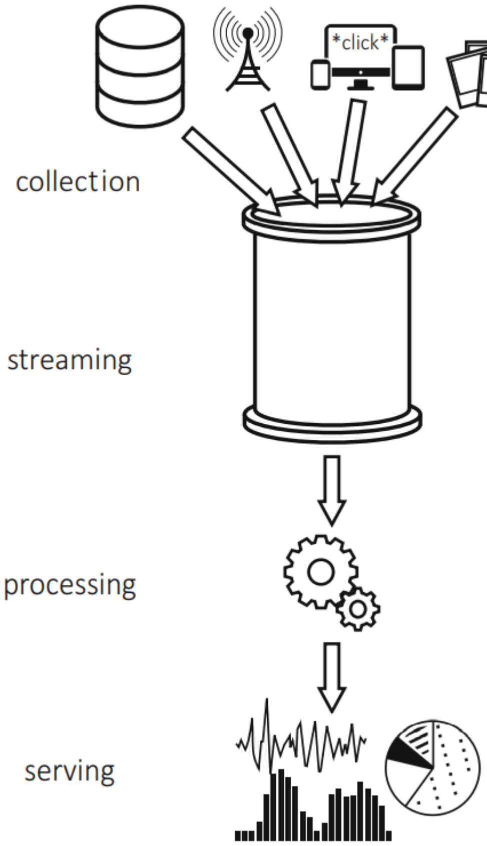


Figure 7: Abstract Schema of Streaming Data Pipeline; from [1, p. 58]

Figure 7 shows main streaming architecture layers mentioned above. Data from the data sources are being collected and moved into the streaming layer. Streaming layer is transporting data to the data processor, or to be more precise, data processor consumes data from the streaming pipeline and processes them. Output from that processing is being forwarder to the serving layer where those data are displayed to the end user as a part of some web app, dashboard, report or event being used by another system. Digging deeper into this architecture, it can be decomposed into more detailed parts in order to better manage the data stream. Streaming data "travels" through the pipeline with very high velocity but not in a high volume at once (high volume is being recognized if the data are stored in a side storage). Thus, the best option would be, besides this architecture to have a solution in which the data can be processed as a very fast stream, and in a batch after collecting all of the data in one storage. Two of the most recognizable architectures - Lambda and Kappa are presented in the following subsections [1, p. 58].

Before presenting these two architectures, it would be perfect to emphasize differences between batch and stream processing [1, p. 58]. Decision of which processing model to implement can have a great impact on the speed of the streaming data and its processing. Processing data stream immediately minimizes latency but provides not so great data quantity in a desired shorter timeframe (window). On the other side, processing data in batches yields efficiency but increases the processing time. Therefore, some of the streaming systems employ micro-batching strategies to trade latency against throughput.

### 3.3.1. Lambda Architecture

Lambda architecture is an architectural pattern for data streaming system where data is being processed in two approaches [1, p. 58]. First approach, or first branch of the architecture consumes data from the streaming pipeline, processing it in a speed layer (also called hot tier) and serving it immediately. Second approach, or second branch of the architecture is a batch layer (also called cold tier) that stores data from the streaming pipeline into the permanent storage in order to take it from the storage and process it in a larger batch. Figure 8 shows architectural schema of the lambda architecture.

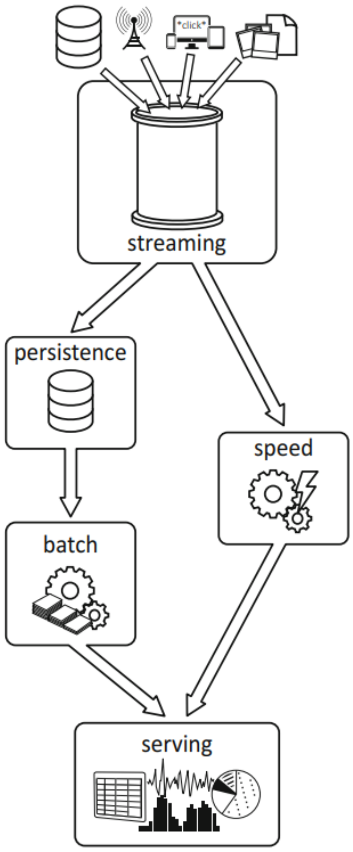


Figure 8: Lambda Architecture; from [1, p. 59]

Lambda Architecture is a hybrid approach to data processing which can, at the very low, technical level, be very complex because engineers have to maintain two separate code bases and computation logic can be duplicated [6]. Lambda architecture solves the problem of high latency during execution of queries. Cases where users need their data as soon as possible, this hybrid approach can be perfectly used to address this data obsolescence problem.

### 3.3.2. Kappa Architecture

As mentioned before, Lambda Architecture is a pattern that solves very important problem, but sometimes it's simply too much for certain cases. One of its largest drawbacks is the previously mentioned complexity and maintenance. Another, alternative architecture is Kappa Architecture [1, p. 59], which addresses the problem of complexity. In the Kappa Architecture, all of the computation is done in the streaming pipeline. Data are ingested as a stream of events into a distrubuted and fault tolerant log. This architecture uses long-term storage only if the recomputation of some part of the data is needed. That means that Kappa emphasizes stream processing but uses batch processing if needed [1, p. 59]. Therefore, data storage is possible but not the critical part of this type of architecture.

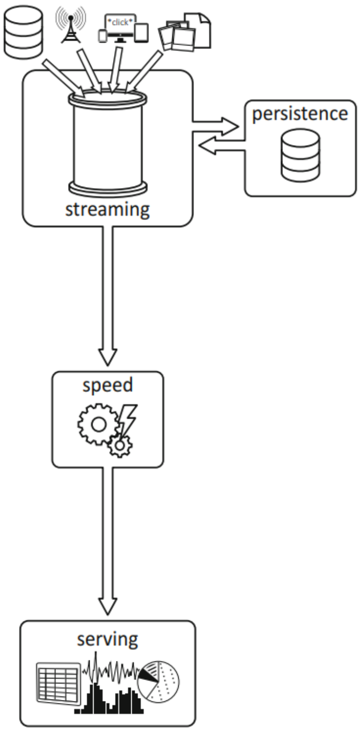


Figure 9: Kappa Architecture; from [1, p. 59]

Data storage in this type of architecture is defined by its retention period setting. That means that after some time, some data will be disposed, usually according to the FIFO[2] rule. Stored data can be used to simply replay the stream [6].

[2] First In First Out

## 3.4. Streaming Data Sources, Ingestion and Transportation

### 3.4.1. Streaming Data Sources

Among many streaming data sources, it is very hard to make the exact classification of them but some of the major categories can be enumerated. Ellis [4, p. 2] in his book describes a few of them that, as he stated, "made streaming data interesting" and they are:

- operational monitoring
- web analytics
- online advertising
- social media
- mobile devices, IoT and biometrics
- financial and sales data

#### 3.4.1.1. Operational Monitoring

Operational monitoring is considered as one of the original applications and usages of streaming data. This term implies monitoring of various physical systems and is being realized using specialized hardware and software. According to Ellis [4, p. 3], operational monitoring would be a continuous process of the tracking and monitoring performance of physical systems, initially used to monitor huge network infrastructure centers. In the cases like this, many sensors are monitoring state of the whole system in real-time and recording parameters like temperature of the hardware, speed of the fan, state of the disks, voltage, network activity and similar features. To successfully monitor systems like these, data have to be collected and proccessed in real-time.

#### 3.4.1.2. Web Analytics

Every activity that any of the users commit on the web generated potentially usable piece of data, which, aggregated with the other parts of same data, can lead to website activity insights. Nowadays, web activity is one of the most valuable and important resources for tracking users' behaviour, habits, routines, fields of interests etc [4, p. 3]. Many of news portals track number of visitors in real-time and e-commerce sites track which product categories or particular products users are mostly interested in. Data collected from various web activities can be used as input to recommender systems, or help to analyze how different user interface schemas impact users (e.g. A/B testing in digital marketing) [4, p. 4].

#### 3.4.1.3. Online Advertising

Advertising is one of the most important activities in business when it comes to client acquisition. Many metrics in digital marketing can be monitored in real time and automatically configured in order to make the advertising process beter. One of the most important goals is to manage and optimize marketing campaigns that are currently active.

#### 3.4.1.4. Social Media

One of the biggest sources of streaming data is definitely the social media environment [4, p. 5]. Social networks like Twitter generate large amounts of data in small time frames. Although data from social networks are mostly unstructured, it can be very useful to do semantic analysis on such kind of data. Social media data can be extremely rich and valuable, but since they are loosely structured, they can also be very hard to process.

#### 3.4.1.5. Mobile Devices, IoT and Biometrics

One of the definitely most interesting and recognizable sources of streaming data are mobile devices, IoT devices and biometrical devices. Smartphones can be very valuable data source since they transpond data through the cellular network generating lots of interesting data about users. Mobile devices are highly interconnected, which makes this story even more interesting. Different mobile devices called "wearables" took this story to the whole new level. Streaming from the large number of small devices, very valuable data about people's physical state, sleeping habits, activities and similar things can be collected [4, p. 5].

Different IoT devices with sensors can record many states of the physical world, transforming them to bytes and displaying as valuable chunks of data. Important biometrical data can be measured in real-time and collected by every user over a long period of time. Examples of this can be heart rythm, blood pressure and similar applications.

#### 3.4.1.6. Financial and Sales Data

The last streaming data sources, but definitely not the least are financial and sales data sources. Every day and every second in the world unimaginable number of transactions is being processed in various industries and fields. From basic markets, grocery stores, shopping centres, restaurants and cafe bars to large B2B transactions. Vast amount of these transactional data is being generated especially when we talk about huge companies. Those data can be excellent input resource for various business intelligence analyses in order to improve business performance, make important decisions or predict some business trends. Also, one of the biggest financial data holders are banks but security standards in banks are really rigorous, which makes this process a bit more complex from the security side.

### 3.4.2. Streaming Data Ingestion - Interactional Patterns with the Sources

When speaking of the way in which data source is communicates or connects to the streaming system, Psaltis [5, p. 16] explained that every single communicational pattern can be classified in one of the following categories (communication patterns). These are:

- request/response pattern,
- request/acknowledge pattern
- publish/subscribe pattern
- one-way pattern
- stream pattern

#### 3.4.2.1. Request/Response Pattern

First, and the most simple one is request/response pattern. This is a very common pattern in e.g. HTTP communication where one connection opens and communication flows with request and response messages. Generally, client is the one who mostly requests some resources and server (service) is the one who responds to the requests. Client makes a request to the service and the service responds to the client.

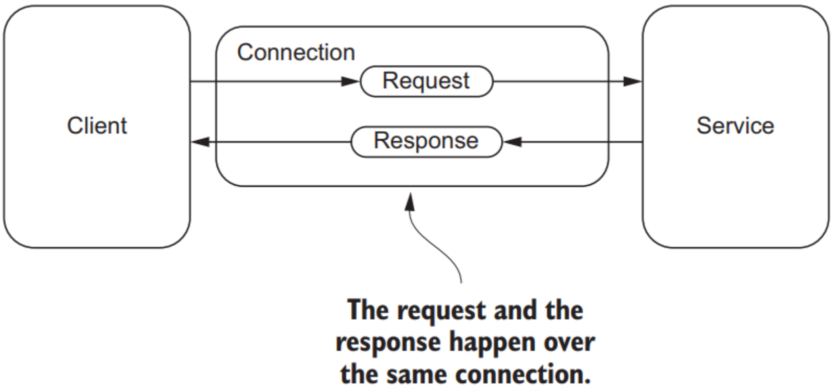

Connection
Request
Client
Service
Response
The request and the response happen over the same connection.

Figure 10: Basic Synchronous Request/Response Pattern; from [5, p. 16]

This is the most basic, synchronous request/response pattern. This is also not the very real-world usable pattern in terms of waiting. It would be very bad if a client would need to wait the service to respond for every single chunk of the resource. That would make a huge waste of time. That is where asynchronous pattern takes place. Asynchronous request is a kind of request where client, after requesting some resource, does not have to wait for the response and therefore, does not have to waste the time. Service responds when it's ready to respond.

#### 3.4.2.2. Request-Acknowledge Pattern

Very similar to the previous one but without the response from the service, request/acknowledge pattern just uses an acknowledgement that the request was received. As an ex-

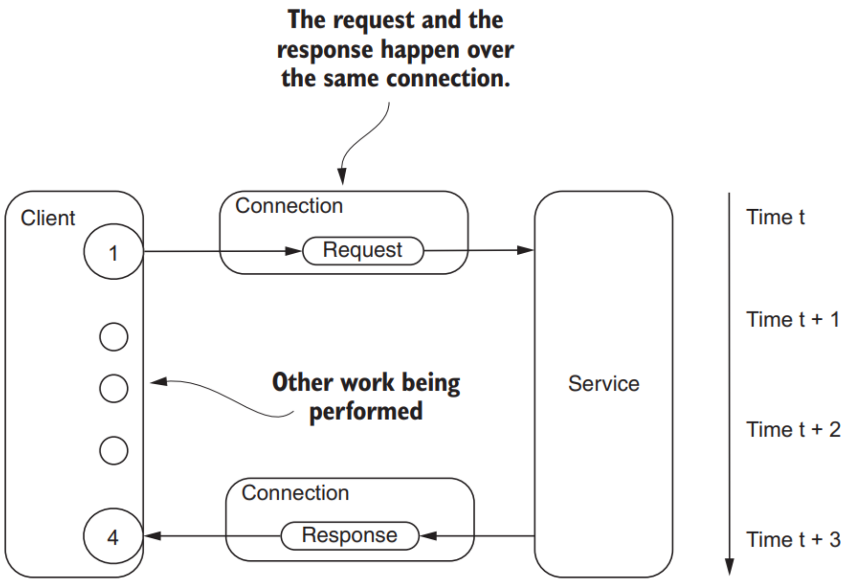

The request and the response happen over the same connection.
Client
1
Connection
Request
Service
Time t
Time t + 1
Other work being performed
Time t + 2
Connection
4
Response
Time t + 3

Figure 11: Basic Asynchronous Request/Response Pattern; from [5, p. 17]

ample, Psaltis [5, p. 19] explained this pattern on the e-commerce web site tracking number of interactions with the site. Using this pattern, the acknowledgement message can be sent to the subsequent pages of the web service in order to deliver some messages useful for the future requests. Example of this is when we purchase something online, we got a confirmation number as an acknowledgement.

### 3.4.2.3. Publish/Subscribe Pattern

A more common communication pattern than previous ones in message exchange systems is publish/subscribe pattern. Main architectural elements of this pattern are:

- producer(s)
- broker(s)
- topic(s)
- subscription(s)
- consumer(s)
- message(s)

Main idea of this pattern is that producers publish a message into a broker. Brokers consist of topics which could be thought as a logical containers for messages. Messages are being reordered, sorted, balanced or something similar being sent to the subscription. Consumers are receiving these messages from the subscriptions that they are subscribed to. An example of this pattern would be a basic subscription to some information like traffic status, where every car produces some new info about the environment and the other cars can consume this information if they are subscribed to the topic [5, p. 22].

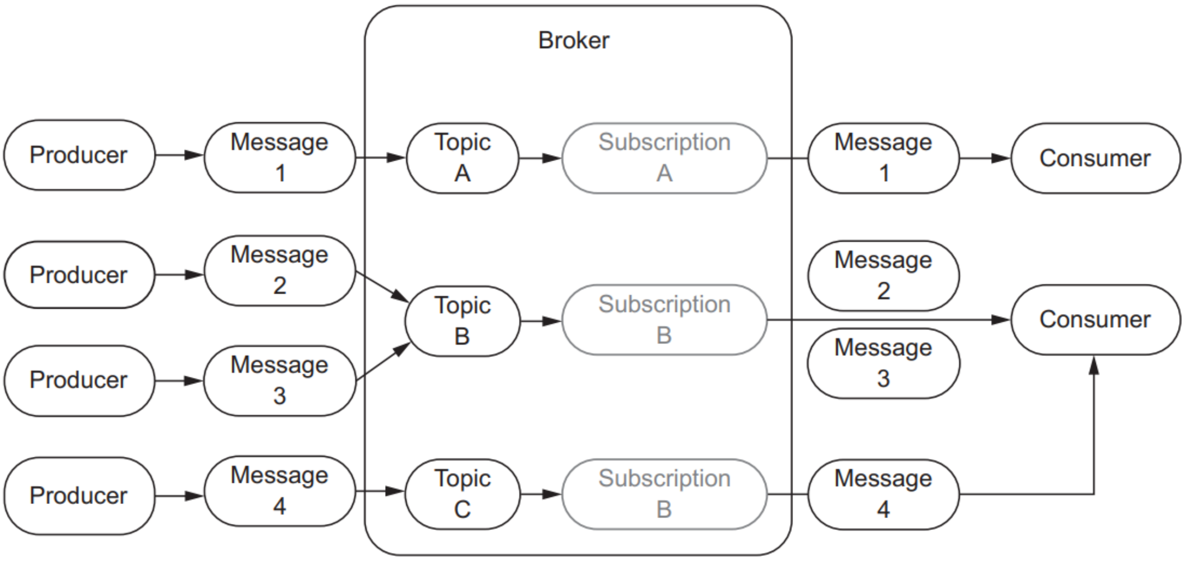


Figure 12: Publish/Subscribe Pattern; from [5, p. 21]

#### 3.4.2.4. One-Way Pattern

This is an example of the pattern within which the client does not need a response at all. It is also called "fire and forget" pattern. This is a pattern which can be used in a systems where a client does not have processing capabilities or needs, therefore, it just sends the information in one way to some processor without the need to know if that service received the data. Useful example would be some IoT devices that scan the environment and just send that data to a hub for processing or transportation [5, p. 23].

#### 3.4.2.5. Stream Pattern

Stream pattern is a very different pattern from the previous patterns. Within this pattern, a client and a service change their roles, in a way. Service sends a connection request to the client as a streaming data source after which client opens a streaming connection to that service, sending data continuously.

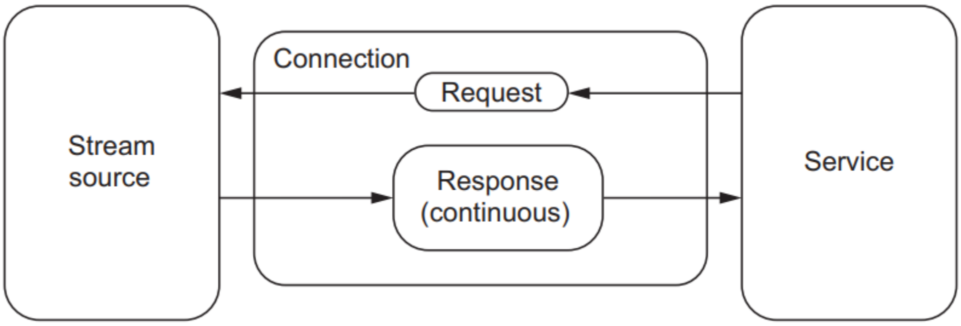


Figure 13: Stream Pattern; from [5, p. 23]

### 3.4.3. Streaming Data Transportation Pipelines

After defining the ways in which data can be collected communicating with the data source, a very important part of the whole streaming pipeline is data transportation to the further parts of the system. The part of the whole system responsible for this task is called message queuing tier, as Psaltis defines it [5, p. 39] or messaging middleware as Wingerath et al [1, p. 49] name it in the literature.

#### 3.4.3.1. Message Queuing Tier (Messaging Middleware)

One of the first questions to arise when thinking about standard streaming system architecture and looking into the Message Queuing Tier (MQT) is "why do we need it?". Previously in this document the most common architectures and their building elements were explained. One of them is an MQT.

After collecting the data, data are going in the direction of the data processor or some other consumer (not only end users, but other systems, APIs etc.). A monolithic system in which every consumer is directly connected to the collection tier can be tremendously hard to maintain. One of the main concepts of any engineering part in computer science is modularity of the components. Modularity of any system implies that elements of that system work together to form a single whole but if needed, they can perform as a completely individual function if not connected with each other [7]. In oder to achieve a modular streaming pipeline, where data consumers are not directly connected to the producers and where subcription to the data is not a complex story, MQT is utilized. In a nutshell, the main idea of MQT is to decouple elements of the system from each other. Streaming systems are mainly distributed between numerous machines what would be way more complicated without a modularity.

Core concepts of the MQT that are critical for it's function are [5, p. 41]:

- producer,
- broker,
- consumer.

These terms when were already explained at the part of exploring the publish/subscribe communication pattern. Producers are the entities that produce (generate) the data, and consumers are the entities that consume (use) those data. Broker is a part of the MQT which is kind of a hub for many data stream queues that are being created inside. The relation broker:queue is 1:N (one broker, more queues in it). An example of a very simple data flow through the MQT would be:

1. producer produces a message and send it to a broker,
2. broker queues that message,
3. consumer consumes (reads) that message from the broker.

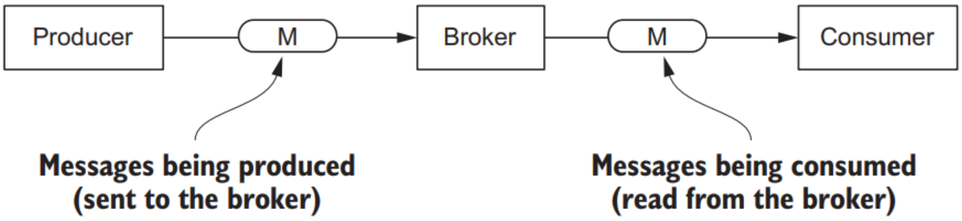

Producer
M
Broker
M
Consumer
Messages being produced (sent to the broker)
Messages being consumed (read from the broker)

Figure 14: Message Queuing Tier Schema; from [5, p. 41]

Now after and MQT is introduced, an importance of such a system element can be explained. Considering speeds in which producers and consumers operate, streaming data system without the MQT would be hardly possible. It is very clear - if a producer produces more than the consumers can consume, without the MQT, with the monolithic approach, streaming data pipeline would "explode" and very important or even critical data could be lost.

Since elements of streaming data system can be geographically distributed, durable messaging should be considered. That means that in case of losing connection between the elements, MQT has enough space to store our data in a permanent storage until connection gets back. Durable messaging provides a degree of fault tolerance and offline consumer scenario.

System failures can happen in any of MQT parts. If the producer fails before sending the message, message are lost. If the producer sends the message and is not sure if broker received it, it can resend the same message. If the network between producer and broker fails, broker never receives the message or producer never gets the response. Message can be sent again. All of the messages could be lost if the broker fails and they have not been stored in a persistent storage. If the network between consumer and broker fails, either the broker can send the message, but the consumer does not receive it, or if consumer don't acknowledge reception of the message, broker can resend it. If the consumer fails, it can request the data from the broker again.

#### 3.4.3.2. Broker Fault Tolerance

After exploring MQTs in depth, what happens when a broker crashes will be elaborated [5, p. 54]:

- if the broker uses some kind of a durable storage, all of the important data should be previously stored there. It can be enforced with waiting for an acknowledgement from the producer
- message queues could be replicated to more than one broker in order to keep the data stored and distributed (significantly reduced risk)
- broker could be configured to hold as little data in memory as possible in order to, if fault happens, small piece of data is lost (performance implications!)
- broker replication could be configured in order to be able to distribute the data in case of network interruption. After network normalisation, brokers will synchronise automatically.

## 3.5. Time Frames and Event Streams

When thinking about streaming data, focus is not on stationary data but on the data that flows over time moment by moment creating data stream. One of the first thoughts of the streams is linear movement of continuous events through the time as presented on Figure 15.

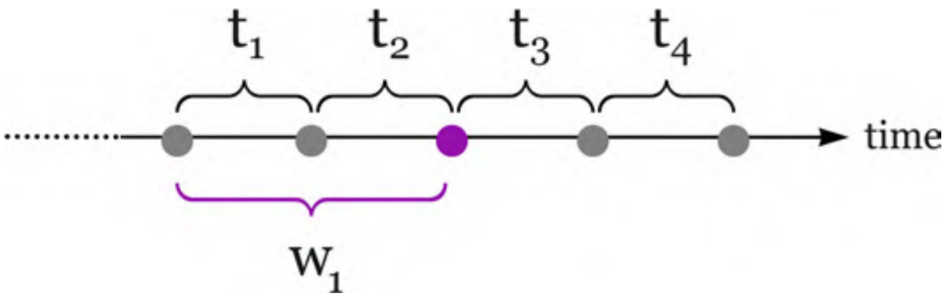


Figure 15: Linear Stream Movement Through Time; from [3, p. 301]

Events (displayed as circles) occur at precise moments in time and can be collected individually ($t_1, t_2, t_3, t_4$) or aggregated across windows over time ($W_1$) [3, p. 23]. These events could be understood as a chunks of data that are being produced at a specific time and place. Time of creation does not have to be the same like time when data stream management system received and registered that event (a chunk of the data) [1, p. 46], but it can. By default, stream events are ordered as they arrive by a timestamp received when entering the stream. Of course, incoming data stream does not have to be provided in a chronological order to the processing services, storage or other consumers. Stream messages can be reordered in a way that is suitable for a specific business case. That is when the concept of window in stream analytics context becomes very important.

Since a data stream is a very dynamic data portion moving constantly, it could be very hard to perform definite operations on a dynamic and in terms of size - undefined data source. Therefore, in order to make that possible, it is necessary to use some kind of a frame called window (or time frames or series) [1, p. 46]. Very important feature of windows in this context is that they are a finite partition of the data stream.

As Wingreath [1, p. 47] defined, there are different dimensions according to which a window can be defined:

- the direction of window movement (fixed, landmark or sliding)
- the way of content definition (time, number...)
- way of window movement (sliding, jumping, tumbling)
- window update frequency

Speaking of window direction, window is a frame between X and Y and it is fixed if both X and Y are fixed. If one of them is moving, that is a landmark window [1, p. 47]. If both of them are moving, that would be a sliding window [1, p. 47]. Furthermore, we can say that a window is expanding if their X and Y are moving in the opposite directions of contracting if their X and Y are going to meet each other.

Speaking of windows' content definition, content of the window can be defined either in terms of time or number of instances. Query could look as follows: "all records from last 2h" or "last 100 records" [1, p. 47].

Sliding window would be a windows that refreshes on every incoming record, and jumping window is a type of window that refreshes after predefined time. Tumbling windows implement an equal window size splitting stream into non-overlapping ranges.

There are two main windowing techniques [5, p. 81]:

- sliding window
- tumbling window

In the following chapter, these two techniques are going to be explained as well as the third one - hopping windowing.

## 3.6. Data Stream Processing

Stream processing is one of the very important elements of the pipeline, or streaming architecture. Data streaming architecture could work without any processing, only storing as-is data into some kind of storage. To make those data usable and to get insights out of them, they have to be processed. Generally speaking, there are two main ways to process data [4, p. 118]:

- batch processing,
- stream processing.

Either way of processing data can be usable and acceptable, depending on the use case and needs. According to Ellis [4, p. 118], batch processing is a way of processing data in (usually) large batches, which extends processing time, but can give more comprehensive insights based on more data and a longer time frame. Sometimes, it also means more accurate metric results. On the other side, stream processing processes data in a flow. That means that stream processing is a very agile and fast way to process data, but also gives less volume of the data to work on over a shorter period, which means less accurate metric results. Usually, batch processing jobs have a really high starting cost while stream processing jobs amortize that cost to zero. Ellis [4, p. 118] states that stream processing is basically a specialized form of parallel computing for processing data in a flow. While in traditional batch systems static data sources are queried for the answers, in streaming system data is moved through the query which is called *continuous query model*.

Table 2: Traditional DBMS vs Streaming System; from [5, p. 62]

| | DBMS | Traditional Streaming System |
|---|---|---|
| Query Model | One-time model, consistent data. User executes a query and gets an answer. Pull based model. | Query continuously executed on the data flowing through the system. Push based model. |
| Changing Data | Data cannot change during downtime. | While streaming tier is down, stream apps can generate the data. |
| Query State | On system crash during the query execution, it is forgotten. App or user has to rerun that query. | May or may not continue where it left off |

One of the most important concepts in stream processing is distributed stream processing architecture. It is possible to run processing on a single computer, but the volume of data usually makes it nonviable. Today's stream processing technologies are by default focused on distributed processing systems. Most of the state-of-the-art stream processing tools and systems have common architectural parts [5, p. 63]. These are:

- component for executing applications
- nodes for algorithm execution,
- data sources as input for streaming algorithms

### 3.6.1. Streaming Queries

Streaming queries are in this thesis presented using Microsoft Azure technologies, to be more precise, ASA (*Azure Stream Analytics*) processing tool. Messages, events or records entering any data stream processor are entering one-by-one, with no fixed end of the stream. That is where time series, time frames and windows explained in the section 3.5 come to the stage. Time windows are used to contain data that flows constantly and to make creating data aggregations possible. Time window has it's shape and relationship with the data. There are several types of time windows which are used based on the use case needs [8, p. 146]:

- tumbling window
- hopping window
- sliding window
- session window

#### 3.6.1.1. Tumbling Windows

Tumbling window starts after every previous window, at the stream analytics job start time. Messages that are contained in that window are used as a set of data to process. For example [8, p. 141], to calculate 150 seconds average, window duration should be set to 150 seconds. Query would look like this:

Listing 1: Tumbling window query example [8, p. 142]

```
SELECT Player, Node, AVG(NodeValue) AS AvgValue
FROM HubsInputBiometrics TIMESTAMP BY EventTime
WHERE Player = 'abera101' AND Node = 12 AND NodeValue > 80
GROUP BY Player, Node, TumblingWindow(second, 150)
```

Query selects Player, average value of the node from AZ Event Hub for player with the exact name and node with grouping defined by player, node and tumbling windows of 150 seconds. To deliver current value and value based on the last 150 seconds, two sources of calculation should be joined. Average pitch can be joined with the current pitch. That is shown in the following query [8, p. 146]:

Listing 2: Tumbling window query with common expression [8, p. 143]

```
WITH PitchAverage AS (
SELECT Player, Node, AVG(NodeValue) AS AvgValue
FROM HubsInputBiometrics TIMESTAMP BY EventTime
WHERE Player = 'abera101' AND Node = 12 AND NodeValue > 80
GROUP BY Player, Node, TumblingWindow(second, 150)
)
SELECT a.Player, a.NodeValue, b.AvgValue
INTO SqlOutputPitcher
FROM HubsInputBiometrics a TIMESTAMP BY EventTime
INNER JOIN PitchAverage b
ON a.Player = b.Player
AND a.Node = b.Node
AND DATEDIFF(second, a, b) BETWEEN 0 AND 150
WHERE a.NodeValue > 80
```

In the above query, first output is being calculated 150 seconds after a job start.

#### 3.6.1.2. Hopping Windows

Hopping windows are a bit different than tumbling windows in a way that their frequency is not determined by duration. This way of setting the windows up leads to more precise sampling of a stream of data. When the time hop is less than the duration of the window, hopping windows are more frequent, and when the time hop is greater than the duration, windows are less frequent.A hopping window query would look like this [8, p. 146]:

Listing 3: Hopping window query with example of usage [8, p. 144]

```
WITH PitchAverage AS (
SELECT Player, Node, AVG(NodeValue) AS AvgValue
FROM HubsInputBiometrics TIMESTAMP BY EventTime
WHERE Player = 'abera101' AND Node = 12 AND NodeValue > 80
GROUP BY Player, Node, HoppingWindow(second, 300, 30)
)
SELECT a.Player, a.NodeValue, b.AvgValue
INTO SqlOutputPitcher
FROM HubsInputBiometrics a TIMESTAMP BY EventTime
INNER JOIN PitchAverage b
ON a.Player = b.Player
AND a.Node = b.Node
AND DATEDIFF(second, a, b) BETWEEN 0 AND 300
WHERE a.NodeValue > 80
```

In the query above, pitch average was created and joined with each event message that enters the ASA job. In HoppingWindow() function, time units were provided (in this case - seconds), along with the duration of the window and the hop (or in other words, time elapsed from the beginning of the previous window until the start of the next one). DATEDIFF is also adjusted to match the window time [8, p. 146].

#### 3.6.1.3. Sliding Windows

A sliding window starts when an event message arrives at the ASA job, which means one time window for each appropriate event. Depending on message times, period of the window updates can vary. SlidingWindow() function takes time units and duration as parameters [8, p. 146].

Listing 4: Sliding window query with example of usage [8, p. 145]

```
WITH PitchAverage AS (
SELECT Player, Node, AVG(NodeValue) AS AvgValue
FROM HubsInputBiometrics TIMESTAMP BY EventTime
WHERE Player = 'abera101' AND Node = 12 AND NodeValue > 80
GROUP BY Player, Node, SlidingWindow(second, 300)
)
SELECT a.Player, a.NodeValue, b.AvgValue
INTO SqlOutputPitcher
FROM HubsInputBiometrics a TIMESTAMP BY EventTime
INNER JOIN PitchAverage b
ON a.Player = b.Player
AND a.Node = b.Node
AND DATEDIFF(second, a, b) BETWEEN 0 AND 300
WHERE a.NodeValue > 80
```

With duration of 300 seconds, around 25% of the data can be covered. DATEDIFF should be 300 seconds in order to match the sliding window. With this approach, average will be calculated over any number of records in the previous 300 seconds, even if only one [8, p. 146].

## 3.7. Streaming Data Storage

After finishing data processing, one of the following actions is reasonable to follow: [5, p. 95]:

- discard the data
- push the data to the streaming platform
- store the data for real-time usage
- store the data for batch or offline access

At first, discarding the data after processing it sounds illogical, but it can be a real use case. Simply, if a piece of data does not meet some of the requirements, it can be discarded. Pushing data to the streaming platform is also an option, as output from one job can become an input for another one. Last two storage options are storing data for real-time access and storing for batch access. Since the topic of this thesis is streaming data, further sections describe storing real-time data.

### 3.7.1. Long-Term Storage

Sometimes, a use case requires a system that can store data in a long-term storage for batch processing, backup or something else. Generally speaking, data can be stored in a long term storage message by message (in a stream) or a number of messages can be stored in a cache container in order to store them in a batch. Even a combination of these two approaches makes sense, if followed by adding some batch loaders as load balancers, to load the data evenly into the batches.

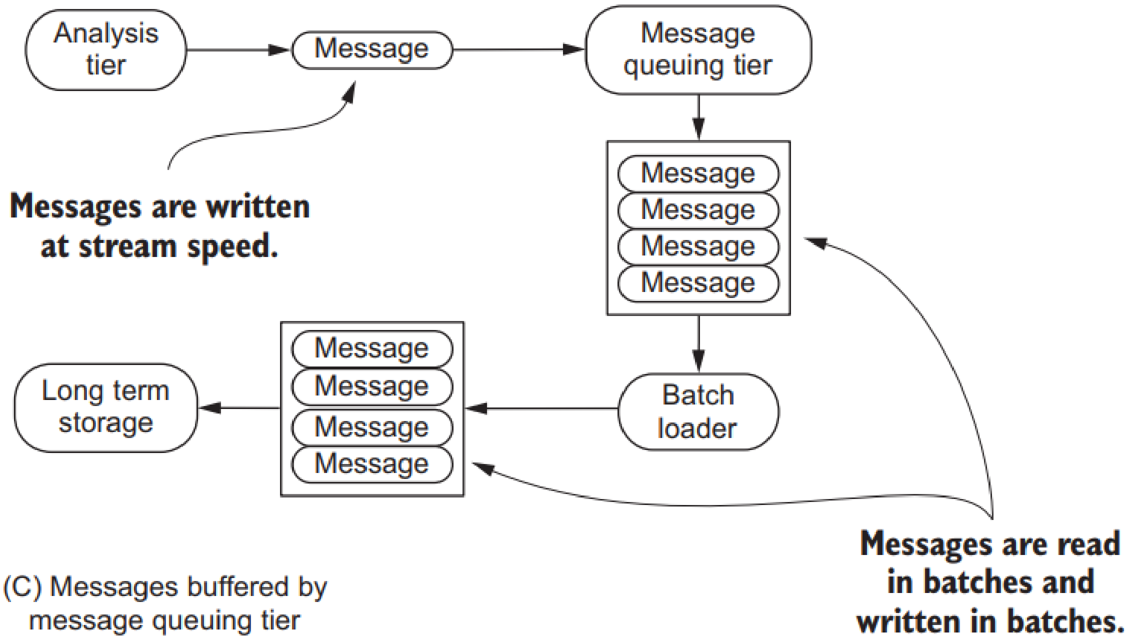

Analysis tier
Message
Message queuing tier
Messages are written at stream speed.
Message
Message
Message
Message
Long term storage
Message
Message
Message
Message
Batch loader
Messages are read in batches and written in batches.
(C) Messages buffered by message queuing tier

Figure 16: Saving Streaming Data to Long-Term Storage; from [5, p. 97]

### 3.7.2. Short-Term Storage (In-Memory Approach)

For most of the real-time analysis use cases, most up-to-date data is needed especially if quick and accurate decisions should be performed. Thus, in order to make it possible, we need our data in-memory. Nowadays, the idea of keeping the entire working dataset in-memory is a reality, unlike 10-20 years ago. Some of the typess of memory to store streaming data in-memory are [5, p. 100]:

- embedded in-memory
- caching system
- in-memory database and in-memory data grid

#### 3.7.2.1. Embedded In-Memory

This category includes memory designed to be embedded into the software with focus on a single node. This approach is not acceptable for distributed stream processing since the data is being stored inside of each node making it very hard to fetch again. Some of the solutons include SQLite, RocksDB, LevelDB, Perset etc.

#### 3.7.2.2. Caching System

Some of the caching system strategies to use are, according to [5, p. 101]:

- read-through
- refresh-ahead
- write-through
- write-around
- write-back

Read through strategy means that caching system reads data from a persistent store for an entry not found in the cache. This can have an impact on performance because persistent storage has to be accessed and usually some updates have to be written back to the cache. Refresh ahead refreshes cache with recently accessed data before it is expired, in order to avoid the read-through performance issues. Write through method writes data to the persistent storage when the cache is updated after which these changes become valid. During the write-around strategy, persistent storage is being updated independently of the cache. Write back strategy is very similar to the write-through strategy, but with acknowledging the write process before it even starts. To summarize, every strategy is based on reading temporary storage and, in case of not finding the right chunk of data, it looking into the permanent storage.

#### 3.7.2.3. In-Memory Database

Based on Psaltis [5, p. 105], in-memory databases are used for non-volatile data, using a storage medium instead of DRAM. Although they use a storage medium, they are developed to use RAM memory first and disk second. Disk storage is used for logging and periodic snapshots. In comparison with the in-memory databases, traditional databases use disk first and RAM memory second mindset.

## 3.8. Streaming Data Visualisation

Last step in the overall streaming data pipeline is making data visible to the end users in a most effective and simple way. As Ellis states in his book [4, p. 227], nowadays even web browsers are so advanced that they can handle rich data applications with streaming data connections. Streaming data visualisation makes perfect sense when it comes to need for a quick reaction based on data. Kaur [9] defined data visualisation as "graphical representation of extensive data and information." Streaming data visualisation would be the same, but in a continuous manner in order to respond according to the data rapid data speed.

### 3.8.1. Visualisation Tasks

Users need to get insights into data streams in many real-time applications to make decisions as quickly as posssible. To give an overview of some of the domain problems, three of the examples that Rohrdantz et al. wrote will be shown [10]:

- emergency management and response
- news and stock market
- server administration

Many social phenomena could be observed in a real-time manner using real-time streaming data. One of them, regarding virus tracking, was already explained in this thesis yet there are many others, like power blackouts, hurricanes and storms, attacks and similar. Each of them can be managed better using user-generated data streams. In some scenarios the acceptable monitoring delay is higher and in some of them not that high. Sometimes, a fully automated alerting system is needed in order to provide continuous control.

Second case is the one regarding analysis of server log data, which is critical from a security and business perspective. An early problem detection is very important to any company. Similar like banking transaction fraud detection, suspicious behaviour or attack in the early stage can be detected and prevented. This is the case where historical data in combination with new, incoming data, can be very important and useful in decision making process.

### 3.8.2. Streaming Data Visualisation Challenges

As in every part of the streaming data pipeline, visualisation has its own challenges. Some of them are related to technical and algorithmic problems. Fast algorithms and data structures are needed in order to process the data in a real-time and incrementally. Second part of the challenge is dealing with a constantly growing amount of data and showing the new data in the context of older data. Most of the available algorithms work with time buffers (windowing), which enables micro-batch analysis. One of the very common problems in streaming data visualisation is lack of data within a data window. This problem can be tackled in a few ways. The simplest way is to visualise as-is which, means visuals are going to lack some elements or can even be not very useful at that moment. Second, a more advanced approach would be approximation of the missing data based on the previous window data. The latter one requires mathematical models for calculating and approximation of the missing data.

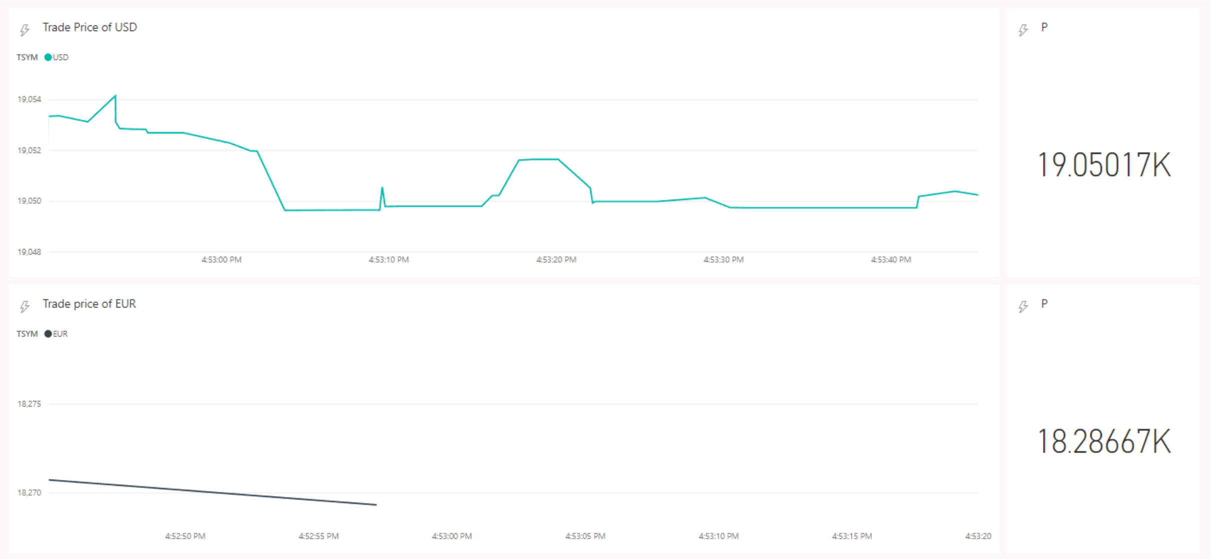


Figure 17: Lack Of Data Within the Stream

Figure 17 shows how lack of data in the stream can cause the visualisation not to be displayed in a proper way with smooth elements.

# 4. Streaming Data Architecture Implementation

This chapter presents an implementation example of streaming data architectures in selected technologies. For the implementation in this thesis, Microsoft cloud technological stack is chosen because this thesis was being done in a company that provided tech stack access and assistance during the preparation of this thesis.

## 4.1. Implementation in Microsoft Azure Technologies

This section introduces at a first place Microsoft Azure Cloud Environment, its fundamental concepts as well as the tools that were used for the implementation of the architectures shown in this thesis.

### 4.1.1. Microsoft Azure Cloud Environment Overview

Senior Technology Editor Stephen Bigelow [11] defines Azure Cloud Environment as "Microsoft's public cloud computing platform." That platform provides a range of cloud services including cloud computational resources, analytics, storage and networking. Using Azure, users can choose either an existing application ready to use (SaaS - Software as a Service) or just the infrastructural services needed to build and scale their applications (PaaS - Platform as a Service and IaaS - Infrastructure as a Service). Very important fact to emphasize is that Azure Cloud is compatible with the other cloud tehchnologies including open source technologies.

Any Azure customer, whether personal or corporate business user, can create an Azure subscription and subscribe to any of the available services. Using these services, any user can create Azure resources which can be e.g. Azure SQL database, Azure data lake, various clusters, virtual machines (VMs) etc.

In Azure, there are five customer support options for users [11]:

1. basic
2. developer
3. standard
4. professional direct
5. premier

Microsoft categorizes Azure services in the following categories [11]:

- **compute** - services that enable user to deploy VMs, containers, jobs,
- **mobile** - environment that provides developers with resources to build mobile cloud applications,

- **web** - environment that provides developers with resources to build and deploy web applications,
- **storage** - services that provide scalable cloud storage for structured and unstructured data,
- **analytics** - services for distributed analytics and storage,
- **networking** - virtual networks, dedicated connections and gateways,
- **content delivery network (CDN)** - streaming digital rights protection
- **integration** - services for backup, site recovery and connecting private and public clouds
- **identity** - services that enables authorization
- **internet of things (IoT)** - services for capturing, monitoring and analyzing IoT data from external devices
- **devops** - project and collaboration services and tools
- **development** - services for development and collaboration
- **security** - services for threat identification and responses
- **artificial intelligence and machine learning** - services for cognitive computing
- **containers** - services for working with containers such as Docker or Kubernetes
- **databases** - database as a service offerings for SQL and NoSQL and other database instances like CosmosDB, PostgreSQL etc.

### 4.1.2. Lambda Architecture Implementation in Azure Using Python, SQL Server and PowerBI

For the first implementation of the streaming data concepts using Azure cloud tools, Microsoft Power BI analytical tool was used. Power BI is Microsoft-built software for data analysis, business intelligence and data visualisation. It is mainly used for creating reports and dashboards, but it also has possibilities for creating real-time data streaming dashboards.

Power BI was used in the first implementation to demonstrate how streaming data works at the very top level, having in mind only data producer (data source), analytical element and visualisation element (both Power BI).

Figure 18 shows a hot path of the architecture, the part closer to the Kappa architecture, based only on continuous datastream with possible but not mandatory data retention. In this case, a Python script generates the data and acts as the data source although the architecture data source separated, to illustrate what it would look like in a real scenario. After generating and structuring the data (or after receiving and transforming/structuring the data), the script

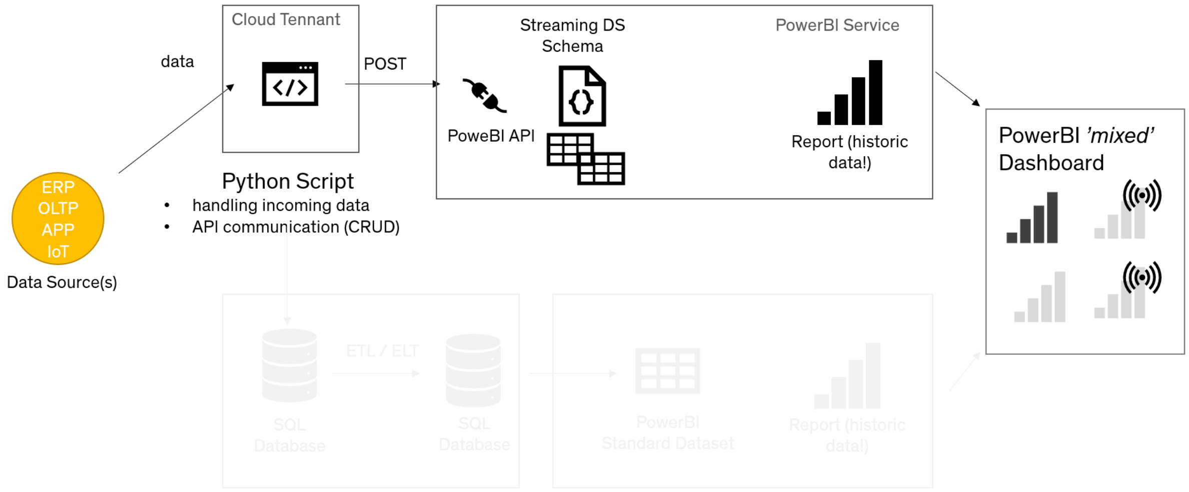


Figure 18: Hot Path Of The Lambda Architecture

sends the data to the Power BI API. Once when the data are in Power BI API, it can be used to create reports or dashboards. Important thing to emphasize is that in this architecture, push datasets receive only the data for one day. After one day, these data are deleted and new data are being received and ingested to the push dataset. Historical data are going to be stored in the SQL database, which is shown later in this thesis.

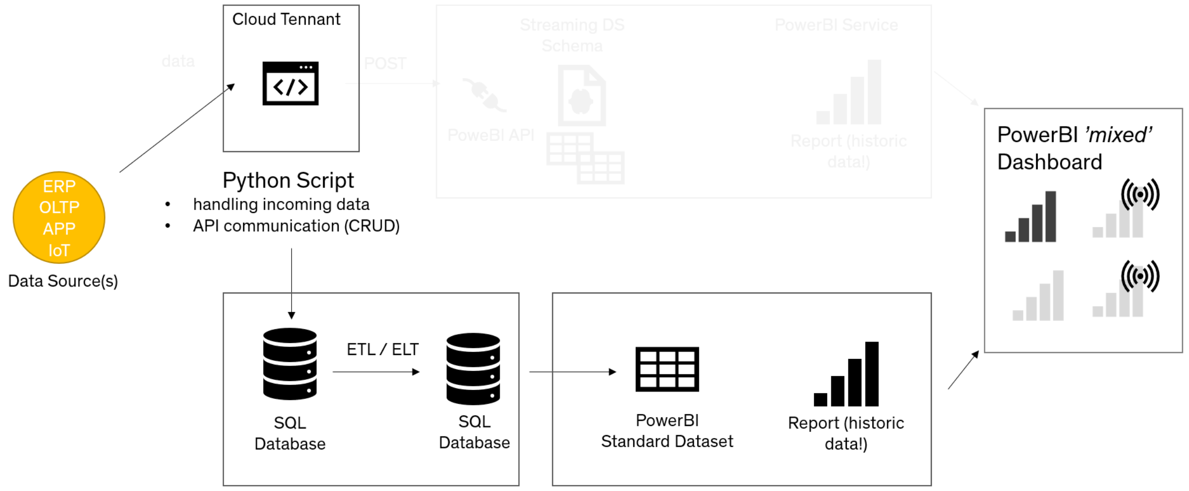


Figure 19: Cold Path Of The Lambda Architecture

Figure 19 shows a cold path of the architecture. Cold path also receives data, but stores them in the SQL database at a first place. Once the data are in the database, they can be used.

On the architecure shown on the figure 19, data are being sent from the SQL database to the Power BI standard dataset. Power BI standard dataset is a fundamental type of the Power BI dataset where the data are not being pushed, but they are being pulled to it. After ingesting the data to that type of the dataset, data can also be consumed by the reports or dashboards.

At the end, there is a Power BI dashboard that contains visuals based on the data from both the cold and the hot path. That means that the dashboard presents real time data for the current day as well as the data for the previous days, weeks, months or even years. Data that are being generated in this example are being sent both to the hot path (pushed to the push dataset) continuously, analyzed and visualised in real-time on the dashboard and to the cold path where data are stored to the database, transformed if needed, pulled into Power BI dataset and then processed in batch and visualised.

#### 4.1.2.1. Streaming Data Source

Generally speaking, real, live streaming data source is not that easy to find as a static one. Most of the streaming data sources are either social media APIs or stock/crypto market APIs. Data source was simulated using only Power BI and Python as a tools. For the purposes of this implementation , a Python script generating domain data was made.

Business case which is base for this data source and implementation is taxy company. Taxy company has a certain number of vehicles and drivers. One taxy ride includes drive ID, driver ID, start and end latitude, start and end longitude, kilometres driven, price of one ride, average speed, average fuel consumption, average engine temperature, start timestamp and weather. It is very important to emphasize type of data in this example called synthetic data. Andrews [12] defined synthetic data as artificial data generated by computer simulations and algorithms as an alternative to real-world data. Synthetic data in this exmaple, as previously mentioned, is generated by Python script. Goal of this example is to show how real-time data can be useful in reporting and monitoring.

The Python script contains a data generator part as well as the part for Power BI API communication. Work principle of the Python script is:

1. generate a random number of daily rides based on the Gauss distribution
2. for each drive
   (a) add a random generated time delta to the previous timestamp to simulate distance between two drives (initial timestamp is generated by taking today's date and setting time to midnight)
   (b) generate data using `DataGenerator(id_drive,date,time)`
   (c) send generated data to the API (POST method)
3. increment day and delete all the data from the push dataset (DELETE method)

Listing 5: JSON structure of the data being sent to the API

```
json_data = {
        "rows":[
            {
                "id_drive": id_drive,
                "id_driver":id_driver,
                "start_lat": dist_lat_long["start_lat"],
                "start_long":dist_lat_long["start_long"],
                "end_lat":dist_lat_long["end_lat"],
                "end_long":dist_lat_long["end_long"],
                "km_driven":distance_km,
                "price":price,
                "avg_speed":avg_speed,
                "avg_fuel_cons":avg_fuel_cons,
                "avg_engi_temp":avg_engine_temp,
                "start_timestamp": date_time,
                "weather": random.choice(["Warm","Hot","Extremely Hot","Heavy Rain",
                 "Moderate Snow","Heavy Snow","Low Rain"])
            }
        ]
    }
```

Listing 5 shows structure of the data that is being sent to the Power BI push dataset API. Python script generates ID of the driver and the ride, random start latitude and longitude as well as end latitude and longitude, kilometers driven in the ride, price of one drive, average speed, fuel consumption, engine temperature, start timestamp and current weather.

#### 4.1.2.2. Power BI Push Dataset API

Data is being ingested in the push dataset through the API. Fundamental API methods for push datasets in the Power BI are [13]:

- `GetToken` (POST) - method for claiming auth token
- `PostDatasetInGroup` (POST) - posts JSON structure of the dataset to the API thus creating a dataset. Main parameter is the group ID
- `PostRowsInGroup` (POST) - ingests rows in a JSON structure to the push dataset through the API
- `DeleteRowsInGroup` (DELETE) - deletes all rows from the dataset. Parameters are group ID, dataset ID and the table name

Connection between the Pyhton script and the Power BI API is as follows: Python script, when started, sends a request to the API to claim bearer authentication token to authenticate to the API. After that, in an infinite loop, Python script generates the data and sends the data to the Power BI API using `PostRowsInGroup` API method.

Listing 6: API call to POST data to the Power BI push dataset API

```
api_endpoint = f'https://api.powerbi.com/v1.0/myorg/datasets/{DATASET_ID}/tables/{
 TABLE_NAME}/rows'
```

```
        header = {"Authorization": BEARER_TOKEN}
        req = requests.post(api_endpoint, json=data_to_send, headers=header)
        print("API Post Status Code: "+""+str(req.status_code))
```

#### 4.1.2.3. PowerBI Streaming Data Processing and Visualisation

Data that arrived in the system either through the cold or the hot path should be processed or visualised in order to give some added value or insights. In this example, processing was implemented in Power BI natively using Power BI DAX[1] language. Some of the visuals were implemented only by selecting the data columns some of which were implemented using DAX.

Figures from 20 to 23 show visuals in the report based on the hot path data. First visualisation shows top drivers by daily revenue and sales share which represents daily share of the revenue by each driver. Share was calculated using the following DAX expression:

Listing 7: Custom DAX metric for calculating single driver share of earned money

```
share_sales = SUM(Drives[price]) / CALCULATE(SUM(Drives[price]),ALL(Drives))
```

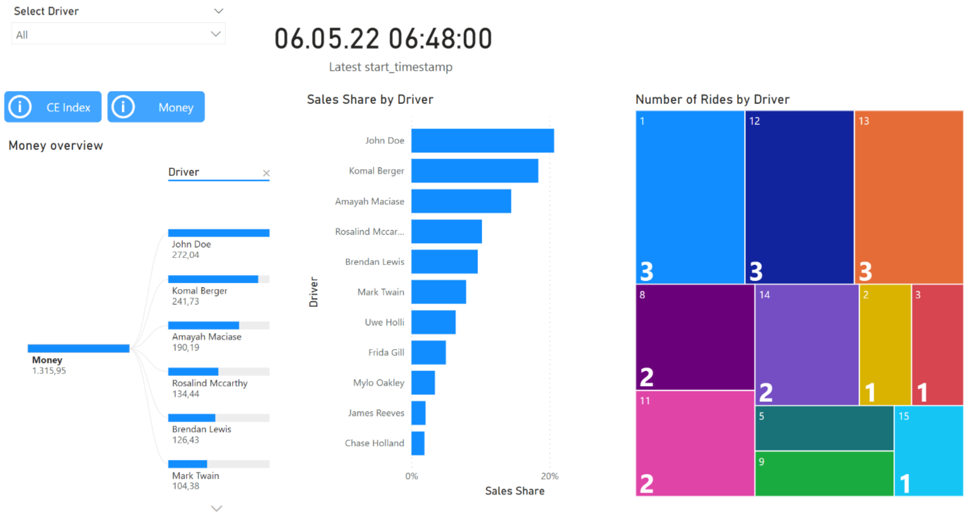


Figure 20: Report visuals showing analyses on results by the quantity of money earned by drivers

Figure 20 shows metrics mentioned above as well as the absolute number of rides by driver.

[1] Data Analysis Expressions - embedded Power BI library with functions and operators for data analysis

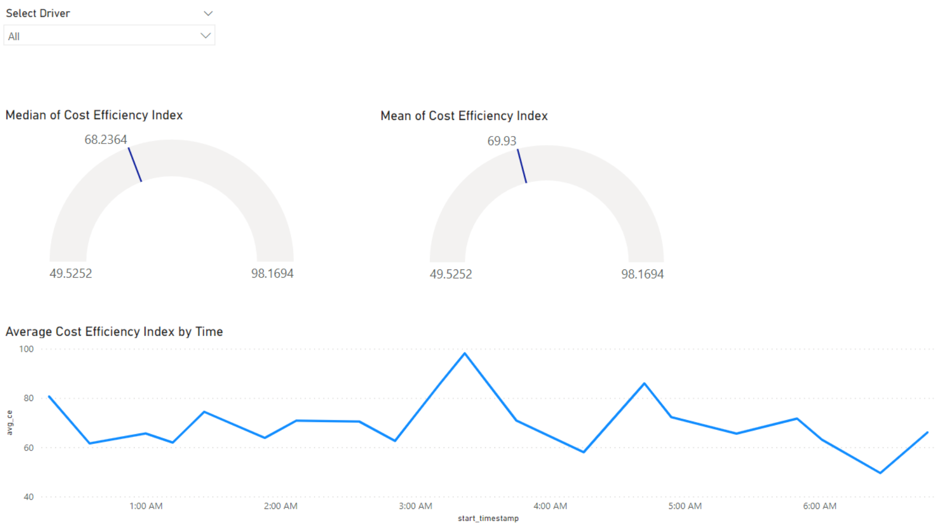


Figure 21: Report visuals showing analysis on cost efficiency

Cost efficiency (CE) is a metric made to demonstrate how non-trivial metrics can be calculated inside the Power BI using DAX. Listing 8 shows DAX metric expression to calculate average cost efficiency on top of the data generated from midnight until the calculation moment.

Listing 8: Custom DAX metric for calculating average CE

```
avg_ce = AVERAGEX(Drives,DIVIDE(Drives[price],(DIVIDE(Drives[avg_fuel_cons],100)*
 Drives[km_driven])))
```

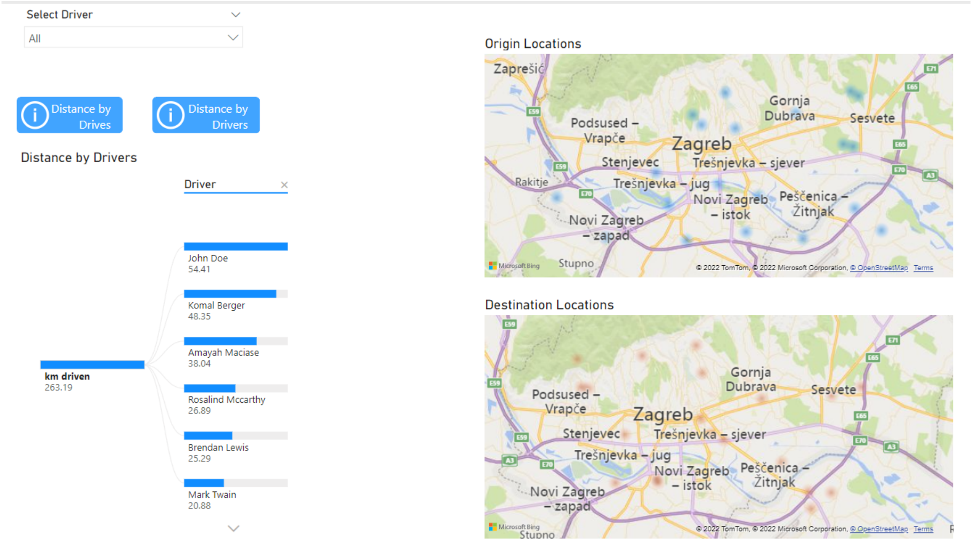


Figure 22: Report visuals showing results of analyses on drivers by the kilometers driven and most frequent pickup and dropoff spots

Figure 22 shows sum of the distances of each driver as well as two heatmaps showing most frequent pick-up and dropoff locations.

Figure 23 shows number of the rides by weather status. Important note to have in mind: since this dataset was artificially made (Python script generator) it is clear on this visual why it maybe makes no sense that maximum number of rides is when the weather is 'extremely hot'.

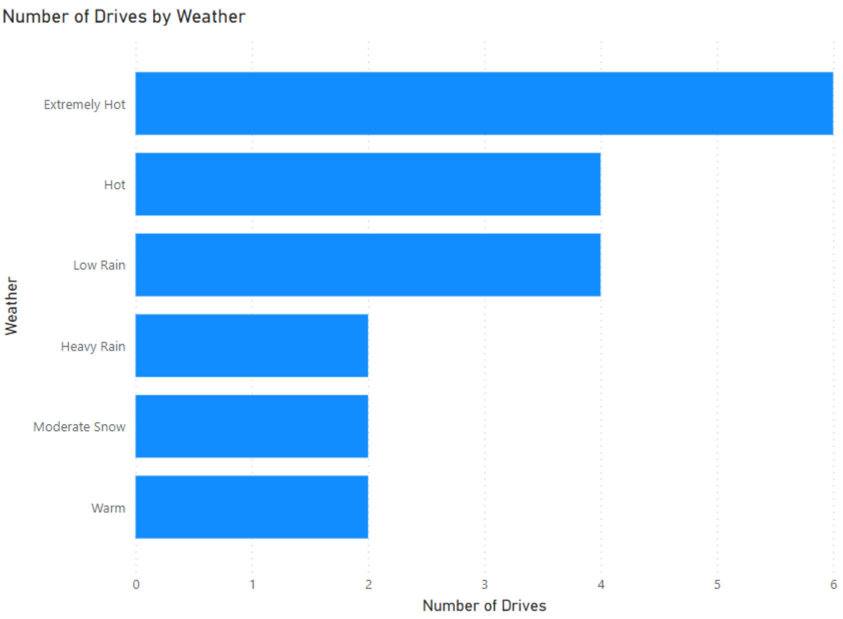


Figure 23: Report visual showing the results of an analysis of the rides by weather

Second report is based on the data arriving to the cold path. These data, as already mentioned above, are being stored in the database and are being used as historical data.

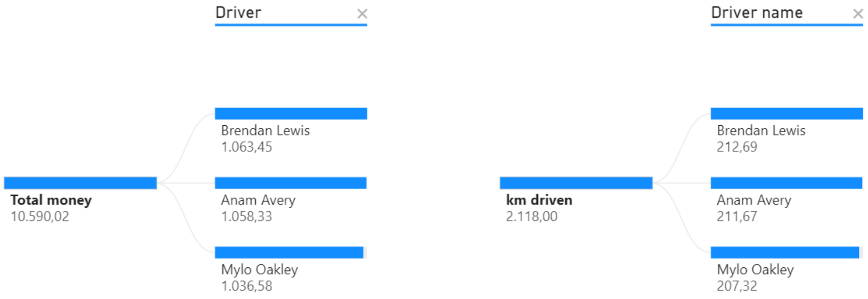
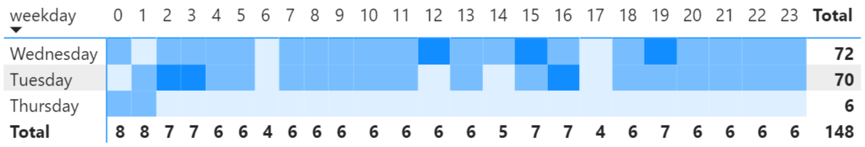


Figure 24: Report visuals showing analyses on drivers by the quantity of money earned and kilometers driven during the period longer than one day

Figure 24 shows a visualisation of the money earned and kilometers driven, but over a period longer than one day. This is possible because of the historical data stored in the database.



| weekday | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Wednesday | | | | | | | | | | | | | | | | | | | | | | | | | 72 |
| Tuesday | | | | | | | | | | | | | | | | | | | | | | | | | 70 |
| Thursday | | | | | | | | | | | | | | | | | | | | | | | | | 6 |
| Total | 8 | 8 | 7 | 7 | 6 | 6 | 4 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 5 | 7 | 7 | 4 | 6 | 7 | 6 | 6 | 6 | 6 | 148 |

Figure 25: Report visual showing analyses on quantity of drives by day and time of the day

Figure 25 shows a matrix that displays the number of rides per day in the week and per hour in the day. Mentioned analysis is based on the data collected over a long term. Darker squares mean higher frequency of the rides in that day and hour, and lighter squares mean lower frequency of the rides. This analysis is possible because of the historical data stored in the database.

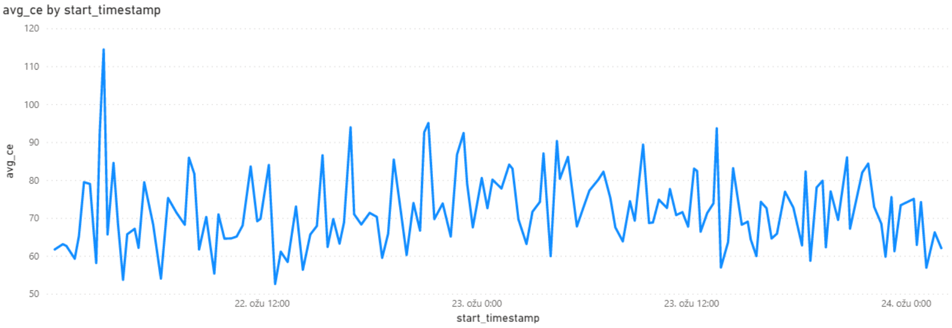
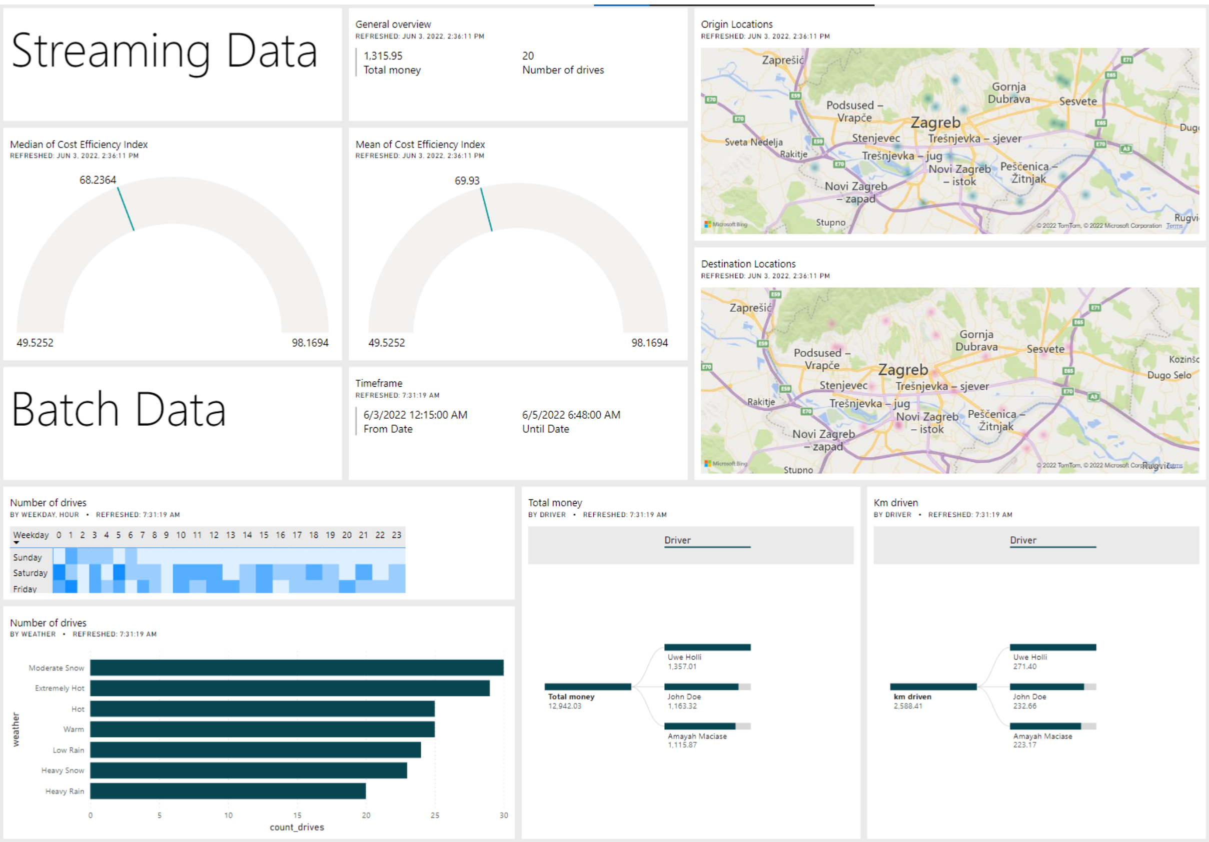


Figure 26: Report visual showing analyses on the CE over a longer period

Figure 26 shows CE index line over a longer period. This analysis possible because of the historical data stored in the database.



Streaming Data

General overview
REFRESHED: JUN 3, 2022, 2:36:11 PM
1,315.95 Total money
20 Number of drives

Median of Cost Efficiency Index
REFRESHED: JUN 3, 2022, 2:36:11 PM
68.2364
49.5252
98.1694

Mean of Cost Efficiency Index
REFRESHED: JUN 3, 2022, 2:36:11 PM
69.93
49.5252
98.1694

Origin Locations
REFRESHED: JUN 3, 2022, 2:36:11 PM

Destination Locations
REFRESHED: JUN 3, 2022, 2:36:11 PM

Batch Data

Timeframe
REFRESHED: 7:31:19 AM
6/3/2022 12:15:00 AM From Date
6/5/2022 6:48:00 AM Until Date

Number of drives
BY WEEKDAY, HOUR • REFRESHED: 7:31:19 AM

Number of drives
BY WEATHER • REFRESHED: 7:31:19 AM

Total money
BY DRIVER • REFRESHED: 7:31:19 AM
Total money 12,942.03
Uwe Holli 1,357.01
John Doe 1,163.32
Amayah Maciase 1,115.87

Km driven
BY DRIVER • REFRESHED: 7:31:19 AM
km driven 2,588.41
Uwe Holli 271.40
John Doe 232.66
Amayah Maciase 223.17

Figure 27: Mixed dashboard with both hot and cold path based visuals

Figure 27 shows final product of this Lambda architecture based pipeline. Upper half of the dashboard shows streaming-wise relevant visuals like average and medial CE index, heatmaps with the pickup and dropoff locations and data about the latest ride. Lower half of the dashboard contains visuals based on the cold path data, which means historical data and analysis over a longer period of time than one day.

It is very important to understand the connection between these two approaches inside of the lambda architecture. First approach, based on the hot path data, is suitable for the metrics based on time-sensitive data and the second one, based on the cold path, is suitable for the metrics that should be based on the historical data collected over a longer period.

### 4.1.3. Kappa Architecture Implementation Using Azure Streaming Data Stack

This subsection presents an implementation of Kappa architecture using Microsoft Azure Cloud tools for streaming data engineering and analytics. In comparison with the previous example, with random generated data and a very high-level architectural approach, the second approach was implemented using more advanced and specialized streaming data tools. These streaming data technologies are state-of-the-art scalable, cloud based tools for handling vast volumes of streaming data in terms of reception, storage and analysis:

- SQL Server
- Event Hubs
- Stream Analytics
- Power BI

#### 4.1.3.1. Azure Event Hub

Azure Event Hubs is a big data streaming platform that exposes a high-throughput endpoint for ingesting and serving event messages [8, p. 93]. Event messages can be understood as records of the application's activities as time-based series of event data. Event Hubs is considered as previously mentioned MQT - tier for message ingestion and propagation.

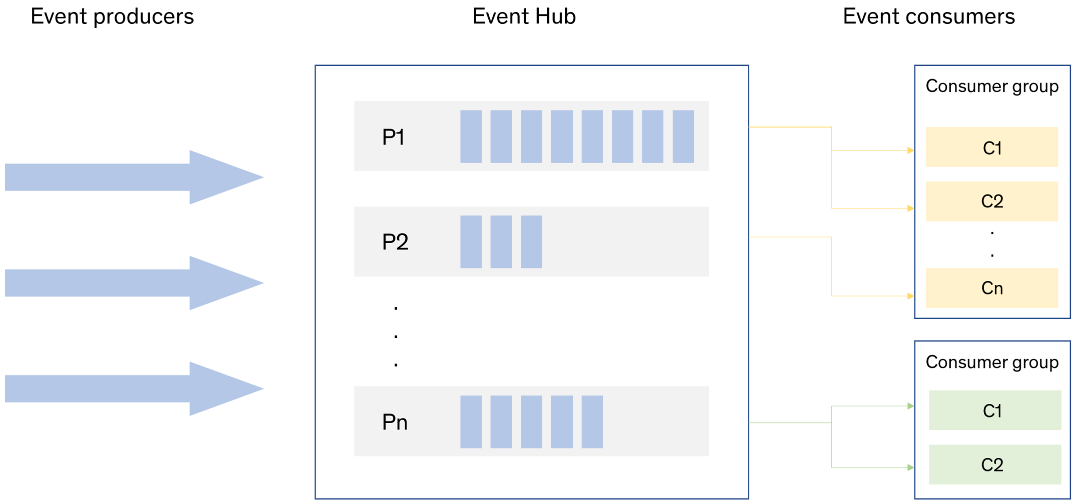


Figure 28: Event hub Schema

Figure 28 shows abstract schema of one event hub instance. Above all, event hubs are placed in a logical container called namespace. One namespace can contain one or more event hubs and acts as an FQDN or Fully Qualified Domain Name endpoint to submit messsages to. Namespaces can also be useful during the outages of the entire namespace by rerouting the traffic to another namespace, if configured. One event hub can contain from minimum 1 up to maximum 32 partitions. Partitions are logical containers inside of an event hub that are part of its load management. They are used as load balancers for the incoming data messages. Events in one partition are ordered chronologically by timestamps. On the left side of event hub are event producers (or event publishers). These can be app services, devices, APIs or something else that produces streaming data or that is a source of the streaming data. On the right side are event consumers (or event subscribers). Those are apps, services or something else that consumes data from the Event Hubs. Consumers can be logically divided into consumer groups, each having its own policy filters, e.g. reading only some of the partitions and similar rules. As Nuckolls stated in his book [8, p. 94], event hubs ingests messages from applications. Main property of the event hubs is that they serve messages on request (consumer request). Multiple consumers can read from the same partition of messages. One partition is also sometimes called a journal of messages. Messages in the Event Hub are retained in the partition until the retention period elapses. Retention period is the time period during which and event hub keeps ingested messages in its own storage. After a retention expires, event hubs deletes all of the messages out of the retention period scope [8, p. 94].

Applications (producers) submit messages to an event hub via:

- API using HTTPS,
- Advanced Message Queuing Protocol (AMQP),
- Apache Kafka open source protocol.

In order to create an event hub using the Azure Portal, it is necessary to create Event Hub Namespace inside of which Event Hubs can be created.

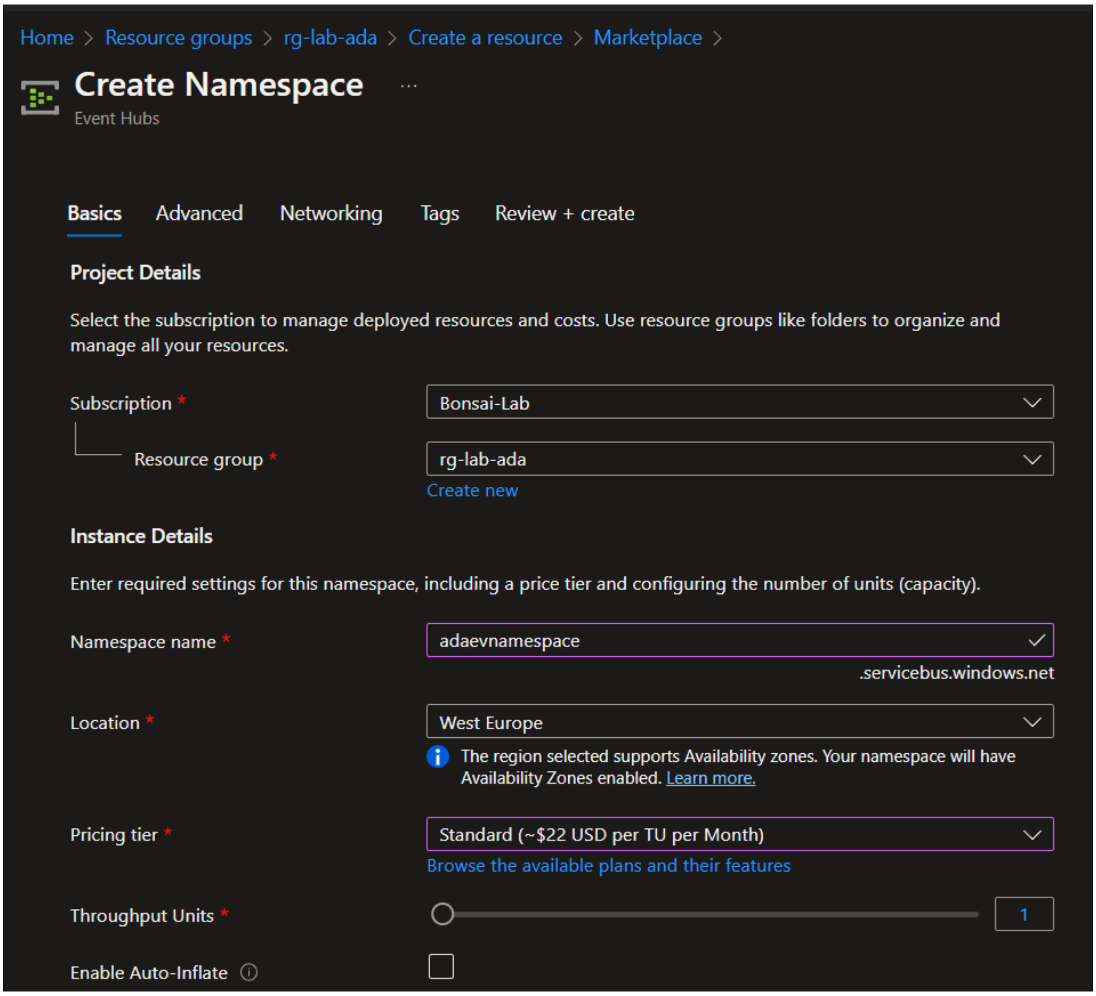
Figure 29 shows the interface for creating an Event Hub namespace. In order to do that, it is necessary to have an Azure Subscription and Resource group. After giving a name, using the Azure portal to the event hub, selecting its resource deployment location, pricing tier and throughput units, an event hub Namespace can be created.

Figure 30 shows the monitoring interface of the Event Hub Namespace showing summary data visuals for all of the Event Hubs deployed in the namespace. It gives an overview of incoming requests, messages and throughput and provides all of the other configurational options for the namespace.

Figure 31 shows the interface for creating an event hub. Event hub instance can be created only inside of a namespace. After giving a name, configuring the number of partitions (journals) and the message retention period, an event hub can be created. There is also a capture option. Capture option enables permanent storage configuration for the Event Hub. That means that incoming data can be stored in permanent storage instead of being disposed after retention period expiration.

#### 4.1.3.2. Azure Stream Analytics

Stream processing implies running an operation on one or more pieces of data either in an endless sequence or time-ordered sequence. The former is called one-at-a-time (real


Home > Resource groups > rg-lab-ada > Create a resource > Marketplace >
Basics
Advanced
Networking
Tags
Review + create

Create Namespace

Event Hubs

Project Details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * : Bonsai-Lab

Resource group * : rg-lab-ada

Create new

Instance Details

Enter required settings for this namespace, including a price tier and configuring the number of units (capacity).

Namespace name * : adaevnamespace ✓ .servicebus.windows.net

Location * : West Europe

The region selected supports Availability zones. Your namespace will have Availability Zones enabled. Learn more.

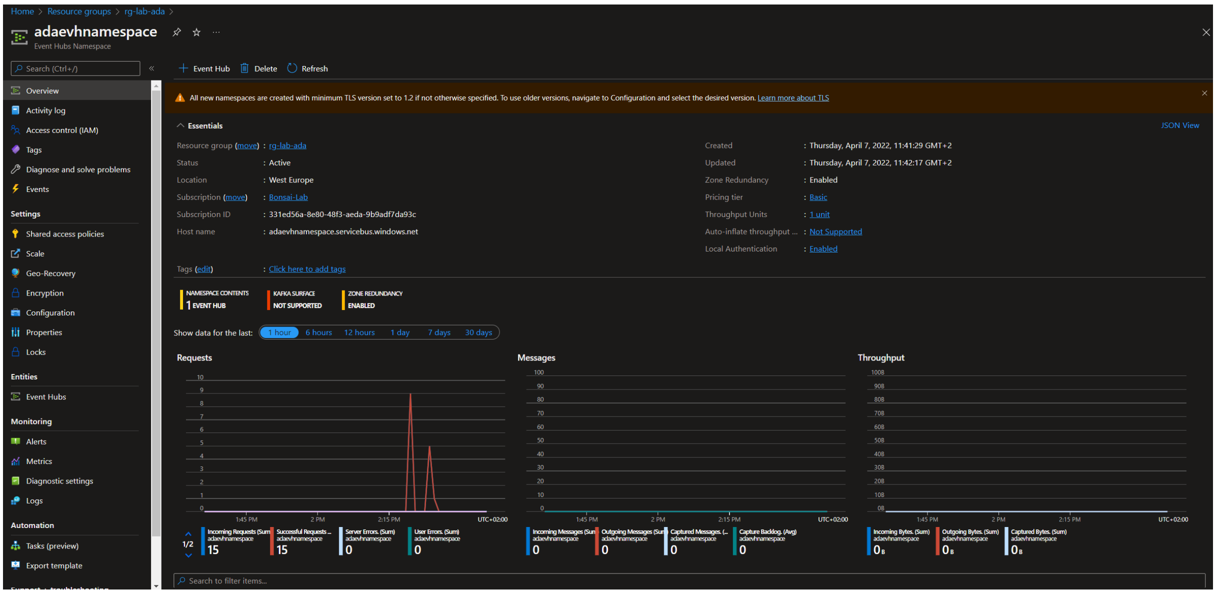
Pricing tier * : Standard (~$22 USD per TU per Month)

Browse the available plans and their features

Throughput Units * : 1

Enable Auto-Inflate ⓘ □

Figure 29: Event Hub Namespace Creation



adaevhnamespace

Event Hubs Namespace

All new namespaces are created with minimum TLS version set to 1.2 if not otherwise specified. To use older versions, navigate to Configuration and select the desired version. Learn more about TLS

Essentials

Resource group (move) : rg-lab-ada

Status : Active

Location : West Europe

Subscription (move) : Bonsai-Lab

Subscription ID : 331ed56a-8e80-48f3-aeda-9b9adf7da93c

Host name : adaevhnamespace.servicebus.windows.net

Tags (edit) : Click here to add tags

Created : Thursday, April 7, 2022, 11:41:29 GMT+2

Updated : Thursday, April 7, 2022, 11:42:17 GMT+2

Zone Redundancy : Enabled

Pricing tier : Basic

Throughput Units : 1 unit

Auto-inflate throughput ... : Not Supported

Local Authentication : Enabled

NAMESPACE CONTENTS 1 EVENT HUB

KAFKA SURFACE NOT SUPPORTED

ZONE REDUNDANCY ENABLED

Show data for the last: 1 hour 6 hours 12 hours 1 day 7 days 30 days

Incoming Requests (Sum) adaevhnamespace 15

Successful Requests ... adaevhnamespace 15

Server Errors. (Sum) adaevhnamespace 0

User Errors. (Sum) adaevhnamespace 0

Incoming Messages (Sum) adaevhnamespace 0

Outgoing Messages (Sum) adaevhnamespace 0

Captured Messages. (... adaevhnamespace 0

Capture Backlog. (Avg) adaevhnamespace 0

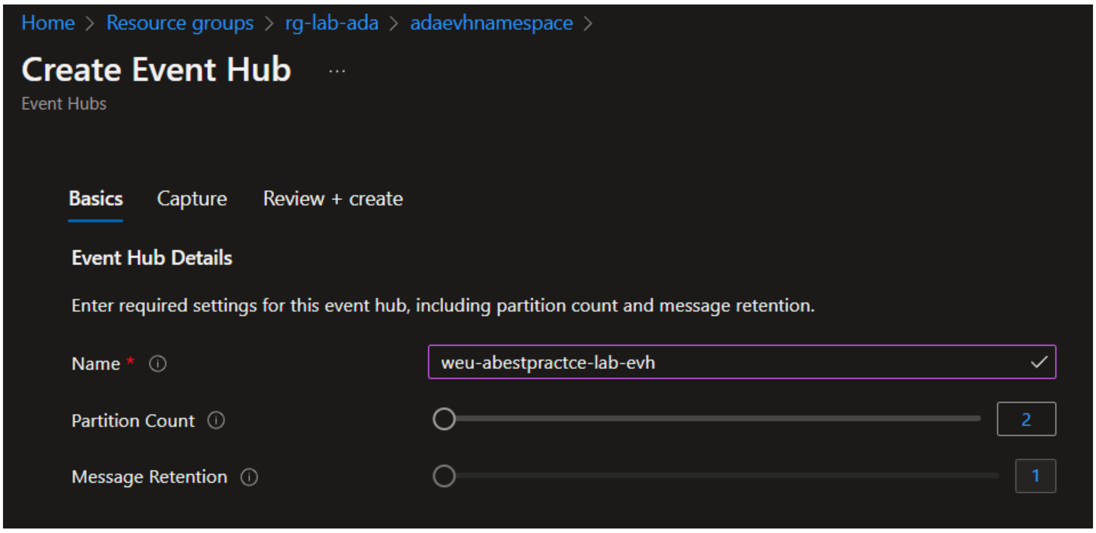
Incoming Bytes. (Sum) adaevhnamespace 0 B

Outgoing Bytes. (Sum) adaevhnamespace 0 B

Captured Bytes. (Sum) adaevhnamespace 0 B

Figure 30: Event Hub Namespace Monitoring Portal

time stream processing) and the latter is micro-batch processing. Stream processors generate results in real time rather than on demand[8, p. 117]. As Nuckolls [8, p. 117] described in his



Create Event Hub

Event Hubs

Event Hub Details

Enter required settings for this event hub, including partition count and message retention.

Name * : weu-abestpractce-lab-evh

Partition Count: 2

Message Retention: 1

Figure 31: Event Hub Instance Creation

book, Azure Stream Analytics (ASA) is a service that reads data sources, executes operations over the data and outputs results to the configured outputs.

For the definition of data operations, ASA uses SQL (Structured Query Language) to execute continuous queries over the stream of data. Instances of the ASA are called jobs and consist of four parts [8, p. 119]:

1. inputs
2. transformations
3. outputs
4. coordination

Inputs are parts of the ASA job that read data into the job, which includes connection to stream sources. Transformations transform input into output and possibly combine input and reference data. Reference data are non-streaming, static data stored into some static storage, whose role is to enrich results of continuous queries. Transformations are performed by writing SQL queries. ASA outputs connect transformed data to the external output sinks. Last part of the ASA job.

Figure 32 shows an abstract schema of one Azure Stream Analytics Job that performs transformations over a stream of data. The yellow stream is called an input stream and represents non-processed data flowing through the stream continuously. Green stream is called an output stream and represents data processed by the ASA job. ASA job, as mentioned earlier, performs continuous predefined SQL queries over a data stream in order to transform data from the input stream to the output stream.

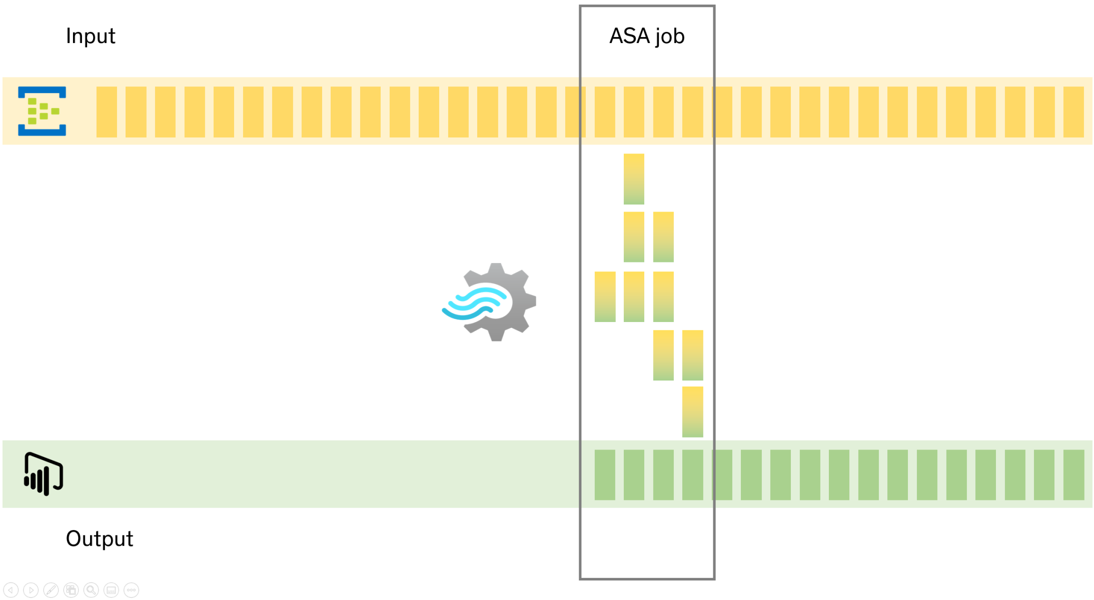


Figure 32: Azure Stream Analytics Job Schema

### 4.1.3.3. Architecture of the Solution

Architectural approach that is implemented using these technologies is, the previously mentioned, Kappa streaming architecture. Kappa does not have to include data persistence and it is a purely streaming architecture. Architecture of this solution consists of, as 33 shows, the data source which is an API that sends the data to the MQT via websocket protocol, an MQT, in this case the Azure Event Hub, an Azure Stream Analytics instance which connects to the Event Hub and receives the data stream from it, and a Power BI tool that offers streaming dashboard creation, while connected to the Azure Stream Analytics job, which outputs the data into the Power BI.

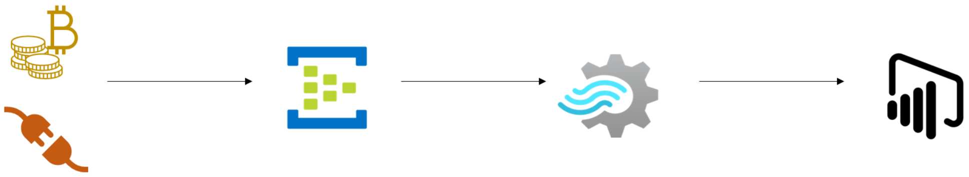


Figure 33: Solution Kappa Architecture

### 4.1.3.4. Streaming Data Source

The data source used in this example is CryptoCompare streaming API. CryptoCompare provides free REST and WebSocket APIs under predefined rules. Websocket protocol is a kind of continuous streaming protocol mentioned in section 3.4.2.5. For the purposes of this implementation, CryptoCompare streaming API is used. Once connected, users can subscribe and unsubscribe to channels. API key is needed to start a data stream. A user can connect to the websocket API using the following endpoint: `wss://streamer.cryptocompare.com/v2/`. Once connected, the user can subscribe to the desired channels using the subscription ID. Subscription message looks as follows:

Listing 9: Cryptocompare Subscription Message

```
{
    "action": "SubAdd",
    "subs": ["5~Coinbase~BTC~USD", "0~Coinbase~ETH~USD"]
}
```

Listing 10: Cryptocompare Unsubscribe Message

```
{
    "action": "SubRemove",
    "subs": ["5~Coinbase~BTC~USD", "0~Coinbase~ETH~USD"]
}
```

When sending a request, what is being sent is an array with subscriptions. For the purposes of this implementation based on free tier API access, few of the possible subscription options were taken.

Python script is used to coordinate all tasks for connection between the WebSocket API and the Azure Event Hub service. Python snippet for connecting to the CryptoCompare streaming API via WebSocket protocol looks as shown in listing 11:

Listing 11: Python Snippet to Connect to CryptoCompare Streaming API via WebSocket (from: [14])

```python
import asyncio
import json
import websockets

async def cryptocompare():
    api_key = "{your_api_key}"
    url = "wss://streamer.cryptocompare.com/v2?api_key=" + api_key
    async with websockets.connect(url) as websocket:
        await websocket.send(json.dumps({
            "action": "SubAdd",
            "subs": ["0~Coinbase~BTC~USD"],
        }))
        while True:
            try:
                data = await websocket.recv()
            except websockets.ConnectionClosed:
                break
            try:
                data = json.loads(data)
                print(json.dumps(data, indent=4))
            except ValueError:
                print(data)

asyncio.get_event_loop().run_until_complete(cryptocompare())
```

#### 4.1.3.5. Processing and Visualising Streaming Data

Data stream processing requires a bit different query writing approach to process data than the static data, although most of the concepts (in this case SQL) are similar. This thesis already elaborated processing in terms of batch and stream processing as well as in terms of windowing. To start with the example, an Azure Stream Analytics job has to be created inside of an Azure Stream Analytics resource in Azure. After creating one, job topology has to be configured. As previously mentioned, ASA job topology includes [8, p. 119]:

- input
- query
- output

That means that one ASA job consists of at least one input (in this case it is event hub MQT), at least one query and at least one output. The simplest ASA SQL query is called "pass-through query". It simply selects everything from the input and sends it to the output.

Listing 12: Simple SQL pass-through query (from: [15])

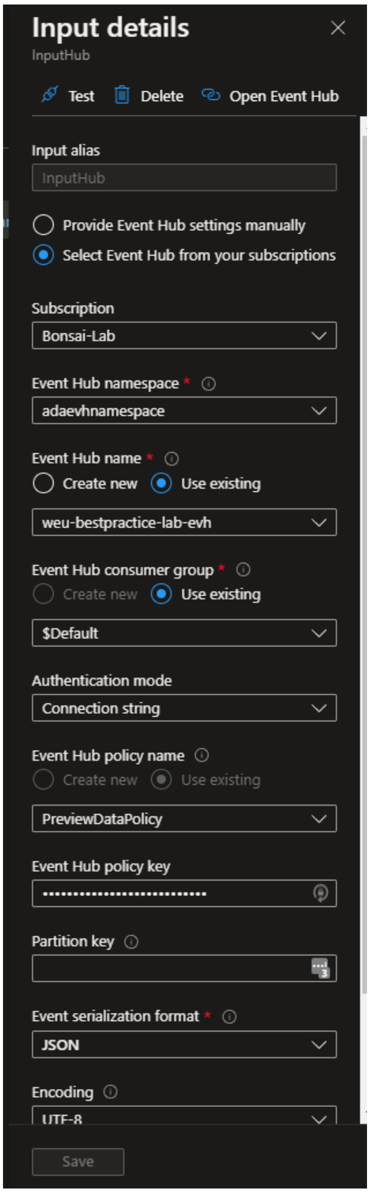
```
SELECT *
INTO Output
FROM Input
```

Before tackling with the queries, data structure of the incoming messages should be well understood. For this particular case, every message ingested into the ASA job has the following attributes:

- TYPE - type of the message (e.g. 0 = crypto trade message, 1 = crypto ticker message...)
- M - exchange market (e.g. Coinbase)
- FSYM - mapped from asset (e.g. BTC)
- TSYM - mapped to asset (e.g. USD)
- F - flag for the trade (e.g. 1 = SELL, 2 = BUY...)
- ID - trade ID
- TS - timestamp in seconds
- Q - from asset volume of the trade
- P - price in the to asset
- TOTAL - total volume in the to asset of the trade (Q * P; how much e.g. USD was paid in total for the volume of BTC traded)
- RTS - timestamp in seconds when trave is received
- CCSEQ - internal sequence number for the trade
- TSNS - nanosecond part of the reported timestamp
- RTSNS - nanosecond part of the received timestamp

For the purposes of this processing example, six ASA SQL queries will be presented. After creating an ASA job, it should have input and output configured. Since this architecture is developed in Azure technologies, it is very easy to deploy and configure ASA job as well as to connect Event Hub and Power BI with the job.

Figure 34 shows an ASA job input configuration and connection. After selecting Azure subscription, Event Hub namespace and Event Hub instance followed by consumer group and authentication mode, input is configured.



Input details
InputHub

Input alias
InputHub

○ Provide Event Hub settings manually
● Select Event Hub from your subscriptions

Subscription
Bonsai-Lab

Event Hub namespace *
adaevhnamespace

Event Hub name *
○ Create new ● Use existing
weu-bestpractice-lab-evh

Event Hub consumer group *
○ Create new ● Use existing
$Default

Authentication mode
Connection string

Event Hub policy name
○ Create new ● Use existing
PreviewDataPolicy

Event Hub policy key
••••••••••••••••••••••••••

Partition key

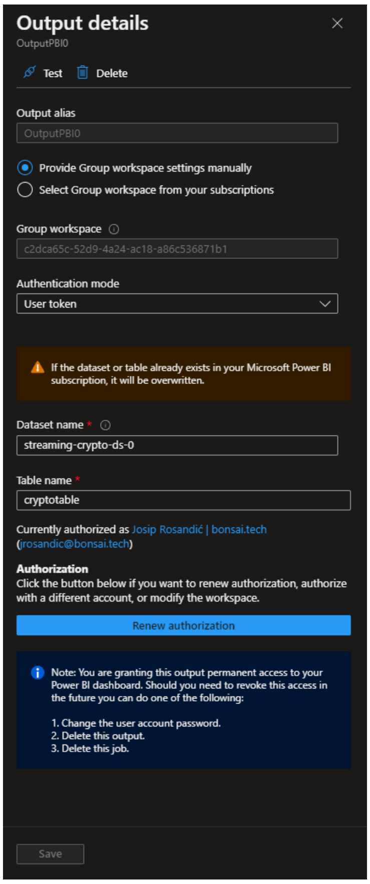
Event serialization format *
JSON

Encoding
UTF-8

Figure 34: Stream Analytics Input Configuration

Figure 35 shows an ASA job output configuration and connection. In this case, since output is going to be Power BI, output type and configuration is also for Power BI. To fill all of the required fields in the configuration, group workspace ID, user token for authentication and dataset and table name have to be provided. Power BI is going to automatically create a dataset and the relevanttable after starting ASA job and receiving inputs.

Central part of an ASA job are queries. Queries receive data from inputs, process that data sending the results to the output. For the purposes of this thesis, six queries were written in order to send processed data to the Power BI for visualisation. Since this implementation is Kappa architecture based, there will be no permanent storage for the data although it can be added either to the database from the Event Hubs or from ASA job to the database.



Output details

OutputPBI0

Output alias: OutputPBI0

● Provide Group workspace settings manually

○ Select Group workspace from your subscriptions

Group workspace: c2dca65c-52d9-4a24-ac18-a86c536871b1

Authentication mode: User token

If the dataset or table already exists in your Microsoft Power BI subscription, it will be overwritten.

Dataset name *: streaming-crypto-ds-0

Table name *: cryptotable

Currently authorized as Josip Rosandić | bonsai.tech (jrosandic@bonsai.tech)

**Authorization**
Click the button below if you want to renew authorization, authorize with a different account, or modify the workspace.

Note: You are granting this output permanent access to your Power BI dashboard. Should you need to revoke this access in the future you can do one of the following:

1. Change the user account password.
2. Delete this output.
3. Delete this job.

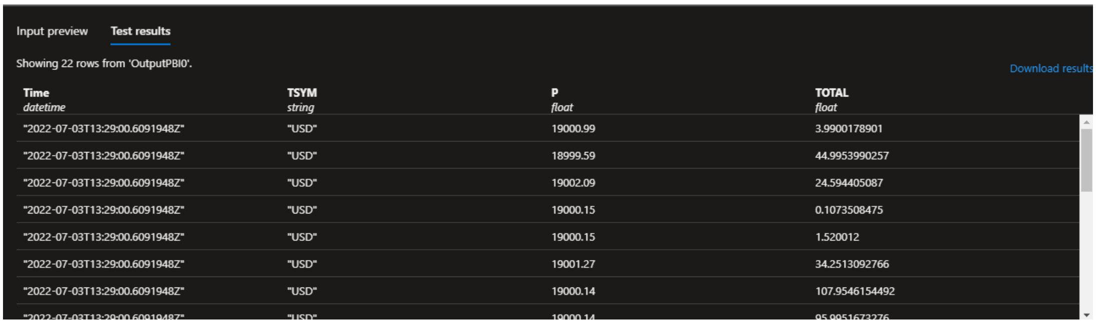
Figure 35: Stream Analytics Output Configuration

### ASA SQL Queries

Query number 0, shown on listing 13, creates dataset projection on Time, TSYM, P and TOTAL attributes, with selection condition TSYM = ’USD’. That means that query returns attribute values for Time, TSYM, P and TOTAL with respect to the condition that TSYM value must be ’USD’. Result of the query is shown on figure 36. This query simply fetches the data based on selected condition and outputs it to the configured output.

Listing 13: Query 0

```
SELECT
    EventProcessedUtcTime as Time, TSYM, P, TOTAL
INTO
    OutputPBI0
FROM
    InputHub
WHERE
    TSYM = ’USD’
```



Showing 22 rows from 'OutputPBI0'.

| Time<br>*datetime* | TSYM<br>*string* | P<br>*float* | TOTAL<br>*float* |
|---|---|---|---|
| "2022-07-03T13:29:00.6091948Z" | "USD" | 19000.99 | 3.9900178901 |
| "2022-07-03T13:29:00.6091948Z" | "USD" | 18999.59 | 44.9953990257 |
| "2022-07-03T13:29:00.6091948Z" | "USD" | 19002.09 | 24.594405087 |
| "2022-07-03T13:29:00.6091948Z" | "USD" | 19000.15 | 0.1073508475 |
| "2022-07-03T13:29:00.6091948Z" | "USD" | 19000.15 | 1.520012 |
| "2022-07-03T13:29:00.6091948Z" | "USD" | 19001.27 | 34.2513092766 |
| "2022-07-03T13:29:00.6091948Z" | "USD" | 19000.14 | 107.9546154492 |
| "2022-07-03T13:29:00.6091948Z" | "USD" | 19000.14 | 95.9951673276 |

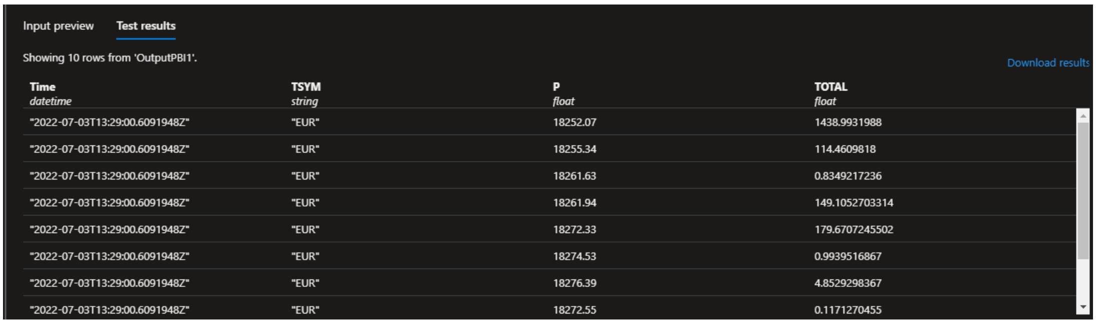
Figure 36: Result preview of query 0

The first query shown on listing 14 calculates dataset projection on Time, TSYM, P and TOTAL attributes with selection condition TSYM = ’EUR’. That means that query returns attribute values for Time, TSYM, P and TOTAL with respect to the condition that TSYM value must be ’EUR’. Result of the query is shown on figure 37.

Listing 14: Query 1

```
SELECT
    EventProcessedUtcTime as Time, TSYM, P, TOTAL
INTO
    OutputPBI1
FROM
    InputHub
WHERE
    TSYM = ’EUR’
```

The second query, shown on listing 15, calculates dataset projection on Time, TSYM and AVG aggregation on P attributes grouped by tumbling window with duration of 5 seconds, TSYM and Time. That means that query returns attribute values for Time, TSYM and calculated average value over P attribute with respect to the condition that calculation has to be done within a Tumbling Window whose duration is 5 seconds. Result of the query is shown on figure 38.



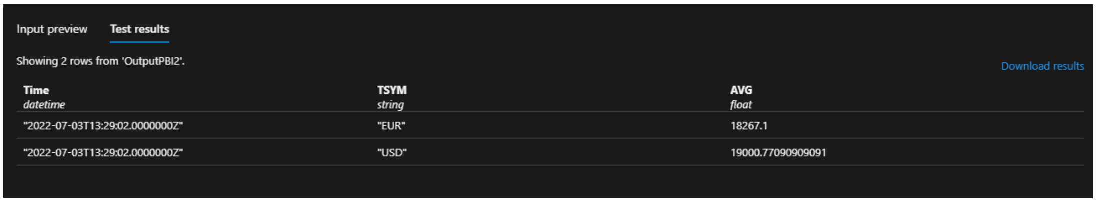
Showing 10 rows from 'OutputPBI1'.

| Time<br>*datetime* | TSYM<br>*string* | P<br>*float* | TOTAL<br>*float* |
|---|---|---|---|
| "2022-07-03T13:29:00.6091948Z" | "EUR" | 18252.07 | 1438.9931988 |
| "2022-07-03T13:29:00.6091948Z" | "EUR" | 18255.34 | 114.4609818 |
| "2022-07-03T13:29:00.6091948Z" | "EUR" | 18261.63 | 0.8349217236 |
| "2022-07-03T13:29:00.6091948Z" | "EUR" | 18261.94 | 149.1052703314 |
| "2022-07-03T13:29:00.6091948Z" | "EUR" | 18272.33 | 179.6707245502 |
| "2022-07-03T13:29:00.6091948Z" | "EUR" | 18274.53 | 0.9939516867 |
| "2022-07-03T13:29:00.6091948Z" | "EUR" | 18276.39 | 4.8529298367 |
| "2022-07-03T13:29:00.6091948Z" | "EUR" | 18272.55 | 0.1171270455 |

Figure 37: Result preview of query 1

Listing 15: Query 2

```sql
SELECT
    EventProcessedUtcTime as Time, TSYM, AVG(P)
INTO
    OutputPBI2
FROM
    InputHub
GROUP BY
    TumblingWindow(s,5), TSYM, EventProcessedUtcTime
```



Showing 2 rows from 'OutputPBI2'.

| Time<br>*datetime* | TSYM<br>*string* | AVG<br>*float* |
|---|---|---|
| "2022-07-03T13:29:02.0000000Z" | "EUR" | 18267.1 |
| "2022-07-03T13:29:02.0000000Z" | "USD" | 19000.77090909091 |

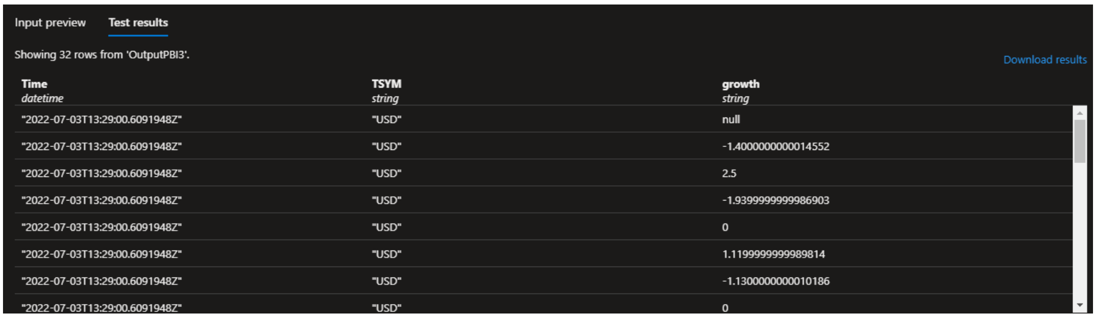
Figure 38: Result preview of query 2

The third query shown on listing 16, calculates dataset projection on Time, TSYM, and calculated attribute "growth" using LAG(<attr>) function. That means that query returns attribute values for Time, TSYM and calculated growth attribute. The LAG function looks at past events within the same time window and compares them against the current event. Result of the query is shown on figure 39.

Listing 16: Query 3

```sql
SELECT
    System.TimeStamp() as Time, TSYM, growth = P - LAG(P) OVER (PARTITION BY TSYM
     LIMIT DURATION(hour, 1))
INTO
    OutputPBI3
FROM
    InputHub
```

The fourth query shown on listing 17, calculates dataset projection on F, count of F and time grouped by tumbling window with duration of 5 seconds, F and Time. That means that query returns attribute values for Trade, calculated count over attribute F and Time with respect to the condition that calculation has to be done within a Tumbling Window whose duration is



Showing 32 rows from 'OutputPBI3'.

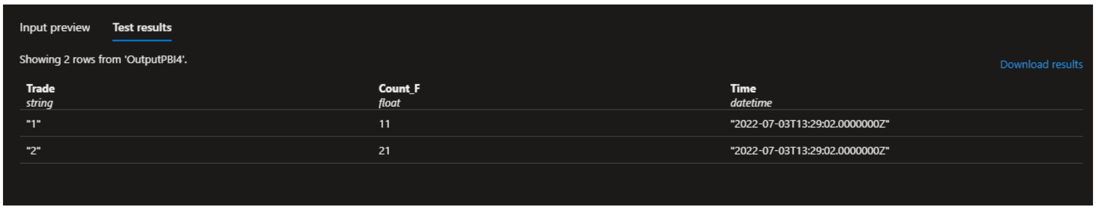
| Time<br>*datetime* | TSYM<br>*string* | growth<br>*string* |
|---|---|---|
| "2022-07-03T13:29:00.6091948Z" | "USD" | null |
| "2022-07-03T13:29:00.6091948Z" | "USD" | -1.4000000000014552 |
| "2022-07-03T13:29:00.6091948Z" | "USD" | 2.5 |
| "2022-07-03T13:29:00.6091948Z" | "USD" | -1.9399999999986903 |
| "2022-07-03T13:29:00.6091948Z" | "USD" | 0 |
| "2022-07-03T13:29:00.6091948Z" | "USD" | 1.1199999999989814 |
| "2022-07-03T13:29:00.6091948Z" | "USD" | -1.1300000000010186 |
| "2022-07-03T13:29:00.6091948Z" | "USD" | 0 |

Figure 39: Result preview of query 3

2 seconds along with the grouping by attributes F and Time. Result of the query is shown on figure 40.

Listing 17: Query 4

```sql
SELECT
     F as Trade, COUNT(F) AS Count_F, EventProcessedUtcTime AS Time
INTO
    OutputPBI4
FROM
    InputHub
GROUP BY
     TumblingWindow(second, 2), F, EventProcessedUtcTime
```



Showing 2 rows from 'OutputPBI4'.

| Trade<br>*string* | Count_F<br>*float* | Time<br>*datetime* |
|---|---|---|
| "1" | 11 | "2022-07-03T13:29:02.0000000Z" |
| "2" | 21 | "2022-07-03T13:29:02.0000000Z" |

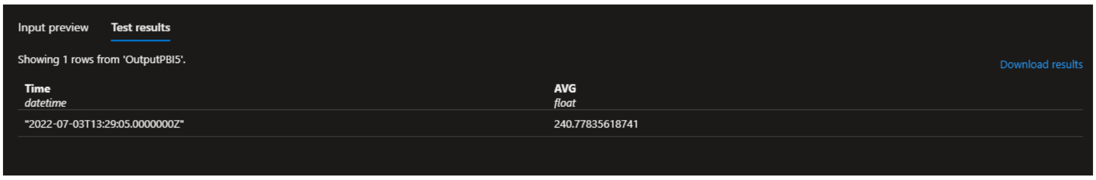
Figure 40: Result preview of query 4

Fifth query shown on listing 18 calculates dataset projection on Time and aggregate function average of TOTAL with selection condition TSYM = ’USD’ and grouped by tumbling window with duration of 5 seconds. That means that query returns attribute values for Time, TSYM and calculated average value over TOTAL attribute with respect to the condition that TSYM value must be ’USD’ and calculation has to be done within a Tumbling Window whose duration is 5 seconds along with the grouping by Time. Result of the query is shown on figure 39.

Listing 18: Query 5

```sql
SELECT
    EventProcessedUtcTime as Time, AVG(TOTAL)
INTO
    OutputPBI5
FROM
    InputHub
WHERE
    TSYM = ’USD’
```

```
GROUP BY
    TumblingWindow(s,5), EventProcessedUtcTime
```



Showing 1 rows from 'OutputPBI5'.

| Time<br>*datetime* | AVG<br>*float* |
|---|---|
| "2022-07-03T13:29:05.0000000Z" | 240.77835618741 |

Figure 41: Result preview of query 5

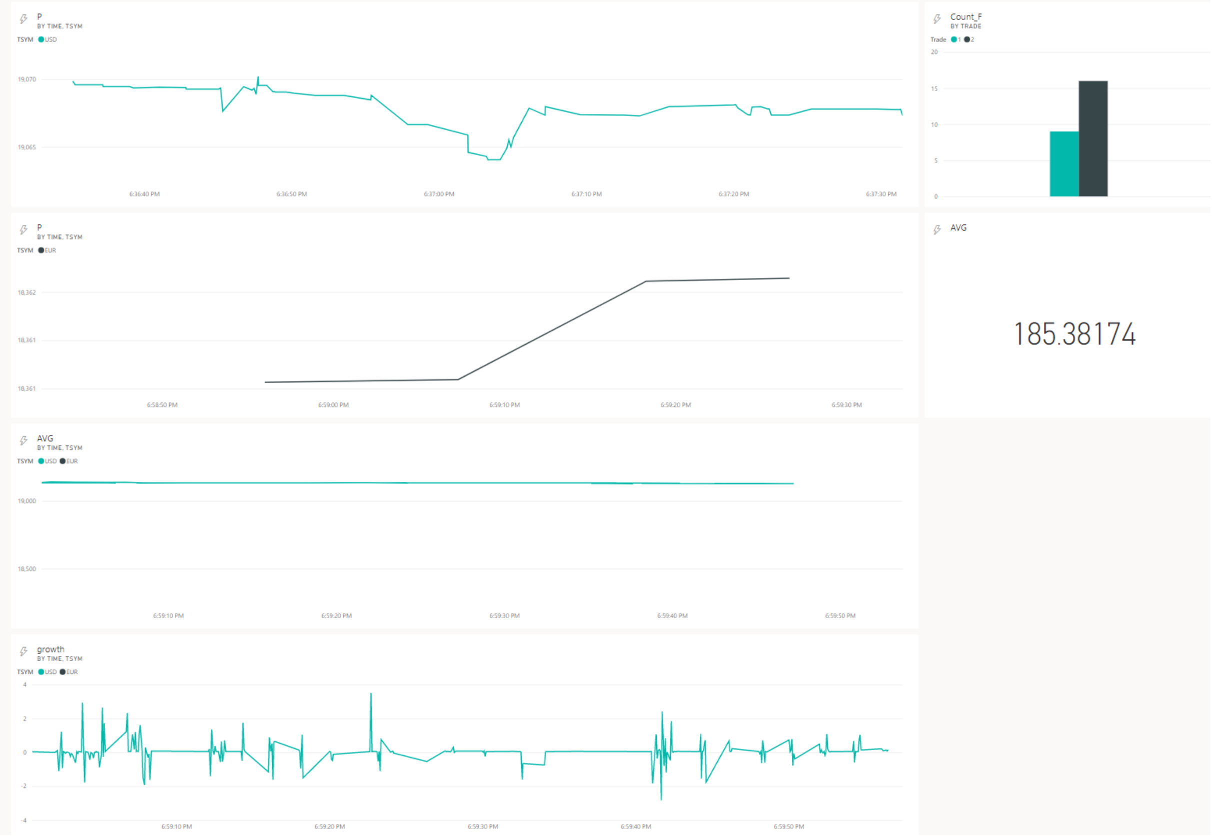


Figure 42: Streaming Dashboard

Figure 42 shows the final streaming dashboard made in Power BI service. Dashboard shows mainly line charts displaying various data from the data stream based on queries. Dashboard receives data constantly, refreshing all of the visuals automatically. Of course, if there is a lack of data for the particular element of the stream, visualisation cannot be drawn properly, which is one of the challenges in streaming data visualisation.

# 5. Conclusion

To summarize, data streams are continuous, always-on, unpredictable by their dynamic and can be complex to work with. Purpose of a data stream is to deliver processed information as soon as possible from the data source all the way to the consumers to bring them added value. It has to be done through the phases of classic data stream architecture: from the data source, through the MQT (Message Queuing Tier), through the stream processor the real-time visuals. To solve most of the challenges that streaming data puts on the table, data engineers created architectural patterns that are used to collect, process and visualise data streams. This thesis explained two of these architectural approaches - Lambda and Kappa. Lambda includes persistent storage like relational a database, while Kappa can include it, but it is not mandatory.

Data streams are being generated from many sources, some of which are mobile devices, social media, financial institutions, websites and web in general, various IoT devices etc. Since these sources generate data in a very fast manner, we can also say that streaming data are large-scale data, thus streaming architectures should be truly robust. The processing approach is similar to the non-streaming data, but it introduces some concepts like windows, which are not considered at all when processing batches of data.

Speaking of the chosen technology - Microsoft Azure Cloud technologies, all of the cloud services are ready to be deployed and configured in a few minutes and at the end used to build data-oriented architectures. The main difference between Azure Streaming stack and technologies like Apache streaming tech stack is that Azure doesn't need any extensive setup or configuration which means that MQT or stream processor can be deployed in a moment, without writing any explicite code. This claim has two sides. On the one side, that is useful to compose the streaming architecture quickly and maybe in a bit simpler way. Less acceptable fact is that proprietary software, such as Azure tools, is not customizable and is a bit harder to manoeuvre. On the other hand, open source technologies provide a kind of freedom when developing the architecture in terms of configuration and setup. Proprietary technologies are strictly limited to their already implemented functionalities which means that they are not customizable in the way some open source solutions are. This difference is mostly present during the visualisation, because all of the visuals are predefined and closed to any customization.

Data stream architectures and technologies are a very engaging data engineering topic to explore more about in order to tame data streams. Data streams can exist and be used without being utilized to their fullest, which is another thing to think about and to work on - how to recognize useful properties of the stream in order to deliver useful and worthful conclusions and to maximize data stream value.

# Bibliography

[1] W. Wingerath, F. Gessert, and N. Ritter, *Real-Time & Stream Data Management: Push-Based Data in Research & Practice*, 1st ed. 2019, ser. SpringerBriefs in Computer Science. Cham: Springer International Publishing : Imprint: Springer, 2019, ISBN: 978-3-030-10555-6.

[2] *Stream*, English, Cambridge, UK, 05/2022.

[3] S. Haines, *Modern Data Engineering with Apache Spark: A Hands-On Guide for Building Mission-Critical Streaming Applications*, English. San Jose, CA, USA: Apress, a Springer Nature company, 2022, ISBN: 9781484274521 OCLC: 1310247043, ISBN: 978-1-4842-7452-1.

[4] B. Ellis, *Real-time analytics: techniques to analyze and visualize streaming data*, eng. Indianapolis, IN: Wiley, 2014, ISBN: 978-1-118-83793-1 978-1-118-83791-7.

[5] A. G. Psaltis, *Streaming data: understanding the real-time pipeline*. Shelter Island, NY: Manning Publications, 2017, OCLC: ocn952385842, ISBN: 978-1-61729-228-6.

[6] E. Price, *Big data architectures - Azure Architecture Center*, en-us, Documentation.

[7] M. Hammad, *Modularity and its Properties*, en-us, Computer Science, Section: Software Engineering, 06/2020.

[8] R. L. Nuckolls, *Azure storage, streaming, and batch analytics: a guide for data engineers*. Shelter Island, NY: Manning Publications Co, 2020, ISBN: 978-1-61729-630-7.

[9] J. Kaur, *Real-Time Streaming Data Visualizations*, en, 02/2021.

[10] C. Rohrdantz, D. Oelke, M. Krstajić, and F. Fischer, "Real-Time Visualization of Streaming Text Data: Tasks and Challenges," 01/2011.

[11] S. Bigelow, *What is Microsoft Azure and How Does It Work?* en, Technological News.

[12] G. Andrews, *What is synthetic data?* en-US, 06/2021.

[13] rloutlaw, *Push Datasets - REST API (Power BI Power BI REST APIs)*, en-us.

[14] *Websockets Documentation | CryptoCompare Cryptocurrency Price Streaming Service for Developers*, en.

[15] Fleid, *Common query patterns in Azure Stream Analytics*, en-us.

# List of Figures

# List of Tables

# List of Listings