

UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Oleg Savik
**Tartu Smart City data integration between the
Cumulocity IoT database and ArcGIS**
Bachelor's Thesis (9 ECTS)

Supervisor:
Pelle Jakovits, PhD

Tartu 2025

Tartu Smart City data integration between the Cumulocity IoT database and ArcGIS

Abstract:

The main goal of this thesis was to develop a scalable and reusable data integration solution between the Cumulocity IoT database and the ArcGIS platform. Based on the initial requirements, the most popular Python-based data orchestration platforms were evaluated. Dagster was selected as the most suitable platform for the problem at hand. Additional custom Python packages were created during the development process to support a reusable configuration. The final solution provides seamless data integration between Cumulocity and ArcGIS while maintaining flexibility for other JSON-based APIs without requiring many changes to the core logic. Additional testing validated the reusability, data integrity, and resource efficiency of the solution and confirmed its robustness.

Keywords: Data pipeline, Dagster, ETL, Smart City

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

Tartu targa linna andmete integratsioon Cumulocity IoT andmebaasi ja ArcGISi vahel

Lühikokkuvõte:

Käesoleva lõputöö peamine eesmärk oli välja töötada skaleeritav ja taaskasutatav andmete integratsiooni lahendus Cumulocity IoT andmebaasi ja ArcGISi platvormi vahel. Esialgsete nõuete põhjal hinnati kõige populaarsemaid Pythonil põhinevaid andmeorkestreerimisplatvorme. Antud probleemi jaoks kõige sobivamaks platvormiks valiti Dagster. Arendusprotsessi käigus loodi täiendavaid kohandatud Pythoni pakette, et toetada taaskasutatavat konfiguratsiooni. Lõplik lahendus pakub sujuvat andmete integreerimist Cumulocity ja ArcGISi vahel, säilitades samas paindlikkuse teiste JSON-põhiste APIde jaoks, ilma et oleks vaja teha palju muudatusi põhiloogikas. Täiendav testimine kinnitas lahenduse taaskasutatavust, andmete terviklikkust ja ressursitõhusust ning kinnitas selle töökindlust.

Võtmesõnad: Andmekonveier, Dagster, ETL, Tark Linn

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Table of Contents

1. Introduction	7
2. Background	9
2.1 Smart City	9
2.2 Data integration.....	9
2.3 Cumulocity.....	11
2.4 ArcGIS	12
2.5 Related works.....	15
3. Requirements and Technology Selection.....	18
3.1 Functional requirements.....	18
3.2 Non-Functional requirements	19
3.3 Data Orchestration Platform	20
3.4 Technology overview.....	23
3.4.1 Monitoring	23
3.4.2 Storage	24
3.4.3 Containerization	25
3.4.4 Dagster components.....	25
4. Implementation	28
4.1 System Architecture of the Final Solution.....	28
4.2 Custom YAML-based configuration for reusable Dagster modules	29
4.3 Overview of created modules	31
4.4 General integration.....	32
4.5 Reusable template	33
4.6 Results.....	35
5. Integration verification.....	39
5.1 Validation against the other API.....	39
5.2 Data integrity validation	47
5.3 Integration resource consumption.....	48
5.4 Future work	50
6. Conclusion	51
References.....	52
Appendices.....	55
License	56

List of Figures

Figure 1. ETL process.....	10
Figure 2. ELT process.....	10
Figure 3. Feature layer [23]	13
Figure 4. Example of sending coordinates without specifying a WKID.	15
Figure 5. Dagster pipeline with partitions.....	26
Figure 6. Cumulocity measurement API call using partition values	26
Figure 7. System architecture of the final integration.....	28
Figure 8. Example of the YAML structure	31
Figure 9. General integration data pipeline.....	32
Figure 10. Example of template usage in the main.yaml file.	34
Figure 11. Example of the map created after the eco_counters job	35
Figure 12. Visualization of all sensors after multiple integrations	37
Figure 13. QuestDB API response [52]	40
Figure 14. Mapped QustDB API response	40
Figure 15. OpenWeather API response	41
Figure 16. OpenWeather API mapping block.....	42
Figure 17. Mapped OpenWeather API response	42
Figure 18. Cumulocity events API response.....	43
Figure 19. Cumulocity events mapping block	43
Figure 20. Mapped Cumulocity events API response	44
Figure 21. Cumulocity http_get module configuration	45
Figure 22. Reaalajaandmed http_get module configuration	45
Figure 23. Cumulocity json_mapper module configuration	46
Figure 24. Reaalajaandmed json_mapper module configuration.....	46

List of Tables

Table 1. Summary of related work search results	16
Table 2. Functional requirements	18
Table 3. Non-Functional requirements	19
Table 4. Comparison of DOP based on functional and non-functional requirements	21
Table 5. Reusable modules	31
Table 6. Example of a general integration of an ArcGIS table.....	35
Table 7. Overview of used datasets	36
Table 8. Overview of implemented requirements.....	37
Table 10. Materialization of 91 partitions in a row for Lights fragment	49

1. Introduction

Urban populations are continuously increasing [1], [2] , leading to new challenges such as sustainability, energy efficiency, and mobility management [3]. With the rapid growth of the Internet of Things (IoT) industry and 5G [4], cities are starting to adopt these technologies for data collection [5]. The collected data can be anything from traffic measurement to energy consumption and solar panel energy production of individual buildings. As a result, local governments analyze this data to make data-driven decisions and address urban challenges. Cities that collect data through IoT devices and leverage big data are generally referred to as Smart Cities (SCs) [6].

One of the major challenges for SCs is effectively utilizing the collected data. The data collected from IoT devices is often stored in various formats and locations, complicating its use across platforms. An example is the use case of Tartu SC, which faces these problems every day. Collected data is stored in the Cumulocity platform, which has limitations in exporting existing data. The platform itself cannot cover all analytical needs, so another platform is used. ArcGIS is a geographic information system used mainly by Tartu municipality for data analysis and visualization of geospatial data on maps. It is not possible to access Cumulocity data directly through ArcGIS, which adds another level of complexity to SC operations. In order to overcome these limitations, other solutions are needed as an intermediary between the two platforms. Several solutions have been proposed in previous theses by University of Tartu students. Previous works primarily focused on extracting data from the Cumulocity platform and storing it as comma-separated values (CSV). These solutions cannot be used directly for data integration because it would mean manually uploading CSV files into ArcGIS. Additional problems arise with these solutions in the future because they are primarily developed explicitly for the Cumulocity platform. A solution that overlaps with the current thesis problem is an integration made for the mobility visualization dashboard [7]. NIFI Data Orchestration Platform (DOP) is used for data extraction, transformation, and loading. The problem with the current integration is that it is complicated, which leads to problems with maintainability and it does not provide any reusability options for further similar integrations. These solutions will be compared in more detail in Chapter 2.5 (Related works).

The objective of this thesis is to address the challenges of reusability and maintainability by building a prototype that provides a solid foundation for future integrations. As a result, future integrations will rely on reusable modules, rather than requiring extensive coding or

exploration of the ArcGIS platform application programmable interface (API). As a consequence, future integrations between any REST API source and the ArcGIS Platform would be created in a very short time. The scope of this work focuses on ArcGIS as the target platform, though the data source may differ from Cumulocity.

To create a reusable integration, popular data orchestration tools were compared based on initial requirements. Dagster was selected as the main data orchestration platform for the integration. Several prototypes were developed to identify the optimal solution. For monitoring and validation, an additional monitoring stack was implemented alongside the final solution to ensure system stability. The system's reusability was tested against various APIs.

This thesis is divided into 6 chapters. Chapter 2 provides the necessary background and insight into previous research related to this topic. Chapter 3 outlines the requirements for the integration, together with the justification of the chosen tools. Chapters 4 and 5 focus on the implementation of the final solution together with the architectural description and validation of reusability, resource consumption, and data quality. Finally, Chapter 6 summarizes the conclusions and key findings of the thesis.

Some parts of this thesis were enhanced using OpenAI's ChatGPT to improve the clarity and fluency of the text. The author originally wrote all ideas and content before requesting any suggestions. Additionally, ChatGPT was used to assist in solving specific programming problems. These contributions are documented and commented within the code. GPT-4o model version **2024-08-06**.

2. Background

This section provides an overview of the Smart City concept, data integration challenges, and the role of data orchestration platforms. It also examines the key features and limitations of both Cumulocity and ArcGIS APIs. Finally, related works and existing solutions are analyzed to highlight their shortcomings and demonstrate how this research addresses these challenges.

2.1 Smart City

The definition of a smart city varies from study to study and the commonly agreed definition of a smart city remains undefined [4]. In this work, a SC is viewed as a city that is capable of collecting data through IoT devices and storing the collected data on platforms such as Cumulocity.

Tartu City started its road toward SC during the SmartEnCity project [3]. Many initiatives were brought to life during the project. Creation of Rattaringlus¹, 312 smart light controllers were placed together with 30 passive infrared sensors (PIR) movement detectors, 6 movement detectors with cameras, 10 noise sensors, 10 light reflection sensors, and 5 environmental sensors for pollution measurement [8], [9].

2.2 Data integration

L. Cai and Y. Zhu state in their study [10] that one of the biggest challenges in the era of big data is data integration across multiple platforms, which is also relevant for SCs. In most cases, it is not possible to move data directly from source A to target B. There are several reasons for this. For example, the data is in an incompatible format for a target, the target only needs to receive a small portion of the data from source A, or the data needs to be modified and aggregated before it is moved to the target.

Data can be processed in batches or streamed. Batch processing is suitable for infrequent data updates. For example, transferring data for the past day, month, or triggering a process manually [11]. On the other hand, data streaming provides near-real-time data transfer [11], which is critical for applications that rely on near-real-time data, such as real-time bus schedules.

¹ Rattaringlus <https://ratas.tartu.ee/>

With the growth of big data, system designs have changed significantly. Now, data is not only stored in a traditional database but also in data warehouses and data lakes. In a data warehouse, filtered, structured, and processed data is stored. On the other hand, data lakes are designed to store undefined raw data. Data warehouses are primarily used to store enterprise data and provide data to analytical applications[12].

To perform integration between A and B, extract, transform, and load (ETL) operations are essential. These operations may vary depending on the goals. At the extract stage, data is retrieved from a data source. Data sources can be anything from a REST API to a SQL database. In a transformation stage (Transform), data is transformed into the required format by applying required data aggregation, cleaning, enriching, and filtering methods. The load stage represents uploading/sending data to the target compatible format. The ETL process is visualized in Figure 1, which illustrates the data flow for each step.

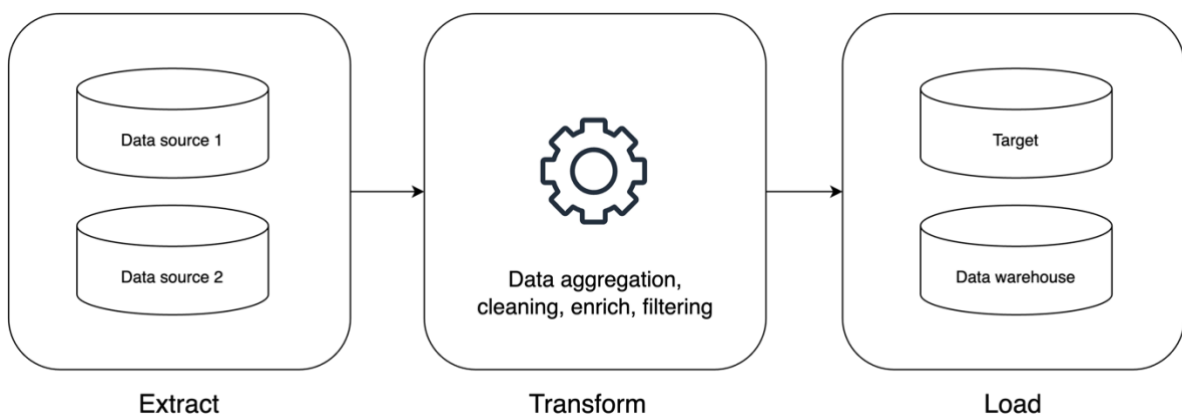


Figure 1. ETL process

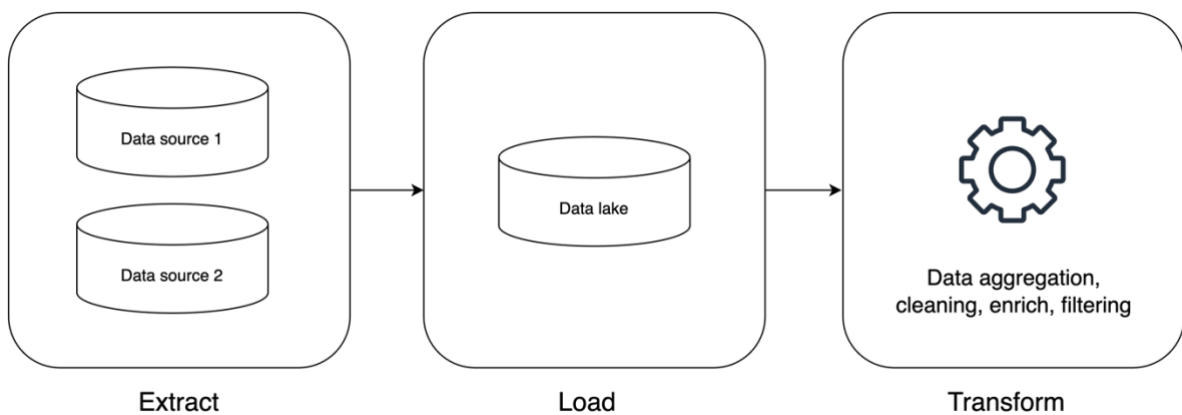


Figure 2. ELT process

In addition, there is another process called ELT (Figure 2), where data is loaded directly into the data lake and then transformed into the required structure. The ELT concept is primarily used in cloud environments [11].

For the ETL process, it is possible to use a Python script that can retrieve data from source A, modify it, and publish the results to target B. This approach may work for simple integrations, but it does not work well for complex integrations where automation, data quality, and task scheduling are critical. This is where data orchestration platforms come in.

A data orchestration platform (DOP) solves these challenges by providing the necessary capabilities to develop reliable workflows. It coordinates job execution, manages scheduling, and addresses issues related to observability, reusability, configuration, and scalability. The DOP is a natural evolution from simpler workflows, such as standalone Python scripts. There is a wide variety of DOPs available, ranging from open-source solutions to commercial, paid options. A single study conducted by M. Matskin et al. [13] identified 37 tools that could potentially be used for data automation, ranging from script-based to declarative solutions.

However, there are still many problems associated with data processing. Challenges such as data quality issues [10], infrastructure scalability [14], geographically distributed data [15], and latency problems [16] still persist.

2.3 Cumulocity

Cumulocity is an IoT platform developed by Software AG. The platform allows IoT devices to connect directly to the platform, monitor activity, manage all remotely connected devices, and aggregate the historical data they collect for future analysis [17]. This platform was used by the City of Tartu as the primary platform for managing IoT devices. According to the Kaspar Kadalipp master's thesis [18], there are about 19000 registered devices and the size of the database is about 1.3 terabytes as of 2024.

Another instance of Cumulocity is running at the University of Tartu Delta Centre². It is mainly used to store data from all sensors located in the Delta building, such as CO₂, temperature, and humidity. In this thesis, Cumulocity is treated as a database rather than a platform and is used as a data source for integration with the ArcGIS platform.

² University of Tartu Delta Centre <https://delta.ut.ee/en/content/discover-delta>

Measurements

Primarily, the measurement endpoint [19] is used to retrieve data from the Cumulocity. Specific measurements can be retrieved using the source or fragment type. One of the limitations of Cumulocity can be considered a paging system, which makes it harder to get a high amount of data at once. It should be noted that this is considered a limitation only from the integration point of view because it requires sending multiple requests. According to Silver Laius' thesis [20], each subsequent request to the Cumulocity server becomes slower

Inventory

The inventory endpoint [19] can be used to retrieve information for a specific sensor or to fetch details for all available sensors, including geospatial data and sensor names. There is no way to query the measurement along with the inventory information.

Authentication methods

Cumulocity API supports the following authentication options [19]:

- Basic authentication (Basic)
- OAI-Secure authentication (OAI-Secure)
- SSO with authentication code grant (SSO)
- JWT authentication with an access token from a IAM (JWT-IAM)

Basic authentication will be used. The reason is that the Cumulocity instance used by the City of Tartu does not support other options. The available authentication options were validated using the **tenant/loginOptions** endpoint. Currently, user credentials are provided directly, which creates a high risk of password exposure. Many large companies, such as Microsoft, have stopped using Basic authentication [21] because of its security issues.

2.4 ArcGIS

ArcGIS is a geographic information system (GIS) platform for visualizing and analyzing geospatial data. Users of the platform can analyze data, create interactive dashboards and maps using geospatial data, and share them with other users [22]. The platform is widely used by the

City of Tartu for data analysis, dashboard, and map creation. Many dashboards and maps are shared with the public through the official ArcGIS portal of Tartu³.

Feature service

According to the official ArcGIS documentation [22], a feature service is a service that is exposed to the Internet and contains datasets with or without spatial data. Datasets with spatial data are known as layers. Datasets without spatial data are known as tables.

Feature service provides the ability to read, edit, and create different types of layers and store data within each feature layer.

ArcGIS documentation [23] describes a feature layer (Figure 3) as a layer containing different features that are grouped within that layer. Features can be points, lines, or polygons. These features are also known as sublayers of the feature layer. Feature layers are used for data visualization on top of maps.

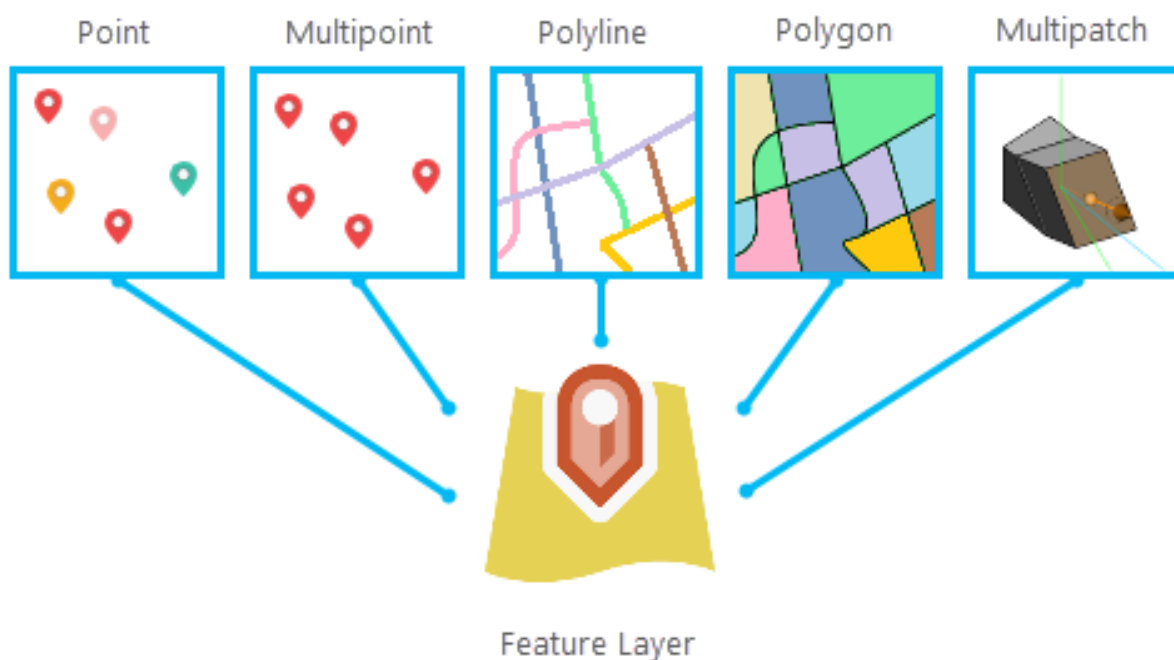


Figure 3. Feature layer [23]

For the integration between Cumulocity and ArcGIS, it is sufficient to use a point sublayer. The point sublayer provides the ability to store data along with a geospatial tag. It will not be available in the data table but will be used to display a point on the map.

³ ArcGIS portal of Tartu <https://tartu.maps.arcgis.com>

Features can be created using the ArcGIS Feature Service API, which accepts a JSON payload containing both attribute fields and geometry data (e.g., point, line, or polygon).

```
{
  "geometry": <geometry>,
  "attributes": {
    "name1": <value1>,
    "name2": <value2>,
  }
}
```

Attributes correspond to predefined table values, if a key does not exist in the table schema, it will not be added to the ArcGIS table [24].

Table data types

Every ArcGIS layer supports table creation. The table has the most common data types [25]:

- String
- Date
- Date only
- Double
- Integer
- Time only
- Timestamp offset

Table dependencies, like those in SQL databases, exist but will not be relevant for this integration.

Coordinates

It is also important to understand the types of coordinates. World Geodetic System 1984 (WGS84) is the most common coordinate system in the world and is used in the Global Positioning System (GPS) [26]. For example, the GPS coordinates of Tartu are 58.38, 26.72 [27]. The well-known ID (WKID) [26] for WGS84 is 4326. And default coordinate system in ArcGIS is Web Mercator Auxiliary Sphere with a WKID of 3857 [28]. The coordinate for

Tartu in the Web Mercator Auxiliary Sphere looks like 6498620.365478803, 3089318.9663040168. This difference should be considered before sending data to ArcGIS.

By default, ArcGIS stores coordinates in a projected coordinate system unless a WKID is explicitly provided [28]. Most Cumulocity sensors have coordinates in WGS84 and the WKID should be specified before sending geospatial data to ArcGIS. Otherwise, the result may not be as expected. As shown in Figure 4, a feature using WGS84 coordinates for the City of Tartu with an incorrect WKID may appear in the middle of the ocean instead of its actual location.



Figure 4. Example of sending coordinates without specifying a WKID.

Authentication methods

There are two authentication methods available: generating an API key or using an OAuth2 application [24]. The key difference is that an OAuth2 application token has a fixed lifetime of 14 days [29], after which it must be regenerated, whereas an API key can be configured for a longer validity period of up to one year [30]. Currently, in the Tartu ArcGIS portal, generating an API key is not possible. Only OAuth2 authentication is supported. While OAuth2 enhances security and access control, it introduces additional complexity since a one-time token cannot be used indefinitely. The OAuth2 client credentials flow [31] should be used for server-side applications, as it enables machine-to-machine authorization, which is required for this integration.

2.5 Related works

To find related work, previous theses from the University of Tartu were searched [32]. The search statistics are presented in Table 1. The following keywords were used: Cumulocity,

ArcGIS, and Data pipeline. The column “Found works” shows the total number of theses found for each keyword, while “Related works” indicates how many of them were relevant to the current thesis.

Table 1. Summary of related work search results

Keyword	Found works	Related works
ArcGIS	0	0
Cumulocity	9	3
Data pipeline	47	0

The results were analyzed for relevance by identifying works that specifically focused on integration use cases. Theses that simply used Cumulocity without creating integration were excluded from the related works category.

The following works were discovered:

- “Solution for Sharing Open Data from Cumulocity IoT Platform” [33]
- “Integration library for analyzing historical data from the Cumulocity IoT platform in Python” [20]
- “Data Acquisition and Preparation Toolbox for Cumulocity-based Solution” [34]

Additional information about the existing NIFI⁴ integration was provided by the supervisor of this thesis Pelle Jakovits.

The work by Silver Laius [20] focused on creating a Python package for communicating with the Cumulocity platform. It used the Cumulocity API with additional abstraction to make it usable for Python code. Although this solution supports measurement requests, it will not be used in this work. The package is outdated and its compatibility with the latest Cumulocity API is uncertain. This thesis focuses on developing a general solution rather than relying on tools designed specifically for platforms like Cumulocity.

The following two theses [33], [34] focus on developing user-friendly web applications that retrieve data from the Cumulocity platform. In these works, the retrieved data is stored as a CSV file, which is then made available to the end user for download. These solutions cannot

⁴ NIFI <https://nifi.apache.org/>

be used directly for integration. The main problem is that solutions are highly coupled to the Cumulocity platform, making it difficult to use with different APIs. An additional problem is that the data retrieval process is focused on manual downloads from the developed web applications.

The NIFI integration is currently being used to move mobility data from Cumulocity to ArcGIS. NIFI, supported by Apache, allows the creation of data pipelines through a web interface. It provides a wide variety of modules for building pipelines, from simple HTTP requests to transforming acquired data into the desired formats and sending it to a broad range of destinations [35].

However, NIFI is not an ideal tool and has its own issues. When the data pipeline becomes larger, it starts to look very complicated with many arrows and boxes. Moreover, it may have grouped tasks, which may be hidden at first glance. Debugging such pipelines is difficult and almost impossible to reuse for other similar solutions. According to NIFI documentation [36], created pipelines can be exported to an Extensible Markup Language (XML) file. Additionally, the flow definition of the process group can be downloaded as a JavaScript Object Notation (JSON) file. These files are huge and cannot be easily modified to transform an existing pipeline into a reusable one using templating tools.

This thesis aims to improve the maintainability and reusability of data pipelines. Instead of creating a new Python library for getting data from Cumulocity, it tries to achieve the same in a declarative way by using Yet Another Markup Language (YAML) configuration. By using the structured configuration in YAML, it would be possible to overcome the maintainability problem and solve the reusability problems that are present in NIFI.

3. Requirements and Technology Selection

This section introduces the general requirements for data integration between the Cumulocity IoT database and ArcGIS. The final chapters discuss the justification for the selected technologies.

3.1 Functional requirements

All requirements are prioritized using the MoSCoW⁵ method. Functional requirements are listed in Table 2, along with their corresponding priorities. A “High” priority is assigned to requirements that must be implemented, “Medium” to those that should be implemented, and “Low” to those that could be implemented. The same prioritization applies to the non-functional requirements described in Table 3.

Table 2. Functional requirements

Functional requirement	Priority
FR1: The integration must remove any duplicate records from the dataset	High
FR2: The integration must support data quality validation (e.g., missing values, incorrect formats) before pushing data to ArcGIS	High
FR3: The integration must ensure that data is sent to ArcGIS in a compatible format	High
FR4: The integration should support data aggregation	Medium
FR5: The integration must support schedule runs	High
FR6: The Integration must be able to upload data to ArcGIS from the previous date	High
FR7: The integration should work with any JSON-based HTTP API as a data source.	Medium
FR8: The integration must be designed to be flexible and reusable by allowing configuration through a configuration file.	High
FR9: The integration should support error detection and allow easy re-execution of failed tasks.	Medium
FR10: The integration should support automatic retry policies in case of temporary failures.	Medium

⁵ MoSCoW method <https://www.agilebusiness.org/dsdm-project-framework/moscow-prioririsation.html>

FR11: The integration should store logs and metadata to provide historical run information for debugging and auditing.	Medium
FR12: The integration could inject sensor name and coordinates, before sending to ArcGIS	Low

Praks mentioned in his thesis [37] that sensors used for traffic monitoring (AVC sensors) in the city are generating many duplicates. For example, some sensors have more than half of their stored data as duplicates. Transferring data to ArcGIS without removing duplicates can lead to data quality issues and provide unreliable data about car traffic to the citizens. To address these problems, the following functional requirements are needed: **FR1**, **FR2**, **FR3**.

Some data can be aggregated before being transferred to ArcGIS. For example, the AVC dataset contains approximately 30000 objects per day. By aggregating this data hourly, the total number of objects is reduced to 576 (**FR4**).

Current NIFI integration transfers past days data from Cumulocity to ArcGIS every day, providing up-to-date data. These functionalities should be taken into consideration in the new data integration as well (**FR5**, **FR6**).

One of the challenges that should be considered is the fact that Tartu City has plans to discontinue Cumulocity. Hower University of Tartu Delta Centre has its instance of Cumulocity, there are no plans to move this data to ArcGIS. To address this challenge and make the thesis solution valuable, the proposed solution should be reusable and allow switching between different JSON-based HTTP API platforms without investing a lot of effort in the new integration creation (**FR7**, **FR8**).

Auto recovery from possible failures is important. If the pipeline could not retrieve data from the source, because of source-related problems, network problems, or DNS resolution problems, the auto recovery should be performed to ensure stability and mitigate anomaly risks (**FR9**, **FR10**, **FR11**).

3.2 Non-Functional requirements

Table 3. Non-Functional requirements

Non-Functional requirement	Priority
NFR1: The integration must be compatible with Kubernetes	High

NFR2: Tokens and other sensitive data must be managed securely	High
NFR3: The integration could provide a user-friendly UI to display job execution status and history.	Low
NFR4: The integration must have monitoring and observability features	High
NFR5: The integration should be scalable to support increased data volume without degradation of performance.	Medium

In addition to the functional requirements, there are also non-functional requirements. These are described in Table 3, along with their corresponding priorities.

Modern data pipelines should be scalable to provide the best possible performance during peak load volume and utilize fewer resources during low volume periods. The Kubernetes-compatible solution should be considered. This technology is widely used in popular cloud platforms, providing all necessary instruments for scaling (**NFR1, NFR5**).

Pipelines, most of the time, request data from sources that require authentication. Keeping credentials safe is a key to company security and public trust. All credentials should be stored safely, especially in a data pipeline, where data could be confidential (**NFR2**).

As current data workflows grow, it is becoming hard to understand how the pipeline is functioning on a higher level most of the time may be viewed as a black box. The modern UI, which provides valuable information about the data pipeline as a whole, is needed to simplify data pipeline understanding to the end users and pipeline maintainers (**NFR3**).

The monitoring is another key aspect that should be addressed as well. Without a proper monitoring and notification solution, pipeline reliability would be questionable. Many issues that occurred during the pipeline exploitation would not be addressed in time. Additionally, monitoring provides valuable insight into pipeline operation problems (**NFR4**).

3.3 Data Orchestration Platform

This thesis primarily focuses on open-source solutions to avoid vendor lock-in and maintain cloud agnosticism whenever possible. This approach ensures that the solution can be deployed in various environments rather than being tightly integrated with a specific cloud provider such as Google Cloud, Azure, or Amazon Web Services (AWS).

Selecting the right Data Orchestration Platform (DOP) is crucial for this thesis. There is no need to develop a custom solution using traditional scripts or build a new web application with similar capabilities, as many existing DOPs already fulfill both functional and non-functional requirements. A well-chosen DOP can provide essential features out of the box, reducing development effort and improving maintainability.

The following DOPs were selected for comparison: Airflow[38], Dagster[39], Prefect[40], Luigi[41], Metaflow[42], and Flyte[43]. These tools were chosen based on two key criteria: they are open-source and they support defining workflows using the Python programming language. The choice of Python was due to its popularity in data processing and the fact that some employees of the City of Tartu are familiar with this programming language. Table 4 presents a comparison of these open-source DOPs, based on the official documentation of each platform. “Yes” in the table means that a specific requirement is satisfied by a given DOP, while “No” indicates that it cannot be satisfied by that DOP.

Functional requirements **FR1**, **FR2**, **FR3**, **FR4**, **FR6**, **FR7**, and **FR8** are not included in the table, as they can be implemented in any DOP using Python. Similarly, **NFR2**, which can be defined via .env files or other Python-based methods, is excluded as well.

Table 4. Comparison of DOP based on functional and non-functional requirements

Requirement	Dagster	Airflow	Prefect	Luigi	Metaflow	Flyte
Web UI (NFR3)	Yes	Yes	Yes	Yes	Yes	Yes
Monitoring (NFR4)	Yes	Yes	Yes	No	No	Yes
Kubernetes ready (NFR1)	Yes	Yes	Yes	Yes	Yes	Yes
Convenient task re execution (FR9)	Yes	Yes	Yes	No	No	Yes
Task scheduling (FR5)	Yes	Yes	Yes	No (Cron job required ⁶)	No (Relies on external tools ⁷)	Yes

⁶ Luigi scheduler https://luigi.readthedocs.io/en/stable/central_scheduler.html

⁷ Metaflow scheduler <https://docs.metaflow.org/production/scheduling-metaflow-flows/introduction>

Previous run history (FR11)	Yes	Yes	Yes	Yes	Yes	Yes
Retry policy (FR10)	Yes	Yes	Yes	Yes	Yes	Yes

The provided table allows to eliminate Luigi and Metaflow because they do not meet the requirements of **NFR4**, **FR9**, and **FR5**. The remaining DOP Dagster, Airflow, Prefect, and Flyte meet all requirements. This means that the decision should be made on the three most important requirements: **NFR1**, **NFR4**, and **FR5**.

While task scheduling is available in each remaining DOP, there is no reason to compare them since they already satisfy the requirement. The same is true for Kubernetes support. Therefore, the final decision will be based on the built-in monitoring capabilities and observability features provided by each DOP.

Dagster has rich metadata support for individual assets within a pipeline, providing better observability over the data pipeline. Dagster allows to define custom metadata for individual assets and displays it in the web UI, providing valuable information about individual assets. Additionally, Dagster has a built-in logger along with a lineage tracing feature. This allows for identifying and debugging potential bottlenecks in the data pipeline. In addition, a data quality feature is built directly into the asset, providing data quality in the same view. The closest competitor is Flyte. It also supports metadata output, but it is basic compared to Dagster. Flyte mainly provides insight into input and output data, while Dagster supports defining custom metadata from a pandas DataFrame⁸ to different plots. Other tools like Airflow and Prefect cannot provide the same level of metadata support as Dagster. The same is true for lineage tracing, where Dagster provides much deeper insight into how data is moved between different steps.

Based on the initial requirements, Dagster was identified as the most suitable DOP for this integration.

⁸ Pandas DataFrame <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html>

3.4 Technology overview

This section provides an overview of the technologies used during development, together with a justification for each technology.

3.4.1 Monitoring

Monitoring tools allow collecting valuable information about virtual machine (VM) performance along with containerized environment metrics. Collecting metrics provides valuable information about performance, storage, and memory issues of VMs and individual containers. Using these metrics, it is possible to identify potential bottlenecks in the system and define alert rules that would be delivered in the event of a rule violation.

Prometheus⁹ is an open-source monitoring system developed at SoundCloud¹⁰. The system uses a pull model over HTTP for metric collections and stores as time series data. It has its own query language called PromQL [45].

The Prometheus was chosen as the core of the monitoring system because it has a wide range of exporters. Node Exporter¹¹ can be used to monitor Linux machines by exposing a wide variety of metrics, ranging from central processing unit (CPU) and random-access memory (RAM) consumption, disk space, to inbound and outbound network traffic. Another valuable exporter is CadVisor¹² [46], which exposes the resource usage of running containers.

Grafana¹³ is an open-source platform [47] used to create dashboards using a variety of data sources, including Prometheus, InfluxDB¹⁴, MongoDB¹⁵, and others. Grafana also includes alerting functionality that allows alerts to be created directly within Grafana. The main reason for choosing Grafana is that it has native support with Prometheus. In addition, there are already built-in dashboards for Node Exporter and CadVisor that can be used for monitoring. All together, these components provide a powerful monitoring system along with visualization capabilities.

⁹ Prometheus <https://prometheus.io/>

¹⁰ Sound Cloud <https://soundcloud.com/>

¹¹ Node Exporter https://github.com/prometheus/node_exporter

¹² CadVisor <https://github.com/google/cadvisor>

¹³ Grafana <https://grafana.com/grafana/dashboards/>

¹⁴ InfluxDB <https://www.influxdata.com/products/influxdb/>

¹⁵ MongoDB <https://www.mongodb.com/>

3.4.2 Storage

The database in Dagster is used for storing metadata, event logs, and other data. Currently, Dagster supports 3 databases as persistent storage. Each database is SQL-based. SQLite¹⁶, PostgreSQL¹⁷, and MySQL¹⁸ [39]. The Dagster documentation does not give any recommendations for databases, leaving the choice up to the user.

SQLite is a file-based database that does not require a separate database server. It is a good option for the development environment, single-user local applications, testing, and mobile applications [48]. In a containerized environment, it is problematic to use SQLite. It is not designed to work over the network with many clients and has problems with high concurrency [48]. It is still possible to share a volume with each container so that it has direct access to the database file, but there may still be problems with concurrency. These problems make working with SQLite at a high scale very unreliable, especially when the jobs are executed in the Kubernetes cluster. PostgreSQL and MySQL are open-source relational database management systems (RDBMSs). Both databases address SQLite's problems with high concurrency and client connections over the network [48]. The preference for PostgreSQL is based on its popularity [49], and it provides stronger support for concurrent queries than MySQL [50], which is important when many parallel jobs are running at the same time.

Another storage solution is required to store the pickle¹⁹ objects that Dagster generates during the runs. Pickle object used for serializing and deserializing Python objects. By default, Dagster stores them on the local file system, which makes it not usable in a containerized environment as well. It is recommended to use Simple Storage Service (S3) buckets to share these objects between different containerized jobs. The MinIO²⁰ is chosen as the object storage. This choice was made primarily because it is open source and compatible with S3 APIs. It provides a highly scalable object storage solution. Additionally, some artifacts generated during the execution of the data pipeline can also be stored in object storage, such as CSV, images, and machine learning models.

¹⁶ SQLite <https://www.sqlite.org/>

¹⁷ PostgreSQL <https://www.postgresql.org/>

¹⁸ MySQL <https://www.mysql.com/>

¹⁹ Python pickle <https://docs.python.org/3/library/pickle.html>

²⁰ MinIO <https://min.io/>

3.4.3 Containerization

Containerization technology is used for integration, packaging and deployment. It simplifies deployment across different environments and ensures consistency by encapsulating each application in its own isolated environment. Each container contains all the necessary dependencies for the application to run, eliminating potential dependency conflicts that can arise when all components are hosted on the same machine. Additionally, containers are lightweight compared to traditional virtual machines, as they do not require the full virtualization of an operating system. Instead, containers share the host's resources, making them more efficient in resource utilization [51]. Docker²¹ has been selected as the primary containerization tool due to its widespread adoption and compatibility with orchestration tools such as Kubernetes.

3.4.4 Dagster components

Dagster data pipelines are typically built using the following components: assets, resources, and partitions. Additionally, Dagster provides several core components that should be considered to fully leverage its capabilities.

Assets and resources

Assets represent the core computational logic within a Dagster pipeline, defining processes such as data retrieval, transformation, and uploading. Resources are configurable dependencies that assets can use. The `ConfigurableResource`²² class allows easy management of external services, providing a structured way to track which assets depend on which resources.

Partitions

Partitions are a powerful feature of Dagster that do not exist in other previously compared DOPs in chapter 3.3. Partitions allow the creation of mini assets based on partition values. This approach enables each mini asset to run in parallel, improving performance and observability compared to a traditional asset. If one of the requests fails during the run, it is sufficient only to re-run the failed partition. Partitions help save computing power and time.

²¹ Docker <https://www.docker.com/>

²² `ConfigurableResource` <https://docs.dagster.io/api/python-api/resources>

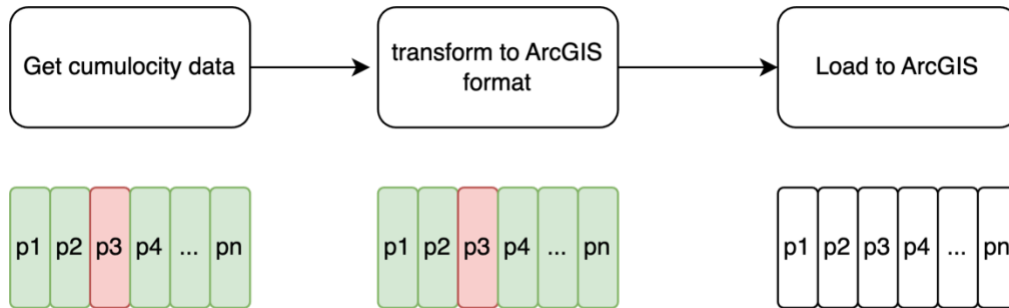
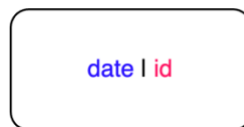


Figure 5. Dagster pipeline with partitions

In Figure 5, partitions are defined with the p character. The green color represents a successfully executed partition. The red color highlights that the partition failed, and partitions without any color have not yet materialized in Dagster. The rounded rectangle represents an asset.

The date partitions can be used as date constraints in API calls if it is supported and static partitions for specific source name or fragment name in the case of the Cumulocity API. This approach is demonstrated in Figure 6, where the partition contains two values. One value is for the date and another value is id. These values can be dynamically passed to the GET request during partitioned asset execution.



GET/measurement/measurements?source={id}&dateFrom={date}&dateTo={date}T23:59:59.999Z

Figure 6. Cumulocity measurement API call using partition values

The partition-based approach has much more control over the data pipeline. Defining a static partition for required sources provides a validation that exactly this data has been moved. Re-execution of the partitions can be done via the Dagster UI.

I/O manager

I/O managers are responsible for reading and storing data between computational steps. By default, Dagster uses the local file system to store generated objects. However, it can also be configured to use other storage solutions, such as an S3 bucket.

Job

A Job in Dagster allows grouping assets into a single logical unit, enabling the execution of all assets within that group as part of one job.

Sensors

Triggers in Dagster are known as sensors, and they enable jobs or assets to be executed based on specific conditions. For example, it is possible to trigger a second job once the first job has completed successfully.

Hooks

Dagster supports two types of hooks: success hooks and failure hooks. These hooks can be triggered based on the outcome of a job. For instance, a failure hook could be used to send an alert if a pipeline run fails.

Schedules

Schedules allow automating job execution at defined intervals, for example, triggering a job every five minutes or once per day at a specific time.

4. Implementation

This section introduces the architecture of the final solution and describes the implementation of the integration. It also presents the custom YAML-based workflow developed to support reusability, along with the implemented modules used in the integration process. Finally, it discusses the results of the implementation, including which Cumulocity datasets were used and how the functional and non-functional requirements were fulfilled.

4.1 System Architecture of the Final Solution

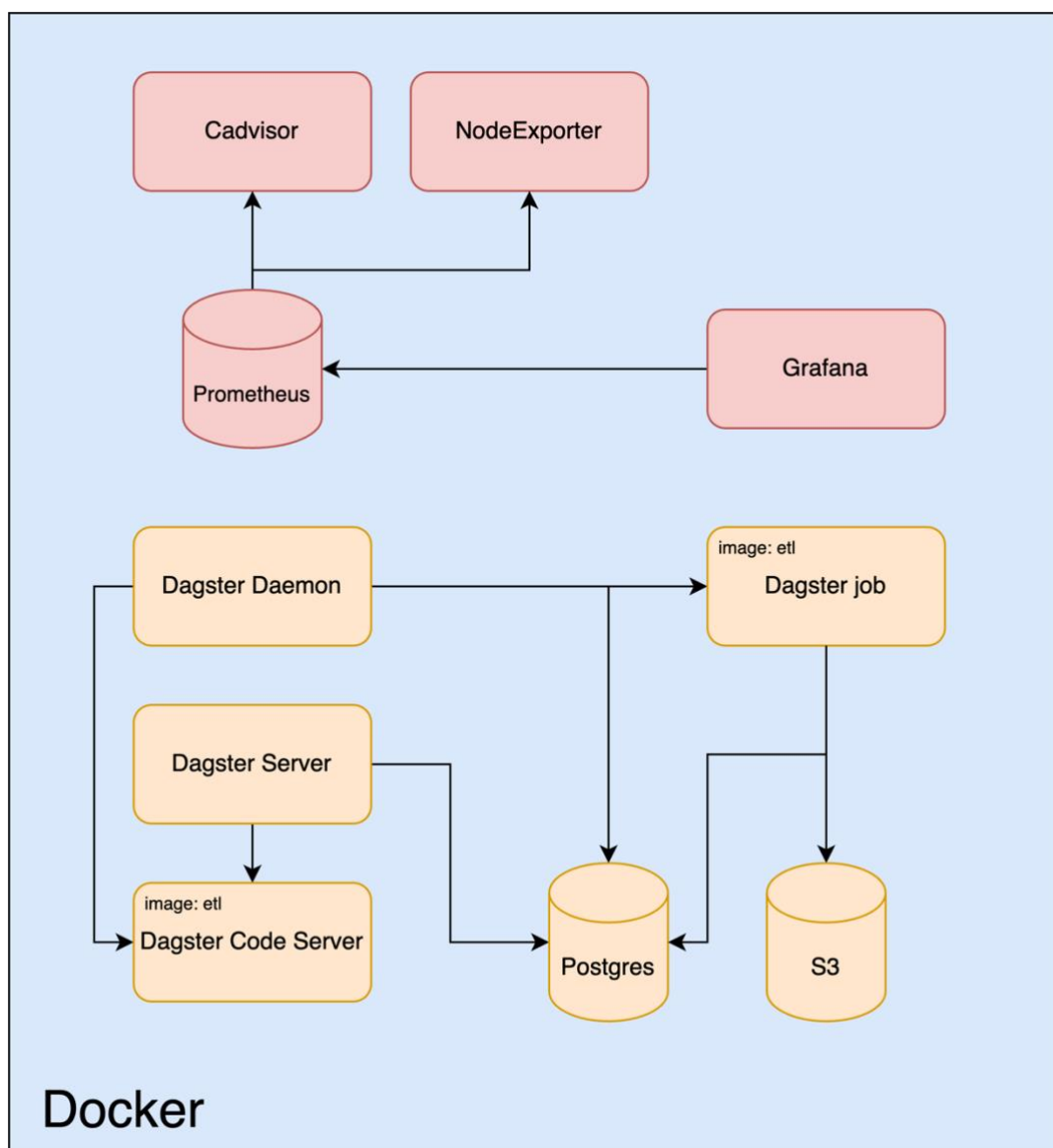


Figure 7. System architecture of the final integration

The final solution architecture demonstrated in Figure 7 consists of 10 containers. The cylinder and circular rectangle represent individual containers. The yellow label represents the Dagster stack, and the red label represents the system monitoring stack. Arrows indicate network communication between containers, showing which container sends requests and which one receives them.

Dagster requires three containers to run containerized jobs in Docker. The Dagster Server provides the Dagster UI. The Dagster Code Server is the container where the user's code is located (can be more than one), and its Docker image is used to execute jobs. The Dagster Daemon coordinates job execution based on schedules, sensor triggers, or user requests. This container can be considered the "brain" and main coordinator of the orchestration platform. Although Dagster Job is represented as a single container, it can run multiple instances of these containers simultaneously.

PostgreSQL stores all Dagster-related data. The S3 bucket serves as a temporary object store where Dagster caches data and pipeline-related artifacts.

Grafana provides dashboards for visualizing system metrics, as well as Docker resource usage, which is provided by the system monitoring stack.

4.2 Custom YAML-based configuration for reusable Dagster modules

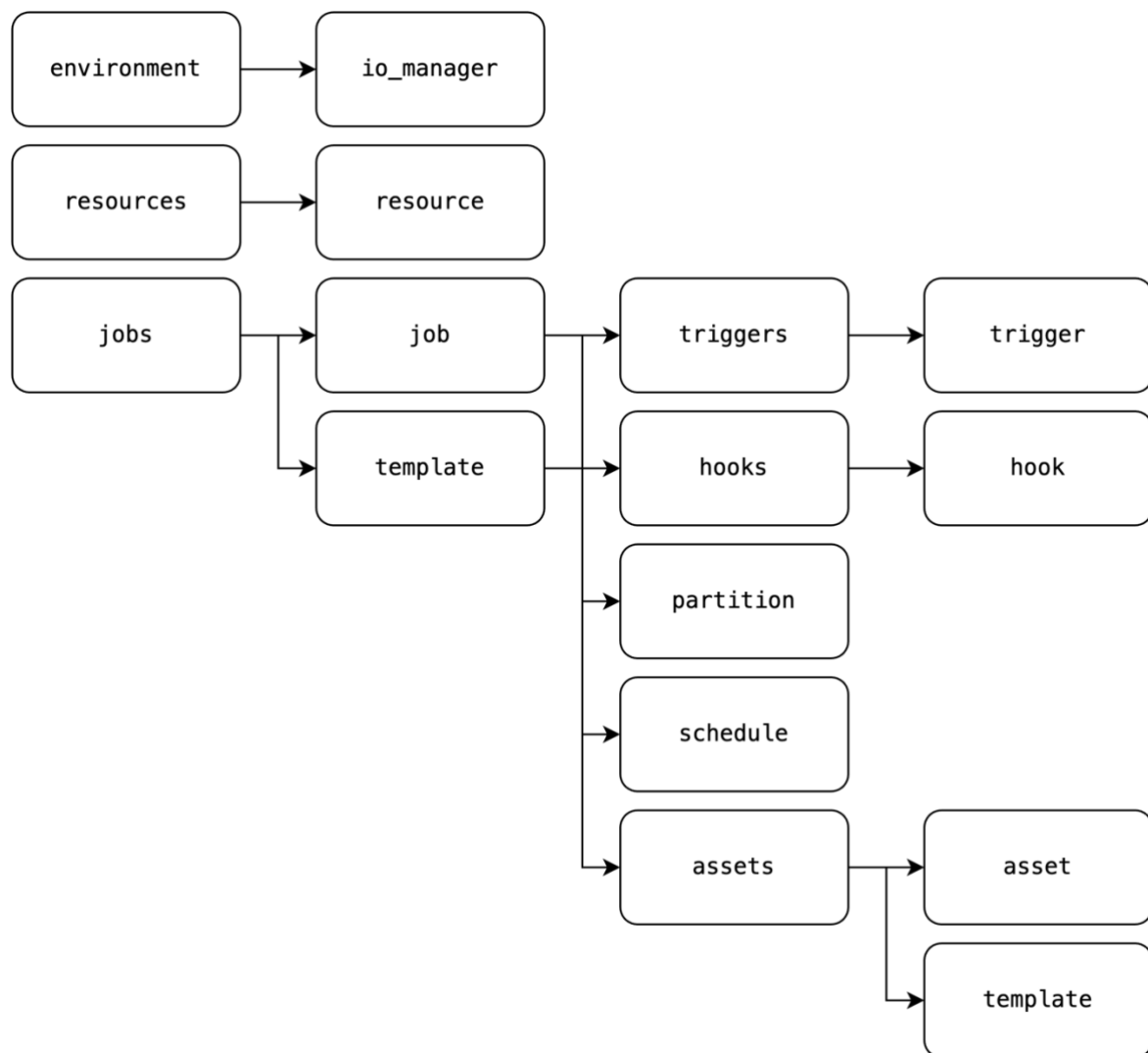
Dagster does not provide a built-in mechanism for reusing pipeline definitions. As a result, creating a new pipeline with similar logic typically involves copying existing code and modifying component names and parameters. This leads to code duplication and complicates debugging and maintenance.

To address reusability issues and simplify new data pipeline creation, the idea of separate reusable modules was introduced. In this context, reusable modules refer to Dagster assets that can be configured via a YAML file. The inspiration for this approach comes from Azure Pipelines²³ and Ansible²⁴ Playbooks, where modular components are used for system configuration. Additionally, some ideas are taken from Dagster official documentation [39] from the section named "Creating asset factories". This approach allows the entire data pipeline logic to be defined directly in a configuration file by using reusable modules.

²³ Azure pipelines <https://azure.microsoft.com/en-us/products/devops/pipelines>

²⁴ Ansible <https://github.com/ansible/ansible>

The implemented solution comes as a Python package and provides a simplified API for creating new modules and direct integration with YAML configuration, together with type and structure validation using the Pydantic²⁵ framework. Pydantic is used for configuration validation, along with object creation after successful validation. Created objects are placed in the repositories from which the final Dagster definition is created. Modules can be created by using abstract classes together with registration decorators. Abstract classes provide a simplified API for modules to retrieve attributes from the configuration if needed, and additionally, they create all required arguments for the asset. Abstract class uses a basic Pydantic class, so it is possible to provide a new module with additional required parameters, which will provide validation for the newly created module during the pipeline initialization stage as well. Registry Decorator is used for module auto-discovery.



²⁵ Pydantic <https://docs.pydantic.dev/latest/>

Figure 8. Example of the YAML structure

Implemented YAML configuration supports almost all Dagster components as demonstrated in Figure 8: I/O manager, resources, jobs, hooks, triggers, assets, partitions. Additionally, support for Jinja templating for jobs and assets has been implemented.

4.3 Overview of created modules

Table 5 presents the implemented modules built using the custom YAML-based configuration. The "Module name" column shows the actual names of the modules as used in the code and configuration files. The "Description" column provides a brief overview of each module's functionality. Modules with names starting with "pd." expect the input data to be in pandas DataFrame format, while modules starting with "ld." expect a list of dictionaries as input.

Table 5. Reusable modules

Module name	Description
http_get	This module allows sending HTTP GET requests to any accessible endpoint. It supports both Basic Authentication and Bearer Token Authentication. Partition keys can be mapped to specific parameters if needed. Additionally, the module is compatible with paginated APIs, such as Cumulocity.
json_mapper	This module allows extracting any type of data from a JSON object using the JMESPath ²⁶ library, a query language designed specifically for JSON.
write_to_csv	Creates a CSV file from a list of objects. Can store CSV in S3 bucket or local environment.
df_to_dict	Transforms pandas DataFrame to list of dictionaries
pd.date_aggregator ld.date_aggregator	Allows aggregation of data based on user requirements.
ld.pandas_ops	Allows to use pandas-based operations as commands.

²⁶ JMESPath <https://jmespath.org/>

questdb.api	This module was implemented specifically for QuestDB during the validation process. It uses the QuestDB API and transforms the response into a pandas DataFrame.
date.timestamp	Module allows to add current timestamp to objects.
pd.nan_check ld.nan_check	It checks certain columns or keys for NaN values. It can also validate the number of rows.
dict.remove_duplicates	Removes duplicates from a list of dictionaries
s3.injection	Injects values from S3 bucket CSV file (inventory) into the current pandas DataFrame using user configuration.
transform_to_argcis_format	This module converts a list of dictionaries into an ArcGIS-compatible JSON format.
send_to_argcis	This module sends data to ArcGIS. Before transmission, it validates that all required columns exist in the target ArcGIS table. Data is sent asynchronously using the aiohttp ²⁷ library.

4.4 General integration

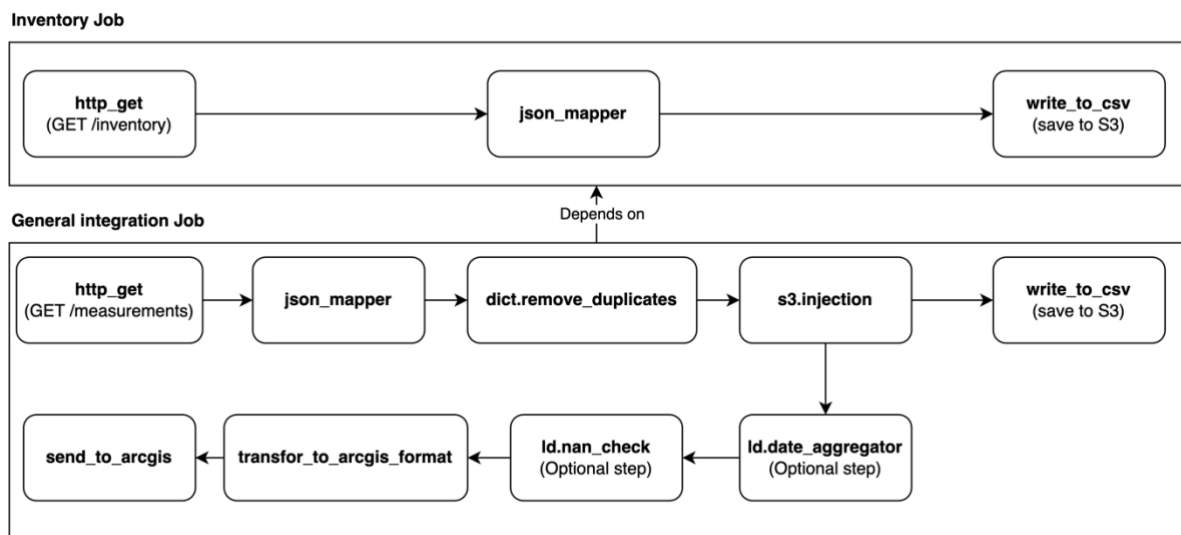


Figure 9. General integration data pipeline

²⁷ Aiohttp <https://docs.aiohttp.org/en/stable/>

General integration consists of two jobs and job logic presented in Figure 9. The rounded rectangle represents a step in the pipeline. The bold text indicates the module names (Table 5) used during the step. The rectangle represents a job and the bold text above the rectangle is the name of the job.

The inventory job creates an inventory file and stores it in the S3 bucket. Inventory is a file that stores metadata about sensors. This job allows the collection of information about all sensors to provide quick access to the name of the sensor, along with coordinates if defined for the sensor. This approach allows for the elimination of unnecessary requests to Cumulocity and acts as a caching mechanism for the data related to the sensor information. This job should be run only once to create an inventory file that will be reused across jobs during the general integration run. It may be necessary to rerun this job as new sensors are added to the Cumulocity.

The general integration job retrieves data from Cumulocity using the **http_get** module and extracts the necessary fields from the received JSON with the **json_mapper** module. Duplicates are removed using the **dict.remove_duplicates** module. Sensor names and coordinates from the inventory file created during the first job are injected using the **s3.injection** module. The retrieved data is stored in the S3 bucket in CSV format for debugging purposes, using the **write_to_csv** module.

If the user defines parameters for aggregation, aggregation is performed to decrease the amount of data before sending it to ArcGIS. Aggregation performed by the **ld.date_aggregator** module. The validation step is optional as well, which can be defined by the user in the configuration file. It can check for missing values and check if the pandas DataFrame has the expected number of rows. The **ld.nan_check** module is responsible for the validation step. Finally, the data is converted to ArcGIS format by using the **transform_to_arcgis_format** module and sent to the ArcGIS server by using the **send_to_arcgis** module. Upon successful pipeline execution, the data appears in ArcGIS.

4.5 Reusable template

The general integration provides a straightforward solution for transferring measurements from Cumulocity to ArcGIS. This solution can be improved by using Jinja support for configuration, allowing the integration fields to be parameterized and optional steps to be included or excluded using Jinja conditions. This makes it possible to reuse existing pipelines for similar use cases. Once the Jinja template is created, it can be directly used in the YAML file by filling

in the required variables. This allows for the creation of a new, similar pipeline. The use of the template is shown in Figure 10.

```
- template: jobs/general_template.yaml
  prefix: _eco
  vars:
    job_name: eco_counters
    partition:
      type: daily
      start: 2025-01-01
      end: 2025-02-19
      elements: ["BicycleIn", "BicycleOut", "PedestrianIn", "PedestrianOut"]
    elements_type: valueFragmentType
    minio_folder: eco_counters
    arcgis:
      layer_name: integratsiooni_näide_(tavaline_andmekonveier)
      sublayer_name: EcoCounters
```

Figure 10. Example of template usage in the main.yaml file.

To reuse this template, the user must create a new template block and fill the vars parameters. In this example, a new job named `eco_counters` has been created. Additionally, a partition with start date 2025-01-01 and end date 2025-02-19 has been defined with type `daily`. This means that each partition corresponds to exactly one day. If only the start date is defined, partitions are automatically created up to the current day, and a new partition is automatically added each following day. This can be used for scheduled jobs.

Elements are static partition values. This concept was discussed in more detail in section 3.4.4. As a result, partitions are created in the following order:

- 2025-01-01 | BicycleIn
- 2025-01-01 | BicycleOut
- 2025-01-01 | PedestrianIn
- 2025-01-01 | PedestrianOut
-
- 2025-02-19 | PedestrianOut

The Minio folder parameter allows specifying where CSV artifacts for debugging are stored. The ArcGIS block is used to configure the destination for the data in ArcGIS. It requires a layer name and a sublayer name.

A table that was created in ArcGIS after integration is presented in Table 6. Additionally, a map of geospatial data is presented in Figure 11.

Table 6. Example of a general integration of an ArcGIS table

Source	Value	FragmentType	Date
Eco counter name	4	BicycleIn	2024-01-01
Eco counter name	3	BicycleOut	2024-01-01
Eco counter name	2	PredestrianIn	2024-01-01
Eco counter name	1	PredestrianOut	2024-01-01



Figure 11. Example of the map created after the eco_counters job

4.6 Results

There were no specific requirements regarding which data should be transferred from Cumulocity to ArcGIS. The author randomly selected datasets from the Cumulocity platform, using the groupings available in the Cumulocity UI and the fragment exploring tool created by Kaspar Kadalipp [18].

It should be noted that the author was not allowed to move a lot of data to ArcGIS. The main reason for this was that many Cumulocity datasets had already been moved to ArcGIS.

Therefore, a small portion of the selected datasets was used as a proof of concept to demonstrate how the integration can work with different datasets.

Table 7 provides an overview of all Cumulocity datasets used in the integration process. It includes a brief description of each dataset along with metadata regarding the number of objects generated per day. Figure 12 displays the locations of all sensors with geospatial information from these datasets in ArcGIS.

Table 7. Overview of used datasets

Dataset name	Description	Objects per day
EcoCounters	Sensors counting pedestrians and bicycles. Four fragments used, each with 768 objects.	~3,072 objects
Telia Cells	Counts of clients connected to specific cell towers (CELL fragment).	~8,837 objects
Rattaringlus	Information on bike-sharing services, such as total members and active users. 9 sources in total.	24 objects per source
Lights	State of city lights in Tartu (Light fragment).	~22,000 objects
Rattaringlus parking slots	Information on Rattaringlus dock stations. 90 sources in total.	~580 objects per source
Buildings	Energy consumption data for buildings. 24 sensors in total.	24 objects per source
AVC sensors	Car traffic counting sensors.	~30,000+ objects

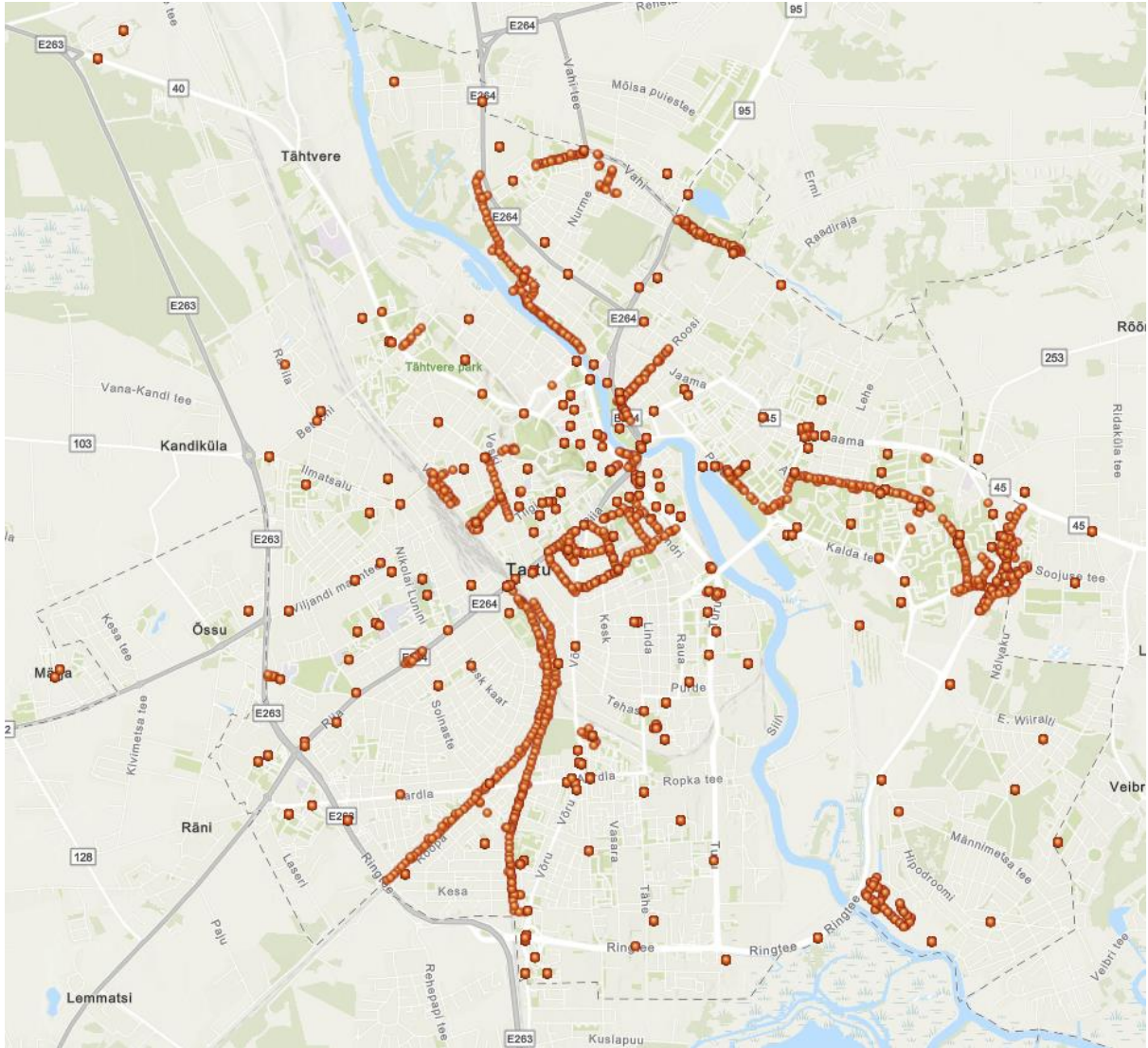


Figure 12. Visualization of all sensors after multiple integrations

The general integration covers all functional and non-functional requirements previously defined in section 3.1. FRs and NFRs that are not covered in this section were already satisfied in the Dagster justification chapter 3.3. Table 8 outlines how both functional and non-functional requirements were met

Table 8. Overview of implemented requirements

Requirements	Description
FR1	The integration removes duplicate entries from pandas DataFrames and supports removal of time series data with constant values over long periods.
FR2	Integration can validate if there are missing columns, values. It is possible to define a pandas DataFrame size for validation to ensure that a panda

	DataFrame does not exceed the required value. Validation allows to detect anomalies in the integration.
FR3	Data is transformed into an ArcGIS-compatible format before transfer.
FR5, FR6	The current solution provides several scheduling options. The first is a simple schedule that uses the CRON scheduler to run the job in a given time frame. Another option is to define a schedule for a partitioned job. In this case, the schedule is triggered based on partition date values. This option is used to automatically move data from the previous date.
FR8	The solution comes with an additional package that was created exactly for this functional requirement. The custom workflow solution provides additional instruments to create reusable modules and define them in the YAML configuration. This option allows to treat each asset in Dagster as a separate module and to define its execution order and dependencies directly in the configuration file with all required parameters.
FR4	The solution supports data aggregation and can be defined by the user in the template.
FR7	Two reusable modules are implemented: one for sending GET requests with multiple authentication methods (Basic, Bearer Token, Parameter-based) and another for data extraction using JMESPath.
FR12	A dedicated job creates an inventory file and a module for injecting inventory data is integrated into the general pipeline.
NFR2	All sensitive information related to data integration is stored in the .env file.

5. Integration verification

This part of the thesis provides an overview of validation procedures. The created solution was validated for its reusability with different APIs. System resource usage validation was performed against 3 different virtual machines (VMs) with their resource configuration. Additionally, the data quality of the integration was evaluated.

5.1 Validation against other APIs

The reusability validation is performed against 4 APIs:

- QuestDB
- OpenWeather
- Cumulocity events
- Reaalajaandmed (New API developed to replace Cumulocity)

QuestDB was selected because a new alternative to Cumuqlocity is using QuestDB. It is crucial to understand how the created solution can work with the new environment, and how many changes should be made to make it fully compatible with the new solution.

OpenWeather API was chosen for the validation because it has some common patterns with IoT devices. API returns data related to the weather, such as wind, temperature, pressure, condition, and many more.

Cumulocity events have different structures from measurement, therefore validating against this endpoint can demonstrate how well reusability where implemented. Mainly Ridango bus validation events will be used. In addition, validation is performed against a newly developed API. This API is intended to replace Cumulocity.

QuestDB

Validation against QuestDB API²⁸ was not successful due to its use of a non-standard JSON response format. The response includes a list of column names and a separate list of lists containing the values (Figure 13). To address this, a dedicated module (**questdb.api**) was developed to convert the QuestDB API response into a pandas DataFrame.

²⁸ QuestDB REST API <https://questdb.com/docs/reference/api/rest/>

```
{
  "query": "SELECT timestamp, tempF FROM weather LIMIT 2;",
  "columns": [
    {
      "name": "timestamp",
      "type": "TIMESTAMP"
    },
    {
      "name": "tempF",
      "type": "INT"
    }
  ],
  "timestamp": 0
  "dataset": [
    ["2010-01-01T00:00:00.000000Z", 34],
    ["2010-01-01T00:51:00.000000Z", 34]
  ],
  "count": 2
}
```

Figure 13. QuestDB API response [52]

```
[
  {
    "columns": [
      {
        "name": "timestamp",
        "type": "TIMESTAMP"
      },
      {
        "name": "tempF",
        "type": "INT"
      }
    ],
    "dataset": [
      ["2010-01-01T00:00:00.000000Z", 34],
      ["2010-01-01T00:51:00.000000Z", 34]
    ]
  }
]
```

Figure 14. Mapped QustDB API response

By default, data is retrieved using the **http_get** module and then processed with the **json_mapper** module. However, this approach leads to issues when working with the QuestDB API. As shown in Figure 14, the mapped output contains keys such as “dataset” and “columns,” which are incompatible with the expected format. By default, the general integration solution expects the **json_mapper** module to return a list of flat objects, where each object contains only simple key–value pairs (e.g., strings, numbers, or booleans), and no nested lists or

dictionaries. This results in a data type problem that was not considered in the initial implementation.

To address the issues with the previous two modules, a dedicated **questdb.api** module was developed. It sends a GET request to the QuestDB API and transforms the response directly into a pandas DataFrame. The pandas framework can convert a list of column names and a list of value lists, where each sublist represents a row, into a pandas DataFrame.

OpenWeather API

The validation of OpenWeather integration was nearly successful. The problem was that the timestamp format was different from that of ArcGIS. An additional module (**date.timestamp**) was implemented to add a timestamp to the OpenWeather API response. A partial API response body from OpenWeather is shown in Figure 15.

```
{
  "coord": {
    "lon": 26.7251,
    "lat": 58.3806
  },
  "weather": [
    {
      "id": 804,
      "main": "Clouds",
      "description": "overcast clouds",
      "icon": "04d"
    }
  ],
  "base": "stations",
  "main": {
    "temp": 5.08,
    "feels_like": 2.2,
    "temp_min": 4.61,
    "temp_max": 6.14,
    "pressure": 1016,
    "humidity": 42,
    "sea_level": 1016,
    "grnd_level": 1009
  }
}
```

Figure 15. OpenWeather API response

```

- asset: open_weather_json
  module: json_mapper
  group: open_weather
  ins: tartu_weather
  params:
    mappings:
      lon: coord.lon
      lat: coord.lat
      weather_condition: weather[0].main
      wind_speed: wind.speed
      temp: main.temp
      pressure: main.pressure
      humidity: main.humidity
      city: name

```

Figure 16. OpenWeather API mapping block

The response can be mapped without any problems. The mapping process is demonstrated in Figure 16. The **date.timestamp** module is used in later integration steps to add the current timestamp to the mapped object, which is shown in Figure 17.

```

[ {
  "lon": "26.7251",
  "lat": "58.3806",
  "weather_condition": "Clouds",
  "wind_speed": "3.6",
  "temp": "5.08",
  "pressure": "1016",
  "humidity": "42",
  "city": "Tartu"
} ]

```

Figure 17. Mapped OpenWeather API response

Cumulocity events

The validation of Ridango events was successful, as it used a similar structure and timestamp format. Only the GET requests and JSON mappings were updated in the configuration file. The Cumulocity events API returns a response similar to Cumulocity measurements, containing a list of event objects. The structure of the first object in this list is shown in Figure 18.

```

"events": [
  {
    "creationTime": "2025-01-20T05:00:13.677+02:00",
    "source": {
      "name": "Route 690667 (Bus 4)",
      "self": "http://tartu.platvorm.iot.telia.ee/inventory/managedObjects/161691521185",
      "id": "161691521185"
    },
    "type": "com_ridango_trip_arrival",
    "lastUpdated": "2025-01-20T05:00:13.677+02:00",
    "self": "http://tartu.platvorm.iot.telia.ee/event/events/1691522066",
    "time": "2025-01-19T21:52:00.000Z",
    "id": "1691522066",
    "text": "Trip arrival",
    "com_ridango_trip_data": {
      "trip_short_name": "",
      "trip_id": "14856346",
      "direction_id": "0",
      "route_id": "690667",
      "arrival_time": "23:52:00",
      "turnr": "4_Ringtee - Kummeli_A>B_c288d781dfe3c8a12c40b6cc943bd6a7|1761068",
      "departure_time": "23:30:00"
    }
  }
]

```

Figure 18. Cumulocity events API response

Figure 19 demonstrates how the objects were mapped using the reusable **json_mapper** module.

```

- asset: event_extract
  module: json_mapper
  ins: get_event
  params:
    mappings:
      id: events[*].source.id
      name: events[*].source.name
      type: events[*].type
      time: events[*].time
      trip_id: events[*].com_ridango_trip_data.trip_id
      direction_id: events[*].com_ridango_trip_data.direction_id
      route_id: events[*].com_ridango_trip_data.route_id
      arrival_time: events[*].com_ridango_trip_data.arrival_time
      departure_time: events[*].com_ridango_trip_data.departure_time
      turnr: events[*].com_ridango_trip_data.turnr

```

Figure 19. Cumulocity events mapping block

After the mapping, a list of mapped events is created. An example of a mapped event is shown in Figure 20. The example contains only one object for demonstration purposes, but in reality, there are more event objects.

```
[
  {
    "id": "161691521185",
    "name": "Route 690667 (Bus 4)",
    "type": "com_ridango_trip_arrival",
    "time": "2025-01-19T21:52:00.000Z",
    "trip_id": "14856346",
    "direction_id": "0",
    "route_id": "690667",
    "arrival_time": "23:52:00",
    "departure_time": "23:30:00",
    "turnr": "4_Ringtee - Kummeli_A>B_c288d781dfe3c8a12c40b6cc943bd6a7|1761068"
  },
]
```

Figure 20. Mapped Cumulocity events API response

Reaalajaandmed

The new API for communicating with IoT databases was developed by Maadis Jaagup Laurson [53] as part of his thesis. This API has some common attributes with the previously used API, but still, it is not an exact copy. The API requires an API key for authorization. Exactly, this feature was not implemented for the **http_get** module. This feature was added to support a new authentication method.

To make a new API work with the general integration (4.4), only two asset configurations needed to be changed. All other parts of the configuration were left untouched.

The new API has different parameters that were updated accordingly in the new configuration file. In total, 3 API parameters, endpoint, and authentication method were updated in the first module responsible for GET requests. The differences between configurations for **http_get** module are presented in Figure 21 and Figure 22.

```

- asset: get_cumu_data
  group: *group
  module: http_get
  params:
    endpoint: https://tartu.platvorm.iot.telia.ee/measurement/measurements
    auth:
      basic_auth:
        username: "{{ env.CUMO_USERNAME }}"
        password: "{{ env.CUMO_PASSWORD }}"
    partition_mapping:
      elements: valueFragmentType #source
      dates:
        - dateFrom
        - dateTo
    params:
      pageSize: "{{page_size | default(2000)}}"
    check_more:
      condition: "{{page_size | default(2000)}} == len(json_data['measurements'])"
      parameter: currentPage

```

Figure 21. Cumulocity **http_get** module configuration

```

- asset: get_cumu_data
  group: *group
  module: http_get
  params:
    endpoint: https://reaalajaandmed.tartu.ee/measurement
    auth:
      api_key:
        key: "{{ env.RA_API }}"
        key_name: "api_key"
    partition_mapping:
      elements: series_type
      dates:
        - date_time_start
        - date_time_end
    params:
      page_size: "1000"
    check_more:
      condition: "1000 == len(json_data['measurements'])"
      parameter: current_page #url param for pages

```

Figure 22. Reaalajaandmed **http_get** module configuration

Additionally, since the JSON response format was different, the JSON mappings were updated accordingly. The differences between configurations for the **json_mapper** module are presented in Figure 23 and Figure 24.

```
- asset: json_mapper
  group: *group
  ins: get_cumu_data
  module: json_mapper
  params:
    mappings:
      source_id: measurements[*].source.id
      value: measurements[*].*.*.value | [[]]
      unit: measurements[*].*.*.unit | [[]]
      time: measurements[*].time
      dal_series: measurements[*].keys(@)[-1]
```

Figure 23. Cumulocity **json_mapper** module configuration

```
- asset: json_mapper
  group: *group
  ins: get_cumu_data
  module: json_mapper
  params:
    mappings:
      source_id: measurements[*].device_identity
      value_: measurements[*].series[0].value
      unit: measurements[*].series[0].unit
      time_: measurements[*].time
      dal_series: measurements[*].series[0].series_type
```

Figure 24. Reaalajaandmed **json_mapper** module configuration

A comparison of the two configurations for the **json_mapper** module shows that Cumulocity has a significantly more complex structure than the Reaalajaandmed API, where the same values can be accessed without using advanced JMESPath queries.

Similar changes were made to the inventory job. As a result, only the first two assets in the configuration need to be updated to begin using the general integration solution with the new API.

This integration continued to function after implementation, uploading the previous day's data to ArcGIS. The integration was functional from 2025-04-25 to 2025-05-15. The integration used the following fragments:

- pedestrian-out
- pedestrian-in
- bicycle-out
- bicycle-in
- truck-in
- truck-out
- car-in
- car-out
- bus-in
- bus-out

The only problem that happened with this integration is the expiration of the ArcGIS OAuth2 token, which had to be manually updated on the running server after 14 days.

5.2 Data integrity validation

The goal of this validation is to ensure that no data is lost during the integration process. The AVC dataset from Cumulocity is used for data integrity validation. A partition is moved directly to ArcGIS. This allows for comparing the moved partition with the actual results in the ArcGIS table. The expected result is that Cumulocity and ArcGIS will have the same number of objects for a given date. If the amount of data does not match, it may indicate that the solution should be improved in the future.

To validate the data integrity of the created pipeline, the loaded data in ArcGIS was compared to the initial data provided by Cumulocity. The first test uses the AVC job and partition with a date of 01.01.2025.

As a result, a total of 33209 records were inserted into ArcGIS, matching the number of records retrieved from Cumulocity. Validation was performed using the Cumulocity API directly. The following parameters were used:

- pageSize = 50
- dateFrom = 2025-01-01T00:00:00Z
- dateTo = 2025-01-02T00:00:00Z

The total number of records was determined by iterating through the API pages. With a page size of 50, the API returned 664 full pages and one additional page containing the remaining 9 records, resulting in a total of 33209 records. This confirmed that the data in ArcGIS matched the original dataset from Cumulocity.

The second test used the partition from 04.01.2025 and included aggregation. Aggregation was configured to group records by one-hour intervals for each source. The AVC dataset consists of data from 24 different sensors. Given 24 hours in a day, the expected result was 576 aggregated records. ArcGIS showed exactly 576 records after the aggregation, confirming the expected number. However, this test focused only on the quantity of records and did not validate the correctness of the aggregation logic itself.

5.3 Integration resource consumption

Understanding the resource consumption of the solution is crucial. It provides information on how well it works on low-tier hardware and provides insights into what hardware to use. This information helps to allocate a sufficient amount of resources to the system and identify problems with the solution.

VMs will be created in the Proxmox²⁹ environment. The operating system for VMs is Ubuntu³⁰ 22.04.5 LTS. Proxmox instance has the following hardware specification:

- Intel Core i3-1220P (10 Cores, 12 Threads)³¹
- DDR4 64 GB
- Samsung SSD 980 EVO

²⁹ Proxmox <https://www.proxmox.com/en/>

³⁰ Ubuntu <https://ubuntu.com/>

³¹ Intel Core I3-1220P <https://www.cpubenchmark.net/cpu.php?cpu=Intel+Core+i3-1220P&id=4819>

The goal of this validation is to indicate how many resources the solution will require to run in a production environment. For Dagster, it is important to define the number of workers to ensure the efficient work of the system. The number of workers depends on the system hardware. As an example, 10 workers on low-end hardware may crush the system or perform slower than a single worker.

For the testing, Cumulocity will be used as a data source. The lights fragment is used, which on average has 10 pages and approximately 19000 objects per day. The pipeline retrieves measurements, extracts required JSON fields and saves CSV to S3. Before the testing sufficient number of workers were identified by materializing 15 partitions with different worker configurations for each VM. The test materializes 91 partitions in a row to simulate data migration for the last 3 months.

Table 9. Materialization of 91 partitions in a row for Lights fragment

Instance	Workers	Time (min:s)
2CPU Cores 4GB RAM (VM1)	4	12:25
4CPU Cores 4GB RAM (VM2)	5	9:28
8CPU Cores 8GB RAM (VM3)	15	5:28

Table 9 provides the results of moving 91 partitions in a row for each VM configuration, along with the number of workers. As expected, in this case, a higher-end **VM3** moved data about 2 times faster than a low-end **VM1**. The main reason is the number of supported workers for each VM. **VM1** can support up to 4 workers at the same time without losing performance. On the other hand, a **VM3** can support 15 workers at the same time. The number of workers is directly related to the number of CPU cores.

It should be mentioned that the low-end VM can handle most of the integration tasks where speed is not critical. For example, **VM1** can be used to move some datasets from Cumulocity to ArcGIS based on a predefined schedule.

In addition, the most resource-intensive container in all the tests was a container in which PostgreSQL was running. After the tests, this was discovered by examining the Cadvisor data in Grafana. This is because all Dagster components rely heavily on the database, and during job execution, all containers in the Dagster stack continuously read from and write to it.

5.4 Future work

The current implementation was only tested on a single VM with Docker. Since Dagster is already Kubernetes-compatible, the solution could be deployed to a Kubernetes cluster in the future to validate the solution's Kubernetes compatibility.

Currently, module creation requires some repetitive definitions from the user. It is not very critical, but still, these problems could be addressed in future work.

Additionally, created YAML workflow solution does not provide a good error description at the moment. That's why it's sometimes very difficult to understand which specific block of the YAML configuration caused an error.

Another problem that should be solved is ArcGIS token rotation. Currently, the OAuth2 token used is only valid for 14 days. After expiration, this token should be generated manually and added to the .env file. After updating the .env file, the Dagster stack should be reloaded.

Documentation is another critical issue with the current solution. Each module should be very well documented, otherwise, there is no practical value in this solution if the end-user cannot understand how to use the software.

6. Conclusion

The goal of this thesis was to create a reusable integration solution between the Cumulocity IoT database and the ArcGIS platform. Dagster data orchestration platform was chosen as the core orchestration platform for the integration after careful consideration of other similar tools.

To address the issue of reusability of data pipelines created in Dagster, a custom YAML workflow configuration solution was implemented. This solution allows treating each Dagster asset as a standalone module instead of defining non-reusable assets directly in Python code. In addition, this solution allows defining Dagster jobs, resources, sensors, partitions, and assets directly from the configuration file, simplifying the whole process of data pipeline creation and automation.

The general integration was successfully implemented as a universal solution, capable of working with any Cumulocity measurement without the need for creating a new pipeline. This approach fully met the initial integration requirements, providing a flexible and reusable solution.

Validation confirmed that the reusability approach works, allowing a new JSON-based API to be used without having to implement a new solution from scratch. It is sufficient to change the initial API requests and JSON parsing in the configuration. The templating feature demonstrated that new general data pipelines can be configured and created simply by using a template, providing reusability and flexibility in job creation. In total, 7 different data pipelines were implemented using this approach and data was successfully transferred to ArcGIS. Additionally, resource consumption was validated, confirming that the integration can function effectively on low-end hardware if the number of workers meets the requirements for the integration use case.

One of the risks of the current solution is the rapid development of Dagster. Many things may change in the future, making this solution irrelevant without continuous support. Considering this, it is obvious that this project should be actively maintained to stay relevant and to keep up with new Dagster features and implement missing features.

References

- [1] ‘Statistika juhtimislauad’. Accessed: Mar. 18, 2025. [Online]. Available: <https://juhtimislauad.stat.ee/et/piirkondlik-statistika-3/tallinn-17>
- [2] H. Ritchie, V. Samborska, and M. Roser, ‘Urbanization’, *Our World in Data*, Feb. 2024, Accessed: Mar. 18, 2025. [Online]. Available: <https://ourworldindata.org/urbanization>
- [3] ‘SmartEnCity Site Tartu | Smart Cities Marketplace’. Accessed: Mar. 18, 2025. [Online]. Available: <https://smart-cities-marketplace.ec.europa.eu/projects-and-sites/projects/smartencity/smartencity-site-tartu>
- [4] ‘Cisco Annual Internet Report - Cisco Annual Internet Report (2018–2023) White Paper - Cisco’. Accessed: Mar. 18, 2025. [Online]. Available: <https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html>
- [5] Z.-G. Chen, Z.-H. Zhan, S. Kwong, and J. Zhang, ‘Evolutionary Computation for Intelligent Transportation in Smart Cities: A Survey [Review Article]’, *IEEE Computational Intelligence Magazine*, vol. 17, no. 2, pp. 83–102, May 2022, doi: 10.1109/MCI.2022.3155330.
- [6] A. Camero and E. Alba, ‘Smart City and information technology: A review’, *cities*, vol. 93, pp. 84–94, 2019.
- [7] ‘Liikuvuse loendurid’. Accessed: Mar. 20, 2025. [Online]. Available: <https://tartu.maps.arcgis.com/apps/dashboards/22fbbfbddda04241921458f6cb04cb3a>
- [8] ‘City Solutions / SmartEnCity.eu’. Accessed: Mar. 18, 2025. [Online]. Available: <https://smartencity.eu/outcomes/city-solutions-best-practices/>
- [9] ‘Kavandatud tegevused – Tark Tartu’. Accessed: Mar. 18, 2025. [Online]. Available: <https://tarktartu.ee/avaleht/kavandatud-tegevused/>
- [10] L. Cai and Y. Zhu, ‘The challenges of data quality and data quality assessment in the big data era’, *Data science journal*, vol. 14, pp. 2–2, 2015.
- [11] H. Foidl, V. Golendukhina, R. Ramler, and M. Felderer, ‘Data pipeline quality: Influencing factors, root causes of data-related issues, and processing problem areas for developers’, *Journal of Systems and Software*, vol. 207, p. 111855, 2024.
- [12] A. Nambiar and D. Mundra, ‘An overview of data warehouse and data lake in modern enterprise data management’, *Big data and cognitive computing*, vol. 6, no. 4, p. 132, 2022.
- [13] M. Matskin *et al.*, ‘A Survey of Big Data Pipeline Orchestration Tools from the Perspective of the DataCloud Project.’, in *DAMDID/RCDL (Supplementary Proceedings)*, 2021, pp. 63–78.
- [14] I. A. Machado, C. Costa, and M. Y. Santos, ‘Data mesh: concepts and principles of a paradigm shift in data architectures’, *Procedia Computer Science*, vol. 196, pp. 263–271, 2022.
- [15] R. Cavicchioli, R. Martoglia, and M. Verucchi, ‘A novel real-time edge-cloud big data management and analytics framework for smart cities’, *Journal of Universal Computer Science*, vol. 28, no. 1, pp. 3–26, 2022.
- [16] V. Sresth, S. P. Nagavalli, and S. Tiwari, ‘Optimizing Data Pipelines in Advanced Cloud Computing: Innovative Approaches to Large-Scale Data Processing, Analytics, and Real-Time Optimization’, *INTERNATIONAL JOURNAL OF RESEARCH AND ANALYTICAL REVIEWS*, vol. 10, pp. 478–496, 2023.
- [17] ‘Introduction to the Cumulocity platform - Cumulocity documentation’. Accessed: Mar. 20, 2025. [Online]. Available: <https://cumulocity.com/docs/>

- [18] K. Kadalipp, 'Knowledge Graphs for Cataloging and Making Sense of Smart City Data'. [Online]. Available: <https://thesis.cs.ut.ee/dedf24ea-13c0-40ad-916f-e6dfc57a1d52>
- [19] 'Cumulocity - OpenAPI'. Accessed: Mar. 20, 2025. [Online]. Available: <https://cumulocity.com/api/core/>
- [20] S. Laius, 'Integration library for analyzing historical data from the Cumulocity IoT platform in Python', UNIVERSITY OF TARTU, 2020. [Online]. Available: <https://thesis.cs.ut.ee/c9bcbf22-5114-4c63-9060-55f2548336b7>
- [21] Ashalyengar21, 'Deprecation of Basic authentication in Exchange Online'. Accessed: Mar. 20, 2025. [Online]. Available: <https://learn.microsoft.com/en-us/exchange/clients-and-mobile-in-exchange-online/deprecation-of-basic-authentication-exchange-online>
- [22] 'Web GIS Mapping Software | Create Web Maps with ArcGIS Online'. Accessed: Mar. 18, 2025. [Online]. Available: <https://www.esri.com/en-us/arcgis/products/arcgis-online/overview>
- [23] 'Feature layers—ArcGIS Online Help | Documentation'. Accessed: May 01, 2025. [Online]. Available: <https://doc.arcgis.com/en/arcgis-online/reference/feature-layers.htm>
- [24] 'Build powerful apps with ArcGIS services', Esri Developer REST APIs documentation. Accessed: Mar. 18, 2025. [Online]. Available: <https://developers.arcgis.com/rest/>
- [25] 'ArcGIS field data types—ArcGIS Pro | Documentation'. Accessed: Mar. 18, 2025. [Online]. Available: <https://pro.arcgis.com/en/pro-app/3.3/help/data/geodatabases/overview/arcgis-field-data-types.htm>
- [26] 'GIS Dictionary'. Accessed: Mar. 18, 2025. [Online]. Available: <https://support.esri.com/en-us/gis-dictionary>
- [27] 'Where is Tartu, Eesti on Map Lat Long Coordinates'. Accessed: Mar. 18, 2025. [Online]. Available: <https://www.latlong.net/place/tartu-eesti-16353.html>
- [28] 'Spatial references | Documentation | Esri Developer', Documentation. Accessed: Mar. 20, 2025. [Online]. Available: <https://developers.arcgis.com/documentation/spatial-references/>
- [29] 'Specify the maximum token expiration time—Portal for ArcGIS | Documentation for ArcGIS Enterprise'. Accessed: Mar. 20, 2025. [Online]. Available: <https://enterprise.arcgis.com/en/portal/latest/administer/windows/specify-the-default-token-expiration-time.htm>
- [30] 'Introduction to API key authentication | Documentation | Esri Developer', Documentation. Accessed: Mar. 20, 2025. [Online]. Available: <https://developers.arcgis.com/documentation/security-and-authentication/api-key-authentication/>
- [31] 'What is OAuth 2.0 and what does it do for you?', Auth0. Accessed: Mar. 20, 2025. [Online]. Available: <https://auth0.com/intro-to-iam/what-is-oauth-2>
- [32] 'ICS Theses Register'. Accessed: Mar. 18, 2025. [Online]. Available: <https://thesis.cs.ut.ee/>
- [33] M. Kahu, 'Solution for Sharing Open Data from Cumulocity Iot Platform', UNIVERSITY OF TARTU, 2023.
- [34] A. Grukhal, 'Data acquisition and preparation toolbox for Cumulocity-based solutions', UNIVERSITY OF TARTU, 2023. [Online]. Available: https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=77934
- [35] 'Apache NiFi', Apache NiFi. Accessed: Mar. 19, 2025. [Online]. Available: <https://nifi.apache.org/>
- [36] 'Apache NiFi User Guide'. Accessed: Mar. 25, 2025. [Online]. Available: <https://nifi.apache.org/docs/nifi-docs/html/user-guide.html>

- [37] J. Praks, ‘Anomaly Detection and Imputation for Tartu Traffic Sensors’, UNIVERSITY OF TARTU, Tartu. [Online]. Available: <https://thesis.cs.ut.ee/43c4d896-e4e1-498b-946e-215467f91866>
- [38] ‘What is Airflow®? — Airflow Documentation’. Accessed: Mar. 18, 2025. [Online]. Available: <https://airflow.apache.org/docs/apache-airflow/stable/index.html>
- [39] ‘Dagster instance | Dagster Docs’. Accessed: Mar. 18, 2025. [Online]. Available: <https://docs.dagster.io/guides/deploy/dagster-instance-configuration>
- [40] ‘Pythonic, Modern Workflow Orchestration For Resilient Data Platforms | Prefect’. Accessed: Mar. 18, 2025. [Online]. Available: <https://www.prefect.io/>
- [41] *spotify/luigi*. (Mar. 20, 2025). Python. Spotify. Accessed: Mar. 20, 2025. [Online]. Available: <https://github.com/spotify/luigi>
- [42] ‘Metaflow - a framework for real-life ML, AI, and data science’. Accessed: Mar. 20, 2025. [Online]. Available: <http://www.metaflow.org/>
- [43] ‘Build production-grade data and ML workflows, hassle-free with Flyte’. Accessed: Mar. 18, 2025. [Online]. Available: <https://flyte.org/>
- [44] Prometheus, ‘Overview | Prometheus’. Accessed: Mar. 18, 2025. [Online]. Available: <https://prometheus.io/docs/introduction/overview/>
- [45] *google/cadvisor*. (Mar. 18, 2025). Go. Google. Accessed: Mar. 18, 2025. [Online]. Available: <https://github.com/google/cadvisor>
- [46] ‘Grafana | Query, visualize, alerting observability platform’, Grafana Labs. Accessed: Mar. 18, 2025. [Online]. Available: <https://grafana.com/grafana/>
- [47] S. T. Bhosale, T. Patil, and P. Patil, ‘Sqlite: Light database system’, *Int. J. Comput. Sci. Mob. Comput*, vol. 44, no. 4, pp. 882–885, 2015.
- [48] ‘Appropriate Uses For SQLite’. Accessed: Mar. 19, 2025. [Online]. Available: <https://www.sqlite.org/whentouse.html>
- [49] ‘Technology | 2024 Stack Overflow Developer Survey’. Accessed: Mar. 19, 2025. [Online]. Available: <https://survey.stackoverflow.co/2024/technology/#1-databases>
- [50] ‘PostgreSQL vs MySQL - Difference Between Relational Database Management Systems (RDBMS) - AWS’, Amazon Web Services, Inc. Accessed: Mar. 19, 2025. [Online]. Available: <https://aws.amazon.com/compare/the-difference-between-mysql-vs-postgresql/>
- [51] ‘What is a Container? | Docker’. Accessed: Mar. 19, 2025. [Online]. Available: <https://www.docker.com/resources/what-container/>
- [52] QuestDB, ‘REST API | QuestDB’. Accessed: May 07, 2025. [Online]. Available: <https://questdb.com/docs/reference/api/rest/>
- [53] M. J. Laurson, ‘Asjade Interneti API-de loomine targa linna andmete halduseks’, UNIVERSITY OF TARTU, 2025. [Online]. Available: <https://thesis.cs.ut.ee/>

Appendices

I. Code Repository

The solution comes as two separate projects. The first solution allows for the creation of a YAML-based data pipeline in Dagster. The second solution is where Dagster data pipelines were implemented.

YAML workflow repository: <https://github.com/lunfman/dagster-factory-pipelines>

Cumulocity and ArcGIS data pipelines repository: <https://github.com/lunfman/cumulocity-arcgis-integration>

Before starting to use the provided solution, the following prerequisites should be met:

- The user has access to ArcGIS
- The user can generate or have a valid OAuth 2.0 ArcGIS token or API key
- The user has access to the Tartu Cumulocity instance (required for Cumulocity-based data pipelines)
- The user has access to the Reaalajaandmed API (required for Reaalajaandmed API data pipelines)
- Docker and Docker Compose are installed on the personal computer/server

Installation guide

1. Git clone the Cumulocity and ArcGIS data pipelines repository
<https://github.com/lunfman/cumulocity-arcgis-integration>
2. Provide all required values to .env file.
3. Run the following command in the terminal from the project root directory

```
docker compose up --build
```

The project should start and the following endpoints should be available:

- Dagster UI URL: localhost:3000
- Minio UI URL: localhost:9001

License

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

I, Oleg Savik _____,
(author's name)

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

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

Tartu Smart City data integration between the Cumulocity IoT database and ArcGIS _____,
(title of thesis)

supervised by _____ Pelle Jakovits _____;
(supervisor's name)

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

Oleg Savik

15/05/2025