

UNIVERSITY OF TARTU  
Institute of Computer Science  
Computer Science Curriculum

Maria Kõusvek

**A Monitoring System for Daily Feedback on IoT Sensor's Status in Tartu City**

Bachelor's Thesis (9 EAP)

Supervisor: Amnir Hadachi, Ph.D.

## **A Monitoring System for Daily Feedback on IoT Sensor's Status in Tartu City**

### **Abstract:**

This thesis explores creating a tool to support the displayed information in the existing ITS lab Modal Split dashboard, which uses data from sensors around Tartu City to produce a modal split estimation of various traffic types (Vehicle, Bike, Pedestrian). This thesis aims to incorporate sensor monitoring to facilitate data quality assessment into Modal Split and in turn, improve the overall quality of Modal Split. This solution comes in the form of an interactive dashboard, which allows users to view the welfare of sensors for multiple dates and filter sensors by type. The finished sensor monitoring dashboard includes a heatmap that displays assessment by the sensor and by the hour for twenty-four hours. In addition, it provides a daily summary of each sensor's welfare in the form of a heatmap. The dashboard can generate a line graph that displays the number of events for a sensor of the user's choosing by the hour. In addition, a map visualisation is also included that highlights the location of a particular sensor to make it easy to identify its location. The dashboard is designed to make it easy for users to identify problems with sensors in the ITS lab system, and thus contribute to the overall quality of the Modal Split dashboard.

**Keywords:** IoT, Monitoring, Cumulocity, Python, Dash, MariaDB

**CERCS:** P170 Computer science, numerical analysis, systems, control

## **Jälgimissüsteem Tartu Linna IoT andurite oleku igapäevaseks hindamiseks**

### **Lühikokkuvõte**

Käesoleva lõputöö raames arendati välja tööriist mis toetab visualiseeritud informatsiooni eksisteerivale ITS laboratooriumi Modal Split töölauale, mis kasutab andmeid kogutud anduritest üle Tartu linna et koostada erinevate liiklustüüpide jaotushinnang (autod, jalgratturid, jalakäiad). Selle lõputöö eesmärk oli lisada andurite andmete jälgimist ja aidata andmete kvaliteedi hindamist Modal Splitil ja omakorda parandada Modal Spliti üldist kvaliteeti. Tehtud lahendus valmis interaktiivse töölauana, kus kasutajad saavad vaadata erinevate andurite kõlbulikust ise valitud aegadel ning filtreerida andureid transpordimeetodi järgi. Tehtud anduri jälgimise töölauda on implementeeritud kuumkaart (ingl *heatmap*), mille kaudu saab interaktiivselt vaadata andmete kvaliteeti konkreetse anduri kaupa ja tunni kaupa. Lisaks annab tehtud töölaud päevase kokkuvõtte iga anduri kõlbulikusest ja sedagi interaktiivse kuumkaardi kujul. Töölaud saab ka genereerida kasutaja poolt valitud konkreetse anduri kohta graafiku, mis näitab selle anduri sündmuste arvu ajas. Lisaks, tuua välja konkreetse anduri asukoha kaardil, aitamaks hõlpsamalt tuvastada anduri asukohta. Töölaua kasulikkus väljendub selles, et ITS lab modal split töölaua kasutajad saavad kergemini ja kiiremini tuvastada probleemseid andureid, suurendades nii kogu ITS Modal split töölaua kvaliteeti ja efektiivsust.

**Võtmesõnad:** IoT, Jälgimine, Cumulocity, Python, Dash, MariaDB

**CERCS:** P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

## Contents

1. Introduction .....	4
1.2 General View.....	4
1.2 Objectives and Limitations.....	4
1.3 Contribution .....	5
1.4 Roadmap.....	5
2. Literature Review and Related Projects .....	7
2.1 Data Quality and Assessment Overview .....	7
2.2 Errors and Problems within IoT Systems.....	8
2.3 Methods and Frameworks .....	9
2.4 Cumulocity.....	10
2.5 Related Projects.....	11
3. System Design and Architecture .....	12
3.1 Modal Split System and Problem Statement.....	12
3.2 Solution Pipeline .....	13
3.2.1 Fetching data from Cumulocity.....	14
3.2.2 Database.....	15
3.2.3 Processing Data .....	18
3.2.4 Displaying data.....	21
4. Results and Discussion .....	24
4.1 Results .....	24
4.2 Testing.....	25
5. Conclusion.....	27
References.....	28
Licence.....	29

# **1. Introduction**

## **1.2 General View**

The term “Internet of Things” (IoT) was first established by Kevin Ashton. It refers to a system where sensors can connect to the internet and are used for various services [1,2]. It is used similarly nowadays to describe scenarios where multiple devices (including sensors) have advanced computing capabilities and the ability to connect to the internet [2].

The Intelligent Transportation System (ITS) lab has developed a real-time IoT system for daily modal split estimations using sensors placed around Tartu City. The results of this estimation and analysis are displayed in a web-based dashboard named Modal Split with the address: <https://its.cs.ut.ee/modsplit/>. The Modal Split dashboard provides a comprehensive view of different patterns and visualisations using the data estimations. Modal Split includes a map-based visualisation, pie and bar charts, aggregate values and numbers of vehicles that enter/exit the city, among other features. The idea of Modal Split is to contribute to the growth of smart cities and smart mobility and has the potential to help Tartu City have a more data-driven decision-making process [3]. This is why reflecting the data quality and ensuring that the data received from sensors are accurate, complete, and consistent is essential.

However, a significant problem with this dashboard is that this sensor data needs to be adequately assessed before being used in the analysis. The data is not guaranteed to be as accurate as needed to reflect good estimation on the dashboard. Data quality assessment is used to ensure data is of high quality for analysis. Data quality assessment can be carried out in a multitude of different ways. The methods used are highly dependent on the needs of the problem and what is required to solve a specific problem. It can be measured in various data quality dimensions [4]. Data quality dimensions and the methods used to evaluate data quality assessment are described in more detail in the literature review of this thesis. The effect of poor-quality data on the Modal Split dashboard is significant. If the data is of good quality, the visualisations shown on Modal Split will likely be accurate as the estimations will not be affected by poor-quality data. The Modal Split dashboard reflects the quality of the data used, and that is why it is important to assess the input from sensors and use it to facilitate the process of data quality assessment. In addition to data quality, Modal Split is also affected by the number of sensors used to generate estimations. It needs a minimum number of sensors input to perform well. Given the significance of the problem, it is clear how creating the sensor monitoring dashboard would notably help improve or reflect the quality index of Modal Split results.

## **1.2 Objectives and Limitations**

This thesis aims to create a tool to support the displayed information in Modal Split in the form of a dashboard with the goal of sensor status monitoring to facilitate data quality assessment. The purposes of this thesis include investigating approaches on how to incorporate a sensor monitoring dashboard that aims to facilitate data quality assessment for the Modal Split dashboard that ITS lab has created and, in turn, contribute to the field of data analysis.

The solution is being created with the goal of sensor monitoring. Still, additional research objectives must also be considered before beginning to build the sensor monitoring dashboard. The first of these research objectives is to investigate data quality assessment and understand what can be assessed by the ITS lab, that the sensor monitoring dashboard can aid in. What sort of data does the sensor monitoring dashboard need to process, and how does this relate to data quality assessment? Next, the errors and problems that may arise need to be considered. As the Modal Split system is an IoT system, issues specific to IoT systems and preferably sensors must be investigated. Another research objective is to investigate methods and frameworks that have been used in the past for data quality assessment and what can be useful in the solution. A brief overview of the platform Cumulocity must also be done to understand better how the Modal Split system utilises this platform.

Outside of research, some aims have been set for the sensor monitoring dashboard that are deemed important and must be fulfilled. The first of these is adequate visualisation. In order for the sensor monitoring dashboard to be useful to not only the ITS lab but also to users who are not experts in the field, the results of the monitoring must be visualised in a clear, concise way. Interactivity is another aim of the dashboard. This was deemed necessary because if the sensor monitoring dashboard is not interactive, there is a minimal amount of data that can be displayed at a time. It will not be easy for users of the dashboard to find problems with individual sensors. The final aim is that the dashboard must work in daily real-time. The real-time aspect of the Modal Split dashboard is relevant to this solution because if this dashboard cannot assess data in real-time, then the Modal Split dashboard will not be able to utilise the findings to display adequately assessed data on time.

### **1.3 Contribution**

The main contribution of this thesis is to create a sensor monitoring dashboard to support and give insights about the quality of the data that is used in the Modal Split system to achieve an adequate understanding of the input data quality used in the main modules of the modal split algorithm. The thesis aims to add sensor monitoring and reflect it as follows:

- Flexible hourly view of the sensors' status and activities.
- Daily distribution of the collected data by the sensors.
- Discovering unusual behaviour or anomalies in the reading from the sensors.
- Interactive map to show the sensors' locations.

### **1.4 Roadmap**

The structure of this thesis is as follows. Section 2 begins with a literature review and provides an overview of data quality assessment, issues that may arise with IoT devices and their processing, Cumulocity and then methods and frameworks that have been used in the past. It will also describe similar projects that have been completed and how these relate to this thesis. Section 3 will describe the architecture of the existing Modal Split system, then go into detail about how the solution to the problem will be developed. This section will go through the pipeline of the developed sensor dashboard and describe each step in detail. Section 4 briefly

describes the obtained result of development, and the final section concludes the thesis. Finally, the thesis is concluded in section 5, which is followed by references and the licence.

## **2. Literature Review and Related Projects**

Data quality assessment in the context of IoT is an important issue that requires attention and thoughtful analysis. In this literature review, several sources have been analysed in detail to provide a comprehensive overview of key findings that relate to the solution.

### **2.1 Data Quality and Assessment Overview**

The concept of data quality has been defined and described in many ways and it is difficult to assign one universal meaning to the concept. It has even been said that data quality depends on the context in which data is to be used [5]. However, two main perspectives have been noted. First is the user perspective, where the quality of data is determined by how the user uses the program and is able to get relevant information from it [1]. This involves measuring dimensions such as accessibility, interpretability, and understandability. Where for example, understandability refers to “the extent to which data is easily comprehended” [4]. These evaluations, which are mostly subjective assessments, usually involve the user giving their own feedback on the various dimensions when using a system or program. The latter perspective involves the system perspective, where the quality is evaluated by basic elements within the system and how they are being used. This involves dimensions such as accuracy, completeness, and consistency. These dimensions and how they relate to this thesis will be described in further detail in a later paragraph. For IoT data quality, the concept is generally the same, where it is described as whether the data collected from IoT devices are suitable for use [1]. This paper will focus more on the latter perspective of data quality, as the evaluation of users using the sensor monitoring dashboard is not so relevant to the solution being created.

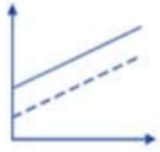
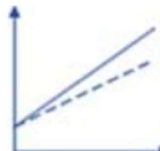
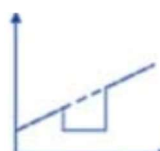
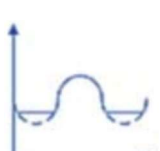
Castillo et al. describe three different IoT layers that can help to break down the assessment process and any problems that may arise. The first layer is the acquisition layer, which refers to the raw sensor data collected from the sensor system. Next is the processing layer, which involves data that has been processed according to the needs of the system. Lastly, the utilisation layer covers how data is delivered to users [5]. In this thesis, the goal is to assess the data that the sensors send directly to the management server before processing. Therefore, the operation of the solution takes place in the acquisition layer. This is important to know, as the problems that may arise with sensors and the data quality dimensions that are explored should be specific to the acquisition layer. Zhang et al. confirm that different types of errors can occur at different layers of IoT systems. However, they propose a slightly different three-layer structure. Zhang et al. introduced the perception layer, the network layer, and the application layer. According to Zhang's description, the perception layer refers to how physical devices gather data. The network layer is how the data is sent over a network and the application layer is the layer that receives the observation results and where processing and analysis take place [1]. According to this description, the layer that most relates to the solution is the application layer, as this is the layer that seems to be receiving the raw data. However, Castillo's description of all the layers applies to Zhang's description of the application layer. This means that Zhang's description is more generalised and related to the hardware of IoT systems, whereas Castillo's description is more oriented to the processing done within the application layer.

Data quality has been described as a multi-dimensional concept [1,4]. These dimensions all present a different aspect or characteristic of data that can be used to determine its overall quality [4]. Pipino et al. highlight sixteen data quality dimensions in a table. This thesis will focus on these dimensions the most: completeness, free-of-error, and timeliness. The other dimensions are arguably equally as important as these three; however, less focus will be put on these. Completeness describes “The extent to which data is not missing and is of sufficient breadth and depth for the task at hand” [4]. Free-of-error describes “the extent to which data is correct and reliable” [4]. Timeliness describes “the extent to which the data is sufficiently up-to-date for the task at hand” [4]. The reason these three dimensions were highlighted specifically is that these relate the most to the tasks that need to be carried out at the acquisition layer [5]. The solution aims to create a tool that aids in monitoring data quality so that the research lab can subsequently use this information to improve estimations in the Modal Split dashboard.

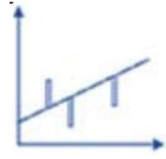
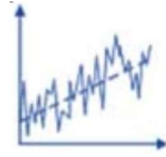
## 2.2 Errors and Problems within IoT Systems

The bigger the IoT gets, the bigger the probability that the system can have failures and errors which lead to poor sensor data quality [1]. It is important to get an overview of problems that may arise from sensor readings so these can be investigated when the solution is developed. When looking into the potential errors and issues that may arise with data, Castillo et al. bring to light several errors from sensors that may be causing data quality issues, shown in Table 1 [5].

*Table 1. SENSORS ERRORS DERIVING DQ PROBLEMS IN SCP ENVIRONMENTS [5].*

Error	Description	Example
Constant or offset error	The observations continuously deviate from the expected value by a constant offset.	
Continuous varying or drifting error	The deviation between the observations and the expected value is continuously changing according to some continuous time-dependent function (linear or non-linear).	
Crash or jammed error	The sensor stops providing any readings on its interface or gets jammed and stuck in some incorrect value	
Trimming error	Data is correct for values within some interval, but are modified for values outside the interval. Beyond the interval, the data can be trimmed or may vary proportionally	



Outliers error	The observations occasionally deviate from the expected value, at random points in the time domain	
Noise error	The observations deviate from the expected value stochastically in the value domain and permanently in the temporal domain	

These errors are useful to keep in mind; however, many of these require comparison to an already existing expected value, which is something that this thesis does not have. Therefore errors such as offset and drifting errors cannot be assessed within this thesis. Kandel et al. created a system with the goal of data quality assessment through visual analysis tools. In their paper, they describe how part of their system, named Profiler, focuses on anomaly detection, and they outline five categories involved in the anomaly detection of Profiler. These were missing data, erroneous data, inconsistent data, extreme values and key violations [6]. Zhang outlines eight types of sensor data errors that partly overlap with Kandel’s categories. These eight were “anomalies, missing values, deviations, drift, noise, constant value, uncertainty, and stuck-at zero”. Zhang mentions that anomalies and missing data fields are the two most common errors [1]. This means that if there is an issue with data quality, it is most likely derived from these two errors; thus, the solution should also consider this.

### 2.3 Methods and Frameworks

Castillo et al. list twenty-three best practices for quality assurance and sensor data field quality control [5]. While these would help with data quality management, most of these would go beyond the scope of this thesis. There are practices such as “BP2: Document all sensor data processing”, “BP7: Provide complete metadata”, “BP20: Retain the original unmanipulated data”, and “BP21: Retain all versions of the input data, workflows, data provenance programs, and models used” which become hard to carry out due to the sheer scale of data that needs to be handled for the solution, and may prove unnecessary to use large amounts of space for these. The ITS lab and their platform have already implemented some practices, such as BP20 and “BP9: Implement an automated alert system to warn about potential sensor network issues”. Practices such as “BP13: Make available ready access to replacement parts” and “BP14: Schedule sensor maintenance and repairs to minimise data loss” are also considered irrelevant, as during the time of writing this thesis, the author will not have access to physical sensors. The only available information is the data provided by the sensors. After excluding all the practices that were deemed irrelevant or challenging to implement, six practices were left. The first two are “BP5: Use flags to convey information about the data” and “BP8: Maintain an appropriate level of human inspection”. These practices are essential to this thesis as the goal of this thesis is to create a sensor monitoring dashboard where potential data quality issues are visually displayed for humans to analyse, and the use of flags or colours will help convey

information to potential users. The practices BP10, BP11 and BP12 are all related to performing checks on the data. Whether this be range checks, domain checks or slope and persistence checks, these are also essential to this thesis as the assessment carried out by the solution will depend on checking the data coming from sensors. The final practice is “BP15: Automating sensor data quality procedures” [5]. One objective of the sensor monitoring dashboard is that it updates daily. Automation is key for this to happen, which is why this practice is also relevant.

Pipino et al. highlight several functional forms that can produce numerical metrics measuring data quality. This source suggests using the functional form of a simple ratio to measure the data quality dimensions mentioned earlier, where 1 represents the most desirable outcome and 0 represents the least desirable. The free-of-error dimension can take this form by dividing the number of data units in error by the total number of data units subtracted from 1, where data units and errors have clearly defined criteria [4]. Similarly, the completeness dimension can also take this form, but there are multiple ways to consider this. Pipino et al. mention that this can be considered as schema completeness (“the degree to which entities and attributes are not missing from the schema”), column completeness (“a function of the missing values in a column of a table”), or population completeness (If, for example, a column requires a value to occur for each element in a set, but one value is missing for an element then population completeness has not been achieved). This is also measured by subtracting the ratio of incomplete data items to complete data items from 1 [4]. These functional forms should be considered when creating the solution.

## **2.4 Cumulocity**

On Cumulocity's official page, Cumulocity is introduced as an IoT platform that allows the user to manage various remote assets quickly and visibly. Cumulocity offers many different uses and applications. Cumulocity offers several services such as software libraries that can be used to bring devices to the cloud, data visualisation in its user interface and graphs, remote control of devices over the web and real-time device data monitoring, among many other features [7]. Cumulocity is suitable for monitoring devices because it provides quick and easy information about specific devices, regardless of the number of devices. It is also possible to change the configuration of any given device and the operations that the device can perform [7]. The ITS lab relies heavily on acquiring and processing Cumulocity's real-time data and uses Cumulocity to monitor the connected devices within the IoT. Within the ITS lab system, the devices are configured, and data is regularly transferred to the system.

Cumulocity uses an interfacing technology known as RESTful API, allowing easy communication between IoT devices and Cumulocity. The protocol REST is used for all external communication, including to and from IoT devices and web applications. REST is a secure protocol based on HTTP(S) and TCP [8]. This information is relevant as this is what will be used to gather data from Cumulocity into the new solution.

## 2.5 Related Projects

Earlier, the system Profiler was mentioned, created by Kandel et al. As their system's goal is similar to the solution's goal, further analysis of their system is required. Their system is highly focused on the visualisation of data and detection of anomalies and how this can be used to easily detect data patterns. For anomaly detection, they use five different data mining routines. These are missing value detection, type verification, clustering, univariate outlier detection and frequency outlier detection. What makes their system unique is that unlike many other systems, where the user has to select variables to visualise themselves, Profiler does this all for the user. Profiler automatically suggests appropriate summary visualisations based on the data types and anomalies that were detected using their anomaly detection algorithms. These visualisations include many different types of charts and will choose what type of chart is shown depending on the data chosen. Notably, Profiler uses “data quality bars to summarise column values as valid, type errors, or missing” [6]. These quality bars are annotated on all visualisations and are useful for data quality analysis. They finally concluded that this system can reduce the amount of time users spend finding data quality issues, leading to more efficient and relevant analysis [6]. Considering this, the solution that is being created should also take into account how to best point out potential issues with sensors to the user.

Several pieces of work related to Cumulocity were found throughout the research. Srirama et al. identified a limitation of Cumulocity, that integrating mobile devices into the platform is more complex. Because of this, an API was developed to easily allow developers to utilise the Cumulocity platform to manage mobile sensors and devices, thus contributing to the development and ease of use of Cumulocity [9]. Fortino et al. carried out an analysis of various industrial and commercial solutions involving IoT platforms, one of which being Cumulocity. They specifically explore the security aspect of Cumulocity, and whether data is confidential and cannot be tampered with. Cumulocity addresses security by providing several levels of authentication, ensuring all connections are established using HTTPS, and allowing tenants to add or terminate users and groups. It was concluded that these factors deem Cumulocity secure, and that data is also well protected from interference from unauthorised parties [10].

### 3. System Design and Architecture

The following section provides an overview of the Modal Split system, as well as an overview of the pipeline that is used for the solution. Aspects of the sensor monitoring dashboard and how it was created are described in detail within the pipeline.

#### 3.1 Modal Split System and Problem Statement

The architecture and data sources of the Modal Split system are described in more detail in a paper written by Khoshkhah et al. [3]. Their system, as illustrated in Figure 1, has multiple raw data sources, most of which are stored in the Cumulocity platform and later used to create modal split estimations displayed on their dashboard. Getting a good overview of the various data sources is important to understand how and where this information can be used to create the sensor monitoring dashboard. The paper describes nine different data sources that they use to produce the daily modal split estimation. However, as mentioned earlier, only the physical IoT devices (sensors) are relevant as the goal of the solution is to assess whether the information sent from these sensors is correct. This means that the most relevant data sources are “Cumulocity ECO Sensor Measurements”, “Cumulocity Thinnect Sensor Measurements” and “Cumulocity AVC Sensor Events”. The reason the data sources related to the bus validation system (“Ridango REST API”) or the bike-sharing system (“Bewegen REST API”) are not explored is that these are considered very accurate already, regardless of weather or other external conditions, and don’t currently require an assessment to see if the information is correct. Khoshkhah et al. describe the three chosen sources as dynamic sources, which means that they “represent stream data fetched on demand from the sensors each time the pipeline runs”. These three sensors all count different types of traffic. ECO sensors count pedestrians and bicycles, Thinnect sensors count vehicles and light traffic (both pedestrians and bicycles), and AVC sensors count only vehicles [3]. The Automatic Vehicle Classification (AVC) sensors differ from the ECO and Thinnect sensors in the fact that these record events as opposed to measurements. This means that these sensors will immediately send data to Cumulocity every time a vehicle is detected, whereas the ECO and Thinnect sensors will only send data after a specific time interval, where the data will contain an aggregated number of events that happened in the time interval, so the exact time of any given event cannot be pinpointed.

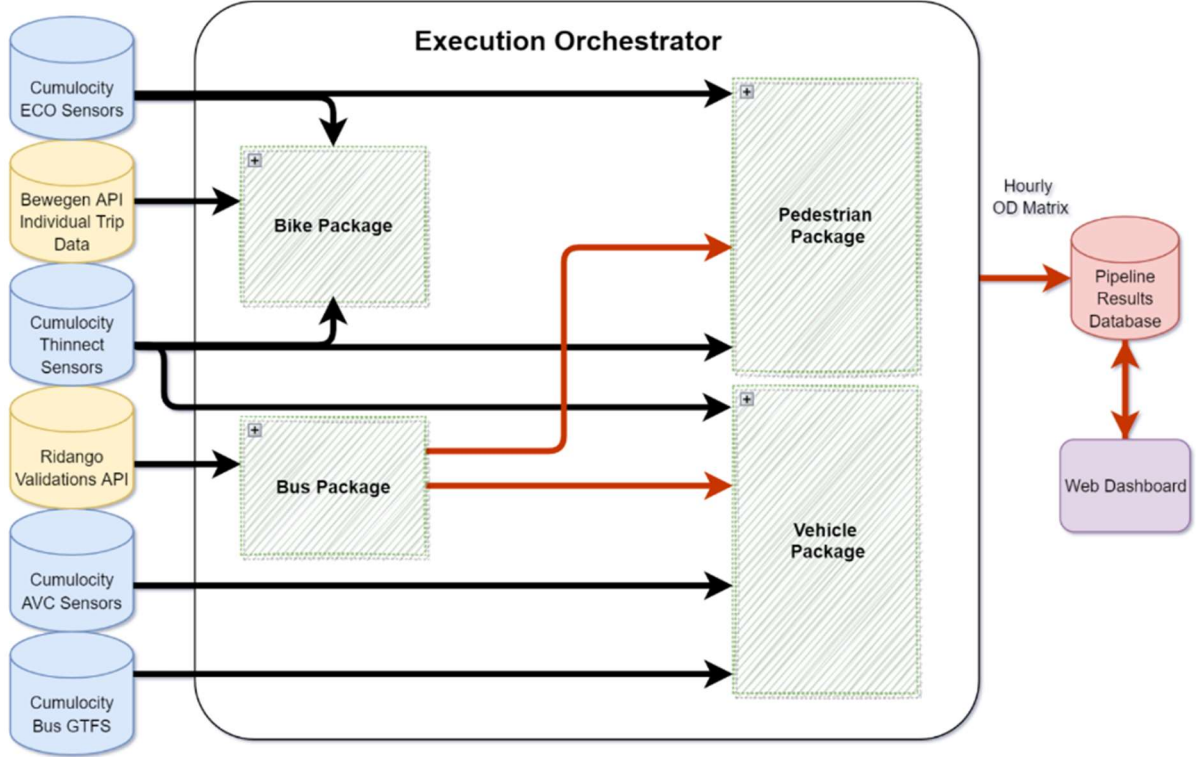


Figure 1. The proposed system architecture [3].

Each package in the execution orchestrator fetches necessary data from the raw data sources and then performs relevant analysis to generate the Hourly OD Matrix, which is used in the web dashboard. In the sensor monitoring dashboard, however, only the raw data from Cumulocity sensors will be used, and outputs from various packages will not be considered. It is also important to note that more sensors are listed in Cumulocity than physical devices placed around Tartu. This is because, in some cases, when a physical device can count traffic moving in multiple directions, it stores each particular direction it records as a separate sensor in Cumulocity. For example, there exists a sensor with the name “KL\_Mõisavahe 1\_jalakäijad\_kesklinna suunast\_EcoCounter” and a sensor “KL\_Mõisavahe 1\_jalakäijad\_kesklinna suunas\_EcoCounter.” Which bear the same location, but the first records traffic moving away from the city centre (“suunast”) and the other records traffic moving towards the city centre(“suunas”).

### 3.2 Solution Pipeline

The pipeline for the solution can now be created using the knowledge and information gained from the Modal Split architecture design. Figure 2 illustrates a proposed pipeline for the solution.

Every day:

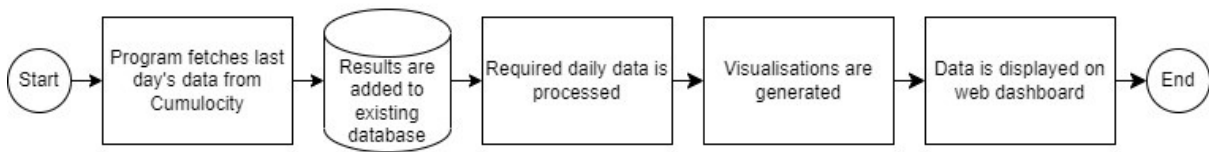


Figure 2. Proposed solution pipeline.

As can be seen in Figure 2, the program begins by fetching data from Cumulocity. It is clear why this step is required, as the only place where the sensor data can be received is from Cumulocity. Next, the current day's data is stored in a database. The data is continually stored here and is fetched from the database as needed for the web dashboard. This is important because it was discovered that fetching data from Cumulocity is a time-consuming process. It would be unrealistic to create a web dashboard that loads for an unreasonable amount of time every time new data is needed. Fetching from an existing database is much quicker, so this approach was chosen. After this, the data is processed as needed for adequate assessment of sensor status. Then relevant visualisations/graphs are created that illustrate the assessments made. Lastly, the created visualisations are displayed on a web dashboard. For the dashboard, the choice was made to program in the Python programming language. This is mostly due to the ease of use and resources available. Using this proposed pipeline, the next section will cover each pipeline step and describe in detail how the dashboard is implemented at each point.

### 3.2.1 Fetching data from Cumulocity

Having looked into different methods to fetch data from Cumulocity in Python, the GitHub repository `cumulocitypython` by the user `SilverLaius` was the best match for the implementation of the dashboard [11]. This Python package allows users to connect to the Cumulocity platform and make queries from the Cumulocity REST API. It is easy to set up, as it just requires installing and importing the `cumulocitypython` package. It includes built-in functions that can make queries for measurements, events and devices from Cumulocity. Figures 3, 4 and 5 show code snippets that were written while creating the dashboard system that uses the `cumulocitypython` package and demonstrate how the package works. Figure 3 demonstrates how to start a connection to the particular Cumulocity server that is being used. Figure 4 demonstrates how to query events for a particular sensor. Measurements are queried in an almost identical way. Figure 5 demonstrates how to get information about a particular device using its ID.

```
tenant_url = "tartu.platvorm.iot.telia.ee"
connection = CumulocityConnection(tenant_url, username, password)
```

Figure 3. Connection to Cumulocity using `cumulocitypython` package.

```
measurement_data = connection.get_events(
    device_id = 117925742,
    date_from="2023-02-20T02:00:00+02:00",
    date_to="2023-02-21T02:00:00+02:00",
)
```

Figure 4. Querying events for sensor with ID 117925742 for 20th of February 2023.

```
device_ids = [661711004, 661711010]
devices = connection.get_devices(ids=device_ids)
```

Figure 5. Querying device information for two devices of ID 661711004 and 661711010.

This package is extremely convenient as it will convert the output of the queries from JSON data to a pandas dataframe, which can easily be used and analysed [11]. The repository *cumulocity-python-api* by Global Competency Center IoT was also considered as a potential way to gather data from Cumulocity [12]. However, the documentation for this option was deemed lacklustre, as there needed to be descriptions of how to query data or in what form data would be handled.

While experimenting with the package, it was discovered that the process takes a while to execute when fetching data from sources with thousands of events (i.e. AVC sensors). For a whole twenty-four hours of sensor data, this could take ten to fifteen minutes. If the web dashboard queried directly from *cumulocitypython*, it would take the webpage ten to fifteen minutes to load and output information to the user. This is entirely unreasonable. Thus, the solution of holding data in a separate database was chosen.

### 3.2.2 Database

The Python programming language supports many different databases, which is convenient as databases can easily be implemented into the pipeline without changing the programming language. Several databases were considered but, in the end, MariaDB was chosen. This was mostly due to the greater performance it provides, and its ease of use in Python. As the webpage being created relies on the speed at which data is fetched and processed, then this factor was prioritised. HeidiSQL is a Windows client for MariaDB and MySQL and comes with the Windows version of MariaDB. A database was created using HeidiSQL. The tables and relations were also configured within HeidiSQL. Figure 6 depicts a database schema of the configured tables in the database. MariaDB's Python library works by first establishing a connection to the created MariaDB database using the *connect()* function, and then retrieving the cursor using the *cursor()* function. The cursor allows the program to interact with the server and is what allows the program to run SQL queries. The snippet of code that was written to connect with the MariaDB database is shown in Figure 7, with the port and password currently hidden.



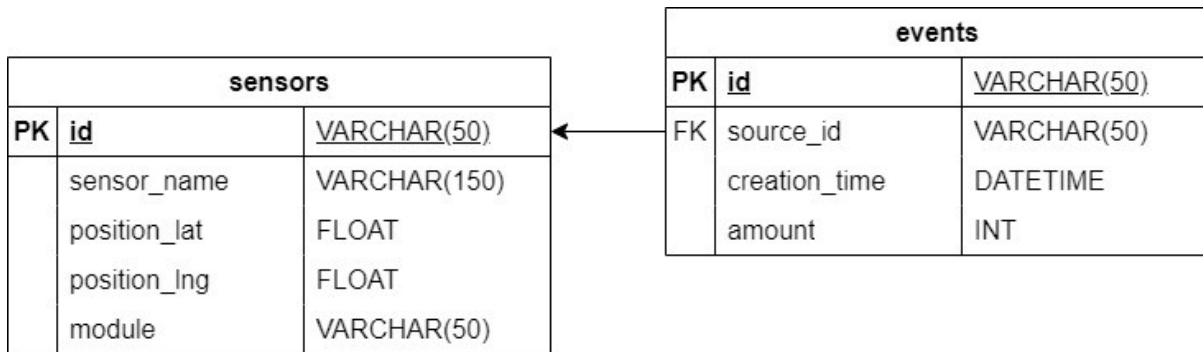


Figure 6. Database Schema.

```
try:
    conn = mariadb.connect(
        user="root",
        password=XXXX,
        host="localhost",
        port=XXXX,
        database="sensordb"
    )
except mariadb.Error as e:
    print(f"Error connecting to MariaDB Platform: {e}")
    sys.exit(1)

cur = conn.cursor()
```

Figure 7. Connection to the database using MariaDB and retrieving the cursor.

The database has two tables, sensors, and events. The sensors table includes the devices in Cumulocity that send data, and the events include each event or measurement that a particular sensor will send to Cumulocity. As mentioned earlier, AVC sensors send individual events, so each row in the events table is a separate event. However, for ECO and Thinnect, each row represents one piece of aggregated data that has been sent. The amount column in the events table represents how many events were recorded in a particular aggregated time frame. For AVC sensors, this column will always be null as they do not send aggregated data.

The sensor monitoring dashboard includes as many sensors as possible; however, some limitations with the sensors needed to be considered before adding them to the database. Firstly, is the problem that Thinnect sensors cannot differentiate between pedestrians and bikes. This is a problem because part of the sensor monitoring dashboard's interactivity involves being able to choose between modules, and it shows the sensors relevant to that module. Instead of displaying Thinnect sensors in multiple modules, it was decided that these Thinnect sensors that detect light traffic fall into the pedestrian module for the purpose of simplicity. There are a few sensors that are listed in Cumulocity that are not included in the final product. This is because they include repeated information. Some of the ECO sensors include readings where other sensor information is added together. For example, there exists the sensor



“KL\_Kroonuaia\_liiklejaid\_kokku\_EcoCounter.” (Kroonuaia traffic total), whose values correspond directly to the sum of the sensors “KL\_Kroonuaia\_jalakäijad\_EcoCounter.” (Kroonuaia pedestrians) and “KL\_Kroonuaia\_jalgratturid\_EcoCounter.” (Kroonuaia bicyclists). Examples like these, where sensor traffic is summed up were omitted. There are also two pairs of Thinnect sensors which both bear the same name, however, in both cases, one did not show any events. The sensor that did not show events was omitted in both cases.

To continue the pipeline, there now needs to be a connection where the Python code both fetches the Cumulocity data and stores it in the database. For this, the file *addtodatabase.py* was created. The workflow diagram for this file is seen in Figure 8. This will run every day. In this file, a connection to MariaDB is started, and the sensors are broken down into groups. First, by type (AVC, Thinnect, ECO), then by module (Vehicle, Bike, Pedestrian). This leaves us with five different groups (AVC, Thinnect Vehicle, Thinnect Pedestrian, ECO Bike, ECO Pedestrian). These sets make it easier to add the sensors to the database. For each set, the relevant devices are found using the cumulocitypython in-built function *.get\_devices()*. The module that the sensors are in is not included in the device information. This is information that is added manually to the sensors. The ITS lab has this information within their documents, so this was done manually later in the program.

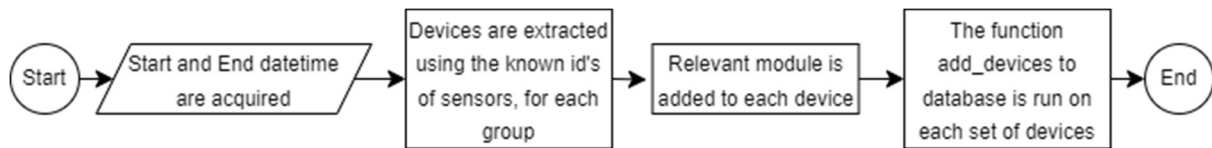


Figure 8. Workflow diagram for file *addtodatabase.py*.

The function *add\_devices\_to\_database()* is run on each dataframe of devices. This is the main function in this file and does the bulk of the processing and inserting into the dataframe. The devices and events that are added to the database are determined by the variables *startDate* and *endDate*, which are of datetime type, and define the range of time where the events in between are extracted and added to the database. Because every type of device and event varies, each needs to be processed differently. This function starts by iterating over each device, and checking if it is of type “AVC”, “ECO” or “Thinnect”. If the device is of type AVC, the sensor is added to the database (if it does not already exist) and then the events of the device are found using the cumulocitypython method *.get\_events()*. Each event is individually added to the database with the “amount” column equalling null. For the other types of devices (ECO and Thinnect), the sensors are also added and the measurements are found using the cumulocitypython method *.get\_measurements()*. Each measurement is added to the events table with the amount listed in the measurement. There was an extra piece of processing that needed to be done with ECO sensors. This was because when extracting measurements from any ECO sensor, each measurement was repeated four times. Here, the fix was simple and involved filtering out only one of each unique measurement and adding these to the database. The workflow diagram for this function is seen in Figure 9. Figure 10 displays an example of a piece of code that was used to insert data into the database using MariaDB in Python.

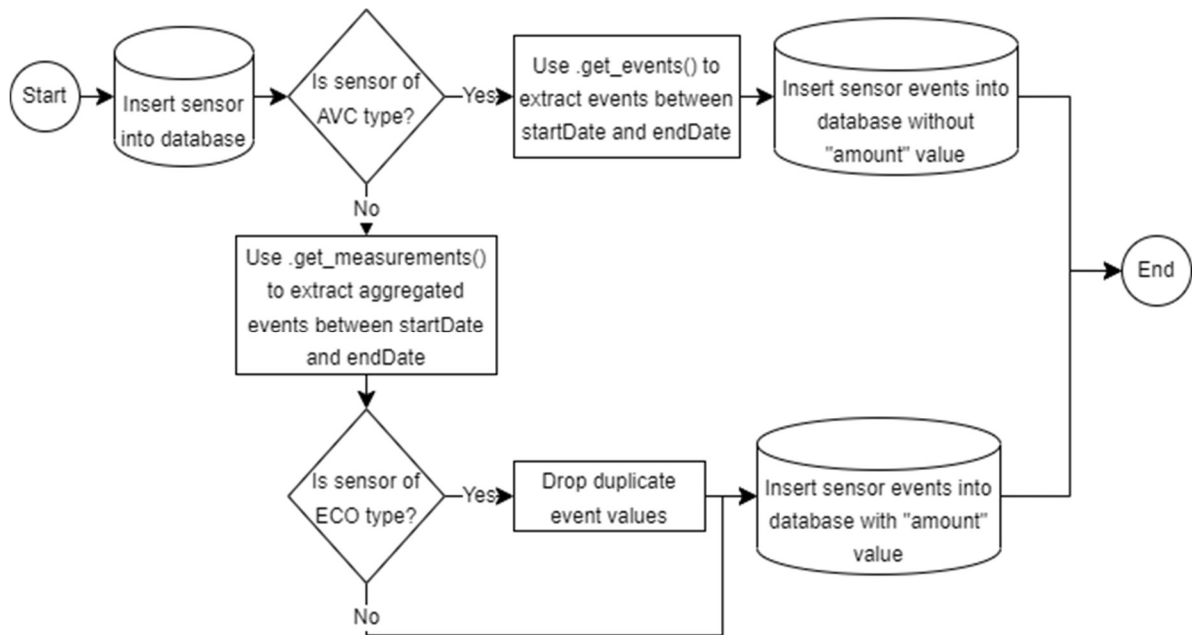


Figure 9. Workflow diagram for function `add_devices_to_database()`.

```
cur.execute("INSERT INTO events VALUES (?, ?, ?, ?)", (row1["id"], row1["source_id"], creation, None))
```

Figure 10. Inserting an AVC sensor into the database using MariaDB.

### 3.2.3 Processing Data

The next part of the pipeline involves accessing the data from the database and processing the data. The data processing all happens in the main `app.py` file. Data is fetched from the database using SQLAlchemy. This is because SQLAlchemy can fetch data from the MariaDB database and immediately place all the data in a pandas dataframe. This process was initially carried out just using MariaDB queries, however, this meant that data needed to be manually placed in a dataframe, and when it comes to thousands and thousands of rows, this process was very slow, thus the new solution was adopted. Figure 11 shows how data was retrieved from the database using SQLAlchemy and connecting to the MariaDB database, with the password and port number currently hidden.

```
engine = create_engine("mariadb+mariadbconnector://root:XXXX@127.0.0.1:XXXX/sensordb")
with engine.connect() as connection:
    query = "SELECT * FROM events WHERE creation_time BETWEEN \"{}\" AND \"{}\";".format(startDate, endDate)
    events = pd.read_sql_query(sql=text(query), con=connection)
    query2 = "SELECT * from Sensors"
    sensors = pd.read_sql_query(sql=text(query2), con=connection)
```

Figure 11. Using SQLAlchemy to retrieve data from a MariaDB database.

The ITS lab Modal Split dashboard only shows one day of data at a time. The user can pick a date, and only that day is shown on the screen. For the thesis solution, a similar approach was adopted. The sensor monitoring dashboard only shows twenty-four hours' worth of data at a time. Therefore, every time a query is made to the database, only twenty-four hours is fetched at a time. It was decided that the data analysis would be done by the hour, as this would help break down the data and see if any trends are worth noting. This information is shown to the user in the form of a heatmap, where for each hour of the day, for each sensor, a different colour

is displayed showcasing the results of the sensor monitoring. Subsequently, it was decided that there should also be an indicator that summarises the results found throughout the day into one comprehensive reading, meaning that the user does not need to assess each individual hour, but can instead look at the daily summary for each sensor, and if needed, then look at details related to each hour.

This approach means that for each hour of the day, the number of events are grouped together. In the program, a matrix is created where each column represents the hour of the day, and each row represents a different sensor. The values in the matrix show how many events were recorded in that hour. The name of this matrix is the `event_number_matrix` and is created in the function `create_event_matrix()`. After this, another matrix is created where the rows and columns are the same as the last matrix, but the number within the matrix now represents a colour that corresponds to the assessed data quality for that hour. This colour can be red, which represents the biggest issues that can occur with a particular sensor, yellow, which represents untypical values or minor problems or green, which means that the sensor delivered events as expected. The colours and how these are classified are explained in more detail in later paragraphs. This second matrix bears the name `assessment_matrix` and is created in the function `create_assessment_matrix()`. The final matrix that is created is not so much a matrix, but rather a list where, similarly to the `assessment_matrix`, the number represents a colour that signifies the assessed data quality for the whole day. This matrix also uses the colours red, yellow, and green and are used in a similar fashion to the `assessment_matrix`. The name of this final matrix is `assessment_summary` and is created in the function `create_day_summary()`. The workflow diagrams for these three functions are visible in Figure 12.

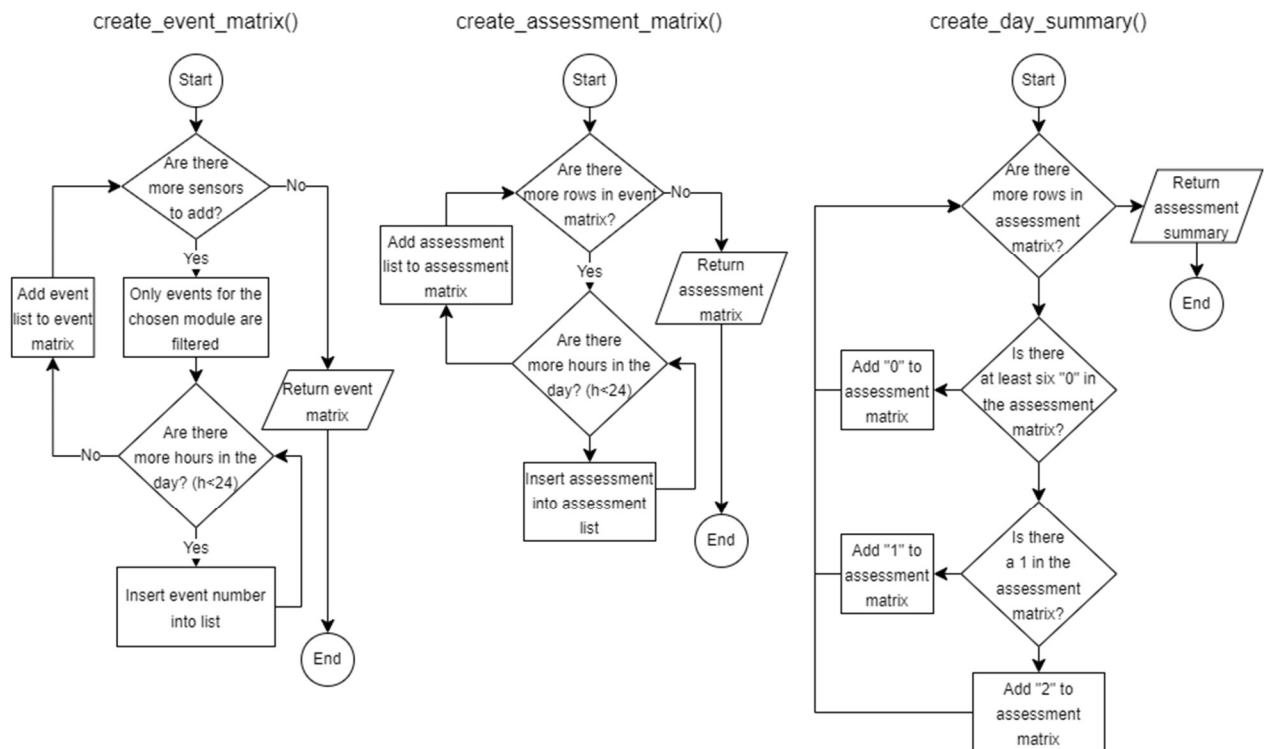


Figure 12. Create matrix functions in the file `app.py`.

The `assessment_matrix` must now be populated using the knowledge gained from the literature review. Potential errors that the sensors could have need to be considered. Firstly, as mentioned by Kandel et al. [6], the first, and arguably most obvious type of error to look for is missing values. In the sensor monitoring dashboard, this would mean a sensor fails to deliver an event that happened in real life. Each individual event that is missed cannot be accounted for, thus instead the dashboard considers if there is a continual error where, for a certain time period, the sensor fails to deliver any events. In this case this period of time would be one hour as the dashboard is broken down into an hour-by-hour assessment. While it is possible that a sensor shows 0 events over the course of an hour, the longer the period of time, the less likely it is that this information is correct. Thus, the approach that was decided on was that the hours that show no events are marked with red in the `assessment_matrix`, and instead the `assessment_summary` will only show red if a significant number of hours throughout the day show 0 events. In this case, it was chosen to be six hours in the day where no events are recorded, as then 25% of the day would have been empty. Next, an error that was mentioned by Castillo et al. was also implemented. Castillo et al. [5] describe the “Crash or jammed error”, wherein a sensor may get stuck on an incorrect value and continuously send only this incorrect value. This error is easy to check, as it just requires the program to control when it seems that event numbers in the `event_number_matrix` start to repeat themselves repeatedly. If this is found to be the case, the cells where the repetition occurs would be marked with a yellow cell. In the `assessment_summary`, if there is at least one yellow cell, the summary already shows a yellow assessment.

When it came to displaying and updating the data, a single function was deemed most suitable to update all data. The function `create_matrices()` was created, which carried out the bulk of the processing of the data. The workflow diagram for the function can be seen in Figure 13. How the input is received and displayed is discussed in section 3.2.4 Displaying Data.

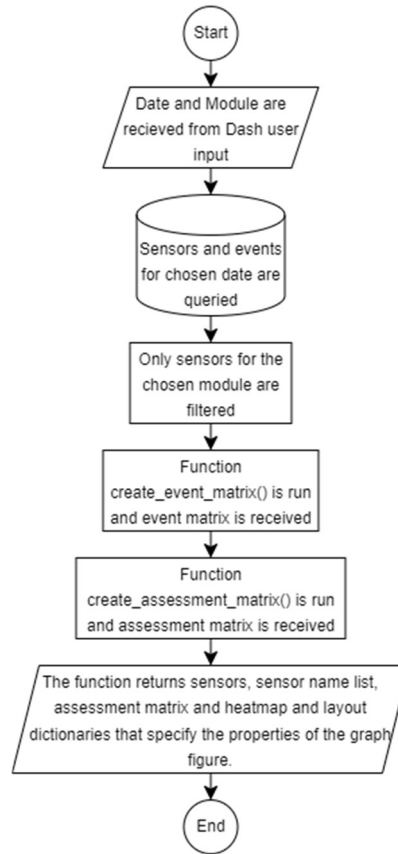


Figure 13. Workflow diagram for the function `create_matrices()` in the file `app.py`.

### 3.2.4 Displaying data

Dash was chosen as the relevant framework for building the sensor monitoring web dashboard. Dash is an open-source enterprise product of Plotly and is used to build web-based analytic applications in Python, R or Julia. Dash is not only useful to create a professional-looking dashboard but also has features that allow for interactivity, which is suitable for the solution [13]. Dash apps are composed of two main parts, the layout and callbacks. The layout is composed of components, and the components can either be HTML components, which generate a HTML element in the app, or a Dash Core Component, which are more complex components generated with HTML, Javascript and CSS. These components all form the general layout of the application and what it looks like. Similarly to HTML, components can be arranged in a multitude of different ways, including embedding components inside one another using the `children` property. The Dash Core Components used in the sensor monitoring dashboard are Graph, Store, DropDown and DatePickerSingle. All other used components are simple HTML components used to aid in the layout of the app. The Dash Graph component renders interactive data viualizations using the Plotly graphing library. The Dash Store component is used to share data between callbacks, as it allows callbacks to store an intermediate value that can then be used by another callback. The DropDown and DatePickerSingle components were used as inputs for a callback, whose value can be manipulated by the user. Figure 14 shows a shortened piece of code that was used in the final dashboard implementation. For the sake of conciseness, sections were removed to only leave

some examples of how Dash was used. In the figure, the Dash app is configured, and the layout includes a H1 title component, a div component (which includes two Dash graph components inside of it), and a Dash store component.

```
app = dash.Dash(__name__)

app.layout = html.Div(children=[
    html.H1(
        id = "title",
        children='Interactive Sensor Data',
    ),

    html.Div(
        className="graph-div",
        children=[
            dcc.Graph(
                id= "sensor_assessment_hm",
                className='sensor-chart',
                figure=sensor_assessment_hm,
                config={"displayModeBar": False}
            ),

            dcc.Graph(
                id = "sensor_assessment_summary",
                className="sensor_day_chart",
                figure=assessment_summary_figure,
                config={"displayModeBar": False}
            ),
        ]
    ),

    dcc.Store(id='heatmap-value'),
])
```

Figure 14. Configuring a Dash app and layout.

The second part of Dash apps is callbacks, which are essentially functions that are called by Dash when a particular input is triggered and will cause other components to change. Callbacks are essential to interactivity in Dash apps [13]. There are three main interactive components in the sensor monitoring dashboard. The first is the ability to choose between dates. This already exists in the Modal Split dashboard and is essential to the sensor monitoring dashboard, so data from the past can be viewed and compared. The second interactivity component is the ability to choose between the mode of transport. This is something that is a new addition compared to the Modal Split dashboard and would be beneficial and aid the problem finding process. The third and final component is the ability to generate line graphs and highlight the location of a chosen sensor. This aids users in seeing general trends with sensors and identifying if there appears to be any problems, as well as easily finding the location of a sensor. Figure 15 shows the beginning of a callback that is used in the dashboard implementation. The *Output()* field represents the components on the dashboard that are going to change, and the *Input()* field

represents a property that can be changed, and can be used to generate information about the output. The function `update_graph` is called when the input property changes.

```
@app.callback(  
    Output("line_graph", "figure"),  
    Output("sensor-map", "figure"),  
    Input("sensor_repetition_summary", "clickData"),  
    Input("heatmap-value", "data")  
)  
def update_graph(clickData, data):
```

*Figure 15. Using a Dash callback to change information on the dashboard.*

Dash apps can be styled using CSS. For the dashboard, a separate CSS file called `style.css` was created that defines various style aspects for the components. The styles that were adopted were chosen with the aim of creating a dashboard with a similar style to the original Modal Split dashboard.



## 4. Results and Discussion

After the pipeline was implemented, a final product was ready. The following describes the results that came from the implementation, and what features are available in the sensor monitoring dashboard. This section also includes a round of testing that was done to check whether the results are produced to an adequate standard.

### 4.1 Results

Putting together the pipeline, there is now a working application that allows the research group to spot issues and inconsistencies with sensors. The finished sensor monitoring dashboard has three main visualisations. The pictures shown below are specifically for the 19th of February 2023, and the heatmap is displaying sensors in the module “Vehicle”. The first of the visualisations is shown in Figure 16. As mentioned earlier, this is displayed as heatmaps, the first showing a colour for each sensor, and each hour. Green shows that there are no issues with the events, red shows there were no recorded events and yellow shows values that are untypical and worth noting. When hovering over a cell, information is displayed about that particular cell. This information includes the name of the sensor, from what starting time are the events recorded, and how many events are recorded. In the graphs, information about the sensor “07 AVC controller (Ringtee 1)” from 10:00 to 11:00 is shown as an example.



Figure 16. Dashboard Component 1. Heatmap Assessment.

The second visualisation is shown in Figure 17. This is a line graph, and the information displayed is dependent on what sensor has been chosen. The chosen sensor can be seen at the top of the graph. Similarly, to the previous graph, hovering over any point will reveal the exact time and number of events for that hour.



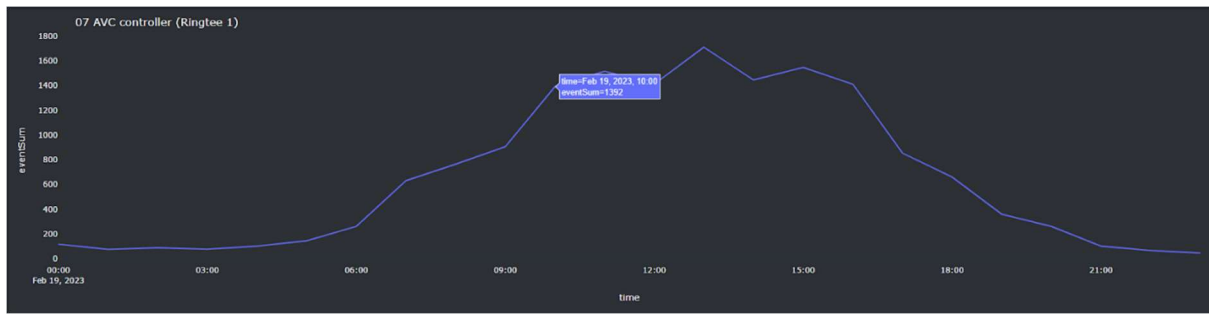


Figure 17. Dashboard Component 2. Line graph.

The final visualisation on the page depicts a map with all the sensor locations. This makes it easier to pinpoint where a particular sensor is. This is seen in Figure 18.



Figure 18. Dashboard Component 3. Map.

The dashboard is also interactive, with users being able to change the date with a date picker and display data for another date, and users are also able to choose between the vehicle, pedestrian and bike module via a drop-down menu. The full sensor monitoring dashboard can be found at the address: <https://its.cs.ut.ee/iotcheck/>

## 4.2 Testing

It was decided that automated unit tests should be implemented to test the sensor monitoring dashboard. For this, the Python library “unittest” was used. Unittest allows users to automate tests for particular functionalities within a Python file. It works similarly to Junit, where test suites can be defined and run, which then output a summary of the tests that were run. Using this, the file *testClass.py* was created to test the contents of *app.py* [14].

Eleven tests were written to check the outputs of functions in the file *app.py*. The first four tests check the sizes of the created matrices. It is known that the events matrix and the assessment matrix should always have the same number of rows as sensors and always have twenty-four columns (for each hour of the day). Only the first five sensors were extracted from the database for these tests, so five rows should be generated in both matrices. Their events were found and then the functions *create\_event\_matrix()* and *create\_assessment\_matrix* were run and the number of rows and columns were controlled. For tests 5 and 6, the event numbers generated from the function *create\_event\_matrix()* were controlled. These tests involved manually

generating dataframes. For both tests, a single sensor was generated. For test 5, a single event that has an amount was created for the sensor. This event represents the type of events that Thinnect and ECO sensors generate. For test 6, two events were generated for the same hour and do not contain a value in the column amount. This represents the type of events that AVC sensors generate.

When running this initial suite, a problem arose with test numbers 5 and 6. In these tests, a sensor is created with a single event, where the time and amount is known. An event matrix is created for this one sensor and event using the function *create\_event\_matrix()*. However, when these tests were run the function ran into an error because only one sensor was inputted. When trying to create a matrix with only one row it is unable to due to a loop that uses *iterrows()* to go through the different sensors. *Iterrows()* cannot be used on a single-row dataframe thus the problem arises. While it is unlikely that only one sensor will need to be inputted into this function, this problem should still be fixed to account for future situations where matrices need to be created for only one sensor. The fix was implemented with a simple if statement that prevents iteration if only one sensor exists. All tests were run again, and everything passed.

Five more tests were then written. The next two tests involved checking the assessment matrix for correct assessments. Again, sensors and events were manually generated to simulate certain situations. For test 7, events were added only to one certain hour, and that hour was checked for a correct assessment, which should not show anything strange, and the assessment matrix should place a “2” for that cell to represent the green colour. For test 8, two hours with the same number of events were generated. This time, the assessment matrix should produce a “1” for the cell where repetition is detected to represent yellow. The final tests involved testing the *create\_day\_summary()* function. Here, different assessment matrices were generated to see if the output of the function is correct. One test checked if the cell is red when the assessment matrix has exactly six red cells, another checked if the cell is yellow if the assessment matrix has exactly one yellow cell, and the last one checked that the cell is red when there is six red and one yellow cell. After running the tests, all tests passed so it is safe to assume that the functions used to generate event numbers and assessments work well.

## 5. Conclusion

This Bachelor's thesis aimed to provide the ITS lab Modal Split dashboard with a tool to monitor the sensor's status and assess the data quality via visualisation and analysis to facilitate data quality assessment. Before this thesis, the data the sensors sent had no prior assessment, so it was difficult to tell if a sensor was working correctly. The solution was to create a sensor monitoring dashboard that fetches sensor data from the Cumulocity platform, processes the data, and then displays the data assessment results on a website using Dash. The resulting dashboard is interactive; users can choose the date and module of which sensors they wish to look at. The dashboard displays information to the user in three main visual components. First, a heatmap displays an hour-by-hour assessment of the events recorded by different sensors, where the level of data quality is shown by three colours. This heatmap is accompanied by another heatmap which provides a summary for each sensor depending on the results throughout the day. Secondly, a line graph allows users to click on a sensor and view the trend of event numbers throughout the day. And lastly, a map showing the sensors' locations to the users.

The aim of the thesis was adequately fulfilled; however, there is still potential for more features in the future. Notably, the assessment done on the sensor data was based on a direct check, and only two aspects were explored. There can be more advanced approaches using probabilistic and statistical methods or machine learning for anomaly detection. In the future, implementing an algorithm that better identifies anomalies could prove useful, similar to the approach mentioned in Kandel et al.'s paper [6].

## References

1. Zhang L, Jeong D, Lee S. Data quality management in the Internet of Things. *Sensors*. 2021;21[17]:5834.
2. Rose K, Eldridge S, Chapin L. The internet of things: An overview. *Internet Soc ISOC*. 2015;80:1–50.
3. Khoshkhah K, Pourmoradnasseri M, Hadachi A, Tera H, Mass J, Keshi E, et al. Real-time system for daily modal split estimation and OD matrices generation using IoT data: A case study of Tartu City. *Sensors*. 2022;22[8]:3030.
4. Pipino LL, Lee YW, Wang RY. Data quality assessment. *Commun ACM*. 2002;45[4]:211–8.
5. Perez-Castillo R, Carretero AG, Rodriguez M, Caballero I, Piattini M, Mate A, et al. Data quality best practices in IoT environments. In: 2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC). IEEE; 2018. p. 272–5.
6. Kandel S, Parikh R, Paepcke A, Hellerstein JM, Heer J. Profiler: Integrated statistical analysis and visualization for data quality assessment. In: *Proceedings of the International Working Conference on Advanced Visual Interfaces*. 2012. p. 547–54.
7. Cumulocity GmbH. Introduction to our IoT platform - Cumulocity IoT Guides [Internet]. 2023 [cited 2023 May 5]. Available from: <https://cumulocity.com/guides/concepts/introduction/>
8. Cumulocity GmbH. Using the REST interface - Cumulocity IoT Guides [Internet]. 2023 [cited 2023 May 5]. Available from: <https://cumulocity.com/guides/microservice-sdk/rest/>
9. Srirama SN, Mass J, Koppel MK, Sepp A, Avanashvili S. Smartphone-based Real-time Sensing and Actuation with the Cumulocity Internet of Things Platform.
10. Fortino G, Guerrieri A, Pace P, Savaglio C, Spezzano G. Iot platforms and security: An analysis of the leading industrial/commercial solutions. *Sensors*. 2022;22[6]:2196.
11. SilverLaius. SilverLaius/cumulocitypython: Repository of the cumulocitypython package [Internet]. 2020 [cited 2023 May 5]. Available from: <https://github.com/SilverLaius/cumulocitypython>
12. Global Competency Center IoT. SoftwareAG/cumulocity-python-api: Python client for the Cumulocity REST API. Created by Global Competency Center IoT [Internet]. GitHub. [cited 2023 May 5]. Available from: <https://github.com/SoftwareAG/cumulocity-python-api>
13. Plotly. Dash Documentation & User Guide | Plotly [Internet]. 2023 [cited 2023 May 5]. Available from: <https://dash.plotly.com/>
14. Python Software Foundation. unittest — Unit testing framework [Internet]. Python documentation. 2023 [cited 2023 May 5]. Available from: <https://docs.python.org/3/library/unittest.html>

## Licence

Non-exclusive licence to reproduce the thesis and make the thesis public

I, **Maria Küüsvek**,

(author's name)

1. grant the University of Tartu a free permit (non-exclusive licence) to:

reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, my thesis,

**A Monitoring System for Daily Feedback on IoT Sensor's Status in Tartu City,**

(title of thesis)

supervised by **Amnir Hadachi Ph.D.**,

(supervisor's name)

2. I grant the University of Tartu the permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work from **09/05/2023** until the expiry of the term of copyright,
3. I am aware that the author retains the rights specified in points 1 and 2.
4. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Maria Küüsvek

**09/05/2023**