UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Artem Grukhal

Data acquisition and preparation toolbox for Cumulocity-based solutions

Master's Thesis (30 ECTS)

Supervisor(s): Pelle Jakovits, Ph.D

Tartu 2023

Data acquisition and preparation toolbox for Cumulocity-based solutions

Abstract:
The development of city infrastructure is growing in both speed and quantity. The concept of making the city a "smart city" is needed, so that the infrastructure development can be efficiently adapted to the growing population, and further directions can be predicted better, with less redundant work from the city officials. However, there are numerous challenges that one has to overcome in order to apply a concept and make it work regardless of the size of the city, its population, or any other parameter. Moreover, to efficiently handle the data from such applications, there needs to be a system that will help data scientists to overview, analyze and operate with data produced by smart cities. This thesis will try to propose a solution with features such a system should possess to maintain smart city applications and outline the most important requirements for a system to be functional. It will look into existing systems for large-scale data visualization and analytics platforms and try to evaluate the working models against a scenario of the Internet of Things system of Delta building in Tartu University, which is currently using Cumulocity Internet of Things platform for working with data. The advantages and disadvantages will be assessed against the above-mentioned scenario, an architecture for the solution to the scenario will be presented, and fruitful conclusions that can be made out of this research will be presented. Further, the solution will be introduced and described along with the reference to the previously made requirements, and finally, an evaluation of such a system against the requirements and for the scenario is to be performed.

Keywords: Cumulocity, smart city, data science

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

Andmete kogumise ja ettevalmistamise tööriist Cumulocity-põhiste lahenduste jaoks

Lühikokkuvõte: Linna infrastruktuuri areng kasvab nii kiiruselt kui ka mahult. Linna "targaks linnaks"muutmise kontseptsioon on vajalik, et linna taristut saaks tõhusamalt kohendada kasvava rahvastiku keskkonnas ning tuleviku suundasid paremini ennustada, vähendades linnaametnike üleliigset tööd. Siiski on palju väljakutseid, millest tuleb üle saada et rakendada targa linna kontseptsiooni ja panna see toimima sõltumata linna suurusest, rahvaarvust ja teistest parameetritest. Veelgi enam, et selliste rakenduste andmeid tõhusalt hallata, peab olema süsteem, mis aitaks andmeteadlastel arukate linnade toodetud andmetest ülevaadet teha, analüüsida ja nendega töötada. Käesolevas lõputöös püütakse välja pakkuda lahendus, mille funktsioonid on vajalikud targa linna süsteemide nutikaks toimimiseks ja kirjeldatakse kõige olulisemad nõuded süsteemi toimimiseks. See töö uurib olemasolevaid suuremahuliste andmete visualiseerimis- ja analüüsiplatvorme ning proovib hinnata nende rakendandatavust Tartu Ülikooli Delta hoone asjade interneti stsenaariumi jaoks, mis praegu kasutab andmetega töötamiseks Cumulocity asjade interneti platvormi. Eeliseid ja miinuseid hinnatakse eelnimetatud stsenaariumi kontekstis, kirjeldatakse lahenduse arhitektuuri ja tehakse järeldusi uuringust. Edasi tutvustatakse ja kirjeldatakse lahendust koos viidetega varasemalt defineeritud nõuetele ning lõpuks viiakse läbi süsteemi nõuete valideerimine eelneva stsenaariumi kontekstis.

Võtmesõnad:
Cumulocity, tark linn, andmeteadus

CERCS:
P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Contents

1 Introduction **6**

2 Background **8**
2.1 Internet of Things 8
2.2 Cumulocity 8
2.3 Smart city 10

3 Related work **11**

4 System Analysis **14**
4.1 Current state of affairs 14
4.2 University of Tartu's Delta scenario 14
4.3 Requirements 15
4.3.1 Navigation 15
4.3.2 Data retrieval 15
4.3.3 Filtering and aggregating 16
4.3.4 Data visualization and analysis 16
4.3.5 Application of user-defined algorithms 16
4.3.6 Dataset information 16
4.4 Validations 17

5 Design **18**
5.1 System design 18
5.2 Desktop application 19
5.3 Web application 19
5.4 Complete solution for a toolset 20
5.4.1 Containerization 21
5.4.2 Selection of an approach 21
5.5 Module architecture 22
5.5.1 Data extraction module 22
5.5.2 Algorithms module 22
5.5.3 Visualization and Inspection modules 23
5.6 Solution architecture 23
5.7 Storage 24
5.7.1 File storage versus database 25
5.7.2 File type 25
5.8 Communication 28
5.9 Job queue 29
5.9.1 Workflow in depth 29

5.9.2 Comparison of RabbitMQ and Apache Kafka 30
5.10 Backend project design 32
5.10.1 Server and framework 33
5.10.2 Credentials 35
5.10.3 Services 36
5.11 Frontend project design 38
5.11.1 Framework choice 38
5.11.2 Pages 38
5.11.3 Visualization 39
5.12 Algorithms implementation 40
5.13 Data function design 42

6 Validation of requirements **45**
6.1 Download a dataset 45
6.2 Read information about a dataset 46
6.3 Visualize a dataset 48
6.4 Apply an algorithm to a dataset 48

7 Evaluation **50**

8 Future work **54**

9 Conclusion **57**

References **60**

Appendix **61**
I. Glossary 61
II. Licence 62

1 Introduction

With the world of data acquisition, transfer, and storage evolving, the need for administering data and processing it is growing exponentially. One such case can be a smart city solution, which organizes the data around many different devices, sensors, statistics, etc.

Contemporary smart city solutions, and the databases they may use typically output a great amount of data that is available for processing in any way possible, however with some complications: data scientists have to be ready to dive into a vast ocean of data, from various devices, and be ready to filter it, organize, transform, visualize and perform many other operations in pursuit of extracting useful knowledge out of it.

One of the examples of databases that the smart city solution may utilize would be the Cumulocity Internet of Things platform, which was adopted by Telia in Estonia, and is used by a number of companies and local governments (e.g. Tarty City). As an example, the University of Tartu has used the platform to collect information on various measurements around the Delta building with the intent to further use this system in further research. Despite Cumulocity having the possibility to export data, collect it and perform simplistic operations, there is often a need to go beyond the services provided by a single vendor.

One of the areas for improvement can be the data extraction feature, which allows downloading Comma Separated Value datasets, which limits the user in the input parameters for creating a dataset to be exported or requires additional effort to provide the parameters needed for this. Additionally, it is not possible to apply processing to datasets other than data aggregation by time, which will later require a user to perform such operations additionally, after the dataset is downloaded.

Such operations may be complicated further when software used to operate with data doesn't allow to fully grasp the data scientist's vision and extra work is required to make use of data, which presents extra losses of time, which in turn cause additional money expenses.

In pursuit to overcome overhead work, and help a data scientist to perform his job, this thesis will focus on creating a tool, which can be used to alleviate complications and save time working with the Cumulocity platform.

The work is going to be distributed in the following manner:

Firstly, the background works will be discussed, in order to form a preliminary opinion on the current state of art and what kind of functionality should be present in the toolset to be developed.

The second section will summarize and propose our own considerations about the system to be developed. The requirements will be presented, as well as an introduction to the current scenario that is the main driver for this work.

The third section will guide through the development process outlining the important aspects that came out during the development phase, and the overall structure of the application will be presented.

Finally, there are conclusions to be made in the final chapter, discussing the platform that was developed and foreseeing the future scope of work that may be chosen to enhance the capability of this toolset.

2 Background

This section will provide the necessary background to further explain the motivation behind this work and provide additional materials for reference.

2.1 Internet of Things

There is no common definition of what the Internet of Things is however, various sources present it as a concept of physical devices connected by any means to form a unified structure, where the nodes may communicate the information to a chosen media and therefore output data. The Internet of Things is an evolving field of academic study and has already received many applications in other fields, such as industry, healthcare, entertainment, military, etc. This is a parent concept to many others, including smart cities, which will be explained in detail in the following sections.

The Internet of Things (later referred to as IoT) allows for a better and automated experience of analyzing and making decisions, a good example to understand what an Internet of Things is would be a system presented in a work B. Nast et al. [31] where the author talks about an IoT setup comprised of various sensors and devices integrated into air conditioning and cleanroom technologies facilities, which later output data to a central hub and provide the possibility to diagnose and suggest optimizations for said facilities.

The result of the IoT setup presented, from this example, may allow to make the manufacturing of such systems more energy efficient, and therefore not only save costs for company, but also benefit the environment.

2.2 Cumulocity

Cumulocity IoT of Software AG [7] is a tool that allows connecting your devices, sensors, and virtually everything that can communicate with Cumulocity through supported protocols, to persist, analyze, visualize, and perform many other operations on data that comes to this platform. There are many possibilities to utilize this tool, which can be used to operate both real-time and historical data remotely.

Figure 1 presents the technical overview of Cumulocity's key concepts and how they interact with each other. Here, the Inventory contains the configurations, device structures, and their connection data. The devices can interact with the tiles placed around the inventory:

- Measurements - data produced by the content of Inventory, for example receiving a thermometer reading.
- Operations - data sent towards the content of Inventory, for example, to send the signal for an update to a device.

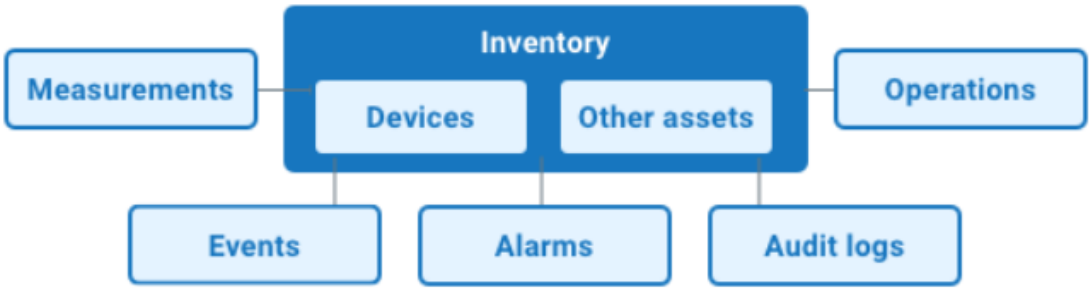


Figure 1. A domain model of Cumulocity structure [7].

- Events, Alarms, and Audit logs - events are configured triggers based on the incoming data. These events can later be an alarm - which is a notification of an event happening, or an audit log, which can be generated for security reasons.

The data type in this thesis will mostly focus on working with measurements datasets, as the operations are used to act on the device and typically rarely used for analysis, and events, alarms, and audit logs are more difficult to analyze, as they only indicate an occurrence of a specific action, and will need additional operations to create a meaningful dataset out of them.

As this thesis will heavily rely on data produced in the Delta building from the University of Tartu system, for the simplicity of understanding this thesis will treat Cumulocity as a real-time database, that stores the information it receives and is able to visualize this information as it arrives. The Delta building stores the data it receives to Cumulocity and later uses it to process the information and supply it to other applications that make use of this data.

As it was described in the introduction, downloading the dataset from the Cumulocity database manually requires additional effort by configuring a UI dashboard in the Data Explorer tool, or by making an export configuration manually selecting each column that needs to be downloaded, and providing query parameters to isolate the dataset to a specific device only, or a specific measurement type only.

In a scenario where a user wants to analyze the measurements for a given time period, considering the growing number of devices that produce measurements it will not be possible to use the configured data dashboard, as it will require a much greater effort to consolidate all the devices on a single dashboard. As for an option for creating an export file and downloading the data later: it allows the user to fine-tune the search parameters for a new dataset, however, the execution is such that the user is required to later download the dataset using a download link that comes to an email after an approval from either a team member or a supervisor, depending on the configuration. This further

creates overhead in scientists' work by having to keep track of emails and requiring attention from another scientist to supply access to a download link.

There are many other applications that Cumulocity has and can be used for, but this work will isolate the database application, since interacting downloading with data for further processing is what this work aims to solve.

2.3 Smart city

The core concept, which inspires this work is the smart city [23] - an IoT system to organize and use data collected from many devices on a large scale.

The project of Tartu (smart) City can be an example, where there are various sensors to monitor the state of several infrastructure services. For example, one of such services provides information on the number of cars leaving and entering the city through major outbound/inbound roads. There are sensors that count passing by cars, and data obtained from this application can get valuable insight on the traffic situation on major traffic nodes, and allow the city to plan additional measures to mitigate the congestion on said nodes, thus elevating the quality of life for guests and residents of the city.

The Delta building can be another good example. It has raw data collected by various sensors, which are then put into a Cumulocity system for storage. In practice, the notion of a smart city usually comprises data collected on a greater scale, bigger than the one of a single building, however for the purpose of simplification, this thesis will use the Delta building as an example of an IoT network, which is one of the parts of a larger smart city network. The same technology is used for the Tartu City network, so the findings of this work will be applicable to a larger scale as well.

3 Related work

To provide additional backbone to the work presented in this thesis, several scientific research materials were used to help see the current problems that need solving and define the requirements for a system that will help maximize the benefit for a data scientist.

To start with, in a work presented by Amy X. Zhang et al [33], we may find some good analysis of what the data scientists' duties are, what the general workflow looks like, and which tools they may use throughout their work. Mainly these three takes will be the central topic in this thesis since the main goal is to create a tool that will help data scientists in the first place.

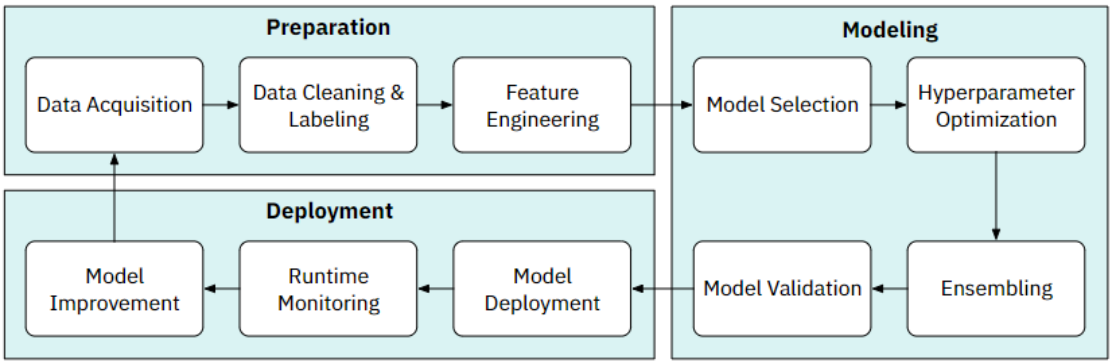


Figure 2. A data science workflow presented in work by D. Wang et al [32].

As we may see from Figure 2 the general workflow for scientists in this field can be expressed in three major definitions: Preparation, Modelling, and Deployment. The toolset developed in this work should concentrate on the Preparation stage, allowing the scientist to shape the data in a desirable way and giving an opportunity to extract data from Cumulocity, which is a limitation we have to abide by in this system. Finally, Feature Engineering is the final step in data preparation, which should also be mentioned in this work, exploring possibilities to give users the power to create versatile tools, to conclude their work.

Another good take from this work is the overview of tools that data scientists use in their work combined with the central topic, the collaborative process in this niche, which can provide good hints about how to create a system, that will be more intuitive to the users and blend into the working environment easier and become more useful. We may see, that the dominant tool for this line of work would be the Jupyter Notebook and similar software, providing the user with the ability to both, perform many different operations together with the ability to instantly visualize their work within the same environment. The collaborative process hints show, that currently, Jupyter notebooks don't provide the desired level of shared work environment, hence this will be another

useful trait for an application to support such nature of work.

Since this work focuses mainly on the preprocessing stage of data analysis, it will be beneficial to discuss the findings presented in the review by Cheng Fan et al [25]. We may see the overview of various algorithms and the classification of tasks one may need to perform with the data. The toolset that will be developed in this work will have to support such functionality or provide a way to create such functionality by means of custom algorithms that may be applied to a dataset. Another point to keep in mind is that since these actions typically are performed with the help of programming languages, authors' work hints that the system to be developed should support the language that may create such algorithms and be widely accepted among data scientists, which ultimately means that the parts of the toolset that deal with algorithms should be handled with Python programming language.

The last work, that should be pointed out in this chapter, is a Triangulum dashboard [26] which has very good examples of how to build a system, that is capable of downloading, processing, and displaying the data on a smart city scale. It is developed under EU Commission's initiative, and its main feature is to provide adequate visualization from large streams of data, and using a regular web browser to be able to provide a user-friendly interface for navigating around visualizations, and providing real-time response to changes.

The whole system is built into 3 layers presented in Figure 3: The Data layer, the Application layer, and the Presentation layer. The data layer is one of the main concerns in this case since it has the most challenges connected to it. It needs to collect data and store it efficiently, so authors suggest their way of implementing such manipulations.

Further, we may see the system overview, and its architecture from the software perspective: the authors introduce the main functionalities of the developed system: Filtering, Aggregation, Time series analysis, and Visualization.

From the technical perspective: Apache Web Services, Kibana, Logstash, and Elasticsearch were used for the server-side technologies, together with Python being used as a primary backend language. Front-end was mostly operated by plain JavaScript and jQuery frameworks, together with Plotly and Leaflet libraries as visualization tools. The backend and frontend were communicating through REpresentational State Transfer(later referred to as REST/RESTful) requests, backed by the Flask framework. This is a good technical example of what the application may be and provides insights on how to work with creating dashboards and having to use big amounts of data underneath them.

One of the closest examples of a similar work to the toolset is a Cumulocity IoT DataHub - a solution that allows to create a bridge between data stored in Cumulocity databases and a business need for the data. It uses an SQL engine, that allows exporting data with the help of various database connectors and is an additional tool, that has to be enabled and configured separately. DataHub is a tool for working with data, as it helps to operate with data in query languages, which are commonly used in data science

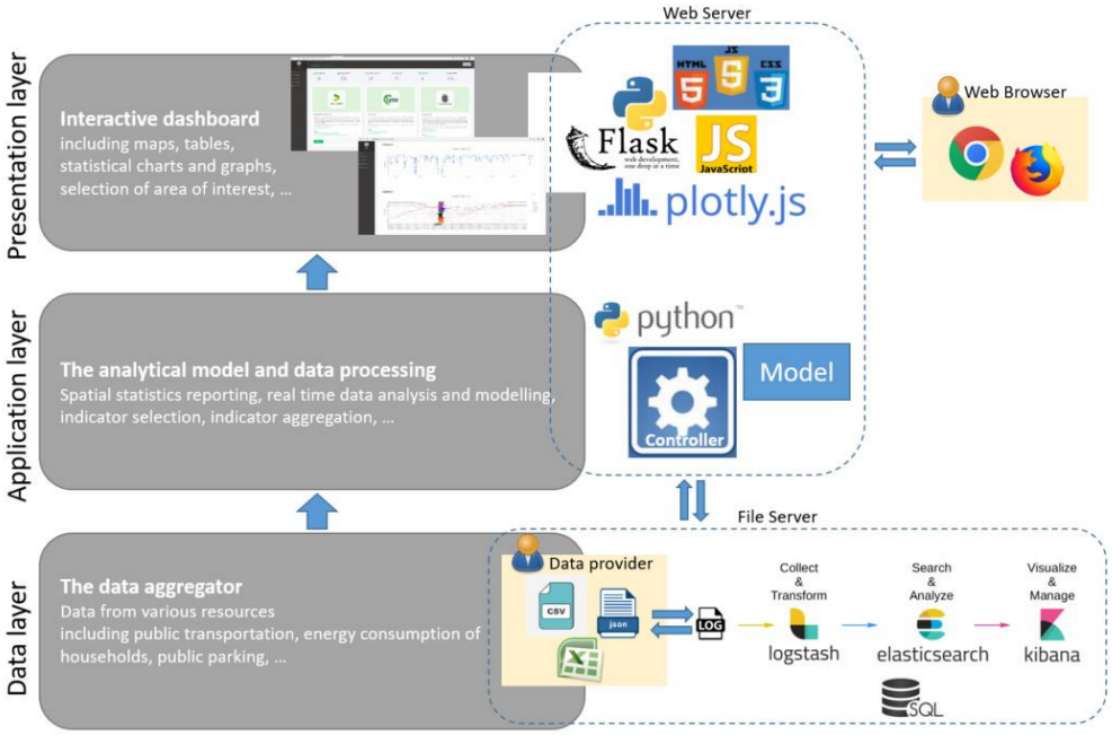


Figure 3. Three layers behind the Triangulum dashboard structure [26].

domain. It is also similar to the goal of this work - to provide easier access to data and opportunities to filter the datasets before they are downloaded. However, this is purely a connector between a database and a business application or a storage, and it is not applicable to data processing needs, as it only allows for querying the data. It also does not allow to visualize and explore data, as it is designed to be used with large-scale applications for a long term archiving, or analyzing real-time data.

To put everything together, these works are a backbone for a collaborative toolset for data scientists, where should be the possibilities to obtain datasets, process them, apply algorithms to them, visualize data, and analyze it. The next sections will dive into the specifics of such a system and suggest the possible features it should have, in order to be able to provide value to data scientists.

4 System Analysis

The system analysis chapter will discuss the requirements a toolbox should have by analyzing the needs of a current scenario that is working with the Cumulocity system to store data. These requirements will shape future work by helping to define a list of functionalities the toolset will possess.

4.1 Current state of affairs

The University of Tartu currently has a system for monitoring various indicators and sensors for display, which is so far difficult to operate with, given that all data is collected into a Cumulocity cluster. The main difficulty arises from the limitations of the Cumulocity system for processing the data that arrives and being able to manipulate and visualize it. It does not provide a convenient user experience to a researcher for doing his everyday work using this tool, often making him write his own software for various needs. Since visualization and data operations are vital in data science, there is an idea to create a toolset for these matters, which will help in everyday work, and reduce the time spent by a researcher for writing code. The monitoring system provided by Tartu will serve as the basis for a bigger target - the modernization of a smart-city project for Tartu City, which currently has several sensory devices as well, that will need to be organized into a centralized system for monitoring them. This system is also using the Cumulocity platform and will have similar limitations as to the university's situation. Therefore, further sections will take a closer look at the scope of the problem, that this work will be trying to solve, and provide some background works, which hold a similar idea and will be helpful for creating software for developers in the University of Tartu.

4.2 University of Tartu's Delta scenario

Delta Center building is a newly constructed building (it became active in year 2020) hosting students of various faculties. It is equipped with many sensors, that form a whole system for monitoring the state of different parts of the building with the intention to further analyze and process data. The system is currently run inside a Cumulocity environment and is open to use for research of students. This thesis will mostly concentrate on using Delta's Cumulocity environment since it will be legally easier to create software for the university's infrastructure. But this also enables to create a system very close to Tartu City's conditions and potentially to use the same system only for different Cumulocity environments.

4.3 Requirements

The driver is the data scientists' need to efficiently manipulate the data, in order to make accurate analysis using data retrieved from the Cumulocity platform. The manipulations, which will be interesting to a user are:

- Navigating categories of devices in the dashboard and observing the data that is present under each device
- Being able to configure data retrieval from Cumulocity
- Being able to filter and aggregate data
- Produce graphs and other visualizations based on the nature of the data
- Applying algorithms to solve various data-related issues like cleaning the data from null values or normalizing the data for later usage in a machine learning algorithm
- Give a compact overview of the dataset's measurement types and their relation to devices

4.3.1 Navigation

Since the work of this thesis will include the development of a dashboard, that is simple to utilize and provides the necessary functionality to be a data analyst's toolset, there has to be an understandable UI/UX (User Interface/ User Experience) along with the needed visualization features. The toolset will also need an overview of devices present in Cumulocity in order to be able to navigate through them with the intention of inspecting potential dataset filters.

4.3.2 Data retrieval

To have additional possibilities to process the data, at some point, a researcher may need to export the information from either Cumulocity or the toolset. For such cases, there has to be an opportunity to download and share the dataset in a universal format, which will be suitable for the majority of data science software (for example, CSV format).

The central source from which the data will come is Cumulocity. Even though there exists a possibility to download the data files from Cumulocity directly, it is not as convenient when it comes to downloading filtered data and/or partitioned data. Moreover, as the toolset may produce resulting datasets of its own, there has to be a simple yet efficient way to export datasets that are both - retrieved from Cumulocity and generated by the toolset.

4.3.3 Filtering and aggregating

The system that will be developed needs to have the filtering possible, in the sense that the data may be allowed to be filtered either when retrieving from Cumulocity, after when forming/preparing datasets, or both.

Aggregation is another requirement, with similar constraints, however, for aggregation it may be best to have a unified logic for aggregating the data, and hence it will make sense to only allow to download raw data, which can be aggregated later, upon requirement.

4.3.4 Data visualization and analysis

Finally, since a dashboard itself is a graphical application and the findings of data scientists are best presented in the form of graphs, maps, and charts, there has to be a functionality to provide the generation of such items. This will not only help to present the findings but also help in researching data for a scientist during the work.

4.3.5 Application of user-defined algorithms

To perform additional processing of data, which is not present in the scope of this thesis, there will be a way to develop algorithms with the intention of reusing them. The algorithms developed will have to be stored for future reusing and have a strict structure, that the user has to follow when creating an algorithm. This will ensure improved stability for the application in general, and allow for simplifying the subsequent user experience when applying these algorithms to a dataset.

User-defined algorithms are an important part of this work since they will be able to provide versatility and reusability to the specific features, that may be well needed by a data scientist. This feature will allow the creation of custom algorithms, save them, generate the UI for running them, and store the results, with the possibility to later reuse them. This way, if a team of data scientists works on a specific task, there is no need to have everyone with practical experience in developing complex algorithms, instead a single developer may populate the system with needed algorithms and the rest of the team may reuse them.

4.3.6 Dataset information

The number of datasets will essentially mean the growth of system complexity, and one of the main issues with such an approach will be to understand which files are the ones that are works in progress, and what these datasets contain. For this reason, there has to be a possibility to review the content and provide general information about a specific file on an on-demand basis.

4.4 Validations

To ensure that system development succeeded, and future development is feasible, there has to be a strategy for validating the results. There are several points that are worth discussing and can be separated into 2 categories: Data and Processor.

To ensure that data collected from the Cumulocity system is correct, there have to be a number of tests performed over the true dataset and the dataset downloaded and saved by the application. There should also be three different sizes for these tests, which would allow us to test the data thoroughly. The reason behind the three different dataset sizes lies in the approach that will have to be taken to data download, which may impact the data quality. Due to the download rate limitation placed by Cumulocity on its services, at some point, it will be needed to create a mechanism for combining this data, which risks data corruption. For this reason, there are three different sizes:

- Data correctness (5000 $\leq$ rows $\leq$ 500,000) - to ensure data received is the same as data sent
- Big data correctness (500,000 rows and more) - test similar to above, only with testing big size for a dataset
- Small data correctness (0 - 5000 rows)

  These tests will help ensure, that the dataset is received and saved consistently, with full data preserved.

Another set of validations will have to assess the downloading time progression over size of the dataset. It is important to inspect the tendency of downloading data with the growing number of records. Ideally, we want to see that the time does not grow exponentially, since otherwise, development may pose difficulties and another approach should be chosen

5 Design

This section will dive deeper into the chosen design for the application. All crucial building blocks will be described, providing reasoning as to why certain decisions were made and key points will be discussed in detail.

5.1 System design

The system design's first goal is to answer the requirements of the application defined in section 4.3. Thus, on the application level, the following modules are defined:

- Data extraction module - an abstraction that contains parts of a system architecture, that are responsible for data download. This module will address the data retrieval requirement and may address the filtering and aggregating requirement, as it was decided in subsection 4.3.3 that it may influence the functionalities of data extraction.
- Algorithm module - an abstraction responsible for running, storing, and loading user-defined algorithms. This module will influence the decisions in pursuit to comply with the requirements set in section 4.3
- Dataset inspection module - an abstraction responsible for getting tabular or textual information about the dataset. This module encompasses the needs of two requirements defined in section 4.3: dataset information and navigation.
- Dataset visualization module - an abstraction that deals with generating a graphical representation of a dataset. Answers directly to the requirement of data visualization capabilities and will also influence decisions related to the data analysis capabilities.

These are the core modules present in an application that should work as a standalone feature, regardless of the logic behind the functionality of one another's. This essentially means that they are part of an application, yet are not considered a feature, but rather a collection of features that each deal with a respective task. These modules may either be integrated as a monolithic service with various classes/functions, or separate microservices. This decision will be made further upon discussion of the limitations that arise from the architectural decisions.

All these components should share the codebase and environment, yet have a distinct functionality, which leaves us with a choice of a monolithic application instead of a set of microservices, that are responsible for each separate module. The monolithic application idea is influenced by the example of Jupyter Notebook[15], where we have a single environment to perform all the procedures that are necessary for any given task.

This is opposed to, for example, having multiple applications like the combination of Grafana [13] and Cumulocity for Dataset visualization and Data extraction respectively. The solution that this work is trying to achieve is one, where the overhead from using several applications is reduced, and collaboration is possible through sharing of resources developed inside an application. As a result, the decision is to make an application where everything can be navigated in a singular environment, yet the modules can be split into microservices to enhance the application performance where it is needed.

Now that it is decided that this will have to be a singular application, there is an important choice to make: a Desktop application, or a Web application.

5.2 Desktop application

Desktop application is a good choice when we want to utilize the productivity of the machine, that will be used for working with the toolset. This is an advantage since thoughtfully developed applications for a specific platform can use computer resources more efficiently, and therefore allow users to have a more pleasant experience. This is a good way to work with data, as, for example, Microsoft Excel [17], a standalone application for several most popular platforms [29] including Microsoft, Macintosh, and Linux which helps to observe the data, process it and visualize offline.

Such multi-platform capability is a desirable outcome, yet in the case of Microsoft Excel, it is achieved by having to develop for all three platforms, taking into consideration their respective complications and differences, which increases development time and maintenance difficulty greatly.

We want to make sure that the application runs flawlessly, and it is also a good idea to have it running offline, given that the dataset is already present, and the remaining work is to process it, visualize, and inspect it. However, we may also find that offline work is often not possible in the field of computer science, since there are many reasons why one would want to have a connection to the Internet: to find the solution to the problem online, to download additional datasets, to communicate with colleagues, etc. Therefore, it is not crucial for this application to have purely offline capabilities.

The desktop version cannot run solely offline and would still need to have a connection to the internet as per requirements. It can still be a viable option since the optimization is better, but there is also a possibility to create a web application which will be discussed in the following subsection.

5.3 Web application

A web application is an application that relies on an internet connection to operate correctly. It is a fully functional application, examples of which may be Google Sheets [12] - a competitor for Microsoft Excel that provides similar functionalities, and Colaboratory [11] - a competitor to Jupyter Notebook, which also provides similar functionality,

but is fully web-based. These applications can also have a desktop version for them, but the idea here is to focus on web variants, which are an alternative to the former.

A web application also has its advantages over desktop applications. Typically, web-based applications are multiplatform, since the UI is rendered with the help of a web browser, which is a more time-efficient way of creating an application that would be supported across different operating systems. It may come with a pitfall when there are certain libraries commonly used for developers that are poorly optimized for one of the popular operating systems, or browsers, which are usually required to have to run a web application. This is, however, a duty of a developer, to use libraries and generate solutions that will be truly multiplatform, and therefore a desirable feature for a toolset.

Web applications also have to reside on a server, which in turn may be a big advantage, especially when working with a big amount of data. Colaboratory, for instance, lets developers tune the amount of computer resources they want to utilize and such a possibility may enhance the processing power over large datasets for a reasonable fee. It is, however, still possible to use a personal computer as a server, limiting one's processing resources, but being able to run web applications.

Web applications are a good way to create more versatile, yet less optimized solutions, so there may be a way to compromise the two approaches and have a hybrid structure, where partly the computer resources will be used, and partly the logic behind a web application, which should allow for a toolset to be multiplatform be present.

5.4 Complete solution for a toolset

Having discussed both options, it is decided to opt for a solution, where both approaches are partially used.

One of the considerations for the application is the computing power it will use, since running algorithms on data of different sizes may use a lot of computer resources. Ideally, that is something that should be configured. Due to the complexity of adding such a possibility to the system, it is best to leave this part for future work, however, it would make sense to allow a user to run an application using his own resources, making user's expectations clear.

Running an application on a server adds complexity as well, since then there will be a need to manage sessions, and users and there will be an elevated security risk, which adds additional work to make an application safe and useful. For such reason, it was decided to run the toolset on a local machine. This way the application acts like a desktop application, yet leaves an opportunity to later be transformed into a web application that can be hosted remotely thus allowing the developer to later address the computing power modifications, as it will become an important feature in future work. Currently, it will not be possible to host an application on a remote server since this will require authentication and multiple-user context, which has to be maintained in order to allow this toolset to be used by many users.

It will be severely beneficial to have a multi-platform application as it greatly reduces the time to develop a product. Additionally, this will help to keep versions between multiple operating systems more consistent, allowing for a lesser margin for errors. There are several possibilities to achieve that: through an application developed with a programming language that is multi-platform and by containerizing an application.

5.4.1 Containerization

With this approach, we can ensure a similar experience on all machines, that are capable of running containers. A big part of containerization is to enable saving the configuration for an operating system in a single container. For example, we may have an application that uses additional libraries, which have to be downloaded independently. Or the environment needs an additional tool, in order to be able to run an application without any errors. Containerization helps to set up the system from a single configuration file and download automatically the necessary libraries and tools into an environment that intends to use a container.

5.4.2 Selection of an approach

It is worth noting, that both approaches are not mutually exclusive. To conclude the multi-platform options, we may see three scenarios: platform-specific application in a container, cross-platform application in a container, and cross-platform application without a container.

As it was noted above, this work will not include deploying an application to a server, and because of this, there is no obvious reason to make a containerized application at this stage of development. For portability reasons, this will be, however, a part of future work that will help to keep the application up to date with modern standards in the industry and improve multi-platform capabilities further.

There still is a need to have a multi-platform application. Platform-specific application in a container would still mean that an application is to be developed for a single platform, and it may also surface deeper problems that may be present if the application uses more specific features of an operating system. A better practice would be to have a cross-platform programming language used, that will still satisfy the requirements.

Finally, a web-based system, which is using a cross-platform language and run as a standalone system seems to be an appropriate choice given the requirements and limitations that are presented. Most common systems have two basic layers for a web application: frontend and backend. There is also a possibility for some languages to create a WebAssembly [27] application, which essentially gives a possibility to develop a web application with a single programming language. However, it is still better to have common layers as backend and frontend in order to be able to better manipulate both parts and have greater control over both layers' capabilities.

5.5 Module architecture

Some modules introduced in previous subsections will require additional considerations, that will influence the overall system architecture. This subsection will further explore what limitations are applied to this work and will add final redacts to a schematic.

5.5.1 Data extraction module

The first thing that should be considered is the backbone of the entire system - data. Data will be mostly acquired through Cumulocity, so this module needs to essentially be a wrapper for downloading datasets from the server. Data will also come in different sizes, so it is important to keep in mind that the download times are to be different. It is a bad user experience, to have to wait until the dataset is loaded in order to use the application, so it would make sense to add a job queue.

Instead of having to wait until the download is complete, a job queue allows storing the parameters that will tell which pieces of data exactly are to be downloaded, and schedule them for later download, asynchronous and independent from the main application.

Once the dataset is downloaded, it needs to be stored somewhere. Since Cumulocity provides data in the form of rows and can be returned as a JavaScript Object Notation (JSON) or as a Comma Separated Value (CSV) file. JSON could be stored in a database, and it can be done in multiple ways: relational database and non-relational database.

During the experiments on choosing the downloading approach, it was found that downloading the data in the form of JSON object will degrade the download times drastically. The reason for this is that Cumulocity is only able to return 2,000 records per page. Suppose we need 100,000 rows of data. This will mean, that the server will have to make $100,000 \div 2,000 = 50$ requests in order to download the needed amount.

The CSV-based approach also has complications. After a certain number of records, Cumulocity will not return the CSV response anymore, but create a file inside the system, and give information about this file. This, however, can be easily handled by having a check for the response and using Cumulocity Application Programming Interface (later referred to as API) to later download the file. However, it only requires one request to download the CSV response, and at most two requests, if it is needed to download the file generated inside Cumulocity, which reduces network overhead that would be caused by JSON responses, and therefore is a preferred approach.

5.5.2 Algorithms module

In the algorithms module, we want to be able to run algorithms over datasets that were downloaded. We also want to be able to create these algorithms and be able to reuse them later. To achieve this, there has to be a system, that will understand how to render

UI for a certain algorithm, and how to supply values and run it programmatically. This means, that a programming language of choice must be able to inspect the code files programmatically, instantiate a class programmatically and run functions all while the user is navigating with UI.

The best practices in the data science domain and such restrictions suggest having Python as a core programming language for a server. It serves multiple purposes and will provide an easy and efficient solution to multiple problems: the possibility to programmatically run algorithms, and work with most libraries used by modern data scientists (such as Pandas, Numpy, scikit-learn, and others). It can also be used as a language for the backend, communicating with the Cumulocity server, and providing information to UI.

5.5.3 Visualization and Inspection modules

These modules both put limitations on the visual layer. The server can handle providing data for both modules easily, with simple client-server communication.

It will be the Visualization module's responsibility to be able to generate graphs for data that is provided to a user. Additionally, it should be extensible, to allow for new types of graphs added, should there be such a need.

When developing the frontend it would be unwise to create custom visualization libraries, since it will require a great effort to create working graphs, that can adapt to different data ranges and be versatile in general. The frontend has to support third-party libraries for graph generation and be written in a language that has such libraries.

The Inspection module also dictates that information about a dataset should be provided in a convenient manner of displaying information about a dataset, which is typically a table, that is easy to read. It does not add any direct limitations, however, it points out that having these modules in addition to visual navigation between different functionalities of a toolset suggests that a third-party component library will also be of help since it will reduce the possibility of components being incompatible with one another, and make the toolset itself more visually pleasant.

5.6 Solution architecture

To summarize the previous subsection, there are now definitive limitations and requirements that can help shape the whole system. There need to be several components:

- Storage - a way to persist datasets that the user downloads or operates with. Storage needs to be able to store CSV files, or equivalent if a better way to store CSV output is found later.
- Job queue - a mechanism to allow for asynchronous handling of download, to later be able to put the result into storage.

- Frontend - a visual aid for a user, to be able to navigate the application and control the processes easily and intuitively.
- Backend - a central server, that will orchestrate all communications between listed components. It will be the core of the application, so it needs to have access to all of the components simultaneously.

Figure 4 provides the chosen approach for building the toolset and contains main components and more detail on how the system will communicate. It is important to note, that the Cumulocity server represents a node, that other components will have to communicate with, and it is needed to showcase the different types of communication and for later justification of chosen technology. The key points from Figure 4 are storage, communication, job queue, frontend, and backend, which can be broken down into 2 systems: data download function and server.

In the following subsections reasoning for technology stack choice will be provided together with more limitations that occurred in the process of technology selection.

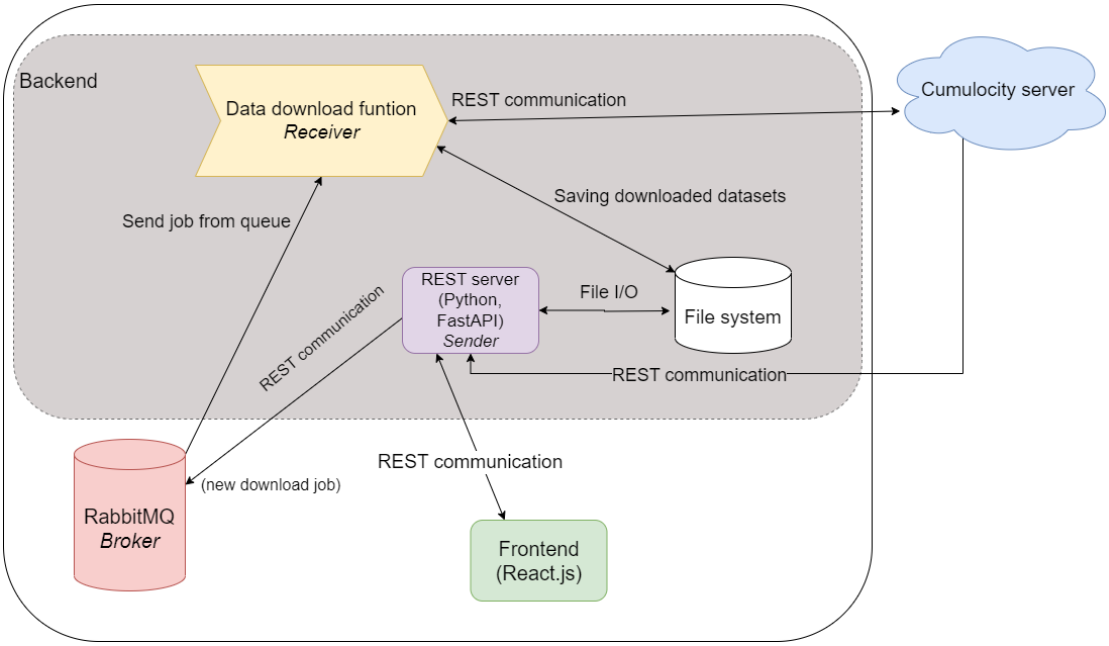


Figure 4. The proposed solution architecture.

5.7 Storage

To store data efficiently, there has to be a system that directly answers the requirements outlined earlier in section 4.3. There are several ways of storing the datasets obtained

from Cumulocity, and this section will provide reasoning and deeper knowledge about the proposed solution.

5.7.1 File storage versus database

The first consideration that comes for storing data is to select whether it will be an internal file system or a database.

The choice in favor of storing data using an internal file system was made because of the nature of a user's work. This toolset assumes that it will be used for pre-processing data, before supplying it to a custom algorithm, custom visualization, or other application for further work with the data. Because of this, having data stored in a database will require additional actions and restrictions from the user.

Suppose we have historical data, downloaded from Cumulocity in CSV format: after data is downloaded, it is stored in a variable, in the program. Consider the scenario, when we use a database as a storage system. CSV format would have to be converted before being inserted into a database. In the future, when a user needs this exact piece of data, he will be required to supply a query to find a dataset downloaded earlier. Later, when the user wants to run an algorithm over this dataset, he will have to be familiar with the client that is used for the database in order to be able to run a query inside the custom algorithm. Finally, to export the data, he will have to make sure an algorithm, instead of saving this data to a database produces a file, that can be later reused elsewhere.

These complications make a potentially bad user experience, and impact negatively the requirement of making this toolset an aid to collaboration between various data scientists. Therefore the idea of having file-based storage proves to be beneficial.

5.7.2 File type

After choosing the file system to be the main persistence mechanism for this solution, it should be decided in which format the datasets that are downloaded from the Cumulocity will be. There are a number of frequently used dataset formats that data scientists commonly encounter during their work. Some of the more popular may be Apache Parquet [3] files, CSV files, and sometimes JSON files. Additionally, it was considered to have a database for storage, that would store permanently the data from Cumulocity without the need to download it again. All of these options have their benefit, but CSV proved to be more beneficial for the purpose of this work.

Apache Parquet. Apache Parquet (later referred to as "parquet") file type, is a specifically organized file for columnar data, that is commonly used in high-performance data input and output (short version - I/O).

This is a good alternative if the developer has plans to operate on real-time data or large datasets, and needs the best way to process files quickly, as the information arrives.

It is common to use this file format in MapReduce [24] model, where the data is analyzed as it arrives, and later only the result is kept. However, such file format is close to the structure that the database has, and essentially, is a simplified version of a file database.

This approach is also not the first choice, since it requires specific software to be read into the program, and will require additional knowledge from a developer to use it in his work.

There is a benefit to using it together with streaming data, for example, Cumulocity provides the possibility to stream real-time data that is received by the server, and there are tools to receive this data. This feature is not used in this diploma, since it adds a great deal of additional difficulty in storing data and overhead software processing.

Streaming is built with the help of WebSocket protocol and an older version that is advised against by Cumulocity - "Bayeux" [28] protocol. Both are used to receive events from the Cumulocity server, and they differ in the age of technology (WebSocket being a newer communication protocol) and performance. Since Bayeux protocol is not advised to use, it would make more sense to discuss WebSocket.

WebSocket in its configuration defined by Cumulocity is limited in its lifetime resulting in a need for a mechanism, that will keep the connection alive by reconnecting every five minutes and checking for potential data misses. Another issue is the double notification when the notifications received by the system have to be checked for potential duplicates and handled accordingly.

Finally, another potential issue is the usage of JSON as the transfer media for data sent by Cumulocity. It was discussed in Section 5.5.1, that the historical data will be downloaded with the help of CSV format, as it is beneficial for bigger dataset sizes. Addition of JSON processing module risks having different storage types, which will be solved either by an additional storage system, that can cope with different file types, or a converter software, that will add additional processing overhead to streaming data and needs to be solved accordingly.

Due to these reasons, it was decided to postpone streaming functionality for future work and will be discussed in greater detail later. As such, the parquet file type loses its value of being efficient for streaming data, and even though it is more performant than the other alternatives, having a need for additional software to parse them makes it more complicated to satisfy the collaborative condition. These reasons make it too complex for a solution presented and not the best candidate for data storage.

JSON. Another file type considered for this application was JSON format. The main reason for this was the possibility to have both historical data and the data that is coming from streaming uniform and eliminate the need for a conversion mechanism.

However, as it was discussed earlier, there are still difficulties with using this format. The first difficulty comes from using the CSV format as a response from Cumulocity for historical data. It means that the conversion back from CSV to JSON has to take place

and it will slow down the application that is already working with data which may take time to download and process. Additionally, the use of streaming technology would still require a great amount of effort to be able to cohesively work with the service that would download historical data.

Additionally, JSON files require more symbols to be stored, which essentially means that the size of an application will be greater. For instance, in its raw implementation, JSON received from Cumulocity can be compared to the same data received in CSV format.

Figure 5. An example of JSON data received.

Figure 5 is an example of what could be sent by Cumulocity. When counting every symbol except for whitespace, tabulations, and new lines (these symbols can be ignored in JSON and are shown in an example purely for a reader's convenience), we may see, that the number of characters is 145.

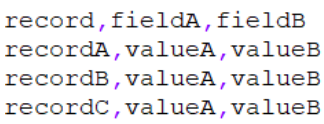


Figure 6. An example of CSV data received.

Counting symbols in the same manner for an example of a CSV formatted response presented in Figure 6 gives us a value of 86 (83 characters and 3 new lines, that are counted for the CSV file).

The reader can see that adding more records to a JSON file yields more symbols than the CSV file, which consequently means a greater file size when storing in JSON format.

Considering these arguments, the final paragraph will showcase a CSV file to conclude the search for a preferred file type.

CSV. CSV is another widely used format for storing tabular data. One of the main reasons it became the choice for this work is that it can be used with many scientific tools and speed of use. Because of CSV's simple structure, most parsing software can ingest the data from a CSV file quickly and is less prone to fail.

There are drawbacks to using such file type, mainly because it is limited in the type of data that can be stored. It is similar to the Apache Parquet format in this regard, where there can only be cells, and if a user wants a specific cell to have extra properties, there should be an additional mechanism for parsing the properties into a single string so the structure is not corrupted. JSON on the contrary is more flexible in this matter and allows for adding more properties effortlessly.

This issue is, however, mitigated by Cumulocity not having the freedom to send free-form data. When data is requested in CSV format, it will automatically convert objects of data into a table.

Resulting decision of using CSV as the main file format was made mainly because of good and consistent interoperability between Cumulocity and the server that it will communicate with. As a result, this file format will satisfy most requirements that will be impacted by it, and not lose performance drastically.

5.8 Communication

Conversation between the server and Cumulocity is possible only in a RESTful way with the help of Hyper Text Transfer Protocol(acronym - HTTP). Another component of the architecture was also using the HTTP protocol underneath - the RabbitMQ [18] broker. RabbitMQ however, has a client library provided for several programming languages, that uses HTTP for communication, so it reaffirms that the server has to be able to fully support HTTP communication, as it will be the main protocol.

A big component that is also dependent on HTTP, and RESTful transfer particularly is the UI layer. This choice was done due to the server limitations, which is mostly working with HTTP protocol otherwise. There could be a possibility to use a gRPC protocol [14] that is faster than RESTful communication, but due to previous examples, where the server has to use HTTP protocol still, it was decided not to add more complications to the system. This, however, can be addressed in future work, provided there is a clear need to switch protocols.

Another way several components communicate is by file I/O. Main programming languages are able to open, read, write, and append to files regardless of the operating

system which doesn't add more limitations to the choice of programming language. There is, however, a need to orchestrate both: the server and the data function to use the same source (file directory in this case) to locate the files they create or modify.

The reason for such a decision can be seen in an example. Suppose a user downloads several datasets and intends to later use them in an algorithm. To do so, a user will need the possibility to list all available datasets or look by a specific name, provided by either a user or a system. The latter experience is not user-friendly, since it may create a clutter of unnecessary files when this scenario is applied repeatedly. Limiting users to only use a dedicated directory handles this issue, and allows for greater security since there is no probability that the important computer's system files get corrupted.

5.9 Job queue

Due to the requirements defined in sections 5.1-5.8, the job queue has to be a major link between the server and a function that handles data downloads. Particularly, this concerns the storage being limited to a local file system, data extraction module and the communication media choice. The task that is solved by a job queue is straightforward - to be able to download data in the background, so there the best candidate would not have to be suited for big-scaled infrastructure.

The reason for this is that in a locally hosted application, every request is typically made by one user. The download queue is not estimated to go beyond the capabilities of simple queues, therefore there will not be much load that has to be handled. As was pointed out in a previous section, a message queue also needs to have access to a local file system which makes it impossible to have a message queue residing on a remote machine.

A RabbitMQ message queue was chosen to handle the jobs that are requested by the user. One of the alternatives that were considered was also Apache Kafka [2] tool, which scalability allows to have a smaller scale application using residing in a local environment as well. It also can act as a database, which potentially can help with file persistence, which can be used as a shared space for the data function and the server. To compare the two options, first, the workflow of the interaction between two components should be defined.

5.9.1 Workflow in depth

To help with understanding the needs of a system and compare possible candidates based on suitability to the workflow, the simplistic schematic for a message queue's involvement is presented.

Consider Figure 7, which focuses on the interaction aspect between key components. The server (later referred to as *producer*) is responsible for creating a message with parameters specified by a user.

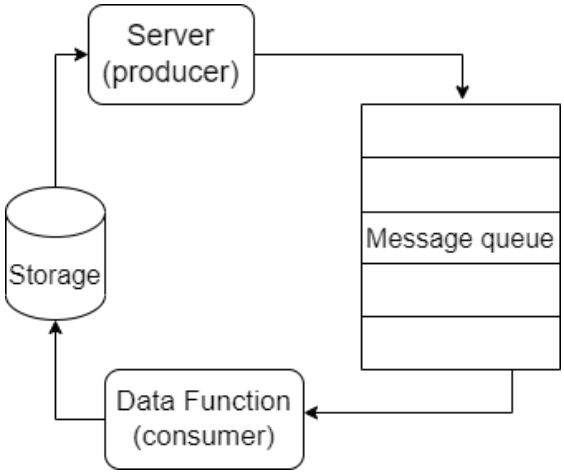


Figure 7. The workflow of a server to data function exchange.

Then, the message queue stores the messages from *producer* and prepares to download them upon the availability of the computer's resources. There are various possibilities as to how the availability can be defined, for example, one of the common ways is to have several working threads, that are able to independently process messages received, however, this will not be considered for the sake of simplicity.

When the resources are available, the message queue will assign the next message to an available data function (later referred to as *consumer*). *Consumer* will process the message to separate the parameters into individual variables and use them to create a job that will run until the dataset is downloaded and finished. This will be explained in more detail in the section dedicated to data function design.

After the download is finished, *consumer* saves a dataset to a specified directory and is available for work again. The saved dataset can now be used by the server to satisfy the user's needs. Note that the *producer* is not restricted from sending more messages while the queue still has previous messages by the same *producer*.

5.9.2 Comparison of RabbitMQ and Apache Kafka

The difference in both systems' architecture is the main driver for choosing the broker. Mainly because Kafka architecture is unnecessarily more complex than RabbitMQ.

The model of Kafka visualized in Figure 8 depicts the simplified version of components present in this software and how they interact. Mainly it is a simple message queue, that relies on the concept of producer and consumer, however, this system was developed to be used in bigger applications than the one created in this work.

In Figure 8 the dashed line represents components that add additional complexity to the solution:

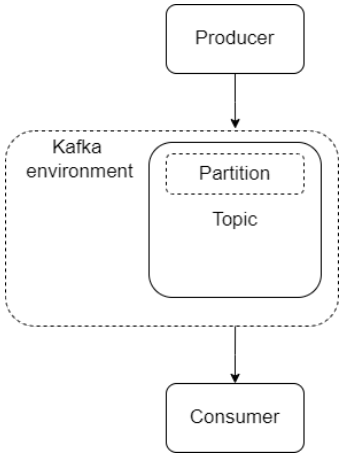


Figure 8. The model for a simple Apache Kafka system.

- Kafka environment - this is essentially a cluster, that may contain configurations for the message queue and will apply them to every topic present. The problem here is that this cluster relies on additional services in order to be operating. Apache Kafka is not mature enough to act as a standalone application and will require extra effort from a user to install and use. One of the main complications is the absence of a certified Docker image developed by the vendor, which could ease the installation and running of this environment.
- Partition - partition is a feature of Apache Kafka to break down the messages into specific partitions that can also be replicated for better system reliability. Since Kafka expects high-volume data, it is designed to accommodate a big number of messages, that are eventually translated to what are called "log files" and stored inside Kafka. Such measures for fault tolerance are not relevant to the toolbox, since it is not expected to work with a high-volume message stream

In Figure 9 main components of RabbitMQ architecture are visualized. It is simpler to create the most basic workflow, since RabbitMQ is designed as a general-purpose broker, and is not as restrictive toward the volume of a message stream.

The diagram presented on Figure 9 contains *Exchange* component which is responsible for another difference from Apache Kafka. *Exchange* component is responsible for the transfer of a message to a specific consumer. It has several parameters which can be used to determine the final destination of a message, which makes it different from Apache Kafka which only allows for a topic-based exchange. This is another benefit of the RabbitMQ system since its general-purpose nature allows testing more communication methods in the early stages of the application and later determining the better-suited one, should there be a need for better performance.

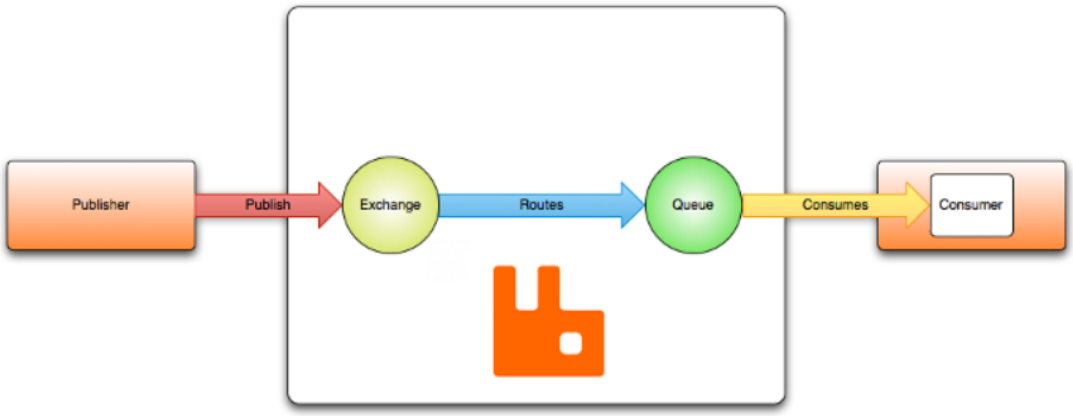


Figure 9. The model of a simple RabbitMQ application. [19]

The discussed reasons were considered during the selection of a message queue. Such queues can also be built manually, which will result in a more configurable system from the developer's point of view. In the case of this work, a third-party system was used to reduce the complexity of development and provide more reliable performance.

5.10 Backend project design

In this section, a detailed overview of the backend for the presented solution will be given. The key aspect of having a reliable and functional application lies in choosing the proper backend, which will be able to satisfy the needs of a developer and be compatible with the tools and other components that are present in the application. As it was presented in Figure 4, the backend is the most connected component of a system, and it will be participating in almost every process which will happen during the runtime.

The most crucial points will be divided into subsections:

- Server and framework - choosing the backbone for the backend. The server and framework are responsible for the application being able to run on top of other software or operating system and essentially dictates the structure of the project.
- Credentials - Cumulocity API requires the user to provide credentials, and this section will discuss how this is managed in the toolbox.
- Services - services are responsible for the main logic happening behind the processes on the backend. They are the reusable parts of the application, as each service can be used for multiple functionalities. This means they have to be built in a manner that they act as independent agents.

- Algorithms implementation - this part will concentrate on the solution for the custom-defined algorithms created by users. This is one of the features that is a requirement to the toolbox.

In general, the choice of backend is often dictated by the limitations which are set on the system. The limitations were presented in the chapters 5.1-5.8 and this section will further introduce additional challenges that were found during the development of the system.

5.10.1 Server and framework

The choice of a server that will host the application developed is dictated by the framework that is compatible with this server. The framework is responsible for an application being able to support other components and interact with them, defines the structure for the project, and is often limited by the programming language that it uses.

The programming language choice was inspired by the Jupyter Notebook software and additionally, the requirements, that data scientists would have for a toolbox they will potentially use. In a review presented by M. Yli-Heikkilä from the University of Helsinki [30] the finding tells us that the dominant language for a data scientist, according to a most recent survey, is Python. Structured Query Language (full version for an abbreviation SQL) and R programming language are also among the most used languages in the industry, however, having support for all of them introduces additional complications which do not bring value to the initial stage of developing a toolbox.

The reason for this is the trade-off between the complexity of developing a solution that would allow users to create algorithms and/or query data using SQL and R languages and the need for them. Currently, the focus of this work is to present a solution that will be compatible with the needs of the majority of users, so the main language considered should be Python programming language. This brings us to the main question of this subsection - the choice of a framework.

After the examination of suggested frameworks that are typically used for creating web solutions for Python, the candidates were:

- Django
- Flask
- aiohttp
- FastAPI

These frameworks are displayed in the order in which they were considered, and each presents its unique features and limitations.

Django. Django [8] is a powerful full-stack web framework, which enables bootstrapping projects and can handle most common tasks such as database entity management, server configuration, type-checking for incoming requests, etc. It uses Model-View-Template (full for abbreviation MVT) architecture which tells a developer of a structure that he has to follow when developing applications. This allows to mitigate human error during development and makes it easier to write readable code, assuming that the structure suggested is followed.

However, being a full-stack solution Django takes away the developer's freedom over the UI manipulation. Because of such limitations, one of the requirements - the visualization feature - may be more challenging to implement. Additionally, Django runs on top of WSGI [22] and ASGI [4] type servers, where the ASGI server is not fully supported yet. This fact creates a problem, as not being able to have an application working asynchronously may impede its performance greatly considering that it may be working with high-volume data.

Flask. Flask [10] is a web framework that only provides the base code structure, without the visual development built-in. This framework is one of the most popular together with the Django framework, and it also provides support to the majority of common tasks that are handled by a developer. Flask is simplistic and has a clear structure, that the user may follow, which makes it also a simple solution that unlocks the freedom over the UI layer, providing the user with the ability to create his own front-end application that will communicate with Flask framework.

In the context of this work, the Flask application can be run as an API server, which will route the requests from frontend to respective components and will be a great tool for future developers because of it being popular and simple. The problem with Flask, however, is that there is no support for asynchronous code. It is possible to declare some of the controllers asynchronous, but it doesn't utilize the full resources of a computer for operations related to data processing and data download. This poses a risk for an application to become unresponsive, should there be multiple operations working at the same time.

aiohttp. Another framework that was a good alternative to choose is aiohttp [1] framework, which is an async web framework for Python. It solves two main issues that were present with other options - it is a non-full-stack framework, meaning that the developer has the freedom to choose custom tools for building UI, and it is a fully-supported asynchronous framework. It can run on top of popular ASGI-type servers, which also makes it easier to consider hosting this application in the future, as the toolbox matures.

The problem for aiohttp stems from having to program most common tasks manually. For example, this framework is configuration agnostic, meaning that it does not have a specified way of having a configuration, and the developer has to create one manually

for the application to run correctly. Not having basic features handled by the framework requires developers to handle them manually and often do extra work, which is always prone to human error, so the decision was to find another web framework that would be more mature and have better support than aiohttp.

FastAPI. FastAPI [9] was a better alternative to aiohttp simply because it is more mature, it has a defined project structure and it retains the most important requirements of a web framework (defined in the scope of this project): to be a non-full-stack web framework and to support asynchronous operations.

Another benefit of the FastAPI framework is the support of newer versions of Python, which enable the developer to have type-checking for variables, and consequently the endpoints. The toolbox uses the feature for type-checking to avoid erroneous requests from the frontend, which is another good way to ensure mitigation of human errors.

FastAPI, being an asynchronous framework, supports ASGI-type servers as well, and the vendor recommends pairing it with a Uvicorn [21] server, which allows the developer to have the reload possibility. Reload possibility enables rebuild for every change in code, which helps with reducing time for manual testing of code and having up-to-date version of code online at all times.

5.10.2 Credentials

The server heavily relies on Cumulocity API. It would make sense to handle the connection to Cumulocity data from a single source. For this reason, a service for requests to Cumulocity API was created. This service needs to include the header with the credentials of a tenant user in order to pass authentication.

Cumulocity API expects the following authentication headers:

- Basic authentication
- OAI-Secure authentication
- SSO with authentication code grant
- JSON Web Token authentication

JSON Web Token authentication is a deprecated way of communication and SSO is not present for the Delta center system. OAI-Secure requires additional setup from Cumulocity to work. The safest option is to go with basic authentication, which is universal and present for every registered user for a specific tenant.

After eliminating non-suitable authentication options, the solution was straightforward.

Figure 10. The example of credentials file.

Figure 10 presents a structure for a credentials file, which is of type JSON. It is shown that there are three main elements for credentials: tenant URL, username, and password. In Figure 10 sequence of "x" letters stands for the tenant id, which is unique to Delta building Cumulocity environment. "y" letter is the username, which exists in the environment. The password field is self-explanatory and the tenant URL is the full URL of the Cumulocity tenant, which contains the tenant ID at the beginning.

```
credentials = b64encode(bytes(username + ':' + password, "utf-8")).decode("ascii")
self.headers = {'Authorization': 'Basic %s' % credentials}
```

Figure 11. The example of credentials encoding.

The credentials need to be passed to a header as an encoded entity. In Figure 11 two lines of code that are handling this process are shown. In the first line, the username and password separated by a colon are translated into bytes and later encoded into a Base64 format. This is a required format for the Cumulocity Basic authentication, which on the second line gets converted into a header and creates a valid request. Now the service which is responsible for communication with Cumulocity is able to send requests to API and receive responses.

Additionally, the tenant URL is not hard-coded, which allows it to connect to different Cumulocity tenants and not be bounded to Delta building's tenant.

Figure 12 showcases the usage of the credentials file, which is intuitive. The benefit of such a solution is the ability to later reuse the system for various Cumulocity environments. Both the server and the data function utilize the same logic at the root level for fetching credentials, which are later used for communicating with Cumulocity within the respective services.

5.10.3 Services

Earlier sections explored the most convenient format for data exchange with Cumulocity. It was decided that CSV format will be used for the data download, however, once the data arrives to the server, it may need to be stored inside a variable, so that further

```
f = open('credentials.json')
credentials = json.load(f)
f.close()

username = credentials['USERNAME']
password = credentials['PASSWORD']
tenant_url = credentials['TENANT_URL']
```

Figure 12. The example of credentials being loaded into an application.

manipulations on this data may take place. For such reason, it was decided that the main library for processing tabular data will be Pandas.

It is a convenient library that is popular among data scientists and is a core technology used in services, that were created on the server. Their main goal is to help the server process the requests from the frontend and responses from Cumulocity.

Below, the services and their brief description may be found:

- Query service - a service used to query the data, and perform operations that prepare the data to be sent to frontend for further visualization. As the dataset stored in a variable is using Pandas library and a DataFrame object specifically, functions of a Query service are used as a wrapper, to restrict data modification and ensure correct work of the frontend, which is typically the receiver of data modified in the Query service.

- CumulocityFetcher service - a utility to download data from Cumulocity API using correct headers. It has a series of functions that build the URL for a request that the user desires to invoke and returns the result in a JSON format, which is commonly expected by the frontend framework. It is different from CSV because these requests typically contain additional metadata, which is required by either frontend or backend services that may rely on it.

- RabbitMQClient service - this is essentially a client for RabbitMQ broker, which allows for a backend to act as a sender. This client is only used by an endpoint, which will send a message about a new job to the job queue. It expects that the broker is running on a local machine with the standard port, and only allows to specify the routing key, which allows to have more job queues for different intentions than creating a new dataset.

The last service present is the AlgorithmRunner, which the author believes should be described in more detail. The reason for this is the complex logic behind the service,

which includes both: specific adjustments in the UI layer and lower-level operations on the computer. For this reason, this service will have its own section after the frontend project design is described.

5.11 Frontend project design

The next crucial component of the system is the frontend. There are many tools, which can be used to build the UI, and there are both benefits and disadvantages to them. However, the goal for frontend in this work is to make development efficient and have a framework that allows to satisfy the visual requirements defined for this toolset. Following subsections will explore the reasoning behind major decisions and the visual representation the author provided.

5.11.1 Framework choice

Most popular frameworks for frontend are relatively similar to each other in a way that they all are able to provide the needed functionality. The author chose to proceed with the React.js [20] library for frontend code. There are several reasons:

- "Configuration over convention" - this is the antipode to the "convention over configuration" approach in software. React is essentially a library one can use to build applications and control the smallest aspects of structuring the application, or decisions on which libraries to use. This is helpful, in case there is a need to use external software, which may not be fully supported by a framework.
- Libraries - React has a variety of libraries that are either developed specifically for it to work with or are adaptations of popular JavaScript libraries that will work with React. This is an important aspect to save development time and make sure there are as few bugs as possible. Notable libraries used in this work are the Material UI [16] component library and Chart.js [5], chart building library.

5.11.2 Pages

The project itself is mainly split into the following categories: pages, components, and services. Pages are essentially containers for components, and components are divided in a meaningful way, to make reusing them in later development possible. Services are a collection of functions, which make requests to the backend for required information. There are five pages in the application, each serving its own purpose (excluding the homepage, which does not contain any functionality).

DeviceListPage. A page that displays devices in their hierarchical structure. The structure comes from Cumulocity objects, which typically have a tree structure, and any node of the tree may have measurements associated with it. *DeviceListPage* in this case is a gateway to a *SingleDevicePage*.

SingleDevicePage. This page is used to output information about a device selected. The information contains a list of child devices, and a chart for the latest N number of measurements, where N is a pre-defined number. It is possible to select a measurement type for which the graph should adapt.

DataStudioPage. Studio is a page where data download parameters are collected. It is possible to specify measurement type, date range, and a name for a dataset, which will later trigger a job for data download from Cumulocity.

AlgorithmSelectionPage. On this page, a possibility to provide parameters to an algorithm, which can further be triggered is granted. At the top of the page, a name for a new dataset is expected. There are also two lists: a list of datasets and a list of algorithms, which is expandable and allows to specify the parameters for an algorithm.

5.11.3 Visualization

The last page is a page for generating graphs for specific datasets. The idea for the graph page is to be able to provide to a certain degree custom components, that users may later also share with each other, and to display the datasets information processed by a specific query.

The Query service, described in the backend section is later used for some endpoints, that have a connection to the frontend's *datasetQueryService*. Charts created as individual components are later added to a list with a specific structure.

In Figure 13 a list for registering the charts present in the application is presented. There are 3 required parameters: a label name, a render method, which includes a component with the arguments propagated, a label name, and an array of data selectors, which are necessary for a chart to operate. The data selectors array is important for a chart to render properly and receive requested parameters. Should any of the parameters be not present, the chart will not render.

It is important to note, that even though graphs can be shared across applications, for stable work they have to use libraries and services already in use or come with a warning that instructs the user to download necessary software.

Once the list is composed, the rendered page presents various fields and can be split into 3 sectors:

```
export const VisualizationMethods = [
    {
        render: (args) => <MinMaxChart {...args} />,
        dataSelectors: ["measurementTypes", "datasetName", "timeAggregate"],
        tabLabel: "MinMax"
    },
    {
        render: (args) => <DoughnutChart {...args} />,
        dataSelectors: ["datasetName"],
        tabLabel: "Doughnut"
    },
    {
        render: (args) => <MeanMeasurement {...args} />,
        dataSelectors: ["datasetName", "measurementType", "timeAggregate"],
        tabLabel: "MeanMeasurement"
    }
];
```

Figure 13. An example of visualization page list with available charts.

- Chart selector - a selector which indicates which item from the visualization list the user wants to see.
- Query parameters - parameters supplied to the render method.
- Chart - a rendered chart with the loaded dataset.

It is also worth noting, that for now only four query parameters are supported: dataset name (has to be filled before other parameters are unlocked) time aggregate unit, measurement type, and a list of measurement types. The example selection of these parameters would be:

- Measurement type - "DP.T" (from dropdown)
- Dataset name - "demo_dataset_name"
- Time aggregate - "1min" (from dropdown)

5.12 Algorithms implementation

The workflow for algorithms execution starts at the backend, where the algorithms are defined. To understand it better, the system can be split into two small logical units: algorithm processor, and algorithm UI. The first unit is defined in the backend and provides capabilities to run algorithms, get the list of algorithms and get the parameters required for a specific algorithm, and the UI part can generate the form for an algorithm and invoke the algorithm run.

Since the algorithms have to be independent and transferable so they can be applied to any downloaded dataset regardless of the action performed in an algorithm, they all possess a structure, defined by the author, which enables to load algorithms dynamically - load them regardless of the functionality within them.

```python
from enum import Enum
import numpy as np

from Algorithms.SystemClasses.DatasetParam import DatasetParam

class LogType(Enum):
    BINARY = 'BINARY'
    DECIMAL = 'DECIMAL'
    NATURAL = 'NATURAL'

# Requires: numpy
class LogarithmicNormalizer:
    def __init__(self, dataset: DatasetParam, log_type: LogType, column_name: str):
        self.dataset = dataset.data
        self.log_type = LogType[log_type]
        self.column_name = column_name

    def Run(self):
        print(LogType.DECIMAL, self.log_type)
        if self.log_type == LogType.BINARY:
            self.dataset[self.column_name] = np.log2(self.dataset[self.column_name])
        elif self.log_type == LogType.DECIMAL:
            self.dataset[self.column_name] = np.log10(self.dataset[self.column_name])
        elif self.log_type == LogType.NATURAL:
            self.dataset[self.column_name] = np.log(self.dataset[self.column_name])
        else:
            pass
        return self.dataset
```

Figure 14. An example of a structure for a custom algorithm.

Figure 14 depicts an example of an algorithm file, with key points highlighted in yellow:

- *DatasetParam* - this is a special class for a dataset, which can be ignored by the processor when compiling a list of arguments the algorithm intakes. The reason for this is that the algorithms operate on the datasets, that are present inside the *SavedDatasets* folder, and the controller which receives a request from the UI loads the dataset manually by name provided by the user. Every algorithm has to receive a dataset as a parameter.

- *class LogarithmicNormalizer* - an algorithm has to be a class. This way the inspector module is able to find the properties of a class and instantiate it later programmatically.
- Parameters - Python does not require to have a type for a variable or a class parameter. However, to enable additional type-checking and help the user debug an algorithm should there be a need for this - types for parameters were added. The backend handles fetching data types and parameter names and providing them to the frontend, but currently, this solution is limited to five main types: string, integer, floating point number, boolean, and enumeration. The dataset parameter, being a special case is ignored.
- *Run* function - this is the main function that gets called when an algorithm is instantiated. It cannot accept any parameters and has to rely on the internal fields of a class.
- Return dataset - the *Run* function always has to return a valid pandas dataset. This later gets saved to a directory, where other datasets reside.

The visual component is responsible for calling the necessary functions on the backend. The most important mechanism that exists on the frontend is the UI generation - a React component that handles form generation for a list of algorithm parameters. The component expects an algorithm name and its type and later generates objects that are used to select a proper display for an input field. There is an exception for enumerated arguments, which also expect a list of values that may be present, and a dropdown is later generated for this input type. Visual examples will be presented in the next chapter.

The visual component heavily relies on the backend for providing correct types and enumeration values, and given that the UI generation process is automatic, it is important to make sure that the values supplied match the desired input type.

5.13 Data function design

There are several key parameters that the user would want to specify based on the nature of the information that the Cumulocity system can provide and the requirements of an application:

- Measurement type - it is possible to filter records by measurement type, which allows for a more specific dataset composition. This is an optional parameter that can be ignored.
- Date from - the starting date from which the dataset will filter measurements. No measurements earlier than this date will be returned.

- Date to - the ending date from which the dataset will filter measurements. No measurements later than this date will be returned.
- Filename - the name of a resulting file, that will be created upon the completion of a job.

These parameters are received as a message through the RabbitMQ broker. The data function is an infinite loop (with the ability to exit by a command from a developer), which actively listens for new messages that arrive from the queue.

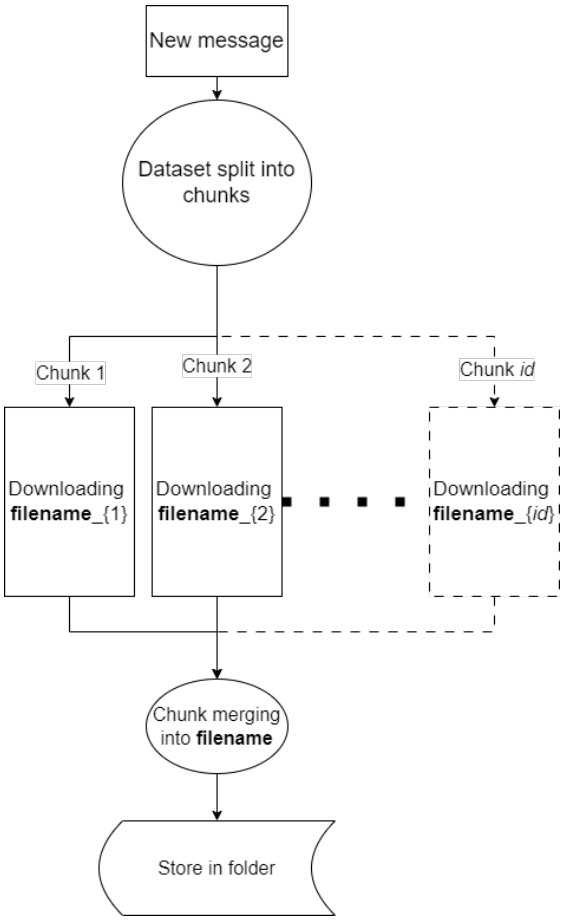


Figure 15. Logic flow for chunking process.

After the message is received, it will try to create chunks as depicted in Figure 15. A full dataset split into several files is done by first making a REST request to Cumulocity, to evaluate the number of records present for a specified date range. Then, the page numbers are calculated based on the estimate of the number of measurements present. The calculation is performed as follows:

1. A response from Cumulocity is returned, displaying the total number of pages present with the requested parameters.

2. The limit of rows per dataset is set to be 100,000, and the number of measurements per page requested is 2,000, so the number of pages per dataset is equal to $100,000 \div 2,000 = 50$ pages/dataset.

3. Next, the list of pages is compiled, which is used in the next stage.

After the pages are split, a series of REST requests are made, to get the first timestamp from the first page and the last timestamp of the last page that will be a part of this chunk. The process is repeated until the date ranges for all chunks are recorded.

After the dataset is split, a CSV file is downloaded and stored in a dataset folder. Since the data function does not run on a server, it is possible to utilize Python's standard multiprocessing library, which allows the spawning of multiple processes and downloads chunks concurrently, which benefits the user by providing shorter file download time.

The process then scans the directory with the saved chunks and merges them based on the filename. It is important not to create datasets with the same filename, as it may result in a defective work of a system.

6 Validation of requirements

This chapter will provide examples on how to use the toolset. There are 4 main functions that a user can perform: downloading a dataset, reading the dataset (displaying various tables about the dataset content), applying algorithms to a dataset and visualization. The functionalities manual will be presented in the logical order, as it would appear in a user's work.

6.1 Download a dataset

The first thing a data scientist would do is download a dataset to work with later. This starts at the *Studio* page, which contains both a utility to download the data and inspect the downloaded dataset. The menu will contain 3 forms that the user has to fill, and one which will be optional.

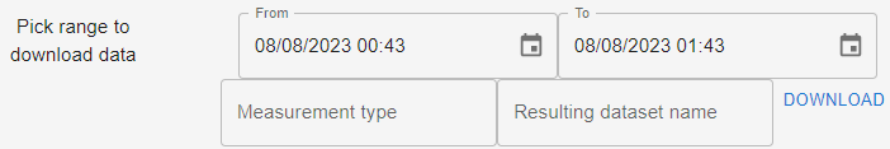


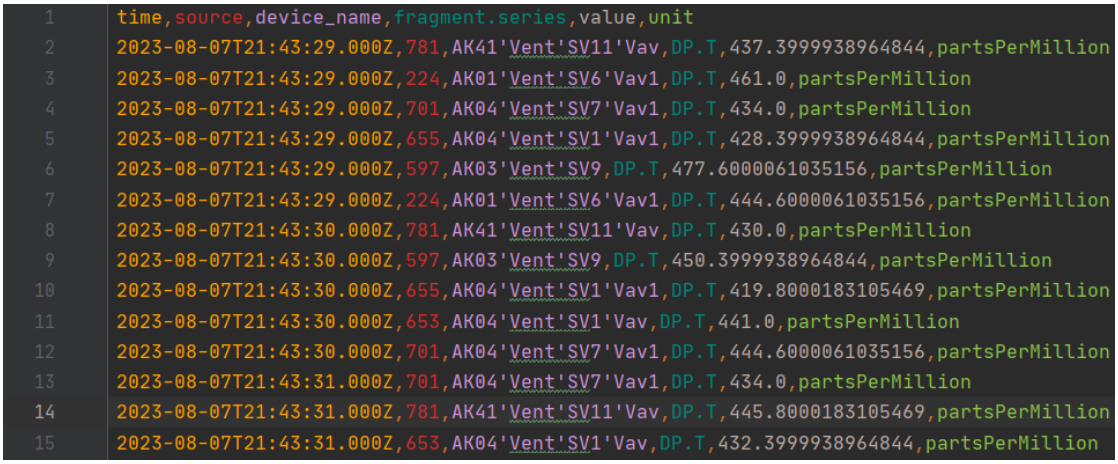
Figure 16. Form for data download.

Figure 16 presents a form that will later be filled. The test values that are used in this example are:

- Date from - 08/08/2023 00:43
- Date to - 08/08/2023 01:43
- Measurement type - DP
- Resulting dataset name - DP_measurements_last_hour_demo

After filling out the form the backend triggers the job and the data function starts to download a file. After the file is downloaded it will appear in the *SavedDatasets* folder in the backend repository. To have this functionality work properly, the function repository and the backend repository have to be on the same level because internally function uses a relative path to save the downloaded data. Figure 17 displays a portion of data obtained (note that the time displayed is in UTC format, as opposed to the time selected in the form).

Data download complies with the requirement of being able to configure data retrieval from Cumulocity, and it is possible to partially filter the data downloaded as it is defined



```
time,source,device_name,fragment.series,value,unit
2023-08-07T21:43:29.000Z,781,AK41'Vent'SV11'Vav,DP.T,437.3999938964844,partsPerMillion
2023-08-07T21:43:29.000Z,224,AK01'Vent'SV6'Vav1,DP.T,461.0,partsPerMillion
2023-08-07T21:43:29.000Z,701,AK04'Vent'SV7'Vav1,DP.T,434.0,partsPerMillion
2023-08-07T21:43:29.000Z,655,AK04'Vent'SV1'Vav1,DP.T,428.3999938964844,partsPerMillion
2023-08-07T21:43:29.000Z,597,AK03'Vent'SV9,DP.T,477.6000061035156,partsPerMillion
2023-08-07T21:43:29.000Z,224,AK01'Vent'SV6'Vav1,DP.T,444.6000061035156,partsPerMillion
2023-08-07T21:43:30.000Z,781,AK41'Vent'SV11'Vav,DP.T,430.0,partsPerMillion
2023-08-07T21:43:30.000Z,597,AK03'Vent'SV9,DP.T,450.3999938964844,partsPerMillion
2023-08-07T21:43:30.000Z,655,AK04'Vent'SV1'Vav1,DP.T,419.8000183105469,partsPerMillion
2023-08-07T21:43:30.000Z,653,AK04'Vent'SV1'Vav,DP.T,441.0,partsPerMillion
2023-08-07T21:43:30.000Z,701,AK04'Vent'SV7'Vav1,DP.T,444.6000061035156,partsPerMillion
2023-08-07T21:43:31.000Z,701,AK04'Vent'SV7'Vav1,DP.T,434.0,partsPerMillion
2023-08-07T21:43:31.000Z,781,AK41'Vent'SV11'Vav,DP.T,445.8000183105469,partsPerMillion
2023-08-07T21:43:31.000Z,653,AK04'Vent'SV1'Vav,DP.T,432.3999938964844,partsPerMillion
```

Figure 17. An example of the final dataset.

in section 4.3. It will be possible to expand the functionality of the system further by allowing filtering by multiple measurement types at once or filtering by device, but this is a consideration for future work.

6.2 Read information about a dataset

On the same page, it is possible to view the information about the dataset. The bottom half contains a simple dropdown, where a user would select the name of the dataset that he would like to explore.

There are three tables that present information about the dataset:

- Measurement type count - it displays how many rows in a dataset for a specific measurement type are present.
- Measurement types per device - a table of devices that contains a list of measurement types.
- Devices per measurement type - a reverse of a previous table.

To showcase this feature, another file can be downloaded, since the previous dataset generated for a single measurement type does not allow for many records in two out of three tables. Consider the following input:

- Date from - 08/08/2023 00:59
- Date to - 08/08/2023 01:59

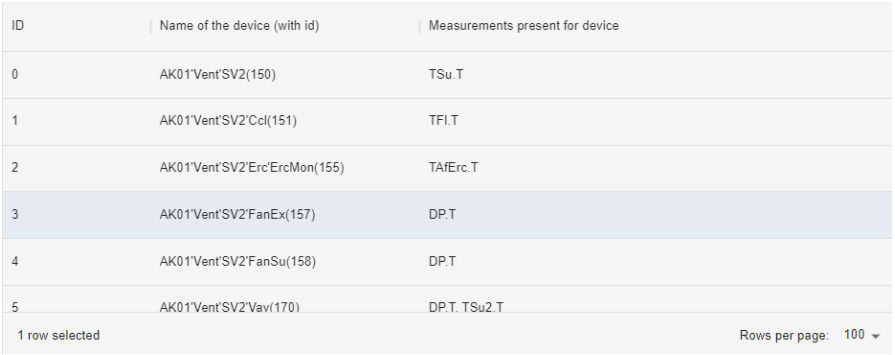
- Measurement type - *Empty*
- Resulting dataset name - multiple_measurement_types

This dataset, once downloaded, can be selected in a dropdown, where the information will populate the tables. Figure 18, Figure 19, and Figure 20 represent the first lines of data that will be presented in the given scenario. The table of devices per measurement type can be particularly useful because Cumulocity does not provide a solution to see a list of such devices out of the box.

ID	Measurement type	Number of ...
0	CO00011.T	727
1	CO00012.T	917
2	CO00013.T	944
3	CumEg1.T	5
4	CumVlm1.T	23

1–5 of 36

Figure 18. An example of the measurement type count table.



ID	Name of the device (with id)	Measurements present for device
0	AK01'Vent'SV2(150)	TSu.T
1	AK01'Vent'SV2'Ccl(151)	TFI.T
2	AK01'Vent'SV2'Erc'ErcMon(155)	TAfErc.T
3	AK01'Vent'SV2'FanEx(157)	DP.T
4	AK01'Vent'SV2'FanSu(158)	DP.T
5	AK01'Vent'SV2'Vav(170)	DP.T, TSu2.T

1 row selected

Rows per page: 100

Figure 19. An example of the measurement types per device.

Finally, there is a Devices page, which allows traversing the device tree, and inspecting the device's measurements, if there are any. After clicking on a device from the device list, a single device page is displayed, where the user at the bottom half of the

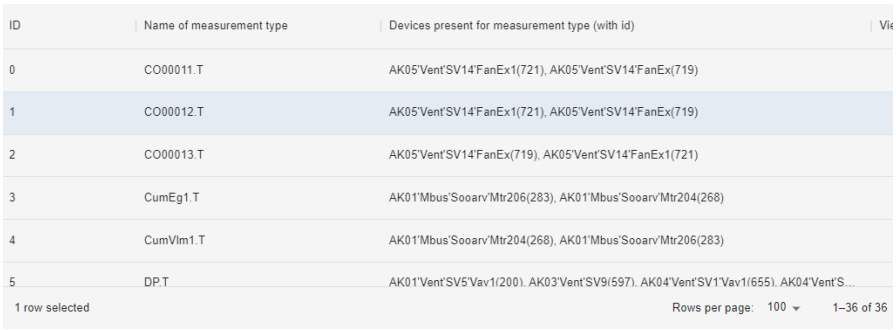


Figure 20. An example of the devices per measurement type.

page may select the measurement type if there are any, and see the graph with the latest measurements' values.

These functionalities answer the requirements defined in section 4.3, since both device navigation and device inspection features are present. There is still, however, room for improvement in terms of providing an overview of the downloaded dataset, and it will be discussed in future work.

6.3 Visualize a dataset

The next feature presented is visualizing the dataset. There are several pre-made graphs present in the system, and they can be accessed from the *Visualization tool* page.

The first thing to do would be to select a graph, the user wants to build from the left side of the interface. In this example, *MinMax* graph will be used. This graph shows both minimum and maximum values at a certain time window on the graph.

Next, a user should specify the dataset name, and click the *Choose* button, to unlock the remaining fields. In this example, a dataset *DP_measurements_last_hour_demo* will be used. The only measurement type that will be allowed to input will be *DP.T*, and the time window selected can be 30 seconds. The result is partially presented in Figure 21.

It applies to the visualization requirements outlined in section 4.3 providing both the ability to generate charts for various datasets and allowing to add custom charts by adding a custom-made chart component according to the rules described in section 5.11.

6.4 Apply an algorithm to a dataset

After the algorithm is downloaded and inspected, one of the following possible applications would be to apply an algorithm to this dataset. There are a few pre-made algorithms to demonstrate the work of the application, this section will use the same algorithm presented in Figure 14.

Figure 21. An example of the portion of a graph, that is generated for a dataset.

This feature can be used from the *Algorithm selection* page. Assume the same *DP_measurements_last_hour_demo* dataset. It is visible that there are a few outliers in the dataset from the previous section. With the algorithm from Figure 14 algorithm, it will be possible to normalize the data and detect outliers better, with the help of a logarithmic scale.

With this feature the final requirement for the application of a custom algorithm to a dataset is satisfied. The system allows using algorithm files as long as they follow the structure described in section 5.12, and the algorithm produces an output that can later be used by a user in his work.

7 Evaluation

In this section, the evaluations for the defined validations will be presented. There will be a series of measurements that will test the desired benchmark described in section 4.4.

Since the chunking mechanism adds complexity to downloading large datasets, it is important to test data integrity and verify that the records are not modified.

The test will be using a *csvdiff* [6] library, to verify the difference between two CSV files in raw format: the file downloaded using software developed in the toolset application, and the file downloaded directly from the Cumulocity system through a web browser. The first file will be downloaded by mimicking the user experience and providing the filter parameters to the dataset from the UI. The second file will be created inside the Cumulocity system with the help of Cumulociy API, thus creating a singular CSV file with all the data, omitting the chunking process. This way it will be possible to see the difference between the native CSV that would be generated inside the system, and the resulting CSV after it has undergone all manipulations performed during the download from within the toolset.

This test will help understand, whether there are any hidden caveats that may impact working with the data in the future.

The tests will be done across three sizes of datasets, defined in section 4.4. It is done this way, because of the unstable work of Cumulocity (returning either a CSV file or a link to a stored CSV file), network stability (given that CSV files are downloaded through the REST requests), and the chunking mechanism. These factors contribute to the potential instability of downloading the data and need to be tested for data correctness in different scenarios.

	Action	# of rows	% from total size
5 minute dataset	Removed:	13 rows	0.3%
	Added:	0 rows	0.0%
	Changed	607 rows	16.7%
6 hours dataset	Removed:	8 rows	0.0%
	Added:	0 rows	0.0%
	Changed	36,610 rows	15.9%
12 hours dataset	Removed:	9 rows	0.0%
	Added:	0 rows	0.0%
	Changed	81,604 rows	16.1%

Figure 22. Data integrity validation table.

The results of the testing are presented in a table in Figure 22. The dataset was compared against the real dataset received from Cumulocity. Dataset sizes were 3,865

rows (data sample per 5 minutes), 230,874(data sample per 6 hours) rows, and 508,414 rows (data sample per 12 hours). It can be seen, that there are a few inconsistencies with downloading data, and upon closer examination of the differences there are two major reasons why that happens.

The changed rows occur because of the rounding of long floating point numbers. This may happen during the upload of the dataset to Pandas Dataframe, due to string parsing and type conversion later. This can be an issue for data that requires extra precision, but this does not pose an issue otherwise, as typically the error occurs after the 6th decimal figure.

The removed rows indicate that the dataset downloaded from Cumulocity contains more rows than are actually present in the dataset. There is almost no difference for different-sized datasets and since the error is consistent, it was found that the issue is due to the fact that the final timestamp in the time range for a dataset is not included in the result. For example, a timestamp *"2023-07-19T06:00:00.000Z"* will only contain results with the last timestamp being *"2023-07-19T05:59:59.000Z"*. During the chunking process, smaller datasets are created and the last date put into the request is *"2023-07-19T05:59:59.000Z"*, which later loses a second as well. Due to this error, the dataset is not complete and will require fixing.

Overall, we may see that the number of missing rows does not grow with the growing size of the dataset, and is more dependent on the number of rows present for the last second of the time range. This means that the percentage of missing rows will only decrease with the growing total number of rows, which is a minor drawback. Additionally, as the data is only impacted for high-precision measurements, and no new rows appear, these results are considered not ideal, but satisfactory, as it does not impede the data scientist's work substantially. The exception would be for scenarios, where the user wants to download the dataset that includes the data for a long period of time which is being sent at once (daily data being sent at midnight, for example) to Cumulocity with the timestamp equal to *"00:00:00 000Z"*. This creates an issue that has to be addressed in future work.

The next tests will measure the performance of the system, and inspect the ability to handle the dataset size progression during the download.

There will be several time ranges that will be used for input: 15 minutes, 30 minutes, 1 hour, 2 hours, 4 hours, 8 hours, 16 hours, 32 hours, and 64 hours.

Downloading time for a dataset will be measured inside the data function, allowing to test of the change in downloading time with the progression of the data size. The changes were analyzed both for a time-difference metric and for a row-difference metric.

Figure 23 presents the graph generated for a time range over the time it takes to download the dataset. As we may see, the change grows in a linear fashion, however, during the attempt of measuring the download time for the 64-hour dataset, the application crashed. This puts a limit on an application, as the chunking mechanism cannot handle

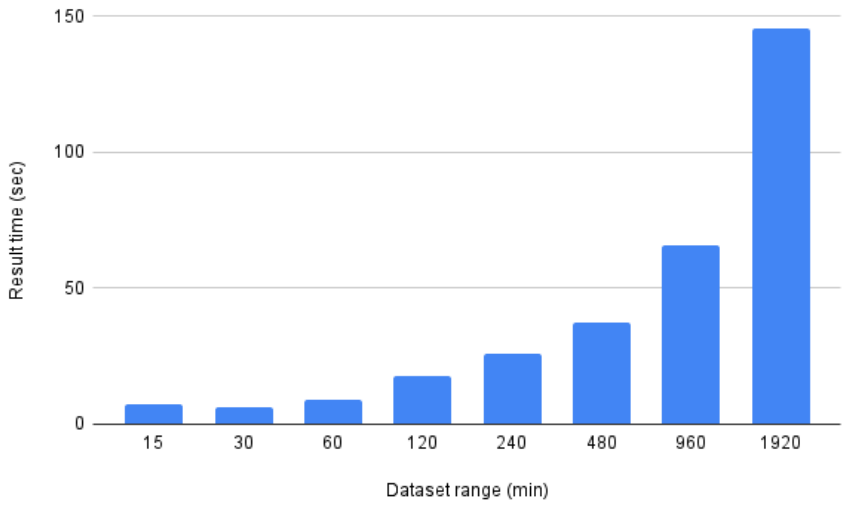


Figure 23. Chart for time range vs time to download test.

too many threads running simultaneously and at some point, the job freezes. At the time of testing, the logs showed that there were 29 processes created for concurrent download, which may be a different number for a different machine. This is a critical issue that has to be handled by organizing thread creation specifically for each device or using additional libraries that would handle this process memory-safe. Due to this issue being discovered late at the validation step, there is no possibility to fix it before the submission of this work. It is, however a high-priority task, that should be handled firsthand upon further development of a toolset, and data function specifically.

Number of rows	Result time (sec)
8961	7,15
17664	6,43
34941	9,08
70933	17,84
142714	25,91
291769	37,23
662274	65,7
1267627	145,63

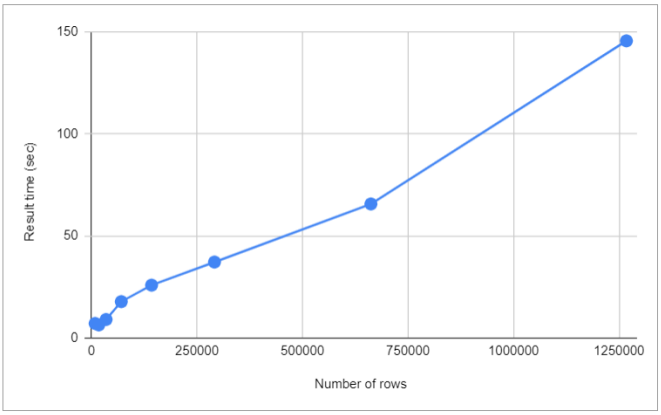


Figure 24. Chart and table of values for the number of rows vs time to download test.

Figure 24 dives deeper into the linear fashion of the result. It also contains a table with the time measurements for each number of rows, where they appear in the same order as the time range results. We can see that in the case of the Delta building Cumulocity's number of rows per hour grows at a linear rate as well, which confirms the stability of the solution until a breaking point.

8 Future work

This chapter will explore possible future work for the toolset. The solution presented is the initial version of a toolset for working with the Cumulocity system for analyzing and visualizing the data stored.

As such there are several main domains that can be explored for expanding the solution functionality that will be discussed in this chapter.

Data download.

- Data quality - the first improvement that can be made was briefly introduced in the previous chapter during the validation of data quality. It was discovered that at some points datasets may lose values by a second, which is a defect, even though the error is minor. Improvement of data quality can further be performed by also using a better parser tool to not lose the true value of 7th and onward decimal points, making the solution more resilient to high-precision data.

- Data download speed - currently, the average speed of download is limited by the device that the application is running on. However, at a certain point, the device hits the limit which allows downloading larger datasets, mainly due to the limitation of a multiprocessing library used. This limitation results in a critical issue, that has to be solved before continuing the development of the application, as the 2-day dataset limit will not be enough, compared to the amount of data it is possible to extract from the Cumulocity system. This results in an application to be not suitable for big-data applications, and therefore does comply with the data download requirement fully. It can be fixed by reworking the chunking process to either use fewer processes in the process pool by making a better queueing mechanism that will not start the processes until the available quota has not increased, or by using adapting the process to another library, which will leverage the power of multiprocessing to its full capabilities, allowing to use additional free resources on a local machine.

- Real-time data download - real-time data download was not in the scope of this work, but there is a possibility to implement data download in an asynchronous way with Cumulocity API. This can lead to further work on creating a solution that will allow processing and visualizing data in real-time, providing tools to handle Big Data tasks to help data scientists with a broader range of tasks that can be performed in the toolset.

Data inspection.

- Additional dataset type support - currently, the system has a central data type used, which may not be enough for some tasks. It would be possible to provide support for other file types, like Apache Parquet, or JSON. This will help users to integrate their current work with other applications easier and more user-friendly.
- Add possibility to query datasets with SQL language - allowing to work with datasets using a common language in the industry can expand the dataset inspection capabilities, and help scientists understand a dataset better by giving them the possibility to write custom queries for summary output.

Algorithm running and visualization.

- Standard algorithm libraries - create a set of algorithms that the users may use to operate with the data. Common algorithms present in a dataset may help to get started working with the tool greatly, and allow for less work with common tasks.
- Visualization of geo data - some of the possible improvements should address the addition of a map that can be used for the visualization entities. Currently, the visualization library does not provide support for geographical plots.

Additional improvements.

- Containerization - the application currently relies on the user to download the repositories with the code and manually set up the environment for the toolbox. Using containerized application can greatly reduce the setup time by allowing to use of a predefined image of an application, that will already have the configuration for the correct environment.
- Authentication - adding the authentication and multi-user system to an application will be beneficial if the application is ever to become hosted on a remote server. If the application is made accessible to everyone remotely, without the need to install anything on the local machine, it will be crucial that users' work does not interfere with each other, and is an important step towards complying with privacy regulations.
- Remote hosting - hosting the application remotely will benefit the users by allowing them to use remote resources for operating with large datasets. It allows handling issues with big-scale data for scientists that do not possess powerful computers and is in general an improvement in downloading speed and processing speed. It is currently not possible to host this application remotely, due to prerequisites of individual application context per user, which typically requires authentication.

For this reason, to enable hosting an application first the prerequisites have to be satisfied.

9 Conclusion

The goal of this work was to create a toolset for working with data of different sizes, which is stored in the Cumulocity system from the Delta building of the University of Tartu scenario. This toolset is supposed to expand the functionality present in Cumulocity and additionally promote collaborative work in the data science domain. The resulting solution allows users to create both, custom algorithm files and custom chart files with the intention to either reuse them for different datasets in the future or to share them with other users promoting collaborative nature of work. The toolset allows downloading data from Cumulocity in CSV format, using filters for the date and time range, and measurement type. It also allows us to inspect the dataset mainly focusing on the measurement types presented and to inspect the devices hierarchy, allowing us to see the possible measurement types that the user may need to use later in his work.

The work was later validated for the data quality and data download speed metrics that may impede the performance of the application. The validations showed, that there are issues that have to be addressed in order to unlock the full potential of the toolset, and the issues with the data quality, that do not pose serious threats to most use cases. The reasoning behind such drawbacks was provided in the respective chapters and future work, which both suggest the course of action on how to fix the issues and outline the ideal behavior for these metrics.

In conclusion, the system does have its flaws because there is a lack of optimization for data download service, which is a major part of an application and negatively impacts the main driver behind such applications - working with large datasets on a smart city scale. However, the skeleton for an application to work with Cumulocity was introduced, which can be developed into a centralized application for data scientists, where day-to-day work can be performed without the need to use multiple services when working with Cumulocity. The steps to further expand the application, and develop it into a feature-rich toolset were introduced in future work. A collaborative narrative of features that work with algorithms and visualizations will also contribute to the development of this application, which passively increases the value that this application provides.

References

[1] aiohttp. https://docs.aiohttp.org/en/stable/. Accessed: 10.08.2023.

[2] Apache kafka. https://kafka.apache.org/. Accessed: 10.08.2023.

[3] Apache parquet. https://parquet.apache.org/. Accessed: 10.08.2023.

[4] Asgi. https://asgi.readthedocs.io/en/latest/. Accessed: 10.08.2023.

[5] Chart.js. https://www.chartjs.org/. Accessed: 10.08.2023.

[6] csvdiff. https://github.com/larsyencken/csvdiff. Accessed: 10.08.2023.

[7] Cumulocity iot. https://cumulocity.com/guides/concepts/introduction/. Accessed: 10.08.2023.

[8] Django framework. https://www.djangoproject.com/. Accessed: 10.08.2023.

[9] Fastapi. https://fastapi.tiangolo.com/. Accessed: 10.08.2023.

[10] Flask. https://flask.palletsprojects.com/en/2.3.x/. Accessed: 10.08.2023.

[11] Google colaboratory. https://colab.google/. Accessed: 10.08.2023.

[12] Google sheets. https://workspace.google.com/intl/en_ie/products/sheets/. Accessed: 10.08.2023.

[13] Grafana. https://grafana.com/. Accessed: 10.08.2023.

[14] grpc. https://grpc.io/. Accessed: 10.08.2023.

[15] Jupyter notebook. https://jupyter.org/. Accessed: 10.08.2023.

[16] Material ui. https://mui.com/. Accessed: 10.08.2023.

[17] Microsoft excel. https://www.microsoft.com/en-us/microsoft-365/excel. Accessed: 10.08.2023.

[18] Rabbitmq. https://www.rabbitmq.com/. Accessed: 10.08.2023.

[19] Rabbitmq schematic. https://www.rabbitmq.com/tutorials/amqp-concepts.html. Accessed: 10.08.2023.

[20] React.js. https://react.dev/. Accessed: 10.08.2023.

[21] Uvicorn. https://www.uvicorn.org/. Accessed: 10.08.2023.

[22] Wsgi. https://wsgi.readthedocs.io/en/latest/index.html. Accessed: 10.08.2023.

[23] Leonidas G. Anthopoulos. Understanding the smart city domain: A literature review. pages 9–21, 2015.

[24] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, jan 2008.

[25] Cheng Fan, Meiling Chen, Xinghua Wang, Jiayuan Wang, and Bufu Huang. A review on data preprocessing techniques toward efficient and reliable knowledge discovery from building operational data. *Frontiers in Energy Research*, 9, 2021.

[26] Mina Farmanbar and Chunming Rong. Triangulum city dashboard: An interactive data analytic platform for visualizing smart city performance. *Processes*, 8(2), 2020.

[27] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. *SIGPLAN Not.*, 52(6):185–200, jun 2017.

[28] Eva Kuhn, Johannes Riemer, and Lukas Lechner. Xvsmp/bayeux: A protocol for scalable space based computing in the web. pages 68–73, 2007.

[29] Jacek Lato, Marek Mucha, and Tomasz Szymczyk. Comparison of the most popular operating systems in terms of functionalities. *Journal of Computer Sciences Institute*, 24:195–202, Sep. 2022.

[30] Yli-Heikkilä M. Data science languages. University of Helsinki; 2019.

[31] Benjamin Nast, Achim Reiz, and Kurt Sandkuhl. Iot-based diagnostic assistance for energy optimization of air conditioning facilities. *Procedia Computer Science*, 219:416–421, 2023. CENTERIS – International Conference on ENTERprise Information Systems / ProjMAN – International Conference on Project MANagement / HCist – International Conference on Health and Social Care Information Systems and Technologies 2022.

[32] Dakuo Wang, Justin D. Weisz, Michael Muller, Parikshit Ram, Werner Geyer, Casey Dugan, Yla Tausczik, Horst Samulowitz, and Alexander Gray. Human-ai collaboration in data science: Exploring data scientists' perceptions of automated ai. *Proc. ACM Hum.-Comput. Interact.*, 3(CSCW), nov 2019.

[33] Amy X. Zhang, Michael Muller, and Dakuo Wang. How do data science workers collaborate? roles, workflows, and tools. *Proc. ACM Hum.-Comput. Interact.*, 4(CSCW1), may 2020.

Appendix

I. Code

Source code is available in GitHub under 3 different repositories:

- Frontend repository: https://github.com/argruk/dashboard-frontend
- Server repository: https://github.com/argruk/thesis-toolset-backend
- Data function repository: https://github.com/argruk/thesis-dashboard-data-function

For the purpose of using an application is is compulsory to have data function and server repositories placed in one directory, and it is advised to have frontend repository placed in the same directory.

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Artem Grukhal**,

1. herewith 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,

   Data acquisition and preparation toolbox for Cumulocity-based solutions, supervised by Pelle Jakovits.

2. I grant the University of Tartu a permit to make the work specified in p. 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 3.0, 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 until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Artem Grukhal
10/08/2023