UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

A DATABASE FOR APPLIANCES REAL-LIFE ENERGY PERFORMANCE AND CONSUMPTION ASSESSMENT

Alexandre Santos Nascimento

Mestrado em Informática

Trabalho de projeto orientado por:
Prof. Doutor Pedro Miguel Frazão Fernandes Ferreira
Prof. Doutor Ana Paula Pereira Afonso

2021

Acknowledgments

First of all, I want to thank my mother and grandparents as they were a main pillar during this journey. They gave me all the support and motivation when I needed it and helped me overcome all the difficulties faced during this thesis.

Secondly, I would also want to thank the professors Dr. Pedro Miguel Ferreira and Dr. Ana Paula Afonso for their dedication, support and guidance during the completion of this thesis.

Finally, I want to thank all my friends for the friendship and support showed in the moments spent together. They helped me overcome the barriers and difficult moments, as well as being present in the best moments.

To my family and friends

Resumo

Um dos objetivos da União Europeia para 2030 é aumentar a eficiência energética em pelo menos 32.5%, tornando assim evidente que a redução do consumo energético é atualmente uma das suas das prioridades [7].

Uma das razões para a ineficiência energética atual nos edifícios é a falta de informação sobre o consumo energético dos eletrodomésticos e outros dispositivos, num cenário de utilização normal. Atualmente a obtenção do desempenho energético de um dispositivo é realizado através de previsões de consumo, que são obtidas durante a fase de design e seguidamente disponibilizadas, pelos fabricantes. No entanto, estas previsões não são suficientes para saber o consumo real, pois já foi provado que o consumo durante a utilização normal tende a ultrapassar as previsões [32].

Existe assim a necessidade de explorar soluções que disponibilizem avaliações do consumo energético num cenário real para um edifício e para os dispositivos elétricos dentro do mesmo. O projeto SATO (*Self Assessment Towards Optimization*) visa desenvolver ferramentas para autoavaliação e otimização energética de edifícios e seus equipamentos / eletrodomésticos. Este objetivo é atingido através da recolha de dados do consumo energético durante a utilização real, permitindo assim avaliações e otimização do desempenho energético.

Esta tese está integrada no projeto SATO, e tem como um dos objetivos a conceção do modelo de dados para representar os dados recolhidos através de sensores e as avaliações energéticas. Outro objetivo é a implementação das bases de dados, para organizar, armazenar e visualizar dados sobre o consumo de energia. É também possível atribuir a classe energética a cada dispositivo, tendo em conta os consumos durante a utilização real.

Para concretizar estes objetivos foi necessário compreender os requisitos para a avaliação de energia de aparelhos reais, deteção de falhas, armazenamento de dados de sensores e, seguidamente, a revisão e comparação dos sistemas de gestão de base de dados apropriados. Adicionalmente foi recolhida informação sobre a base de dados EPREL (*EU Product Register for Energy Labelling*) que armazena a informação dos rótulos energéticos dos produtos à venda no mercado europeu. Finalmente, também foi recolhida informação sobre o processo de rotulagem energética utilizado nos rótulos energéticos da União Europeia.

De modo a realizar uma análise aos diversos sistemas existentes, foi necessário estabelecer quais os requisitos para a gestão de dados de sensores. Os requisitos refletem as diferenças deste tipo de dados em comparação a outros. Alguns dos requisitos recolhidos são o suporte de um elevado e em expansão volume de dados, a possibilidade de visualização em tempo real, conexão em simultâneo de vários dispositivos e a proteção dos dados contra falhas.

Após a análise dos requisitos, foram explorados diversos sistemas de gestão de bases de dados, tais como os sistemas de gestão de séries temporais (do inglês *Time series database* (TSDB)) que são desenhados e desenvolvidos com o objetivo de serem utilizados para armazenar e realizar operações com dados de séries temporais, como é o caso dos dados provenientes dos sensores. Foi realizada uma análise aos sistemas InfluxDB, TimescaleDB, CrateDB, Prometheus e VictoriaMetrics, com base em casos de uso, documentação e artigos. Também foram estudadas algumas comparações já realizadas.

Os requisitos acima enumerados juntamente com os requisitos recolhidos de projetos anteriores, que envolviam dados provenientes de sensores, benchmarks realizados, documentação técnica e reuniões feitas com os restantes membros do projeto SATO, foram utilizados como critérios de comparação entre as várias bases de dados de modo a fazer a saber qual o sistema mais adequado. Esta comparação permitiu concluir que o sistema mais adequado para os critérios enunciados é o InfluxDB.

Como referido anteriormente, foi realizada uma análise à bases de dados EPREL na medida em que armazena a informação sobre a eficiência energética dos dispositivos a quem foi atribuído um rótulo energético da União Europeia. O propósito destes rótulos é indicar a eficiência energética dos dispositivos, permitindo assim que os consumidores façam a escolhas informadas no momento de comprar um dispositivo, levando assim a uma redução das emissões de gases com efeito de estufa. Quando um dispositivo é colocado no mercado europeu deve obrigatoriamente ser acompanhado por este rótulo.

A base de dados EPREL é constituída por uma parte pública, uma parte de verificação e um portal online. Este portal permite aceder a ambas as partes e aceder a informação de um modelo específico de um dispositivo, que foi previamente inserido pelo fornecedor. Nem todos os tipos de dispositivos estão disponíveis na EPREL, uma vez que é necessário terem o novo rótulo energético que tem algumas diferenças em comparação com o anterior. Este novo rótulo energético tem uma nova escala para a classe energética do dispositivo, um código QR que será utilizado para aceder à página web correspondente, e possivelmente métricas diferentes. Atualmente os tipos de dispositivos suportados são as máquinas de lavar louça, máquinas de lavar roupa, máquinas de lavar e secar roupa, frigoríficos e ecrãs eletrónicos.

Tendo em conta que o trabalho desenvolvido é uma das componentes da plataforma SATO, foi necessário efetuar o planeamento e desenho do sistema e fluxo de dados, de modo a compreender a interação entre os diversos componentes. A arquitetura da plataforma é composta por três blocos principais, o SATO *middleware* onde é realizada a ingestão, organização e preparação de dados, de modo a estes poderem ser consumidos pelas tarefas de processamento. O SATO *Self-Assessment framework* é o principal bloco de processamento, onde se realiza o processamento dos dados recebidos de modo a obter avaliações do desempenho energético dos edifícios. Finalmente, no bloco Self-optimization services são tomadas decisões relativas à gestão de energia. As bases de dados desenvolvidas nesta tese fazem parte da secção da *self-assessment framework*. Uma das alterações feitas para este propósito foi, a utilização da plataforma Kafka implementada para uso no projeto SATO, para o envio e receção de dados.

Chegou-se à conclusão que seria necessário existir duas bases de dados, uma para armazenar os dados dos sensores e outra as métricas e meta dados dos dispositivos. Em termos de visualização de dados, foram construídos dois dashboards, um para visualizar os consumos do dispositivo em tempo real e os cálculos necessários para obter as métricas e um segundo para visualizar a comparação entre as métricas da EPREL e as métricas obtidas dos dados de sensores.

Adicionalmente, foi desenvolvida uma API REST que permite aceder aos dados armazenados sem ter que recorrer aos dashboards e utilização de queries. Existem quatro chamadas que permitem acesso às métricas e meta dados dos dispositivos, e uma chamada para aceder aos dados provenientes do sensor de corrente na última semana.

A inserção dos dados nas bases de dados é realizada recorrendo a scripts Python, enquanto o cálculo de métricas e os dashboards são desenvolvidos utilizando Flux, a linguagem de scripts utilizada no InfluxDB. A conecção e interação entre as duas bases de dados é realizada recorrendo à biblioteca Flux SQL. Os dados da base de dados EPREL, são recolhidos recorrendo a uma API REST disponibilizada também pela EPREL, enquanto os dados dos sensores são obtidos através de um ficheiro de texto no formato CSV.

Devido a algumas limitações, como o número de dispositivos dos quais foi possível recolher dados, apenas foi possível implementar e testar o cálculo e comparação de métricas para ecrãs eletrónicos e frigoríficos. O cálculo das métricas foi implementado seguindo os procedimentos estabelecidos pelo regulamento de rotulagem da União Europeia, podendo assim serem diretamente comparadas com métricas do rótulo energético.

De modo a suportar dispositivos não presentes na EPREL, como um computador ou um micro-ondas, logo sem um rótulo energético, foram recolhidas as métricas mais comuns de todos estes tipos de dispositivos. Com esta informação é possível ter informação sobre os consumos, mas não é possível comparar com o rótulo energético.

Devido à pouca quantidade de dados provenientes dos sensores, não foi possível implementar a deteção de falhas, no entanto no relatório estão descritos quais os passos para concretizar esta funcionalidade.

Para os dispositivos para os quais foi possível realizar os cálculos das métricas e comparação, os resultados não foram os esperados devido a apresentarem um consumo

real menor do que o retratado no rótulo energético. Estes resultados podem ser justificados devido ao volume reduzido de dados ou algumas definições dos aparelhos, como o brilho do ecrã, no momento da recolha de dados.

Palavras-chave: Base de dados de séries temporais, Rótulo energético, Metricas de consumo, Dados de sensores

Abstract

Nowadays, the reduction of energy consumption is a pressing matter for the European Union and one field that remains a large part of this energy contribution is the EU buildings and the energy consuming equipment inside. The use of energy labels, in order to assess the energy efficiency on European buildings, has proven to not be enough due to the difference between prediction consumption and real life energy consumption.

The SATO platform aims to provide cloud-based self-assessment and optimization of buildings and equipment/appliances energy. It achieves this goal by collecting data about the energy performance of various devices, both legacy and new, through the use of sensors. One of the objectives of this platform is to develop the databases and its dashboards to store and visualize data on the energy consumption of building appliances. With this goal in mind, this thesis will be able to provide the users a comparison between the design predictions and the real live consumption.

This comparison is done resorting to the EU energy label metrics, that are stored on the EPREL database. This database contains the energy efficiency information about the devices covered by the Energy labelling regulation, and it was created with the goal of providing the public with this information. The sensor data metrics are obtained by applying the formulas in the regulation on energy labelling for the respective device type. Additionally, since this thesis is meant to be integrated in the SATO platform, this databases and metrics will be used in the assessment and development of optimization algorithms for devices and buildings.

The current implementation makes available two databases, one to store sensor data and another for metadata and metrics, as well as two different dashboards, one to show device real time consumption's, consumption metrics and device meta-data, and another to provide the users with a simple comparison between the expected consumption and the real live consumption.

Testing was done with two different device types, electronic displays and household refrigerators, with the expectations in mind of the real life consumption being above the displayed the EU energy label. The results ended up being different from the predicted, with the real live consumption obtained from the sensor data being the highest. This may be due to the low amount of collected data.

Keywords: Time series Databases, Energy label, Consumption metrics, Sensor data

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter presents the motivation for the SATO project and therefore this thesis, more specifically the gaps in the current solutions, for example the lack of knowledge about a device consumption. The objectives and contributions of this work are also detailed as well as the structure of the document.

1.1 Motivation

The reduction of energy consumption has become a number one priority for the European Union, as it can be seen by the European Union goals for 2030, more specifically, to increase the energy efficiency by at least 32.5% [7]. In the European Union the consumption of buildings makes up for 40% of the final energy consumption [6]. This high percentage can be explained by the fact that the buildings remain largely inefficient [13], partially due to a deficit in knowledge about the real-life energy consumption and performance of buildings and their energy consuming devices.

Currently the only way for a consumer to discover the real-life energy performance of appliances and buildings is by resorting to their design predictions, that have proven not to be a reliable source of knowledge to understand the real-life energy efficiency since they tend to exceed the design predictions by more than 100% [32]. Due to this fact, reducing the disparities between the designed and the measured energy use has become a number one priority to increase energy efficiency.

An increase in the consumer knowledge about the real energy consumption of their device or house, will help them make better decisions about what to acquire to reduce their personal energy consumption.

Due to the limitations of the solutions currently used for managing energy consumption, the opportunity appears for a new solution to provide an assessment of real-life energy use of electrical devices including the energy consuming equipment in each building. This assessment will facilitate a better management of the energy resources by the building owner or manager since they will be able to identify what areas of the building and at what time can, and maybe need to be optimized. The SATO (Self Assessment Towards Optimization of Building Energy) project [45] overcomes the limitations of current solutions in the market by collecting real-life data in order to provide cloud-based self-assessment and optimization of buildings and equipment/appliances energy.

With the SATO platform it will be possible to collect energy consumption information at the building, technical equipment and appliance level. Additionally, it will be possible to assess building energy related features, perform optimal control of both legacy and smart systems in most buildings and provide interfaces for the analysis of the assessments. Finally, it will promote the creation of independent scalable building energy resource assessment and management services [10].

This work is integrated in the SATO project by implementing the databases that will store and allow the visualization of the data on the energy consumption of building appliances. This database gives the possibility of assigning energy labels to each electrical device based on their real-life consumption and detect, preemptively, device failures. It is also possible to compare the real-life consumption with the design predictions on the EU energy labels inside the platform, obtained from the EPREL (EU Product Register for Energy Labelling) database [15].

1.2 Objectives

The main objective of this project is to design and develop a database and corresponding dashboards, to organize, store and visualize data on the energy consumption and operating environment of building appliances. This database will enable the comparison between the real-time energy performance and the design prediction and consequently give the right energy label to the appliance. Another objective is to detect appliance failures and provide, when available, reparation steps that help mitigate the problem.

To achieve these objectives the project goals are:

- Review and compare the available and appropriate database engines;
- Collect requirements for real-life appliance energy assessment and failure detection database;
- Design of the data model to represent the data collected, the real-life assessments, and failure detection models;
- Implement the database to collect sensor data and its corresponding dashboards;
- Relate real-life assessment of devices to the European Union energy labelling data in EPREL database;
- Perform testing and validation of data collection in real-life use-case scenarios;

- Development and testing of assessments and failure detection algorithms;
- Integration of the work in the SATO platform through the development of an API to access the data.

1.3 Contributions

The main contributions of this thesis are as follows:

- Study and discussion of database engines for storing sensor data, by providing a detailed list of the requirements that were used as a comparison criteria and a comparison of the database engines.
- Implementation of two databases, a time series database (Sensor DB) to store the real-life energy consumption and a relational database (Metrics and metadata device DB) to store the consumption metrics and metadata.
- Development of two different dashboards. One has the purpose of providing a detailed view of device real time consumption's, the calculations done to obtain the consumption metrics and the device meta-data. The second dashboard provides a comparison between the expected consumption metrics and the real-life consumption metrics.
- Development of an API REST to access data stored in both databases. This API makes available five separated calls, that can return the device metrics, metadata and consumption signatures from the last week.

1.4 Structure of the document

The structure of the document is organized in 6 chapters. This chapter contained the introduction, objectives and contributions. The following chapters are organized as follows:

- **Chapter 2 -** Presents the current state of the art in database engines that are suitable to store sensor data. Additionally, it also presents the EPREL database and the EU labelling process.
- **Chapter 3 -** Begins with the collection of requirements for the databases, that are then used as a comparison criterion to choose the adequate database engine for this work. Then, it is presented a detailed description of the integration of the work done in this thesis in the SATO project, the dataflow and a description of components of the solution that was developed.

- **Chapter 4 -** Presents the formulas used for the calculation of metrics and their implementation in the system. It also describes how the failure detection can be implemented. Finally, there is also the implementation of the external API.
- **Chapter 5 -** Presents a comparison between the design predictions and the real-life consumption's into two real-life use case scenarios.
- **Chapter 6 -** Presents the conclusion and the future work that can be done.

Chapter 2

Related work

This chapter presents the databases for storing sensor data, time series databases, specific database engines and the comparison of various databases. The research done on the EPREL database, and the energy labelling process is also described in this chapter.

2.1 Databases for storing sensor data

The data provided by the sensors is in the time series format, that can be defined as a sequence of data points indexed in time order. In the case of the sensors these consist of successive measurements made from the same source at regular time intervals. The main difference of this data type with regular data is that all the operations are done using time [12].

Some database requirements to store the sensor data are the large and increasing amount of data that needs to be stored, a large number of connected devices simultaneously and the possibility of adding new types of sensors, as stated in [16]. It should also support the possibility of an expansion in the future with minimal downtime. Additionally, the data should be protected against failures, so the database should support backups and restorations. These requirements are used in the comparison done in the Section 3.2, together with other ones that are more specific to this work.

There are many database engines, each one made to serve a different purpose and with their own advantages and disadvantages. Time series databases are different from other models, like relational and noSQL databases, since they are designed and developed with the goal of providing the best performance and experience when working with time series data [36].

According to [16], to obtain the best performance, when performing tasks that involve handling metrics, events or measurements that are time-stamped, time series databases have some specific characteristics, like the association of a timestamp to each data point, to signal the inserted time, and the automatic sorting of data points by this timestamp. The data in this type of database is primarily inserted, usually there are no updates and the deleting of records are mainly done in bulk data. The automatic sorting allows a better performance when inserting the data at the end, with the latest timestamp, but worst when doing inserts of data with random and dispersed timestamps since the data needs to be sorted again. The operations done with time series type data tend to be over a range of values, with timestamps close to each other, instead of dispersed data points with distant timestamps.

2.1.1 InfluxDB

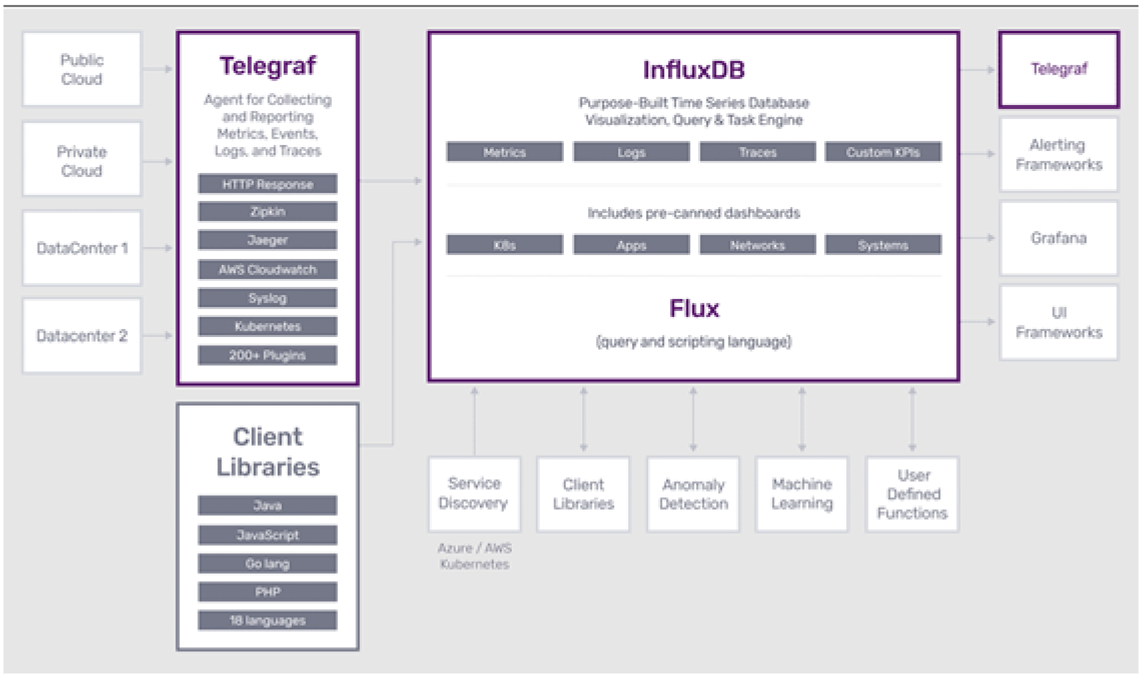


Figure 2.1: InfluxDB diagram [24]

InfluxDB is currently the most popular time series database according to DB-Engines Rankings [36]. It is an open source schemaless time series database that has its own query language called Flux. Since it is a schemaless database, it provides great flexibility and scalability which is essential when working with sensor data. It can also be considered as the essential time series toolkit since it makes available dashboards, querying, tasks and agents in one single platform, as it can be seen in Figure 2.1.

All the data is stored in a bucket, that combines the concept of a database with a retention period. Inside this bucket, the data is organized in data points that have multiple columns. The “_time” column stores the timestamp, the “_measurement” has the measurement name, the “_field” has the field key and “_value” the value of the associated field. The data points structure can also contain tags, that differentiate themselves from fields since they are indexed. This means that queries on tags are faster than queries on fields, so they are ideal for storing metadata that is often queried. Additionally, there are also series keys that are a collection of points that share a measurement, a tag set and a field set. The collection of timestamps and field values for a series key is called a serie.

Since performance is important, especially when working with large volumes of data that are provided by the sensors, it became a main goal when making design decisions for InfluxDB. Some of these decisions are storing data in a time-ascending order, since most of the operations are done over data aggregated by time, and restricting both update and delete permissions, since time series data is predominantly new data that is not updated. Another design decision that improves the write performance is the assumption that data sent multiple data is duplicate data, therefore if a new field value is submitted for a point, this point will be updated with the most recent value [22].

There are many ways to insert data into InfluxDB, namely using Telegraf. Telegraf is a plugin-driven agent that can used to collect and send metrics, events or logs to InfluxDB. Another option for inserting data is to use one of the client libraries that exist for multiple programming languages like Python, Java and JavaScript.

A visualization component is also included in InfluxDB which allows the development of web-based dashboards to visualize the stored data. Each dashboard can have multiple cells, each showing the result of a query. Each cell can show the data using multiple visualization types like a graph, histogram, heatmap, a single value or a table. Additionally, with the use of dashboard variables, it is possible to easily alter a query without the need to edit the code, therefore simplifying the interaction with the dashboard and exploration of data by the user.

In terms of security options, it is possible to set up access control using authentication tokens, that allow the management of permissions for the different organizations. In the context of InfluxDB, an organization is a workspace for a group of users [22].

Unlike some of the other time-series database engines, InfluxDB developed its own querying language called Flux. It was developed with the goal of being a functional data scripting language that can be used for querying, analysing and acting on data. One major advantage of Flux is the support for multiple data sources, for example, it is possible to retrieve data from a InfluxDB bucket and then join it with data stored on a relational database. InfluxDB also makes available a task engine that can be used to process and analyse data. This is done using tasks, that are basically Flux queries that can be scheduled, for example, running every week.

Using the tools available in Influx, it is also possible to monitor the data stored and alert accordingly. To implement this feature, a check is used to read the data and assign a status code, with the possibilities being crit, info, warn or ok, that is then stored and possibly sent to a user either by a third-party service or email.

2.1.2 TimescaleDB

TimescaleDB is an open-source relational time-series database that is built as an extension to PostgreSQL [50]. Therefore the query language used is SQL and it is also compatible with the current PostgreSQL ecosystem and tooling. It provides the full power and usability of traditional relational databases while having a single node architecture. The way TimescaleDB organizes the data is by storing the data in individual tables, called chunks, that are exposed to the user as a single table.

Some design decisions made specifically for time-series data are:

- Automated time partitioning, where the tables are automatically and continuously partitioned into smaller intervals in order to improve performance and unlock data-management capabilities.

- Native columnar compression, providing a datatype specific compression allowing for storage reduction and faster queries to compressed data.

- Automated time-series data management features, providing data retention policies, data reordering policies, aggregation and compression policies and downsampling policies.

- In-database job-scheduling framework, supporting native policies and user-defined actions

2.1.3 CrateDB

CrateDB is an open-source distributed SQL database with a dynamic schema and that allows for simple scaling and high data availability [9]. The architecture combines the familiarity of SQL with the scalability and data flexibility of NoSQL and was built for Industrial internet of things workloads that work with mixed data types and frequent querying and ingestion of data.

This database engine automatically deals with distributes sharding, partitioning, replication, and indexing. It also spreads data processing across a cluster of identical, shared-nothing nodes. Additionally, it offers real-time performance receiving and querying high volumes of complex data in real-time, via parallel processing, in-memory columnar indexes.

2.1.4 Prometheus

Prometheus is an open-source system monitoring and alerting application that provides no reliance on distributed storage since it follows a single node architecture [39]. The query language used is PromQL and is supported by Grafana, that provides the visualization.

Despite being built with the main goal of being used system monitoring, there are multiple cases of it being used for IoT.

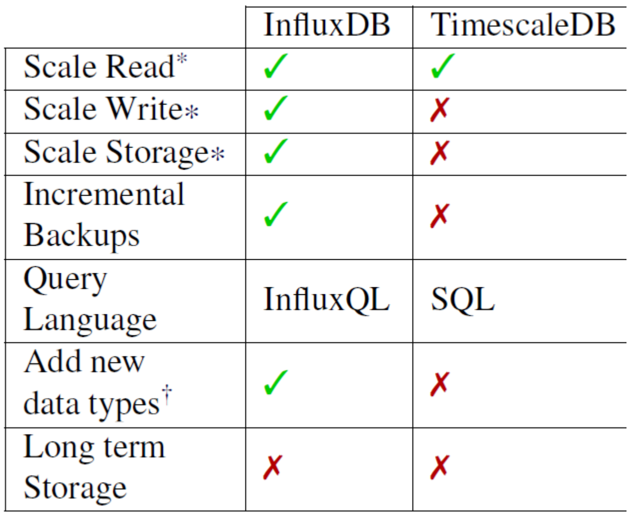
2.1.5 VictoriaMetrics

VictoriaMectrics is an open-source monitoring solution that was designed based on the principles of being fast, cost-effective and scalable [51]. This solution can also be used as a long-term storage for Prometheus since it provides the scalability that is missing in Prometheus. It provides both single node and cluster versions.

2.1.6 Database comparison

In [16], multiple databases are compared using information available in research papers and documentation with the final objective of finding the best solution for storing and handling sensor data. The comparison was done based on the properties of scalability, backups, maintenance, support for new types, query language and long-term storage. The table with the comparison results can be seen in Figure 2.2.

The criteria used are the scalability due to the increasing amount of data, the possibility of backups to protect the data against failures, the query language to know how to access the data, the possibility of adding new data types is needed due to the need to store data from new sensor types. Finally, the long term storage is needed to keep historical data in the storage [16].



	InfluxDB	TimescaleDB
Scale Read*	✓	✓
Scale Write*	✓	✗
Scale Storage*	✓	✗
Incremental Backups	✓	✗
Query Language	InfluxQL	SQL
Add new data types†	✓	✗
Long term Storage	✗	✗

Figure 2.2: Database for sensor data comparison table [16]

The author of [16], reached the conclusion that InfluxDB was the best choice.

In the article [48], a tool called SimpleMetric was developed for measuring the performance of time series databases. The authors then performed a comparative study, using this tool, between InfluxDB [24], KDB+ [30], Graphite [21], TimescaleDB [49], KairosDB [29] and CrateDB [8]. Considering the experimental comparison results, the authors reached the conclusion that there is no single ”best” solution. While InfluxDB provides the best performance in a monitoring scenario, other solutions provide better performance when inserting data (i.e. Graphite) or performing data aggregation operations (i.e. TimescaleDB).

In the article [26], the authors identify software quality attributes that can be used when evaluating time-series databases. They divided the attributes into different categories of performance, usability, maintainability and security and then performed an evaluation of InfluxDB, MariaDB, Redis and TimescaleDB.

In terms of performance, the databases were ranked in this order: InfluxDB - MariaDB - TimescaleDB - Redis. Based on usability attributes, they were ranked as MariaDB - InfluxDB - TimescaleDB - Redis. Based on the maintainability attributes, they were ranked as InfluxDB - MariaDB - Redis - TimescaleDB. Based on the security attributes, they were ranked as MariaDB - TimescaleDB - InfluxDB - Redis.

In the survey [5] the authors perform a comparison between the 12 most prominent time-series databases based on 27 criteria. The criteria were divided in the following groups:

1. **Distribution/Clusterability**: High Availability, scalability and load balancing
2. **Functions**: Availability of INS, UPD, READ, SCAN, AVG, SUM, CNT, DEL, MAX and MIN functions
3. **Tags, Continuous Calculation and Long-term storage**: Continuous calculation, tags, long-term storage, and the support of matrix time series
4. **Granularity**: Granularity, downsampling, smallest possible granularities for operations/storage
5. **Interfaces and Extensible**: Application Programming Interfaces, interfaces, client libraries
6. **Support and License**: Availability of a stable version, Commercial support and License.

They reached the conclusion that no database supports all features from any of the criteria. However, the best choice in all criteria is Apache Druid [4], which is a analytics database designed for large datasets that has been previously used as a time series database for device metrics. However, it does not yet have a stable version. If a stable version is needed the best choices are InfluxDB or MonetDB.

2.2 EPREL and energy labels

The European Product Database for Energy Labelling (EPREL) [34] is an online product registration database managed by the European Commission that contains energy efficiency information about the devices covered by the Energy Labelling regulation (EU) 2017/1369 [40]. This regulation replaces Directive 2010/30/EU maintaining essentially the same scope but updating the energy labelling framework taking into account the technological progress for energy efficiency in products achieved over recent years.

Since 1 January of 2019, all the EU suppliers are legally required to register the new models covered by this Energy Labelling regulation (including second hand imported models) in the EPREL database before they are placed on the EU market for the first time.

The main proposes of the EPREL database are among others: to provide the public with information about products placed on the market and their energy labels and to provide the European Commission with up-to-date products energy efficiency information for reviewing energy labels [34].

The purpose of EU energy labels is to indicate the energy efficiency of products, enabling consumers to make informed choices on products more energy efficient while also reducing greenhouse gas emissions. When a supplier places an energy related product on the market, each unit should be accompanied by a label in paper format and should be easily recognizable. The energy label contains information separated in at least four sections, as it can be seen in Figure 2.3:

1. Product details: specific information about the device like the brand and model.

2. Energy class: a green to red color scale associated to a letter, from A (most efficient) to G (least efficient) that gives an idea of the product's energy consumption. In the old label the class is on a scale from A+++ to D, while in the rescaled labels scale is from A to G.

3. Consumption, efficiency, capacity or other information depending on the product type.

4. QR code: placed on the top right corner in the rescaled energy labels, that gives access to additional information about the model, depending on the type of device.

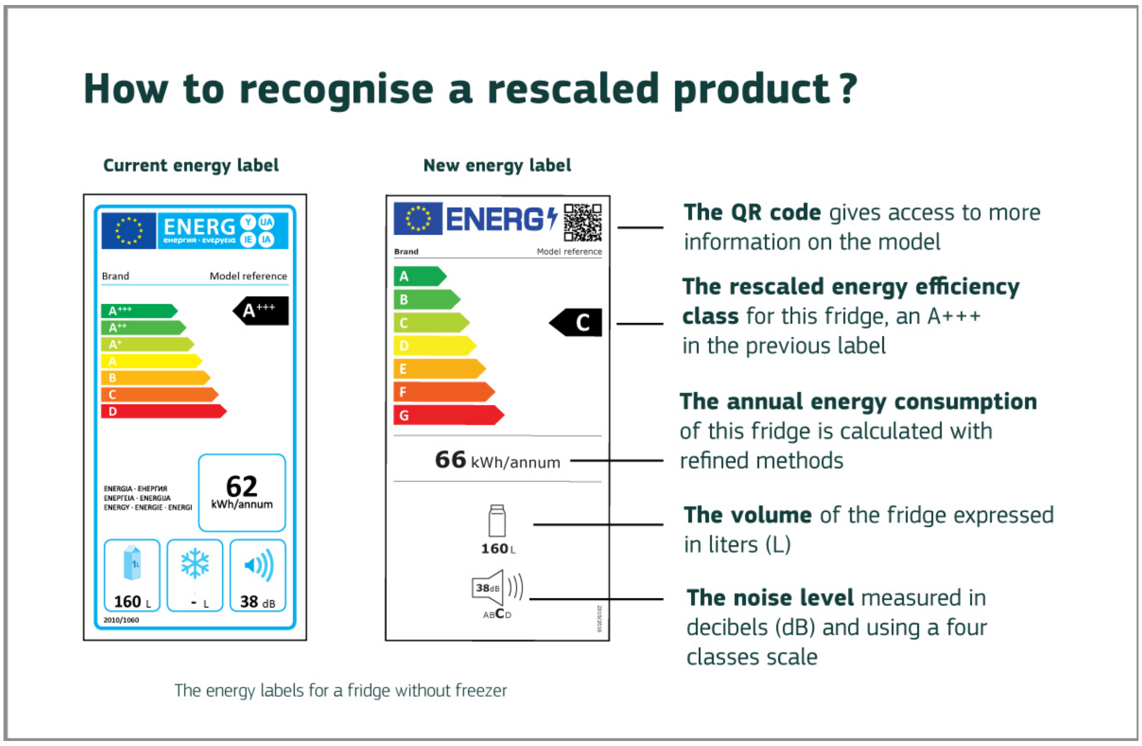

How to recognise a rescaled product?
Current energy label
New energy label
The QR code gives access to more information on the model
The rescaled energy efficiency class for this fridge, an A+++ in the previous label
The annual energy consumption of this fridge is calculated with refined methods
The volume of the fridge expressed in liters (L)
The noise level measured in decibels (dB) and using a four classes scale
The energy labels for a fridge without freezer
62 kWh/annum
66 kWh/annum
160 L
38 dB

Figure 2.3: Example of the past energy label (left) and new label (right) for a fridge [1]

The EPREL database consists of a public part, a compliance part and an online portal [15]. The portal gives access to those two parts, and it is possible to consult the model information that was previously entered by the supplier and the European Commission if accessing to the public part. In the compliance part, it is possible to register and manage an organization, to register a device and enter information related with the energy label, technical documentation, and compliance monitoring [34].

The public information available for each device depends on its type, however there is some information that is common to all, namely [34]:

- Name, address, contact details of the supplier.
- Energy label in electronic format.
- Energy efficiency class and other parameters of the label.

The supplier also enters some information that will not be publicly available, but necessary for the market surveillance authorities:

- A general description of the model.
- References to the harmonized standards applied.
- Specific precautions to be taken when the model is assembled or installed.

- Measured technical parameters of the model.
- Calculations performed with the measured parameters.

Additionally, the EPREL system associates to each device, some important information:

- Model registration number, used to uniquely identify the device in the database. It is also associated with the QR code on the device energy label.
- Product group code, used to identify the device type. Currently, the product groups codes supported are:
  - HOUSEHOLD_DISHWASHER_2019
  - HOUSEHOLD_WASHING_MACHINE_2019
  - HOUSEHOLD_WASHER_DRIER_2019
  - ELECTRONIC_DISPLAY
  - HOUSEHOLD_REFRIGERATING_APPLIANCE_2019

respectively, for dishwashers, washing machines, washer dries, electronic displays, and refrigerator devices. Lamps are planned to be added in 1 of September of 2021 and other device types in the following years. This means that 5 product groups will be rescaled in 2021.

To insert a device in the EPREL database it must receive a new rescaled energy label that has some differences from the previous one, such as a QR code, a new class scale and, possibly, different metrics. Therefore, despite already having information about the metrics and metadata for a device type, this information will probably be altered when added to EPREL [2].

One important thing to note is that the measurement units for each metric are not stored on EPREL and are instead available on the labelling documentation. For example, the measurement units for electronic displays are available on the Regulation on energy labelling for electronic displays (EU) 2019/2013 [41]. This measurement units must be the ones used by all suppliers when inserting the device info into EPREL.

For the currently supported device types, the information and metrics registered in the EPREL database are [31]:

- Electronic displays:
  - Energy class (A, B, C, D, E, F, G)
  - On mode power demand (Watts) (numeric)
  - Off mode power demand (Watts) (numeric)

- Standby mode power demand (Watts) (numeric)
- Networked standby mode power demand (Watts) (numeric)
- Visible screen area (dm^3) (numeric)
- Category (Television, Monitor, Signage, Other) (varchar(15))
- Panel technology (LCD, LED LCD, QLED LCD, OLED, MicroLED, QLED, FED, EPD, Other) (varchar(15))
- Size ratio x (int)
- Size ratio y (int)
- Resolution horizontal (Pixels) (int)
- Resolution vertical (Pixels) (int)
- Screen diagonal (cm) (numeric)

• Dishwashers:

- Energy class (A, B, C, D, E, F, G)
- Energy efficiency index (numeric)
- Rated capacity (place settings, a set of tableware for one person) (int)
- Energy consumption [per cycle, based on the eco programme] (Kilowatts) (numeric)
- Energy consumption [per 100 cycles, based on the eco programme] (Kilowatts) (int)
- Water consumption [per cycle, based on the eco programme] (Liters) (numeric)
- Programme duration for the eco programme (Hour:Minutes) (int)
- Off mode power demand (Watts) (numeric)
- Standby mode power demand (Watts) (numeric)
- Networked standby mode power demand (Watts) (numeric)
- Delay start mode power demand (Watts) (numeric)
- Noise emission class for the eco programme (A, B, C, D)
- Noise emissions for the eco programme (Decibel) (int)
- Height (cm) (int)
- Width (cm) (int)
- Depth (cm) (int)
- Cleaning performance index (numeric)

  - Drying performance index (numeric)

- Refrigerator devices:
  - Energy class (A, B, C, D, E, F, G)
  - Energy efficiency index (int)
  - Annual energy consumption (Kilowatts) (numeric)
  - Noise emission class (A, B, C, D)
  - Noise emissions (Decibel) (int)
  - Low noise appliance (Yes, No)
  - Total volume (Liters or dm^2) (int)
  - Height (mm) (int)
  - Width (mm) (int)
  - Depth (mm) (int)
  - Design type (Built in, Free standing)
  - Compartment:
    * Compartment volume (Liters or dm^3) (numeric)
    * Compartment type (Pantry, Wine storage, Cellar, Fresh food, Chill, Zero star, One star, Two star, Three star, Four star, Two star section, Variable temp)

- Washing machines:
  - Energy class (A, B, C, D, E, F, G)
  - Energy efficiency index (numeric)
  - Rated capacity (Kg) (numeric)
  - Energy consumption [per cycle, eco 40-60 programme] (kilowatt-hour) (numeric)
  - Weighted energy consumption [per 100 cycles, eco 40-60 programme] (kilowatt-hour) (int)
  - Water consumption [per cycle, eco 40-60 programme] (Liters) (int)
  - Washing efficiency index (numeric)
  - Rinsing effectiveness (g/kg) (numeric)
  - Maximum temperature inside the treated textile [Rated capacity] (Celsius) (int)

  - Maximum temperature inside the treated textile [Half capacity] (Celsius) (int)
  - Maximum temperature inside the treated textile [Quarter capacity] (Celsius) (int)
  - Remaining moisture [Rated capacity] (%) (int)
  - Remaining moisture [Half capacity] (%) (int)
  - Remaining moisture [Quarter capacity] (%) (int)
  - Spin-drying efficiency class (A, B, C, D, E, F, G)
  - Spin speed [Rated capacity] (rpm) (int)
  - Spin speed [Half capacity] (rpm) (int)
  - Programme duration [Rated capacity] (Hour:Minutes) (int)
  - Programme duration [Half capacity] (Hour:Minutes) (int)
  - Programme duration [Quarter capacity] Hour:Minutes) (int)
  - Off mode power consumption (Watts) (numeric)
  - Standby mode power consumption (Watts) (numeric)
  - Networked standby power consumption (Watts) (numeric)
  - Delay start power consumption (Watts) (numeric)
  - Noise emissions class (A, B, C, D, E, F, G)
  - Noise emissions (Decibel) (int)
  - Height (cm) (int)
  - Width (cm) (int)
  - Depth (cm) (int)

- Washer drier machines:
  - Energy class for complete/washing cycle (A, B, C, D, E, F, G)
  - Energy efficiency index for complete/washing cycle (numeric)
  - Rated capacity for complete/washing cycle (Kg) (numeric)
  - Energy consumption for complete/washing [per cycle, eco 40-60 programme] (kilowatt-hour) (numeric)
  - Weighted energy consumption for complete/washing [per 100 cycles, eco 40-60 programme] (kilowatt-hour) (int)
  - Water consumption for complete/washing [per cycle, eco 40-60 programme] (Liters) (int)
  - Washing efficiency index for complete/washing cycle (numeric)

  - Rinsing effectiveness for complete/washing cycle (g/kg) (numeric)
  - Maximum temperature inside the treated textile for complete/washing cycle[Rated capacity] (Celsius) (int)
  - Maximum temperature inside the treated textile for complete/washing cycle [Half capacity] (Celsius) (int)
  - Maximum temperature inside the treated textile for complete/washing cycle [Quarter capacity] (Celsius) (int)
  - Remaining moisture [Rated capacity] (%) (int)
  - Remaining moisture [Half capacity] (%) (int)
  - Remaining moisture [Quarter capacity] (%) (int)
  - Spin-drying efficiency class (A, B, C, D, E, F, G)
  - Spin speed [Rated capacity] (rpm) (int)
  - Spin speed [Half capacity] (rpm) (int)
  - Programme duration for complete/washing cycle [Rated capacity] (Hour:Minutes) (int)
  - Programme duration for complete/washing cycle [Half capacity] (Hour:Minutes) (int)
  - Programme duration for complete/washing cycle [Quarter capacity] Hour:Minutes) (int)
  - Off mode power consumption (Watts) (numeric)
  - Standby mode power consumption (Watts) (numeric)
  - Networked standby power consumption (Watts) (numeric)
  - Delay start power consumption (Watts) (numeric)
  - Noise emissions class (A, B, C, D, E, F, G)
  - Noise emissions (Decibel) (int)
  - Height (cm) (int)
  - Width (cm) (int)
  - Depth (cm) (int)

It is possible to access the data stored in EPREL either by scanning the QR code on the energy label, which redirects to a web page with all the information, or by resorting to the EPREL REST API [14], which is the solution used in the SATO project.

This API makes available five possible request calls:

1. **Get list of products groups** - Request used to get full list of product groups codes.

2. **Get product group's models** - Request used to search for all the models. The mandatory parameter needed is the product group code. This request is the only one that is restricted, since it needs an API key to be used. The request for this key is not yet available, and it is not known for when it will be available.

3. **Get product group's model by registration number** - Request used to get the data for a single model in a product group. The mandatory parameters are the product group code and the model registration number.

4. **Get product group's model label by registration number** - Request used to get the label of a single model in a product group. The mandatory parameters are the product group code and the model registration number.

5. **Get product group's model fiche by registration number** - Request used to get the fiche (product information sheet) of a single model in a product group. The mandatory parameters are the product group code and the model registration number.

The data fetched from the EPREL database will be used in the SATO real-life cloud-based self-assessments of appliances, as a reference for comparison to the real-life metrics that will be calculated using sensor-measured data. Additionally, some of the metadata of the device model is needed for the calculation of some metrics. For example, in the case of the electronic displays, the visible screen area is used to obtain the energy class. That measure is part of the metadata stored on the EPREL database.

2.3 Summary

This section presented the requirements for a database used to store sensor data and presents the time series database engine currently used in this use case. Time series databases engine differentiates itself from other types due to the type of data they are made to work with.

Additionally, this chapter also presented a comparison between multiple time series database engines, each one with different advantages and disadvantages.

Finally, there is the research done on the EPREL database, what information is available for each device type and how to obtain it using the API.

Chapter 3

Database system design

This chapter presents the database for storing sensor data (Section 3.1), a comparison between the database engines (Section 3.2) and the database for storing metrics (Section 3.3). Additionally, it presents an overview of the SATO platform (Section 3.4) details and the main components of this work and their integration in the whole data flow (Section 3.5).

3.1 Database requirements for storing sensor data

To find the adequate database engine for the purpose of storing the sensor data, the first step is to define the system requirements, so they can then be used as a comparison criterion. Through the research of previous projects that also worked with sensor data, benchmarks, technical documentation of the database engines and meetings done with other participants on the SATO project, the collected requirements were:

- Data schema, a structure that describes the elements, for example, tables and rows before adding data. The existence of a data schema allows for a higher degree of control, therefore improving the performance, but increases the work needed for inserting data since it needs to be formatted to fit the schema. In a schemeless database there are no constrains for inserting data.
- Real-time visualization capabilities, offering the resources to build a dashboard where the results of a query can be displayed using a various number of visualization types. These results should also update in real-time with the insertion of new data.
- Query language, the system must provide a efficient language to query the data. SQL might provide a better experience since its one of the most common languages used when working with relational databases. However, it may prove to become too complex when working with time series data, like the one received from the sensors.

- Scalability, due to the need to store the data from an increasing amount of connected sensors, increasing number of users and the need keep historical data.
- High availability, since even a short downtime can cause some problem in the system, for example a miscalculation in a metric.
- Load balancing, distributing the workload between the multiple database servers. It is an additional resource to help with the handling of the large size of sensor data.
- Data retention policies, used to describe how long the database keeps the data. A longer retention time leads to a worse performance while the opposite also happens.
- Downsampling, used to reduce the overall disk usage as data accumulates over time.
- Access control, used to limit the access of the different types of users to the data.
- Documentation and technical support, to facilitate the implementation and management of the database.
- Long-term storage, it is also important to provide the capacity for supporting automatic long-term storage using data retention, downsampling and other tolls.
- Dashboard support for plugins, to allow possible extensions of the dashboards making available new functionalities. These can add new visualization types or new external sources.
- Maintenance without downtime, that is the possibility of doing maintenance, for example, increasing the cluster size, without any downtime.

3.2 Database engines comparison for sensor data

Before comparing various database engines using the requirements previously defined, it is necessary to find what is the most appropriate type of database engine. There are many options like the relational database, document base database, graph database, time series database, among others.

The decision was to use a time series database due to the large volume of data provided by the sensors. This type of database are the most adequate [12] to track, monitor and aggregate timestamped data over time and the multiple number of successful use cases of IoT applications that use time series databases for working with sensor data [23].

The next step was to find what databases engines to consider for comparison. Research was done to find which ones have various use case examples of being used in the development of IoT applications, the different advantages and disadvantages and different performance levels for different types of operations. The databases engines considered for the comparison were InfluxDB, TimescaleDB, CrateDB, Prometheus and VictoriaMetrics.

The Table 3.1 presents the comparison considering the requirements defined in Section 3.1.

Table 3.1: Database comparison table

	InfluxDB	TimescaleDB	CrateDB	Prometheus	VictoriaMetrics
Data schema	×	✓	✓	✓	✓
Real time visualization	✓	✓	✓	✓	✓
Query language	FLUX	SQL	SQL	PromQL	MetricsQL
Scalability	✓ e.d	✓	✓ e.d	×	✓
High availability	✓	×	✓ e.d	×	✓
Load balancing	✓	×	✓	✓ n.a	✓
Data retention policies	✓	✓	×	✓	✓
Downsampling	✓	✓	✓	×	×
Access control	✓	✓	✓ e.d	×	✓ v.a
Documentation and technical support	✓	✓	×	×	✓
Long term storage	✓	✓	✓	×	✓
Dashboard support for plugins	✓	✓	✓	✓	✓
Maintenance without downtime	✓	×	✓	×	×

e.d - Enterprise edition, n.a - Not automatic

t.k - Using tasks to perform downsampling and retention policies

v.a - Using the auth proxy vmauth [52]

Analysing the table, it is possible to see the diversity in benefits and limitations of each database engine. InfluxDB checks all the criteria and has the benefit of not needing a data schema, allowing for an easier and faster setup, with all the other criteria are also checked. One disadvantage of the InfluxDB is the need of the enterprise edition for the database scaling, that even if it does not limit this work, it is needed in the context of the SATO project. The use of Flux, although it may prove to be a better scripting language it implies a initial effort learning a new language.

TimescaleDB checks all the requirements besides the high availability, load balancing and maintenance without downtime which limits the use of this database in the SATO project. CrateDB, similarly to InfluxDB, checks all the requirements also needing the enterprise edition for the scalability and high availability. However, since it is a new database engine it does not have a complete documentation and it still has a low user base to provide with help.

Prometheus does not meet any of the scaling requirements so it is not a viable choice, since it cannot be used in the SATO project. The VictoriaMetrics database provides an improvement on the TimescaleDB, with the two being able to being used together, however it does not provide with native access control.

After analysing and comparing the database engines it was decided to consider the InfluxDB, since it checks all requirements, offers the best performance and it has already been used in many projects that work with sensor data and technologies.

3.3 Database engines for storing metrics and metadata

Another objective of this work is to implement a database to store the data from the EPREL DB and also to store important metrics and metadata from the sensor DB and EPREL DB. This database will also support the establishment of relationships between the metrics, the device and the building.

Since the data needs to be accessible using the Flux query language, the database engine must be supported by the Flux SQL package [18]. Therefore, the possible database engines are PostgreSQL [38], MySQL [35], Snowflake [46], SQLite [47], Microsoft SQL Server [33], Amazon Athena [3] and Google BigQuery [20]. A limitation of the Flux SQL package is the lack of of support for the SQLite database in the InfluxDB open source version. Due to this SQLite was excluded since it cannot be used when making a query from the dashboard.

After analysing all the possibilities, PostgreSQL was the chosen database engine, mainly due to the fact of it being an object-relational database, meaning that it features table inheritance which will be useful in the context of the work due to the multiple types of devices considered. Another advantage of this database engine is how well it handles concurrency, which is a big requirement for this database due the high number of accesses needed.

3.4 Overview of the SATO architecture

This section describes briefly the SATO architecture and its main components to understand the integration and the dependencies with the DB components that were developed in this work.

The SATO platform aims to provide cloud-based self-assessment and optimization of buildings and equipment/appliances energy.

As already mentioned, the SATO project implements a appliance and building energy consuming equipment self-assessment and optimization platform, and since this work is part of the SATO project, it is important to design it in order be integrated now and in the future. To achieve this goal, the first step is to understand how the thesis work will fit in the SATO architecture and interact with other components.

The SATO platform comprises six blocks of components and players: external data sources, buildings, SATO middleware, self-assessment framework, self-optimization services, and actors [11]. This blocks are represented in Figure 3.1.

External data sources are publicly available databases that provide context and environment information for the SATO platform. They are read-only data that will be forwarded to the SATO platform. Examples of foreseen external sources include (but are not limited to) energy labelling of appliances (e.g. EPREL), weather data, and energy prices.

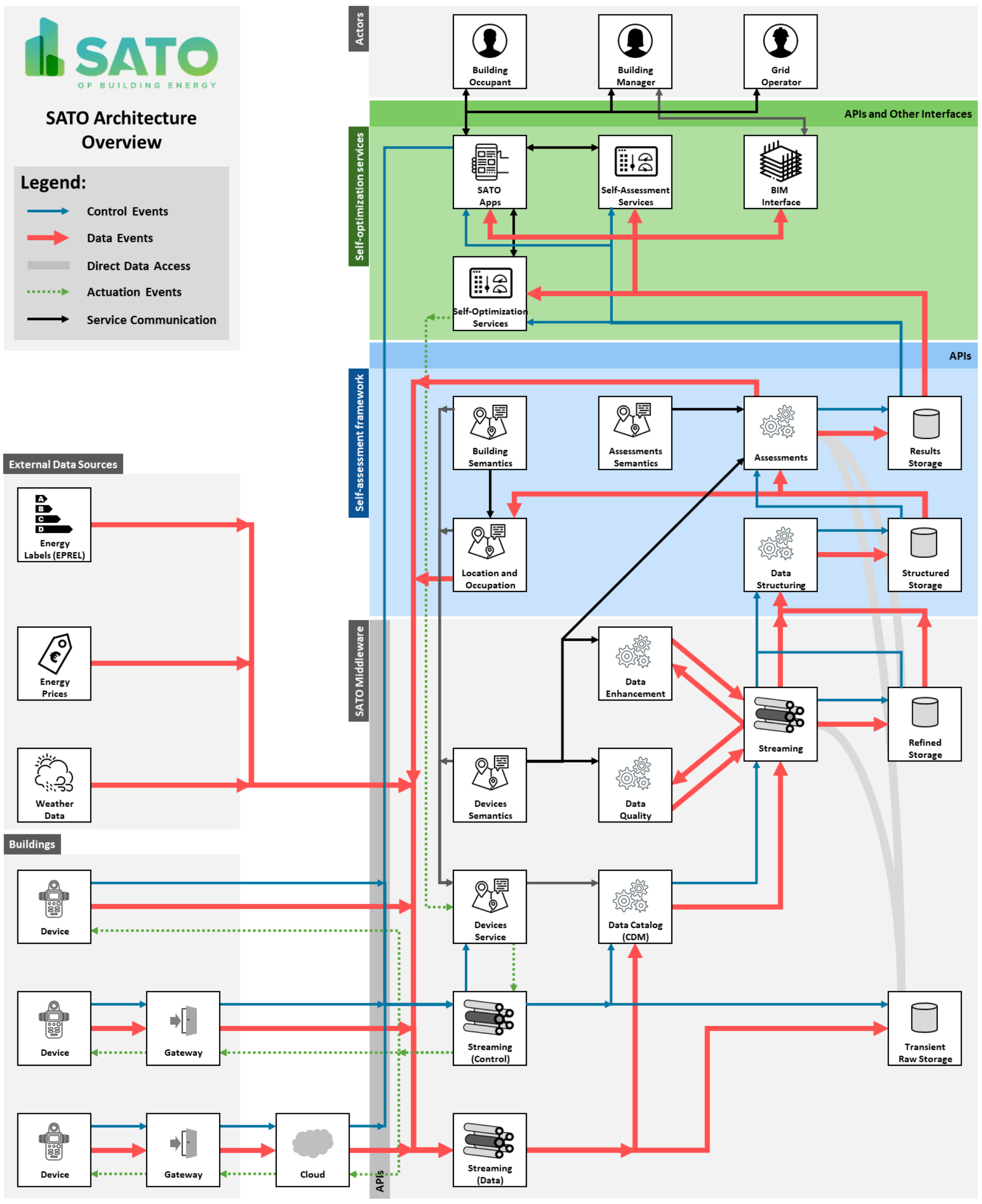


Figure 3.1: SATO architecture

SATO middleware is the core block from the SATO architecture in terms of ingesting data, organizing it, and preparing it to be consumed by the main processing tasks of the self-assessment framework and optimization services. This block is subdivided into event streaming components that will bridge buildings and processing components, services that standardize events being ingested by the platform, services for keeping up-to-date snapshots of devices on each building (and providing semantically coherent data about them), and data enhancing components that will assess and improve the quality of data being provided by the platform.

The Self-Assessment framework is the main processing block of the platform, both in terms of quantity and variety of data. It will use the data ingested by the SATO platform through data analysis and machine learning to build the assessments that will report on the energy performance of buildings, on building occupancy and on equipment faults. This block provides a series of assessment results that complement the available building data, serving as the inputs for the self-optimization services block that will make energy management decisions.

Finally, the Actors block represents the users of the SATO platform. It includes building occupants/owners, building managers, and grid operators, which are humans that will directly benefit from the SATO project (and its platform) through assessments and optimized energy management decisions, improved comfort, or more flexible infrastructures than the ones currently available.

Arrows in Figure 3.1 depict data flows of different event types that will become clearer with the descriptions provided. They include control events (blue lines) related to the status of the monitored buildings and of the platform, data events (red lines) that provide most of the data and building measurements that will be ingested and processed by the SATO platform, and actuation events (green dashed lines) that represent actions and control commands taken by the self-optimization services and managers to change the behavior of buildings.

The SATO platform architecture is designed to deal with incoming data events and outgoing actuation/control commands. It offers support for different types of integration with devices, including: i) direct communication between devices and the platform; ii) communication between a building gateway and the platform, and/or iii) communication between cloud services offered by third parties (e.g. device suppliers or other platforms) and the platform.

3.5 Database components

The work done in this thesis can be divided into two data flows, one for the collection of data from EPREL and insertion into the Metrics and Metadata device DB and another for the collection of the sensor data insertion into the Sensor DB. Both these databases are located in the structured storage block, as seen in Figure 3.2.

When working with data from the EPREL database as a soft sensor in the SATO platform, simulating the behavior of a sensor and thus following the same procedures in the SATO dataflow for inserting data, several steps are executed for registering a device through the Device Services, namely:

1. When a device is registered, it triggers the connector that first checks, by comparing the registration number, if the device model data is already stored in the Metrics and

Metadata device DB, and if not, it fetches the EPREL data. Otherwise, no request is issued.

2. The EPREL data is used to populate the ontology on the Device Semantics component. A device identifier in the SATO platform (SATO ID), which is assigned by the Data Catalog (CDM), is used to correlate the device with the semantic information.

3. The *device id* is sent through the system, using Kafka, and used to insert the EPREL data on the Structured Storage component.

Kafka is an event streaming platform that was implemented in the SATO platform as part of the thesis work of André Gil [19]. It is used in the SATO platform for sending data as events, for example sending the EPREL info to be inserted in the database. The CDM that is presented in the steps above stands for common data model, a unique data format that makes possible the integration of heterogeneous commercial platforms and smart devices.

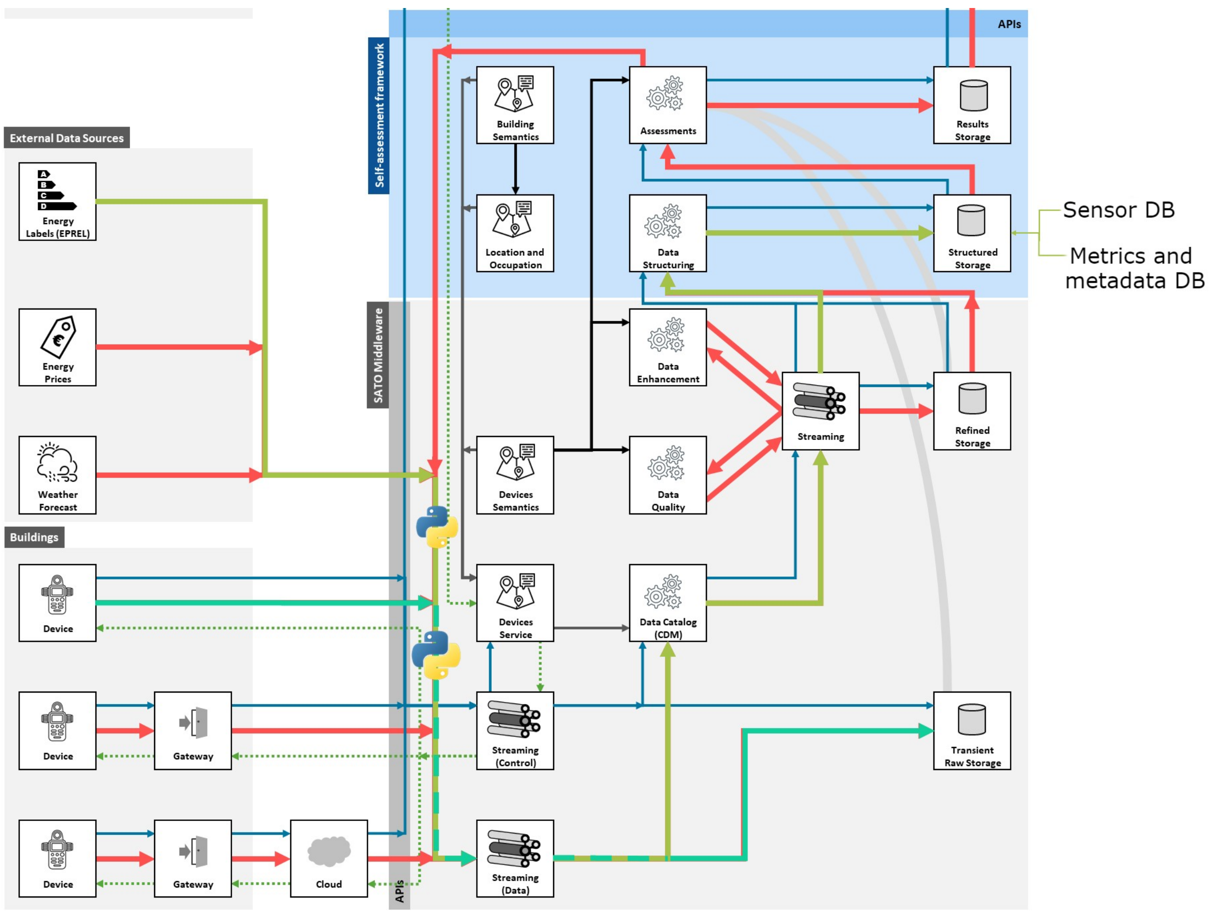


Figure 3.2: Detailed SATO platform: Sensor DB and EPREL

Figure 3.2 represents a detailed view of the SATO platform with the main components related to the EPREL DB, Metrics and metadata DB and the Sensor DB. The EPREL data insertion is represented by the green arrow and the sensor data in the dark green arrow. Additionally, the Python icons are used to represent the adapters that collect and insert the EPREL data and the one that fetches the EPREL data. This Figure is only used to provide a better understanding about how the current data flows would fit the current SATO architecture and in no way represents how it will function in the finalized SATO project.

The SATO project collects real-life data to perform the energy consumption self-assessment and optimization. One objective of the SATO project is to relate real-life assessment of devices to the EU energy label data in EPREL. To accomplish this objective, it was implemented a time series database (Sensor DB) to store real-life energy consumption and operating environment data of building devices.

Additionally, a relational database (Metrics and metadata device DB) is also implemented to store the consumption metrics obtained from the real-life data and the energy labelling data from EPREL. Figure 3.3 shows a dataflow diagram with these databases and the data exchanged between components.

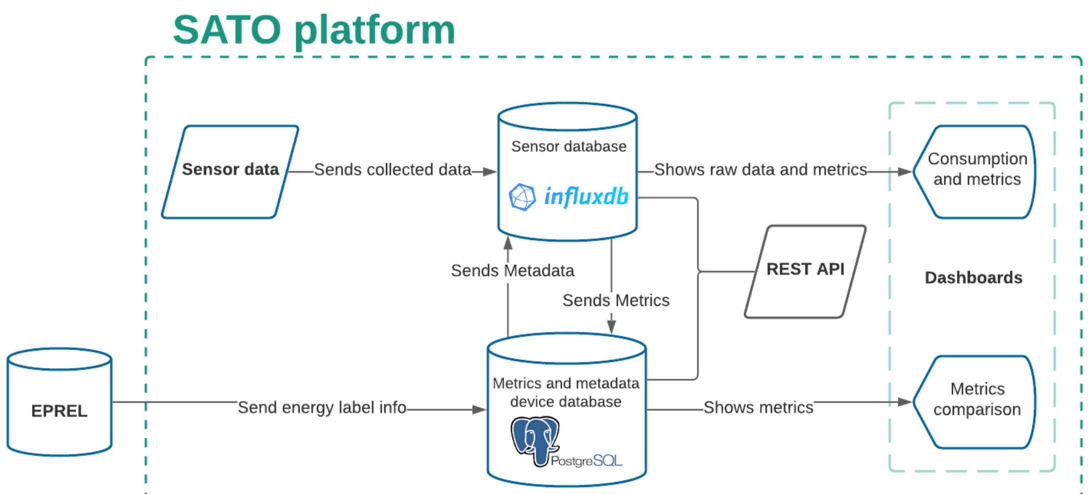


Figure 3.3: System dataflow

This data flow begins with the sensor data sent to the Sensor DB, where calculations are done to obtain the consumption metrics. This metrics are then shown in the consumption and metrics dashboard and stored in the metrics and metadata device DB, that can be accessed through the metrics comparison dashboard. Also present in this dataflow, although external to the SATO platform, is the EPREL database, where the data is fetched when a device is registered.

To access the data without making queries or using the dashboards, a REST API is developed to give this simplified access to the data stored in both the databases. It can be used in the SATO platform to supply the self-assessment framework with access to the real-life metrics that are then used for the assessments.

The insertion of the sensor data and the energy label info in the respective databases is done using Python scripts. The connection and interaction between the two different databases are done using the Flux SQL package. The Flux language is also used to develop the dashboards.

The following subsections present the details of the data model and the dashboards, starting with the description of the data and the DB.

3.5.1 Sensor data

To obtain real-life consumption metrics of the devices, it is crucial to exist a large collection of data about the device consumption. This data is collected using sensors, more specifically, an accelerometer sensor, a temperature sensor, a waterflow sensor, a current sensor and noise sensor. These sensor types were chosen since they collect the required data to calculate the same consumption metrics as the ones that are present in the EU energy label. The installation of sensors and collecting of sensor data was part of the thesis work of [25], that is also part of the SATO project.

Additionally, since these sensors are mounted on the device instead of relying on sensors previously mounted by the manufacturer, it is possible to support both new and legacy devices. It is essential that all the data collected by the sensors is timestamped, since the calculations done to obtain the consumption metrics use data grouped by time. The following sections details the data models of the data and the DB.

1. Accelerometer sensors collects:
    - Timestamp
    - X value
    - Y value
    - Z value
2. Temperature sensors collects:
    - Timestamp
    - Temperature value
3. Waterflow sensors collects:
    - Timestamp
    - Quantity of water
4. Current sensors collects:

- Timestamp
- Amperes
- Kilowatts
- Kilowatts per hour

5. Noise sensors collects:

   - Timestamp
   - Decibels

During the implementation of the database, data was available for a monitor AOC 27B1H and a Laptop Asus Rog GL503GE, that was collected using the current sensor and stored in a file using the XML data format. This work was developed has part of the thesis work in [25]. This storing method will be modified when integrating in the SATO platform, so that the insertion is done directly into the database, without the need to use a file as an intermediate.

Additionally, there was also data available for a fridge Beko TS190030N that was retrieved from the ACS-F2 dataset [44]. This dataset is composed by the electrical consumption signatures of devices in different categories. This fridge was chosen since it is the only device type from the dataset that has data in the EPREL database, making possible a comparison. The data collected for both devices are collected with a 1 second interval during 1 hour of continuous collecting.

Since there was sensor data for only two device types, this caused some limitations for the development of this work. Therefore, it will be only possible to calculate, store and compare metrics for Electronic Displays and Refrigerators.

3.5.2 EPREL data

The EPREL database stores the information about the products placed on the EU market and their energy label. This data will be compared with the real-life metrics obtained from the sensor data. For this comparison to be possible, the data must be retrieved and inserted in the Metrics and metadata device DB.

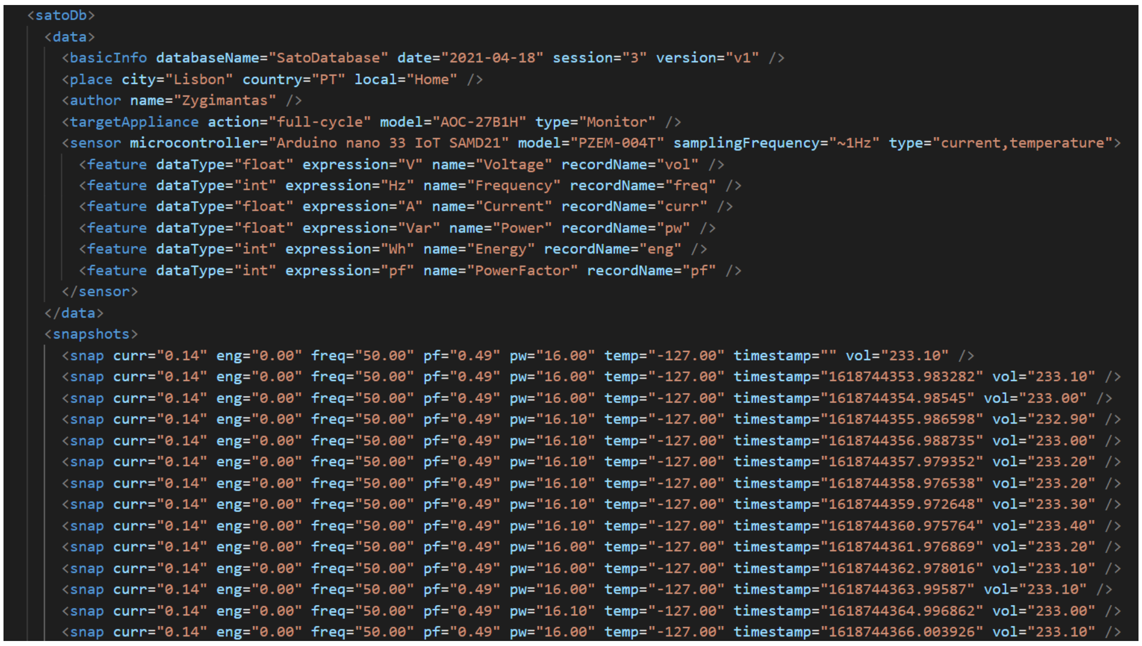
The insertion of the EPREL data for a device in the Metrics and Metadata device DB happens when registering the device in the SATO platform. The registration message contains the device model, brand and type that are used in the API call *Get product group's info*. However, in the current EPREL API version, this call is unavailable until the end of 2021. Therefore, with this restriction in place a temporary amendment needed to be done, that was associating the device brand, model and type with the registration number and the use the call *Get product group's model by registration number*.

3.5.3 Sensor DB

The sensor DB stores all the data collected by the sensors and is developed using InfluxDB. As already mentioned, the Influx engine was chosen, because is a time series DB that allows better performance on time series data and to calculate device energy assessments.

The data is stored in a bucket, called SATO_bucket, were it is possible to set a retention period, that is currently set to retain the data forever. The choice of keeping the data forever was done since data size during this work was not large enough to warrant the deletion of data to improve performance. However, this retention period can be later changed to a specific period, for example a week or month, therefore improving the performance.

The device name, model and brand, that are collected from a XML file with the sensor data like the one in Figure 3.4, are stored as tags in order to improve query performance, due to the fact of tags being indexed. This decision was made since this meta-data is used in most queries to filter the results.



```xml
<satoDb>
  <data>
    <basicInfo databaseName="SatoDatabase" date="2021-04-18" session="3" version="v1" />
    <place city="Lisbon" country="PT" local="Home" />
    <author name="Zygimantas" />
    <targetAppliance action="full-cycle" model="AOC-27B1H" type="Monitor" />
    <sensor microcontroller="Arduino nano 33 IoT SAMD21" model="PZEM-004T" samplingFrequency="~1Hz" type="current,temperature">
      <feature dataType="float" expression="V" name="Voltage" recordName="vol" />
      <feature dataType="int" expression="Hz" name="Frequency" recordName="freq" />
      <feature dataType="float" expression="A" name="Current" recordName="curr" />
      <feature dataType="float" expression="Var" name="Power" recordName="pw" />
      <feature dataType="int" expression="Wh" name="Energy" recordName="eng" />
      <feature dataType="int" expression="pf" name="PowerFactor" recordName="pf" />
    </sensor>
  </data>
  <snapshots>
    <snap curr="0.14" eng="0.00" freq="50.00" pf="0.49" pw="16.00" temp="-127.00" timestamp="" vol="233.10" />
    <snap curr="0.14" eng="0.00" freq="50.00" pf="0.49" pw="16.00" temp="-127.00" timestamp="1618744353.983282" vol="233.10" />
    <snap curr="0.14" eng="0.00" freq="50.00" pf="0.49" pw="16.00" temp="-127.00" timestamp="1618744354.98545" vol="233.00" />
    <snap curr="0.14" eng="0.00" freq="50.00" pf="0.49" pw="16.10" temp="-127.00" timestamp="1618744355.986598" vol="232.90" />
    <snap curr="0.14" eng="0.00" freq="50.00" pf="0.49" pw="16.10" temp="-127.00" timestamp="1618744356.988735" vol="233.00" />
    <snap curr="0.14" eng="0.00" freq="50.00" pf="0.49" pw="16.10" temp="-127.00" timestamp="1618744357.979352" vol="233.20" />
    <snap curr="0.14" eng="0.00" freq="50.00" pf="0.49" pw="16.10" temp="-127.00" timestamp="1618744358.976538" vol="233.20" />
    <snap curr="0.14" eng="0.00" freq="50.00" pf="0.49" pw="16.10" temp="-127.00" timestamp="1618744359.972648" vol="233.30" />
    <snap curr="0.14" eng="0.00" freq="50.00" pf="0.49" pw="16.10" temp="-127.00" timestamp="1618744360.975764" vol="233.40" />
    <snap curr="0.14" eng="0.00" freq="50.00" pf="0.49" pw="16.00" temp="-127.00" timestamp="1618744361.976869" vol="233.20" />
    <snap curr="0.14" eng="0.00" freq="50.00" pf="0.49" pw="16.00" temp="-127.00" timestamp="1618744362.978016" vol="233.10" />
    <snap curr="0.14" eng="0.00" freq="50.00" pf="0.49" pw="16.00" temp="-127.00" timestamp="1618744363.99587" vol="233.10" />
    <snap curr="0.14" eng="0.00" freq="50.00" pf="0.49" pw="16.10" temp="-127.00" timestamp="1618744364.996862" vol="233.00" />
    <snap curr="0.14" eng="0.00" freq="50.00" pf="0.49" pw="16.00" temp="-127.00" timestamp="1618744366.003926" vol="233.10" />
```

Figure 3.4: XML file with sensor data for the monitor AOC 27B1H

Using the InfluxDB tasks it is possible to schedule the calculations and insertion in the metrics and metadata device DB. For each device type there are two Flux scripts, one for the device metrics and another for the cycle metrics, that make the necessary queries. The time between each insertion of the device metrics is 1 week, in order to obtain a large quantity of data and obtain reliable metrics. This time period can also be later changed, changing the task option ”every”, if it proves to not be the best choice. The cycle metrics are calculated and stored after each device cycle ends. A device cycle can be defined as a period where the device is considered turned on and, consequently, with the consumption's rising.

Despite not being implemented, due to the lack of a large data quantity, the InfluxDB checks can be used to detect and warn users about device failures. Section 4.5 will explain, how this process can be implemented using the Influx checks.

3.5.4 Metrics and metadata device DB

The metrics and metadata device DB stores the metrics and metadata for the devices registered in the SATO platform is implemented using PostgreSQL. The metadata for the device comes from the EPREL database and the metrics can come from the sensor database or the EPREL database.

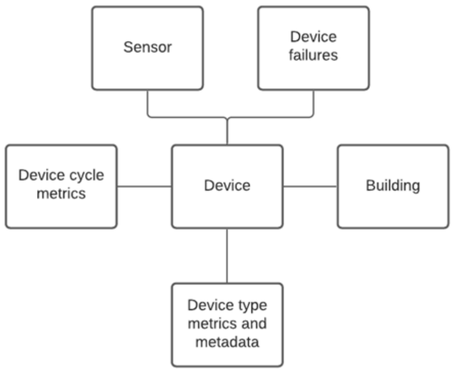


Figure 3.5: Database block diagram

Figure 3.5 presents a block diagram with the main entities and corresponding connections that will be detailed next. The full entity-relationship (ER) model for the metrics and metadata device DB is presented in Appendix A.

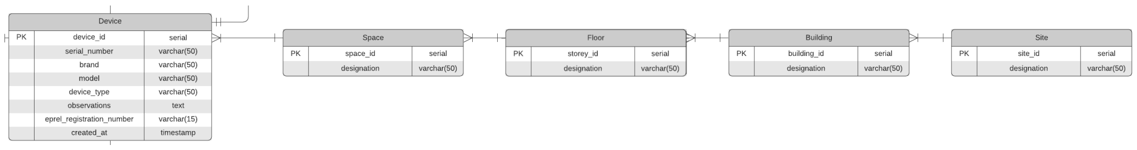


Figure 3.6: ER model related with the Device and Building

Figure 3.6 presents ER model associated with the device and the building block of Figure 3.5. The device block represents the metadata about the device like the model, brand and EPREL registration number. This number is a unique identifier for a device model in the EPREL database, that is obtained by scanning the QR code in the energy label.

The building entity represents the metadata about the location of the device, like the designation. This block contains the space, floor, building and site entities. Each of these entities has a many to one relationship, for example, a space can be a kitchen on the second floor, the building is a house and the space a city.

The relationship between these two entities is one or many to one since one building can have one or many devices.

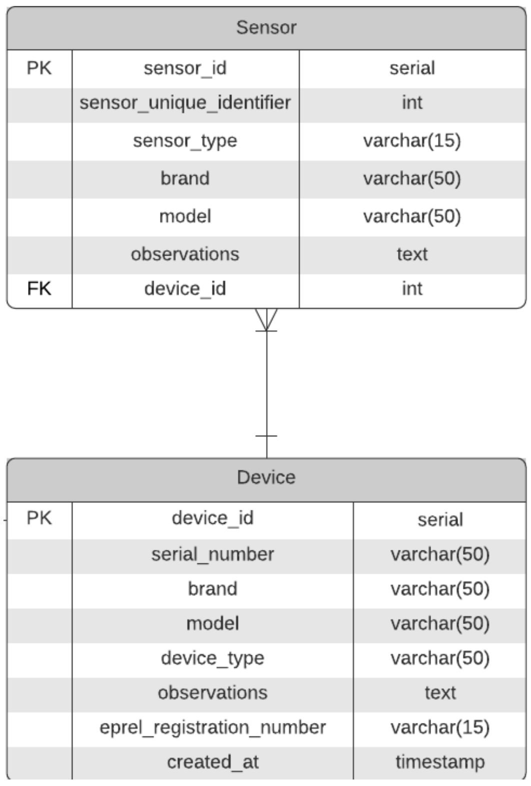


Figure 3.7: ER model related with the Device and Sensor

The sensor entity, showed in Figure 3.7, represents the sensors metadata like observations, sensor type , brand and model. It also contains the sensor unique identifier, which is an id used to identify the sensor in the SATO platform.

The relationship between these two entities is one to many, so one device can have one or multiple sensors.

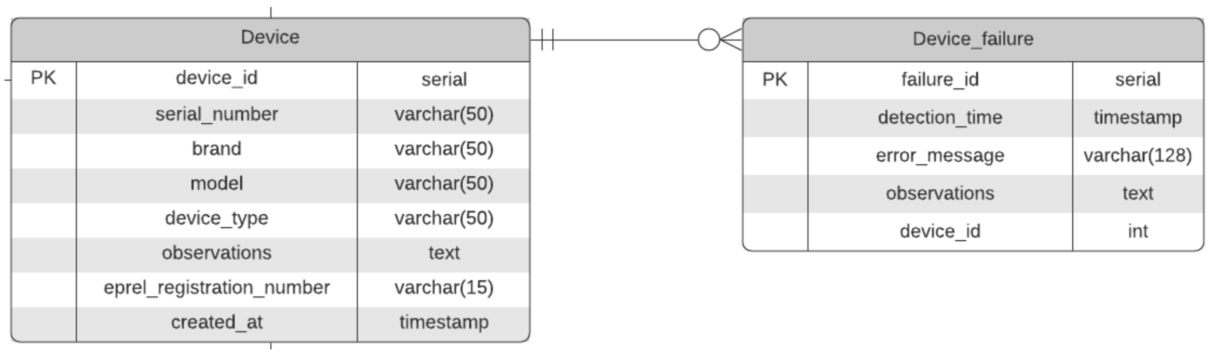
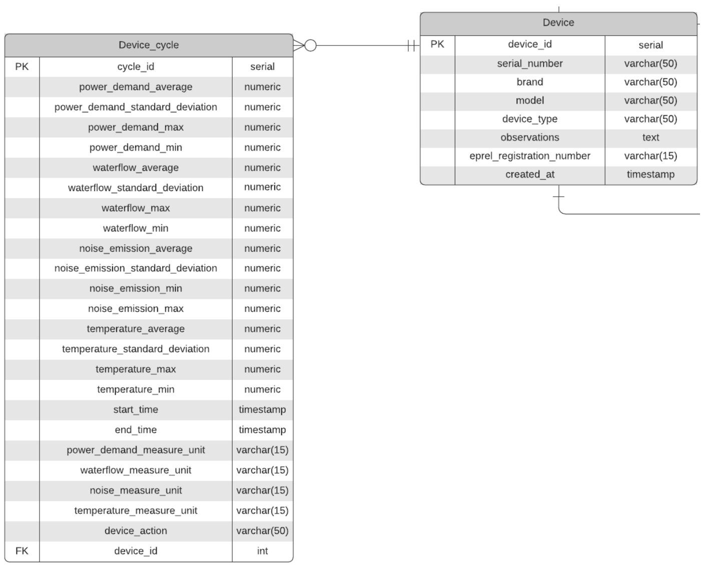


Figure 3.8: ER model related with the Device and Failure

The device failure entity, showed in Figure 3.8, represents the failures metadata, like when was it detected ($detection\_time$), an error message and observations. However, since the failure detection part was not possible to implement there may be some missing information that is needed for the system.

The relationship between these two entities is many to one, but it is not mandatory that a device has failures.



Device_cycle		
PK	cycle_id	serial
	power_demand_average	numeric
	power_demand_standard_deviation	numeric
	power_demand_max	numeric
	power_demand_min	numeric
	waterflow_average	numeric
	waterflow_standard_deviation	numeric
	waterflow_max	numeric
	waterflow_min	numeric
	noise_emission_average	numeric
	noise_emission_standard_deviation	numeric
	noise_emission_min	numeric
	noise_emission_max	numeric
	temperature_average	numeric
	temperature_standard_deviation	numeric
	temperature_max	numeric
	temperature_min	numeric
	start_time	timestamp
	end_time	timestamp
	power_demand_measure_unit	varchar(15)
	waterflow_measure_unit	varchar(15)
	noise_measure_unit	varchar(15)
	temperature_measure_unit	varchar(15)
	device_action	varchar(50)
FK	device_id	int

Device		
PK	device_id	serial
	serial_number	varchar(50)
	brand	varchar(50)
	model	varchar(50)
	device_type	varchar(50)
	observations	text
	eprel_registration_number	varchar(15)
	created_at	timestamp

Figure 3.9: ER model related with the Device and Cycle metrics

The device cycle metrics entity, showed in Figure 3.9, represents the metrics that are calculated for a device during each running cycle, like the average (e.g. $power\_demand\_average$), max (e.g. $temperature\_max$), min (e.g. $noise\_emission\_min$) and standard deviation (e.g. $waterflow\_standard\_deviation$). Additionally, there is also some metadata about the cycle like the start time, end time and measurement unit used (e.g. $temperature\_measure\_unit$).

The relationship between these two entities is one to many, but it is not mandatory that a device has a device cycle.

Finally, part of the device type metrics and metadata entity is showed in Figure 3.10. The entity represents the metrics and metadata that comes from EPREL for the device types that have an energy label. The data for each type was defined to be equivalent to the EPREL side. Additionally, devices without EU energy labels are also supported having the most common metrics from all the device types. Using the sensor data, some of the metrics in this entity (e.g. $energy\_class\_sdr$, $energy\_class$, $power\_demand\_standy$) are also calculated and stored, in order to be compared with the metrics of EPREL.

The relationship between the device type metrics and the device is done through the entity $Device\_data\_source$, that it has the attribute $created\_at$ to register the inserted time. Additionally, it has the attribute $source$ to distinguish between the real-life metrics, with the value $sensors$ and the EPREL metrics, with the value $EPREL$. The relation-

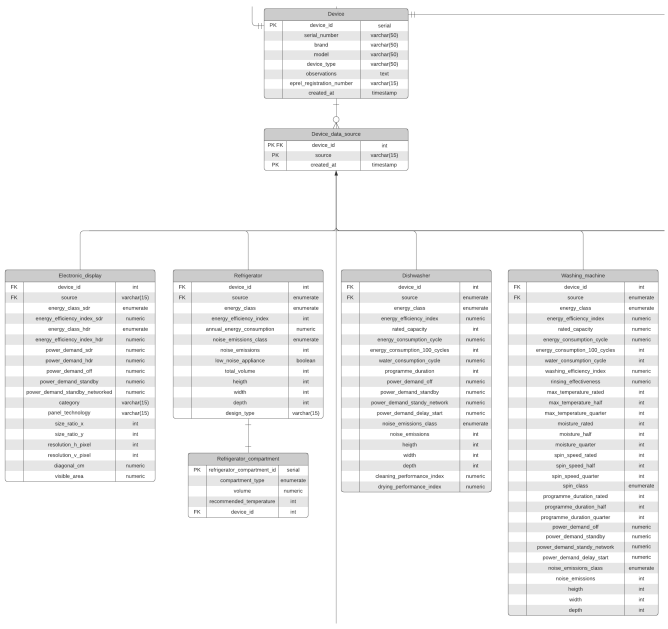

Device
PK device_id serial
serial_number varchar(50)
brand varchar(50)
model varchar(50)
device_type varchar(50)
observations text
eprel_registration_number varchar(15)
created_at timestamp
Device_data_source
PK FK device_id int
PK source varchar(15)
PK created_at timestamp
Electronic_display
FK device_id int
FK source varchar(15)
energy_class_sdr enumerate
energy_efficiency_index_sdr numeric
energy_class_hdr enumerate
energy_efficiency_index_hdr numeric
power_demand_sdr numeric
power_demand_hdr numeric
power_demand_off numeric
power_demand_standby numeric
power_demand_standby_networked numeric
category varchar(15)
panel_technology varchar(15)
size_ratio_x int
size_ratio_y int
resolution_h_pixel int
resolution_v_pixel int
diagonal_cm numeric
visible_area numeric
Refrigerator
FK device_id int
FK source enumerate
energy_class enumerate
energy_efficiency_index int
annual_energy_consumption numeric
noise_emissions_class enumerate
noise_emissions int
low_noise_appliance boolean
total_volume int
heigth int
width int
depth int
design_type varchar(15)
Refrigerator_compartment
PK refrigerator_compartment_id serial
compartment_type enumerate
volume numeric
recommended_temperature int
FK device_id int
Dishwasher
FK device_id int
FK source enumerate
energy_class enumerate
energy_efficiency_index numeric
rated_capacity int
energy_consumption_cycle numeric
energy_consumption_100_cycles int
water_consumption_cycle numeric
programme_duration int
power_demand_off numeric
power_demand_standby numeric
power_demand_standy_network numeric
power_demand_delay_start numeric
noise_emissions_class enumerate
noise_emissions int
heigth int
width int
depth int
cleaning_performance_index numeric
drying_performance_index numeric
Washing_machine
FK device_id int
FK source enumerate
energy_class enumerate
energy_efficiency_index numeric
rated_capacity numeric
energy_consumption_cycle numeric
energy_consumption_100_cycles int
water_consumption_cycle int
washing_efficiency_index numeric
rinsing_effectiveness numeric
max_temperature_rated int
max_temperature_half int
max_temperature_quarter int
moisture_rated int
moisture_half int
moisture_quarter int
spin_speed_rated int
spin_speed_half int
spin_speed_quarter int
spin_class enumerate
programme_duration_rated int
programme_duration_half int
programme_duration_quarter int
power_demand_off numeric
power_demand_standby numeric
power_demand_standy_network numeric
power_demand_delay_start numeric
noise_emissions_class enumerate
noise_emissions int
heigth int
width int
depth int

Figure 3.10: ER model related with the Device and Device type metrics

ship between the device and the device data source is one to many, so a device can have multiple instances of device type metrics.

3.5.5 Dashboards

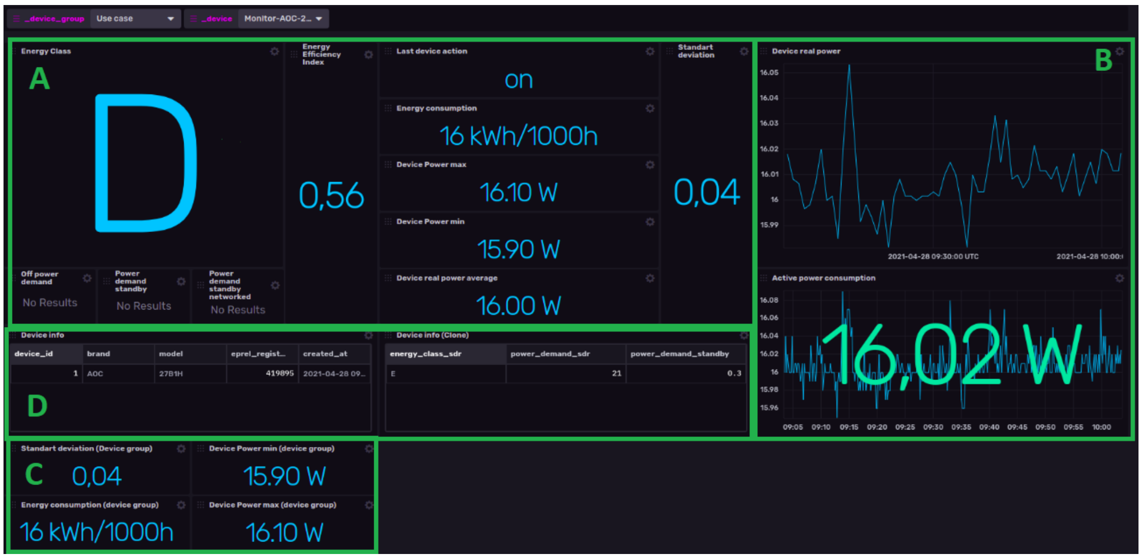
There are many ways to access the data stored in the database, for example, using dashboards. These dashboards are developed in InfluxDB using the Flux language to query the data, with the results being displayed using various visualization types. The access to these dashboards is done through a browser, but there also exists the possibility of developing dashboards in a mobile app using the same queries and the InfluxDB API. A user must also login using a username and password in order to be possible to manage the access permissions for the dashboards.

Consumption and metrics dashboard

The main goal of the consumption and metrics dashboard is to show the device real time consumption's, the calculations that are done to obtain the consumption metrics and the device meta-data. In Figure 3.11 it is possible to see an example of the dashboard for a monitor. It is possible to select what device group and device is showed using the dashboard variables on the top bar. Alongside with the sensor data that is stored on InfluxDB, it is also possible to visualize the device meta-data stored on the PostgreSQL database.

In this dashboard it is possible to visualize in Figure 3.11 - A, for a selected device, the energy class, energy efficiency index, the last device action, energy consumption in kWh/1000h, max power demand, min power demand, average power demand and the power demand standard deviation. There are also two graphs in Figure 3.11 - B, one for the device real power demand and for the device power demand when turned on.

Additionally, it is also possible to visualize in Figure 3.11 - C, for all devices from the device group of the selected device, the energy consumption in kWh/1000h, max power demand, min power demand and power demand standard deviation. Finally, it is also possible to see in Figure 3.11 - D the device metadata and device EU energy label metrics, all obtained from Metrics and metadata device DB.


_device_group: Use case
_device: Monitor-AOC-2...

Energy Class: D
Energy Efficiency Index: 0,56
Last device action: on
Energy consumption: 16 kWh/1000h
Device Power max: 16.10 W
Device Power min: 15.90 W
Device real power average: 16.00 W
Standart deviation: 0,04
Off power demand: No Results
Power demand standby: No Results
Power demand standby networked: No Results

Device info

device_id	brand	model	eprel_regist...	created_at
1	AOC	27B1H	419895	2021-04-28 09...

Device info (Clone)

energy_class_sdr	power_demand_sdr	power_demand_standby
E	21	0.3

Standart deviation (Device group): 0,04
Device Power min (device group): 15.90 W
Energy consumption (device group): 16 kWh/1000h
Device Power max (device group): 16.10 W

Active power consumption: 16,02 W

Figure 3.11: Consumption and metrics dashboard

Metrics comparison dashboard

The main goal of the metrics comparison dashboard (Figure 3.12) is to provide a comparison between the expected consumption metrics, obtained from the EPREL database, and the real-life consumption metrics, obtained from the sensor data. This comparison must be simple and easy to understand by any user, since it will be accessible by the occupants of the house that may have different levels of understandability of a device consumption metrics.

In this dashboard the data visualized depends on the device type of the selected device, since the metrics that were calculated change depending on the device type. All the calculations done in order to obtain these results on the dashboard are presented in Section 4.3. The metrics visualized are divided into two groups, the EU energy label metrics, on the left, and the metrics obtained from the sensor data, on the right. In this example the selected device is an electronic display and the metrics showed are the energy class, power demand in standard dynamic range, power demand in off mode and power demand in standby mode.

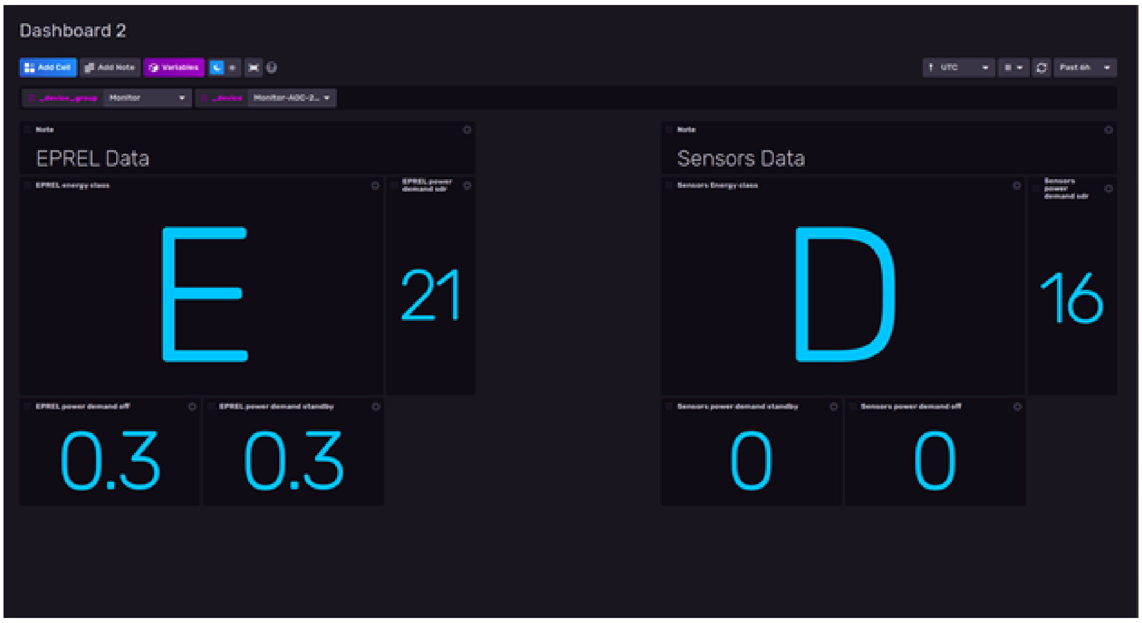


Figure 3.12: Metrics comparison dashboard

3.6 Summary

The section presented the requirements for a database used for storing sensor data, a comparison between multiple database engines and the choice made for this work. Additionally, it also presented the database for storing metrics and metadata. Finally, there was an overview and description of the SATO architecture, how the database components developed in this work are integrated and a description of each component.

Chapter 4

Database system implementation

This chapter presents the details about the process of fetching the data collected from the EPREL database and from the sensors. Also presented is the process and implementation of the consumption metrics calculation, for the device metrics and cycle metrics. Additionally, it also presents a description of how a failure detection could be implemented and the implementation and available calls for the external API. Finally, it presents a comparison between the calculated real-life metrics for two devices and their corresponding values in the energy label.

4.1 Databases implementation

The initial step in the implementation was the creation of the databases in order to start receiving and storing data. As already mentioned, the Sensor DB is implemented using InfluxDB and the Metrics and metadata DB using PostgreSQL.

4.1.1 Sensor DB

This creation of this database follows a simple process, since a data schema is not needed. The only required operations to create this database is the creation of an admin user and a bucket to store the data. When creating a bucket, the only parameters are the name and retention period. All these operations, as well as the setup of the dashboards and tasks, was all done through the InfluxDB interface. A command line interface is also available, despite being currently limited in some operations due to recent release of the InfluxDB version being used.

4.1.2 Metrics and metadata DB

This database was created using the administration and development platform pgAdmin [37]. After the creation of a database using this platform, a SQL script is executed with the creation of all the tables and enumerated types. This script is in Appendix B.1.

4.2 Data insertion

4.2.1 EPREL data insertion

The implementation of the EPREL data insertion is divided in two scripts, one to collect the data using the EPREL API ($eprel\_kafka\_producer.py$), in Appendix B.2, and send it via Kafka acting as a producer and the other script ($eprel\_kafka\_consumer.py$), in Appendix B.3, consumes the data and stores in the metrics and metadata device DB.

The data stored on the EPREL database is fetched by using the request call *Get product group's model by registration number*, referenced in Section 2.2. The call must have the format:

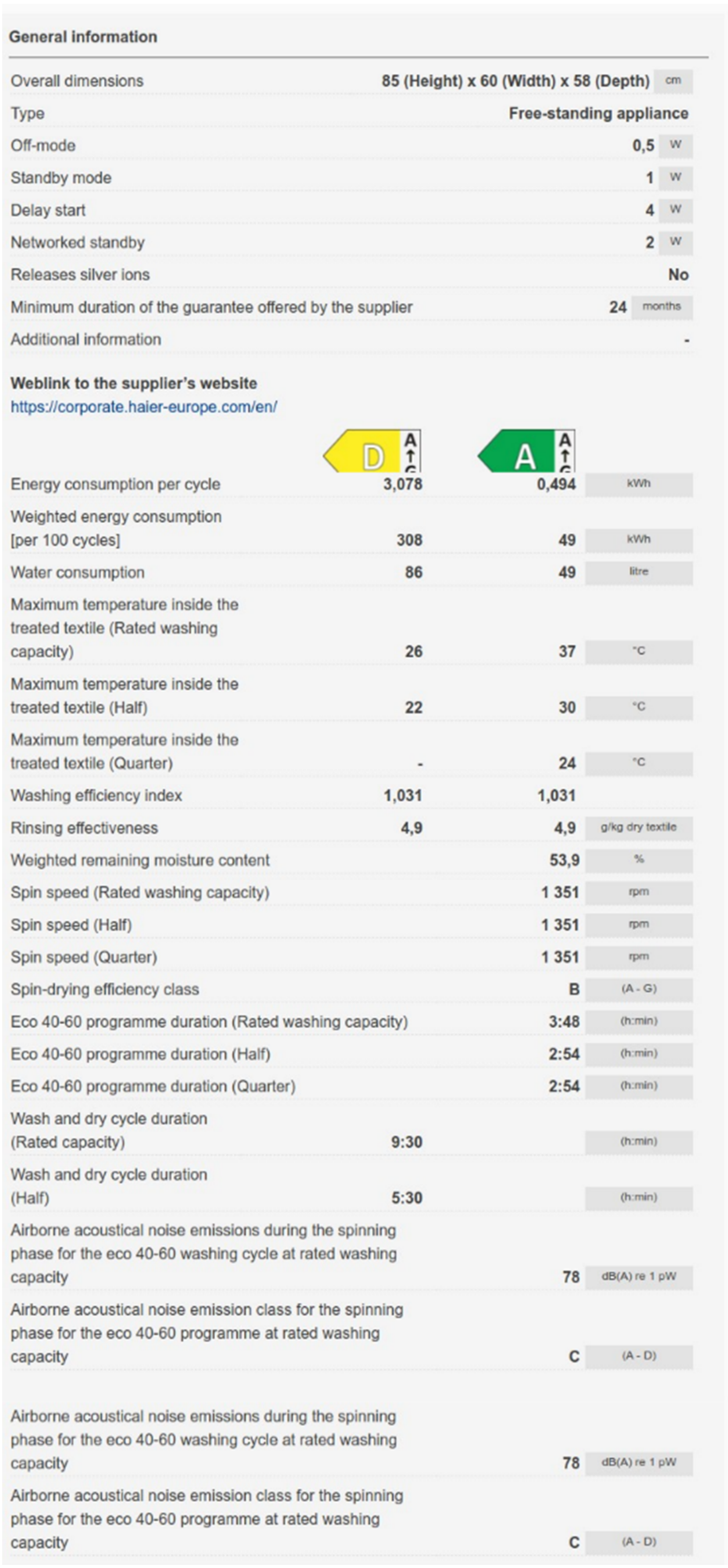
https://eprel.ec.europa.eu/api/products/[PRODUCT_GROUP]/[REGISTRATION_NUMBER]

The PRODUCT_GROUP field must be a valid product group code. The REGISTRATION_NUMBER field is the model registration number used in the EPREL database as a unique identifier for the model [31]. To obtain this number it is necessary to scan the QR code on the energy label, which redirects to the EPREL public website that gives the registration number of the model in the corresponding URL.

For example, for the case of a washer drier energy label (right part of Figure 4.1), the URL for the model page is:

https://eprel.ec.europa.eu/screen/product/washerdriers2019/300268

The data from EPREL will be fetched when a device is first registered in the SATO platform, by using the previously described call. The output of the call is shown in Figure 4.1.



General information

Overall dimensions	85 (Height) x 60 (Width) x 58 (Depth)	cm
Type	Free-standing appliance	
Off-mode	0,5	W
Standby mode	1	W
Delay start	4	W
Networked standby	2	W
Releases silver ions	No	
Minimum duration of the guarantee offered by the supplier	24	months
Additional information	-	

Weblink to the supplier's website
https://corporate.haier-europe.com/en/

	D	A	
Energy consumption per cycle	3,078	0,494	kWh
Weighted energy consumption [per 100 cycles]	308	49	kWh
Water consumption	86	49	litre
Maximum temperature inside the treated textile (Rated washing capacity)	26	37	°C
Maximum temperature inside the treated textile (Half)	22	30	°C
Maximum temperature inside the treated textile (Quarter)	-	24	°C
Washing efficiency index	1,031	1,031	
Rinsing effectiveness	4,9	4,9	g/kg dry textile
Weighted remaining moisture content		53,9	%
Spin speed (Rated washing capacity)		1 351	rpm
Spin speed (Half)		1 351	rpm
Spin speed (Quarter)		1 351	rpm
Spin-drying efficiency class		B	(A - G)
Eco 40-60 programme duration (Rated washing capacity)		3:48	(h:min)
Eco 40-60 programme duration (Half)		2:54	(h:min)
Eco 40-60 programme duration (Quarter)		2:54	(h:min)
Wash and dry cycle duration (Rated capacity)	9:30		(h:min)
Wash and dry cycle duration (Half)	5:30		(h:min)
Airborne acoustical noise emissions during the spinning phase for the eco 40-60 washing cycle at rated washing capacity		78	dB(A) re 1 pW
Airborne acoustical noise emission class for the spinning phase for the eco 40-60 programme at rated washing capacity		C	(A - D)
Airborne acoustical noise emissions during the spinning phase for the eco 40-60 washing cycle at rated washing capacity		78	dB(A) re 1 pW
Airborne acoustical noise emission class for the spinning phase for the eco 40-60 programme at rated washing capacity		C	(A - D)

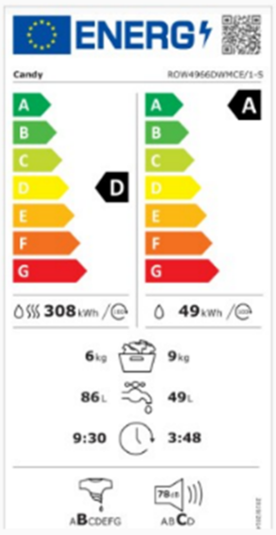


Figure 4.1: Example output of an EPREL database API call. Left: assessment data of the washer drier. Right: the corresponding EU energy label

The producer script can be executed with the following command:

python3 $device\_type$ $device\_brand$ $device\_model$

And it executes the following steps:

1. Receive the device type, brand and model in the command line arguments.
2. Get the device registration number through the device brand and model. This is done using a Python dictionary with the brand and model as keys and the registration number as value, that was previously manually created.
3. Make a request to the EPREL API call *Get product group's model by registration number* using the device type and registration number as arguments and receive the response has a JSON object.
4. Inject the response as an event in the Streaming Data component of the SATO platform, implemented using Kafka. This event contains a header with a field identifying that it is an EPREL event and the whole response (unmodified) is placed in a field called payload (outside the header). The event is associated with the topic "eprel_data".

The consumer script executes the following steps:

1. While the script is running is consuming events from the topic $eprel\_data$ and extracting the payload
2. This data is then processed and inserted in the Metrics and metadata device DB

4.2.2 Sensor data insertion

The implementation of the sensor data insertion is done using a Python script ($insert\_sensor\_data.py$) that reads a XML file with the sensor data and insert it in the Sensor database.

The script executes the following steps:

1. An Influx client instance is created using the InfluxDB-Python library.
2. The file name of the XML file with the sensor data, is set manually.
3. The file is opened, the contents turned into a Python dictionary object and the device type, brand and model recovered. The brand and model are then joined to form the $device\_name$, a variable that is then used in the dashboard for the user to easily identify the device.

4. The *device id* for the device data entry in the Metrics and metadata device DB is recovered, using the device brand and model in the query fields to filter the results.
5. The device metadata is collected from the device file and joined with the device id obtained in the step 4.
6. The XML file is formatted in order to be inserted in the database.
7. For each line a Influx point is created with the sensor data and the device metadata, including for example the device name, collection timestamp, power value and $deviceid$. Then each point is inserted.

4.3 Consumption metrics calculation

This section presents how the consumption metrics, for each device types where data was available (e.g. electronic displays), are calculated using the Flux language. All the formulas and tables used were collected from the EU regulation for labelling, more specifically, the Regulation on energy labelling for electronic displays (EU) 2019/2013 [41] and the Regulation on energy labelling for fridges and freezers (EU) 2019/2016 [43].

4.3.1 Device metrics

A set of consumption metrics, the same that are on the corresponding energy label, are calculated on a defined schedule, currently every day, using InfluxDB tasks and then inserted in the Metrics and Metadata database. The set of metrics depends on the device type, so each device type has its own script (e.g $load\_electronic\_display.flux$, $load\_refrigerator.flux$). These metrics are used in the comparison of the real-life device consumption with the expected consumption that are present on the device energy label.

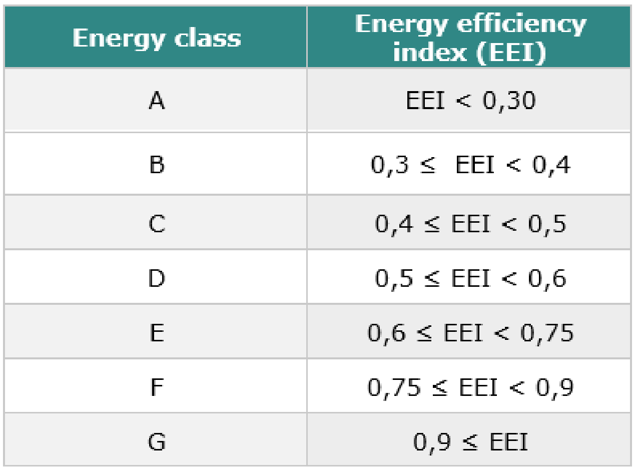
This subsection details the metrics that are calculated the device type, namely the electronic displays, fridges and untyped devices.

Electronic displays

Energy class SDR/HDR: In the case of electronic displays, the energy class is given when the display is in SDR (Standard display range) or in HDR (High dynamic range). Since the data available was for a screen with no HDR mode this was not considered during the implementation. The device energy class is given accordingly to the EEI (Energy efficiency index). The association can be seen in Figure 4.2:

To obtain the EEI the following formula is used:

$$EEI_{label} = \frac{(P_{measured} + 1)}{(3 * [90h(0,025 + 0,0035 + (A - 11) + 4)] + 3) + corr_i)}$$



Energy class	Energy efficiency index (EEI)
A	EEI < 0,30
B	0,3 ≤ EEI < 0,4
C	0,4 ≤ EEI < 0,5
D	0,5 ≤ EEI < 0,6
E	0,6 ≤ EEI < 0,75
F	0,75 ≤ EEI < 0,9
G	0,9 ≤ EEI

Figure 4.2: Energy class for electronic displays

where:

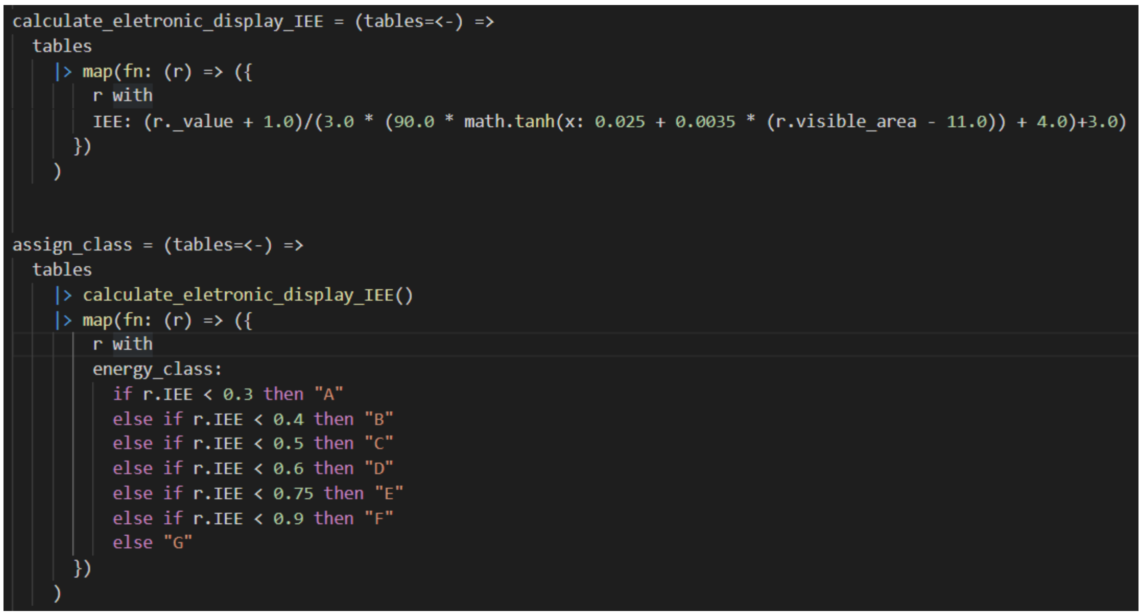
A represents the viewing surface are in dm^2;

$P_{measured}$ is the measured power for the normal configuration, expressed in Watts;

and $corr_i$ is a correction factor

In this current version the $coor_l$ is set to 0, since that is the value for both televisions and monitors. This value only changes when working with digital signage, like outdoor panels.

The correspondent Flux code is shown in Figure 4.3.



```
calculate_eletronic_display_IEE = (tables=<-) =>
  tables
    |> map(fn: (r) => ({
        r with
        IEE: (r._value + 1.0)/(3.0 * (90.0 * math.tanh(x: 0.025 + 0.0035 * (r.visible_area - 11.0)) + 4.0)+3.0)
      })
    )


assign_class = (tables=<-) =>
  tables
    |> calculate_eletronic_display_IEE()
    |> map(fn: (r) => ({
        r with
        energy_class:
          if r.IEE < 0.3 then "A"
          else if r.IEE < 0.4 then "B"
          else if r.IEE < 0.5 then "C"
          else if r.IEE < 0.6 then "D"
          else if r.IEE < 0.75 then "E"
          else if r.IEE < 0.9 then "F"
          else "G"
      })
    )
```

Figure 4.3: Flux code for the assignment of the energy class for electronic display

Power demand SDR/HDR: Since the data available was for a screen with no HDR mode this was not considered during the implementation.

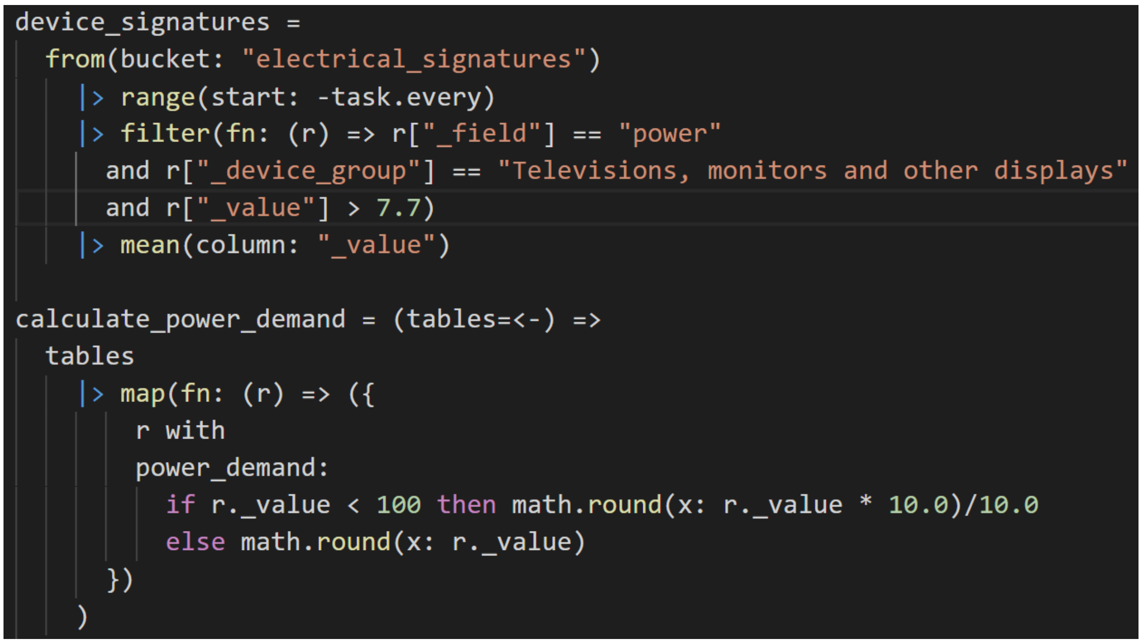
In the case of electronic displays, the power demand is measured in Watts. Since the sensor also detects the power in Watts, the only calculations needed were the average and the rounding the result, accordingly to the rules used in the energy label.

Since this power demand metric is set with the device turned on, the data must be filtered. This is done using the maximum consumptions, set by the regulations of the European Union, for each mode. These limits are:

- Off mode with a maximum of 0,30 Watts
- Standby mode with a maximum of 2,20 Watts
- Standby networked mode with a maximum of 7,70 Watts

It was decided to consider that consumptions above 7,70 Watts refers to a device turned on.

The correspondent Flux code is shown in Figure 4.4.



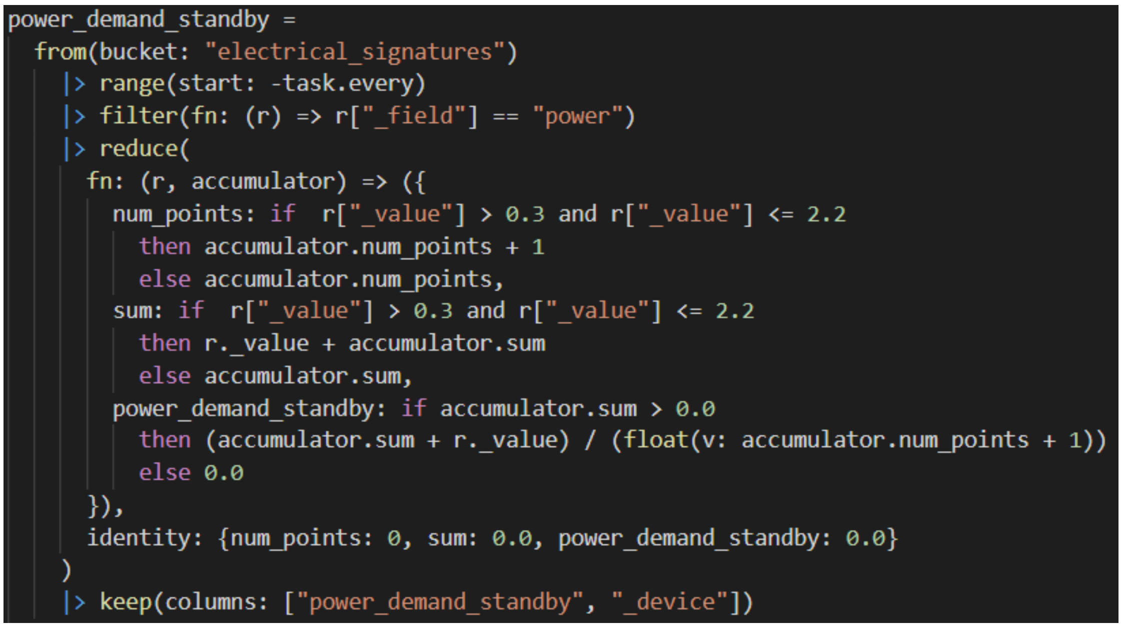
```
device_signatures =
  from(bucket: "electrical_signatures")
    |> range(start: -task.every)
    |> filter(fn: (r) => r["_field"] == "power"
      and r["_device_group"] == "Televisions, monitors and other displays"
      and r["_value"] > 7.7)
    |> mean(column: "_value")

calculate_power_demand = (tables=<-) =>
  tables
    |> map(fn: (r) => ({
        r with
        power_demand:
          if r._value < 100 then math.round(x: r._value * 10.0)/10.0
          else math.round(x: r._value)
      })
    )
```

Figure 4.4: Flux code for the calculation of the power demand for electronic display

Power demand standby: The procedure to calculate the power demand in standby mode is the same for the power demand with the device turned on. In the beginning, the data is filtered for the power above 0,3 Watts and bellow 2,2 Watts and then the average is calculated.

The correspondent Flux code is shown in Figure 4.5.

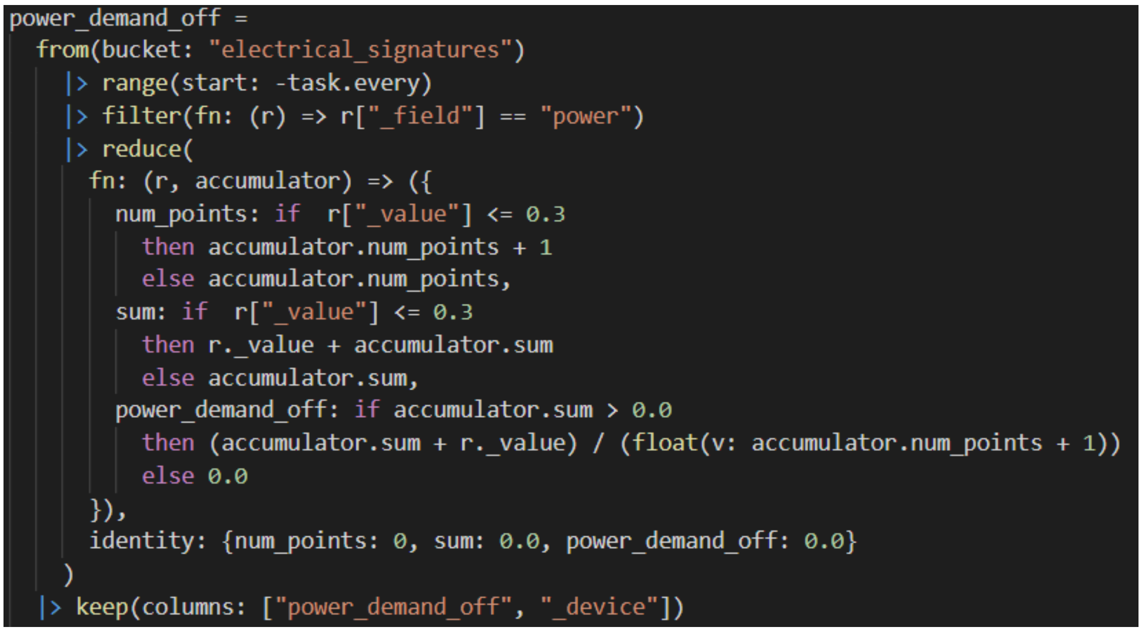


```
power_demand_standby =
  from(bucket: "electrical_signatures")
    |> range(start: -task.every)
    |> filter(fn: (r) => r["_field"] == "power")
    |> reduce(
      fn: (r, accumulator) => ({
        num_points: if  r["_value"] > 0.3 and r["_value"] <= 2.2
          then accumulator.num_points + 1
          else accumulator.num_points,
        sum: if  r["_value"] > 0.3 and r["_value"] <= 2.2
          then r._value + accumulator.sum
          else accumulator.sum,
        power_demand_standby: if accumulator.sum > 0.0
          then (accumulator.sum + r._value) / (float(v: accumulator.num_points + 1))
          else 0.0
      }),
      identity: {num_points: 0, sum: 0.0, power_demand_standby: 0.0}
    )
    |> keep(columns: ["power_demand_standby", "_device"])
```

Figure 4.5: Flux code for the calculation of the power demand in standby mode for electronic display

Power demand off: The procedure to calculate the power demand in off mode is the same as the power demand with the device turned on. In the beginning, the data is filtered for the power bellow 0,3 Watts and then the average is calculated.

The correspondent Flux code is shown in Figure 4.6.



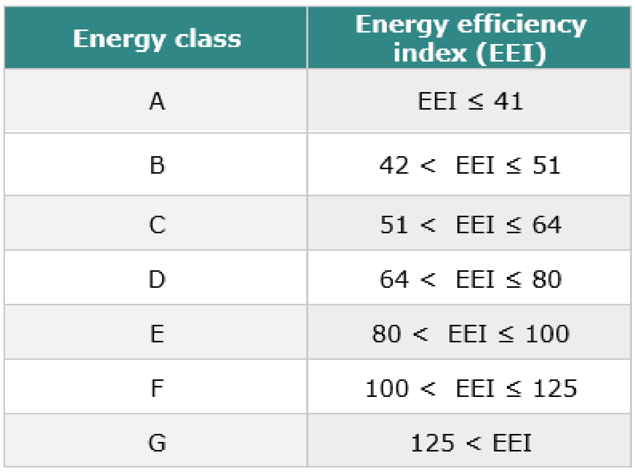
```
power_demand_off =
  from(bucket: "electrical_signatures")
    |> range(start: -task.every)
    |> filter(fn: (r) => r["_field"] == "power")
    |> reduce(
      fn: (r, accumulator) => ({
        num_points: if  r["_value"] <= 0.3
          then accumulator.num_points + 1
          else accumulator.num_points,
        sum: if  r["_value"] <= 0.3
          then r._value + accumulator.sum
          else accumulator.sum,
        power_demand_off: if accumulator.sum > 0.0
          then (accumulator.sum + r._value) / (float(v: accumulator.num_points + 1))
          else 0.0
      }),
      identity: {num_points: 0, sum: 0.0, power_demand_off: 0.0}
    )
  |> keep(columns: ["power_demand_off", "_device"])
```

Figure 4.6: Flux code for the calculation of the power demand in off mode for electronic display

Power demand networked standby: Since the data available was for a screen with no networked standby mode, this metric was not considered during the implementation.

Fridges and refrigerators devices

Energy class: The device energy class is given accordingly to the EEI. The limits of each class can be seen in Figure 4.7.



Energy class	Energy efficiency index (EEI)
A	EEI ≤ 41
B	42 < EEI ≤ 51
C	51 < EEI ≤ 64
D	64 < EEI ≤ 80
E	80 < EEI ≤ 100
F	100 < EEI ≤ 125
G	125 < EEI

Figure 4.7: Energy class for fridges and refrigerators devices

The EEI is expressed in $\%$ and rounded to the first decimal place, is obtained employing the following formula:

$$EEI = AE/SAE$$

where:

AE is the annual energy consumption is expressed in kilowatts hour per year (kWh/annum)

SAE is the standard energy consumption is expressed in kilowatts hour per year (kWh/annum)

The SAE is calculated using the following formula.

$$SAE = C * D * \sum_{c=1}^{n} A_c * B_c * [V_c/V] * (N_c + V * r_c * M_c)$$

where:

c is the index number for a compartment type ranging from 1 to n, with n the total number of compartments type;

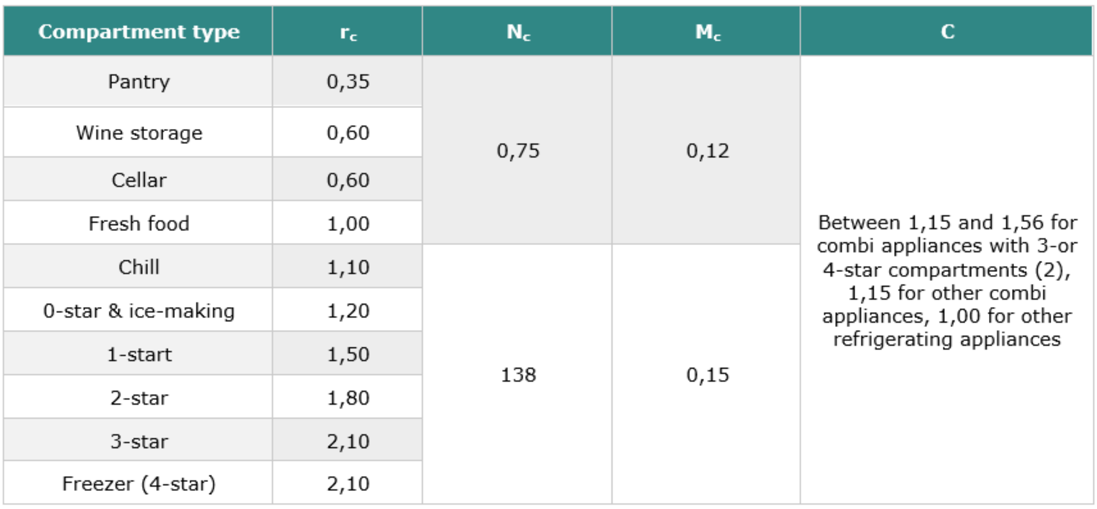
V_c, expressed in dm^3 or litres and rounded to the first decimal place is the compartment volume;

V, expressed in dm^3 or litres and rounded to the nearest integer is the volume with $V \leq \sum_{c=1}^{n} V_c$;

r_c, N_c, M_c and C are modelling parameters specific to each compartment

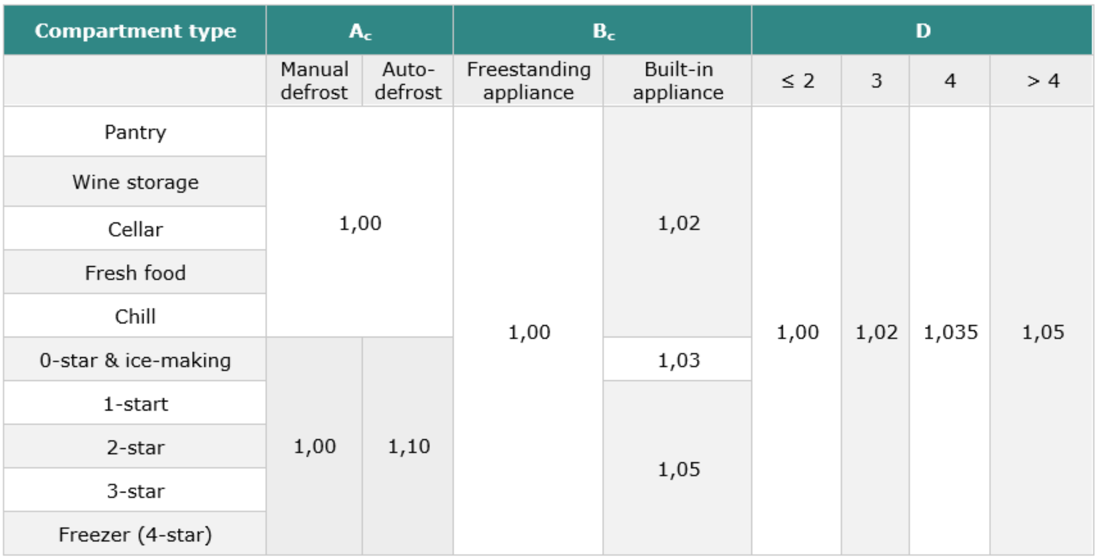
A_c, B_c and D are the compensation factors with values.

The tables in Figures 4.8 and 4.9, are used for assigning the variables values accordingly to the compartment type.



Compartment type	r_c	N_c	M_c	C
Pantry	0,35	0,75	0,12	Between 1,15 and 1,56 for combi appliances with 3-or 4-star compartments (2), 1,15 for other combi appliances, 1,00 for other refrigerating appliances
Wine storage	0,60			
Cellar	0,60			
Fresh food	1,00			
Chill	1,10	138	0,15	
0-star & ice-making	1,20			
1-start	1,50			
2-star	1,80			
3-star	2,10			
Freezer (4-star)	2,10			

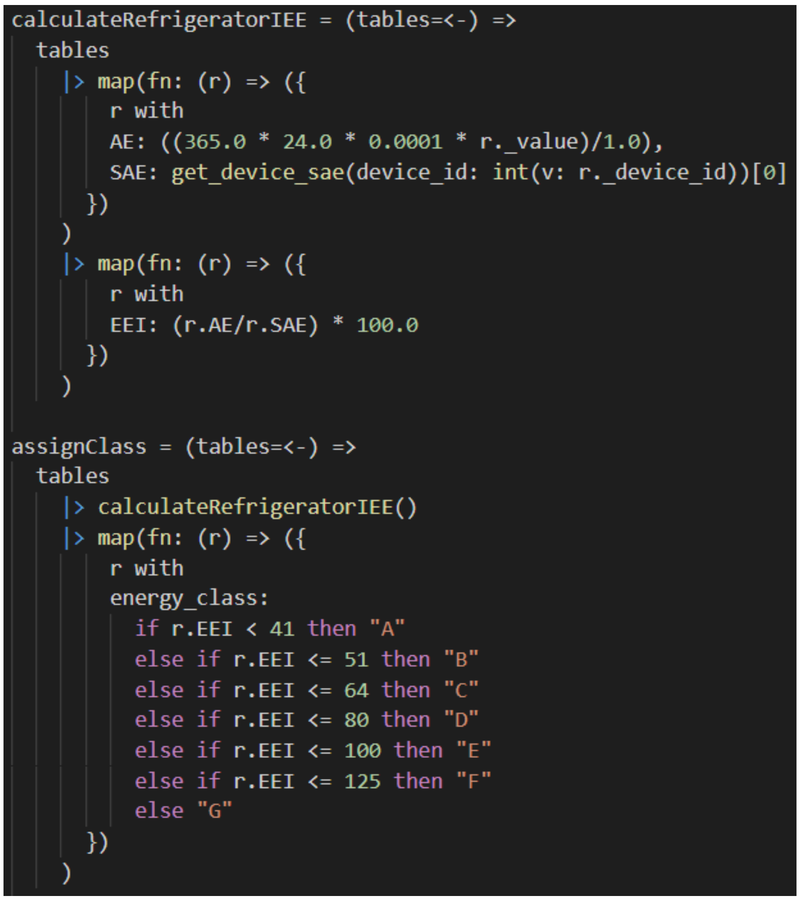
Figure 4.8: Variable assignment table for fridges and refrigerators [42]



Compartment type	A_c		B_c		D			
	Manual defrost	Auto-defrost	Freestanding appliance	Built-in appliance	≤ 2	3	4	> 4
Pantry	1,00		1,00	1,02	1,00	1,02	1,035	1,05
Wine storage								
Cellar								
Fresh food								
Chill								
0-star & ice-making	1,00	1,10		1,03				
1-start				1,05				
2-star								
3-star								
Freezer (4-star)								

Figure 4.9: Variable assignment table for fridges and refrigerators [42]

The Flux code to calculate the EEI is shown in Figure 4.10.

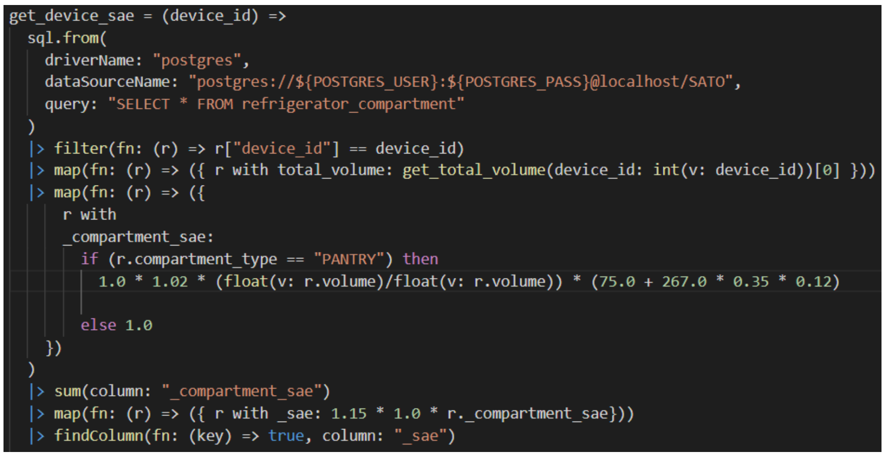


```
calculateRefrigeratorIEE = (tables=<-) =>
  tables
    |> map(fn: (r) => ({
        r with
        AE: ((365.0 * 24.0 * 0.0001 * r._value)/1.0),
        SAE: get_device_sae(device_id: int(v: r._device_id))[0]
      })
    )
    |> map(fn: (r) => ({
        r with
        EEI: (r.AE/r.SAE) * 100.0
      })
    )

assignClass = (tables=<-) =>
  tables
    |> calculateRefrigeratorIEE()
    |> map(fn: (r) => ({
        r with
        energy_class:
          if r.EEI < 41 then "A"
          else if r.EEI <= 51 then "B"
          else if r.EEI <= 64 then "C"
          else if r.EEI <= 80 then "D"
          else if r.EEI <= 100 then "E"
          else if r.EEI <= 125 then "F"
          else "G"
      })
    )
```

Figure 4.10: Flux code for the assignment of the energy class for fridges and refrigerators

The Flux code to calculate the SAE is in Figure 4.11 (code altered to provide better readability).



```
get_device_sae = (device_id) =>
  sql.from(
    driverName: "postgres",
    dataSourceName: "postgres://${POSTGRES_USER}:${POSTGRES_PASS}@localhost/SATO",
    query: "SELECT * FROM refrigerator_compartment"
  )
  |> filter(fn: (r) => r["device_id"] == device_id)
  |> map(fn: (r) => ({ r with total_volume: get_total_volume(device_id: int(v: device_id))[0] }))
  |> map(fn: (r) => ({
      r with
      _compartment_sae:
        if (r.compartment_type == "PANTRY") then
          1.0 * 1.02 * (float(v: r.volume)/float(v: r.volume)) * (75.0 + 267.0 * 0.35 * 0.12)

        else 1.0
    })
  )
  |> sum(column: "_compartment_sae")
  |> map(fn: (r) => ({ r with _sae: 1.15 * 1.0 * r._compartment_sae}))
  |> findColumn(fn: (key) => true, column: "_sae")
```

Figure 4.11: Flux code for the calculation of the SAE for fridges and refrigerators

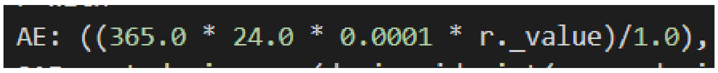
The AE calculations are presented in the sub section below.

Annual energy consumption: To obtain the AE the following formula is used

$$AE = 365 * E_{daily}/L + E_{aux}$$

The E_{daily} stands for the daily energy consumption and is expressed in kWh/24h. The L stands for the load factor and changes from 0,9 when the appliance contains only frozen compartments or 1,0 for all other appliances. The E_{aux} stands for the energy used by an anti-condensation heater. Since this value was not present in the data used the variable was not considered.

The Flux code to calculate the AE is in Figure 4.12.



```
AE: ((365.0 * 24.0 * 0.0001 * r._value)/1.0),
```

Figure 4.12: Flux code for the calculation of the AE for fridges and refrigerators

Sound class: Since there was no data available from the sound sensor, this metric was not considered during implementation.

Untyped devices

It is also important to support devices that have no correspondent type in the EPREL database, for example a laptop. Since these devices have no energy label, the metrics calculated are based on the most common ones. The metrics considered were: Power demand on mode, Power demand off mode, Power demand standby mode, Energy annual consumption.

4.3.2 Cycle metrics

Besides the consumption metrics a set of metrics are calculated after every device cycle, also using InfluxDB tasks. A device cycle can be defined as a period where the device is considered turned on and, consequently, with the consumption's rising. There are two types of devices, the ones with alternate cycles, that is, a device that is turned off and on multiple times over its normal operation like a television or a dishwasher, and the ones with a continuous cycle that are only turned on once, except for a device failure, like a fridge.

These metrics are a set of summary statistics, more specifically the average, min, max and standard deviation. These were the chosen metrics since they allow an analysis of the device consumptions over the entire cycle.

The cycle metrics collection script ($load_{d}evice_{c}ycle.flux$), in Appendix B.5 is executed every 30s in order to ensure that all cycles are accounted, no matter the execution duration. The script steps are as follows:

1. Check if the device already has the metrics for a previous device cycle. This is done by checking if there is a cycle in the Metrics and metadata DB with the same $device\_id$.

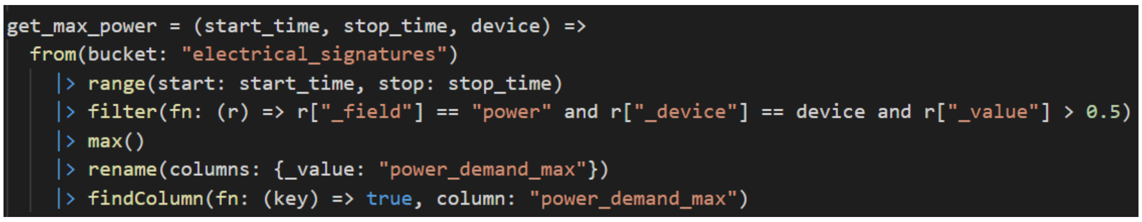
2. If no previous cycle exists, search for the first data point with a power value above 0,5, in all the sensor data available for the device. If a previous cycle exists only the entries since the end of that cycle are searched. This will define the start of the device cycle.

3. The last data point with a power value above 0,5 is then searched and defined as the end of the cycle.

4. Operations are made over all the data points between the start and end of the cycle to obtain the average, max, min and standard deviation.

5. The results are stored in the $cycle\_metrics$ table of the Metrics and Metadata device DB.

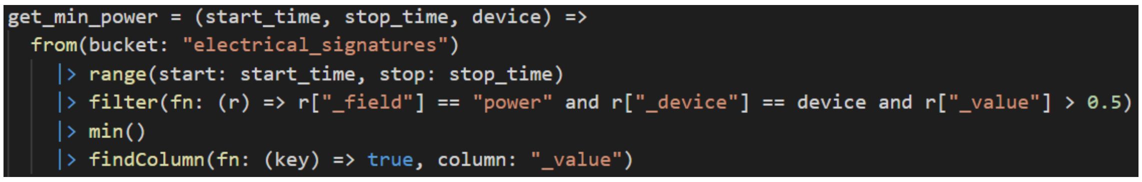
The code to calculate the max value is in Figure 4.13.



```
get_max_power = (start_time, stop_time, device) =>
  from(bucket: "electrical_signatures")
    |> range(start: start_time, stop: stop_time)
    |> filter(fn: (r) => r["_field"] == "power" and r["_device"] == device and r["_value"] > 0.5)
    |> max()
    |> rename(columns: {_value: "power_demand_max"})
    |> findColumn(fn: (key) => true, column: "power_demand_max")
```

Figure 4.13: Flux code for the calculation of the max value for untyped devices

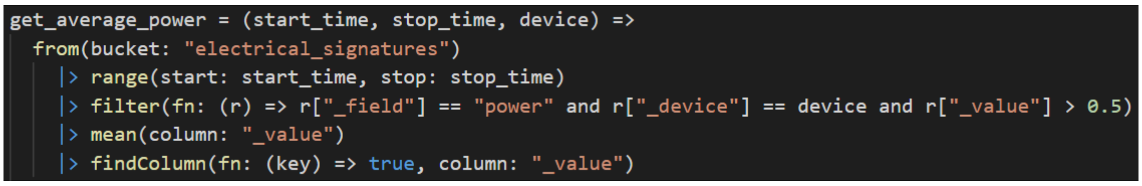
The code for the min value is in Figure 4.14.



```
get_min_power = (start_time, stop_time, device) =>
  from(bucket: "electrical_signatures")
    |> range(start: start_time, stop: stop_time)
    |> filter(fn: (r) => r["_field"] == "power" and r["_device"] == device and r["_value"] > 0.5)
    |> min()
    |> findColumn(fn: (key) => true, column: "_value")
```

Figure 4.14: Flux code for the calculation of the min value for untyped devices

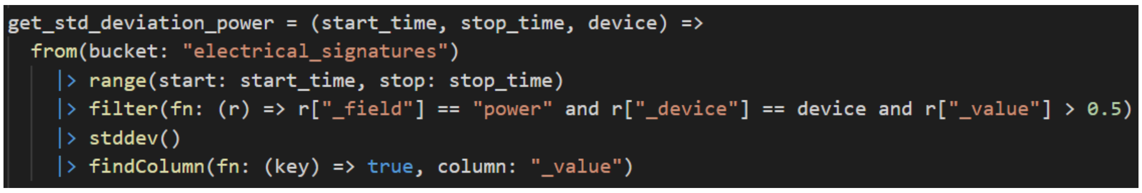
The code for the average value is in Figure 4.15.



```
get_average_power = (start_time, stop_time, device) =>
  from(bucket: "electrical_signatures")
    |> range(start: start_time, stop: stop_time)
    |> filter(fn: (r) => r["_field"] == "power" and r["_device"] == device and r["_value"] > 0.5)
    |> mean(column: "_value")
    |> findColumn(fn: (key) => true, column: "_value")
```

Figure 4.15: Flux code for the calculation of the average value for untyped devices

The code for the Standard deviation value is in Figure 4.16.



```
get_std_deviation_power = (start_time, stop_time, device) =>
  from(bucket: "electrical_signatures")
    |> range(start: start_time, stop: stop_time)
    |> filter(fn: (r) => r["_field"] == "power" and r["_device"] == device and r["_value"] > 0.5)
    |> stddev()
    |> findColumn(fn: (key) => true, column: "_value")
```

Figure 4.16: Flux code for the calculation of the standard deviation value for untyped devices

4.4 External API

A REST API is developed to give external access to the data stored in the databases. This API can be used in the SATO platform to supply the self-assessment framework with access to the real-life metrics that are then used for the assessments. There are four calls to access the data in the Metrics and metadata device DB and one call for the Sensor DB. The code of API is in Appendix B.6.

The REST API was developed using Python and the Flask framework. Flask was chosen due to being a highly extensible microframework. It also has a small number of dependencies to external libraries and it is not restricted to use a specific library, for example to access a specific type of database.

These characteristics are important in the development of this API, since it is not a API with an high number of accesses or with the need of security, so other libraries would bring unnecessary dependencies. Furthermore, since there are two libraries used to access the databases, $psycopg2$ for the Metrics and metadata device DB and influxdb_client for the Sensor DB, the extendibility component is important [17].

Additionally, Flask comes with a built-in server that is designed to be used during local development. However, it should not be used when deploying in a production environment, as in the case of this API being used in the SATO project, since it is not designed to be efficient, stable, or secure. Options for the deployment are using Gunicorn, uWSGI or Gevent [17].

For all the calls the response format can be a numeric value, a float, a JSON object, or an array of JSON objects. Each response also has a status code depending on the type of response, with 200 being used when the request is completed successfully and 404 when the request resource is not found.

The available calls are:

1. **Get device metrics** – Request used to get all metric values for a specific device. The format for the call is:
   /metrics/[device_serial_number]

   A call example is:
   http://127.0.0.1:5000/metrics/123

   A response example is:
   "category":null,"created_at":"Mon, 31 May 2021 22:24:32 GMT","device_id":3, "diagonal_cm":null,"energy_class_hdr":null,"energy_class_sdr":"D" ,"panel_technology":null,"power_demand_hdr":null ,"power_demand_off":0.0,"power_demand_sdr":16.0,"power_demand_standby":0.0, "power_demand_standby_networked":null,"resolution_h_pixel":null,"resolution_v_pixel":null, "size_ratio_x":null,"size_ratio_y":null,"source":"sensors","visible_area":null

2. **Get device single metric** – Request used to get a single metric value for a specific device. The format for the call is:
   /metrics/[device_serial_number]/[metric]

   A call example is:
   http://127.0.0.1:5000/metrics/123/energy_class_sdr

   A response example is: "D"

3. **Get device metadata** – Request used to get all the metadata for a specific device. The format for the call is:
   /metadata/[device_serial_number]

   A call example is:
   http://127.0.0.1:5000/metadata/123

   A response example is:
   "brand":"AOC","building_id":1,"created_at":"Mon, 31 May 2021 19:25:43 GMT" ,"device_id":3,"device_type":"ELECTRONIC_DISPLAY","eprel_registration_number":"419895" ,"model":"27B1H","observations":null,"serial_number":123

4. **Get device single metadata** – Request used to get a single metadata value for a specific device. The format for the call is:
/metadata/[device_serial_number]/[metadata]

   A call example is:
http://127.0.0.1:5000/metadata/123/brand
A response example is: ”AOC”

5. **Get device power data from last week** – Request used to get the device power consumption data points of the last week, for a specific model. The format for the call is:
/device_power/[device_brand]/[device_model]

   A call example is:
http://127.0.0.1:5000/device_power/AOC/27B1H

   A response example is:
*[”_device”:”Monitor-AOC-27B1-27”,”_device_action”:”on”,”_device_brand”:”AOC”
,”_device_group”:”Monitor”,”_device_id”:”3”,”_device_model”:”27B1H”,”_field”:”power”
,”_measurement”:”mem”,”_start”:”Mon, 30 Aug 2021 21:30:11 GMT”,”_stop”:”Mon,
06 Sep 2021 21:30:11 GMT”,”_time”:”Mon, 06 Sep 2021 21:18:51 GMT”,”_value”:16.0,
”result”:”_result”,”table”:0,
...
”_device”:”Monitor-AOC-27B1-27”,”_device_action”:”on”,”_device_brand”:”AOC”
,”_device_group”:”Monitor”,”_device_id”:”3”,”_device_model”:”27B1H”,”_field”:”power”
,”_measurement”:”mem”,”_start”:”Mon, 30 Aug 2021 21:30:11 GMT”,”_stop”:”Mon,
06 Sep 2021 21:30:11 GMT”,”_time”:”Mon, 06 Sep 2021 21:30:10 GMT”,”_value”:16.0
,”result”:”_result”,”table”:0]*

4.5 Failure detection

One of the goals of this work was to implement device failure detection. The device failure can be caused by a malfunction in the device that then leads to abnormal consumption’s values. This detection is designed to be implemented in a Worten use case. In this use case Worten provides the costumers with the option to take part of this use case and those who accept will be provided with a device with sensors.

This device failure detection will allow Worten to provide remote costumer support even before the failure is detected by the costumer.

The process for the failure detection could work as follows:

1. Compare the cycle metrics of a specific device with the metrics of other devices of the same model

2. If the metrics values differ in a large way flag it has a potential failure
3. Save details about this failure in the Metrics and Metadata device DB
4. Notify via email or third party applications the device owner and supplier

Unfortunately, the device failure detection was not possible to implement due to the lack of data, more specifically, from multiple devices of the same brand and model.

4.6 Validation

With the sensor and EPREL metrics stored in the Metrics and Metadata device DB, it is now possible to proceed with the comparison of the predicted and real-life consumption. This section presents the comparison of real-life assessment and EPREL data, to a monitor AOC 27B1H and a fridge Beko TS190030N as these are the only devices with sensor data available.

4.6.1 Monitor AOC 27B1H

This is a monitor with a 1920x1080 Full HD resolution on its 27" IPS panel. It also offers Flicker-Free and Low Blue Light technologies. The metrics obtained from the EPREL database (`https://eprel.ec.europa.eu/screen/product/electronicdisplays/419895`) for the Monitor AOC 27B1H are:

- Energy class - E
- On mode power demand in Standard Dynamic Range (SDR) - 21 W
- Off mode power demand - 0.3 W
- Standby mode power demand - 0.3 W

The metrics obtained from the sensor data are:

- Energy class - D
- On mode power demand in Standard Dynamic Range (SDR) - 16 W
- Off mode power demand - /
- Standby mode power demand - /

As already mentioned, the sensor data for this device was collected using the sensors that were as part of the thesis work of [25], that is also part of the SATO project. It is also from 1 continuous hour of collecting with the device always on. That is why there are no metrics for the power demand of both the off mode and the standby mode. As it can be seen the metrics obtained from the sensor data show a lower power consumption when compared to the EPREL metrics.

The results of the sensors data metrics were not the expected ones, since the expectations were for the power consumption in a real-life use case scenario to be above the performance displayed in energy label metrics, obtained from EPREL.

These expectations come from the difference between the scenario conditions when obtaining the data, since when calculating the EU energy label metrics, the conditions remain in a stable predefined state. For example, in the case of electronic displays a fixed brightness and contrast, which does not happen in real-life since a user may alter these parameters leading to an increase in the power demand.

However, in this case the change in conditions may also lead to this lower power consumption, since the settings of the screen during the period in which the data was collected could for example, have a lower brightness.

4.6.2 Fridge Beko TS190030N

This is a fridge with a total volume of 88L, a single door and LED lighting. The metrics obtained from the EPREL database (`https://eprel.ec.europa.eu/screen/product/refrigeratingappliances2019/341954`) for this device are:

- Energy class - F
- Energy efficiency Index (EEI) - 124
- Annual energy consumption - 106 kWh/annum

The metrics obtained from the sensor data are:

- Energy class - C
- Energy efficiency Index (EEI) - 63
- Annual energy consumption - 79 kWh/annum

As a reminder, the sensor data for this device was collected from the ACS-F2 dataset. However, the device from the dataset is from an equivalent fridge with the same number and type of compartments but from a different manufacturer. Despite not being a perfect comparison, it is still the best possible solution, since there is no device in the dataset that is still in the current market, therefore with the new EU energy label. It is also from 1 continuous hour of collecting with the device always on. As it can be seen the metrics obtained from the sensor data shows a lower energy consumption when compared to the EPREL metrics.

The metric results for the sensor data where not ideal mainly due to the small data size, with the data only being from 1 continuous hour when the ideal would be 1 week and the fact that it was not possible to simulate a normal use of the device.

However, it is still possible to reach the conclusion that the metric calculations are correctly implemented since the results are valid and, in the limits, stipulated in the EU labelling regulations. For example, the energy class is being correctly assigned according to the power demand in both device types.

4.7 Summary

This section presented the implementation details of the database system, starting with a description of the data insertions for the EPREL and sensor data. Then, it was presented a description of the process to calculate and implement, in the Flux language, the consumption and cycle metrics. Additionally, it was described the external API and how the failure detection can be implemented. Finally, the validation was presented with a comparison of the real-life metrics of two different devices with the values on the energy label.

Chapter 5

Conclusion and future work

This chapter presents the conclusions and the future work. The Section 5.1 summarizes the selection process of the databases engines, integration in the SATO project, the description of the dataflow and its components, implementation of the data insertion and consumption metrics calculation, implementation of the API and finally the comparison of the results. Section 5.2 presents the limitations of the current work as well as possible paths to overcome them in future work.

5.1 Conclusion

The main objective of this thesis was to provide a comparison between a device real-life consumption and the predicted consumption. The real-life consumption will provide the consumers with the knowledge they need to make better decisions when buying an appliance, discover which operating mode and options offers the lower consumption and even pre-emptively detect a possible device failure. This knowledge is much more valuable than the design predictions and since this comparison will be part of the SATO platform, it will be available for both new and legacy devices. The SATO platform performs self-assessments and optimizations of devices and buildings using the data stored in the databases developed on the work of this thesis.

The analysis of database engines to store sensor data made it possible to bring together various solutions for the implementation of the project. These where then compared using a set of application requirements as comparison criteria, namely, the data schema, query language, scalability, load balancing, data retention policies, among others. The database system used is InfluxDB, a time series database, because it provides the best performance when working with time series data, has already been used in projects that work with sensor data and provides all the resources for scaling.

Additionally, to store device metrics (e.g. energy class, power demand) and device metadata (e.g. brand, model and the EPREL energy label) the PostgreSQL relational database engine is used.

The work done in this thesis work makes available the databases where all the sensor, metrics and metadata can be stored and accessed. This includes a DB for the sensor data with the scripts needed for the metrics calculation and insertion in a relational database. A DB with both the sensor data metrics and EPREL metrics was also implemented, alongside with the scripts needed for integration in the SATO platform and an entity relationship diagram, including the future integration with the building assessments. In order to access the data stored in the databases, besides the use of queries, an API is made available as well as two different dashboards. This API was developed using the framework Python Flask.

For each device, a set of metrics is calculated, depending on the device type over a certain time frame. The metrics calculations are the same as the ones used in the EU labelling process, so they can then be compared with the ones in the device energy label. An additional set of metrics composed of summary statistics is also calculated during each device cycle. The implementation of this metrics calculations was done using Flux language, InfluxDB data scripting language.

The results of the comparison between the predicted and the real-life consumption were not the expected, since the sensor data metrics showed a lower consumption than the EU energy label metrics, due to the low amount of available data and possible device settings that may lower the consumption. Despite this it was possible to conclude the metrics calculations are correctly implemented and that the EU energy labelling metrics are a viable comparison criterion.

The EPREL database stores EU device energy label and other energy metrics. This data is extracted using the provided API and is a baseline for the consumption assessment.

The dashboards provide a simple real-time visualization meant to be used by the users of the SATO platform since it is simple and easy to interact with. On the other side the API provides access to the metadata, metrics and sensor data through five API calls. For a SATO user, it is not as simple to interact with the API as it is with the dashboards, as the API is instead meant to be used by other developers.

5.2 Future work

To correctly compare the sensor data metrics with the EU energy label, it is necessary to have a large volume of data available collected during a normal use of the device. This will lead to more viable consumption metrics that will provide more viable comparisons. It is also fundamental to collect data from the other types of devices supported by the EPREL database, since these are the only ones that can be used for the comparison.

Additional, it is important to consider other types of sensors, in addition to those considered in this work (e.g. current sensor). The list of types of sensors that must be considered is detailed in Section 3.5.1. This is crucial to obtain the data needed for the calculation of some metrics, namely the waterflow and noise data.

The ER model presented in Appendix A describes the entities and attributes for the all the devices types eligible for an energy label, both the ones on the EPREL database and the ones not on the EPREL database. When a device type is added to EPREL database, the model must be altered to update the metrics and how they are calculated, due to the release of new Energy Labelling Regulations. Therefore, this ER model will need to be altered to reflect these updates.

Currently the insertion of sensor data is done by reading the data from a CSV file, but in the future this insertion should be done using Apache Kafka and sending each data point individually or grouped by 10 seconds. Due to these planned changes, the script currently in use will also need to be altered or even discarded in favour of using the Telegraf Kafka plugin to write to a Kafka broker [28] and the Kafka Consumer Input to read from Kafka [27].

Bibliography

[1] *About the energy label and ecodesign*. URL: https://ec.europa.eu/info/energy-climate-change-environment/standards-tools-and-labels/products-labelling-rules-and-requirements/energy-label-and-ecodesign/about_en (cit. on p. 12).

[2] *About the energy label and ecodesign*. URL: https://ec.europa.eu/info/energy-climate-change-environment/standards-tools-and-labels/products-labelling-rules-and-requirements/energy-label-and-ecodesign/about_en (cit. on p. 13).

[3] *Amazon Athena database webpage*. URL: https://aws.amazon.com/pt/athena/ (cit. on p. 22).

[4] *Apache Druid website*. URL: https://druid.apache.org/ (cit. on p. 10).

[5] Andreas Bader, Oliver Kopp, and Michael Falkenthal. “Survey and Comparison of Open Source Time Series Databases”. In: (2017) (cit. on p. 10).

[6] European Commission. *Commission Recommendation (EU) 2019/1019 of 7 June 2019 on building modernisation*. Tech. rep. 2019 (cit. on p. 1).

[7] European Commission. *Fourth Report on the State of the Energy Union*. Tech. rep. 2019 (cit. on pp. v, 1).

[8] *CrateDB webpage*. URL: https://crate.io/ (cit. on p. 10).

[9] *CrateDB: Technical overview*. Tech. rep. Apr. 2020 (cit. on p. 8).

[10] *Description of action*. Tech. rep. June 2020 (cit. on p. 2).

[11] *Description of the system architecture of the SATO platform - Confidential*. Tech. rep. July 2021 (cit. on p. 22).

[12] Paul Dix. *Why Time Series Matters for Metrics, Real-Time Analytics and Sensor Data*. Tech. rep. July 2021. URL: https://www.influxdata.com/what-is-time-series-data/ (cit. on pp. 5, 20).

[13] *Driving energy efficiency in the European building stock: New recommendations on the modernisation of buildings*. 2019. URL: https://ec.europa.eu/info/news/driving-energy-efficiency-european-building-stock-new-recommendations-modernisation-buildings-2019-jun-21_en (cit. on p. 1).

[14] *EPREL API*. URL: https://webgate.ec.europa.eu/fpfis/wikis/display/EPREL/EPREL+Public+site+-+API (cit. on p. 17).

[15] *EPREL Product database — European Commission*. URL: https://ec.europa.eu/info/energy-climate-change-environment/standards-tools-and-labels/products-labelling-rules-and-requirements/energy-label-and-ecodesign/product-database_en (cit. on pp. 2, 12).

[16] Jimmy Fjällid. "A Comparative Study of Databases for Storing Sensor Data". 2019 (cit. on pp. 5, 9).

[17] *Flask Documentation*. https://flask.palletsprojects.com/en/2.0.x/ (cit. on pp. 50, 51).

[18] *Flux SQL package*. URL: https://docs.influxdata.com/influxdb/cloud/query-data/flux/sql/ (cit. on p. 22).

[19] André Gil. "Platform architecture and data management for cloud-based buildings energy self-assessment and optimization". 2021 (cit. on p. 25).

[20] *Google BigQuery database webpage*. URL: https://cloud.google.com/bigquery (cit. on p. 22).

[21] *Graphite webpage*. URL: https://graphiteapp.org/ (cit. on p. 9).

[22] *InfluxDB documentation*. https://docs.influxdata.com/influxdb/v2.0// (cit. on p. 7).

[23] *InfluxDB Siemens customer story*. URL: https://www.influxdata.com/customer/siemens/ (cit. on p. 20).

[24] *InfluxDB website*. https://www.influxdata.com/products/influxdb/. 2020 (cit. on pp. 6, 9).

[25] Žygimantas Jasiūnas. "Building appliances energy performance assessment". 2021 (cit. on pp. 27, 28, 54).

[26] Janaki Joshi, Lakshmi Sirisha Chodisetty, and Varsha Raveendran. "A Quality Attribute-based Evaluation of Time-series Databases for Edge-centric Architectures". In: (2019) (cit. on p. 10).

[27] *Kafka consumer telegraf plugin*. URL: https://github.com/influxdata/telegraf/tree/master/plugins/outputs/kafka (cit. on p. 59).

[28] *Kafka telegraf plugin*. URL: https://github.com/influxdata/telegraf/tree/master/plugins/inputs/kafka_consumer (cit. on p. 59).

[29] *Kairos webpage*. URL: https://kairosdb.github.io/ (cit. on p. 10).

[30] *kdb+ webpage*. URL: https://code.kx.com/q/ (cit. on p. 9).

[31] Olivier Mathieu et al. *EPREL Exchange Model Documentation*. 2019 (cit. on pp. 13, 38).

[32] Anna Carolina Menezes et al. "Predicted vs. actual energy performance of non-domestic buildings: Using post-occupancy evaluation data to reduce the performance gap". In: *Applied Energy* 97 (2012), pp. 355–364. DOI: https://doi.org/10.1016/j.apenergy.2011.11.075. URL: https://www.sciencedirect.com/science/article/pii/S0306261911007811 (cit. on pp. v, 1).

[33] *Microsoft SQL Server database webpage*. URL: https://www.microsoft.com/pt-pt/sql-server/sql-server-2019 (cit. on p. 22).

[34] Oscar Miralles. *European Product Registry for Energy Labelling (EPREL) Compliance Site Description*. 2018 (cit. on pp. 11, 12).

[35] *MySQL database webpage*. URL: `https://www.mysql.com/` (cit. on p. 22).

[36] Syeda Noor Zehra Naqvi and Sofia Yfantidou. *Time Series Databases and InfuxDB*. Tech. rep. 2017 (cit. on pp. 5, 6).

[37] *pgAdmin website*. URL: `https://www.pgadmin.org/` (cit. on p. 37).

[38] *PostgreSQL database webpage*. URL: `https://www.postgresql.org/` (cit. on p. 22).

[39] *Prometheus Documentation*. https://prometheus.io/docs/introduction/overview/ (cit. on p. 8).

[40] *Regulation (EU) 2017/1369 of the European parliament and of the council*. 2017. URL: `https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=uriserv%5C%3AOJ.L_.2017.198.01.0001.01.ENG` (cit. on p. 11).

[41] *Regulation on energy labelling for electronic displays (EU) 2019/2013*. 2019. URL: `https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=uriserv:OJ.L_.2019.315.01.0001.01.ENG&toc=OJ:L:2019:315:TOC` (cit. on pp. 13, 41).

[42] *Regulation on energy labelling for fridges and freezers (EU)*. URL: `https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32019R2016&from=EN#d1e32-112-1` (cit. on p. 46).

[43] *Regulation on energy labelling for fridges and freezers (EU) 2019/2013*. 2019. URL: `https://eur-lex.europa.eu/legal-content/EN/TXT/?qid=1575537791838&uri=CELEX%3A32019R2016` (cit. on p. 41).

[44] A. Ridi, C. Gisler, and J. Hennebert. “ACS-F2 - A new database of appliance consumption signatures”. In: *Soft Computing and Pattern Recognition (SoCPaR), 2014 6th International Conference of*. IEEE, Aug. 2014, pp. 145–150 (cit. on p. 28).

[45] *Self Assessment Towards Optimization of Building Energy web page*. 2020. URL: `https://www.sato-project.eu/` (cit. on p. 2).

[46] *Snowflake database webpage*. URL: `https://www.snowflake.com/` (cit. on p. 22).

[47] *SQLite database webpage*. URL: `https://www.sqlite.org/index.html` (cit. on p. 22).

[48] Ilya Sychev et al. “Closed Loop Benchmark for Timeseries Databases”. In: (2020) (cit. on p. 9).

[49] *Timescale webpage*. URL: `https://www.timescale.com/` (cit. on p. 9).

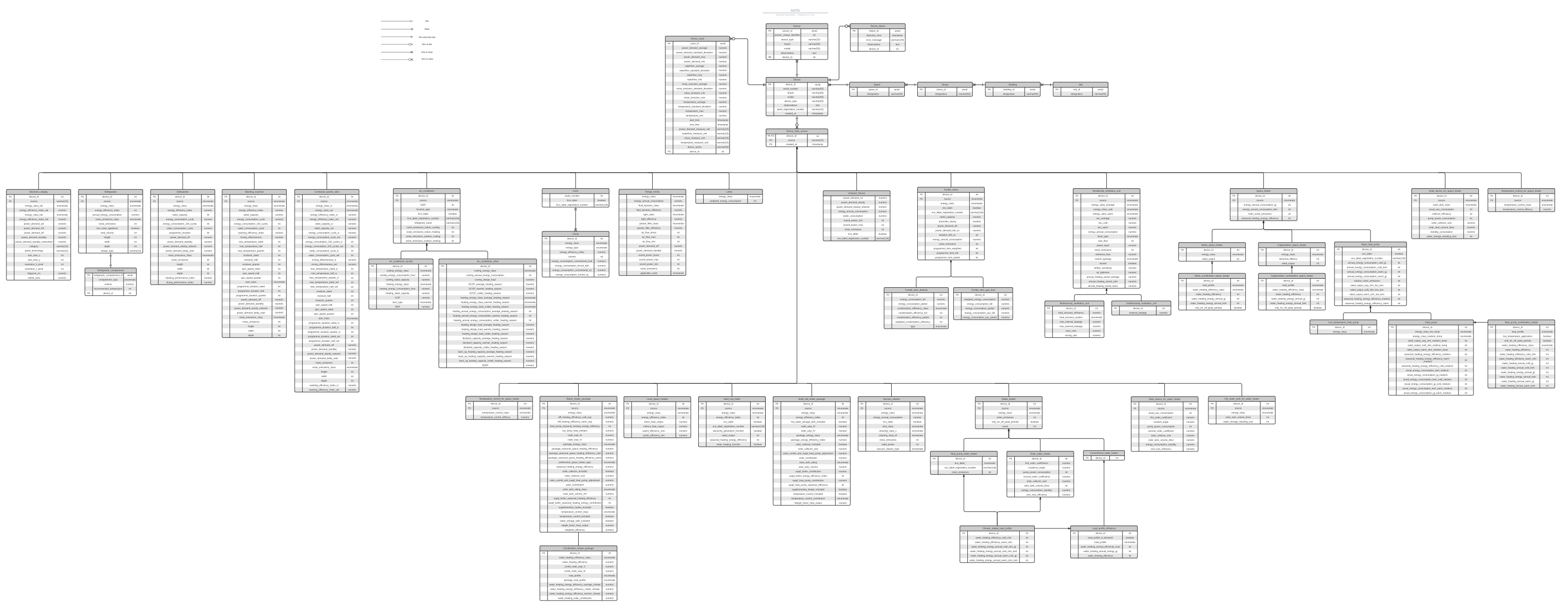
[50] *TimescaleDB Documentation*. URL: `https://docs.timescale.com/latest/main` (cit. on p. 8).

[51] *VictoriaMetrics Documentation*. https://victoriametrics.github.io/ (cit. on p. 9).

[52] *vmauth auth proxy*. URL: `https://docs.victoriametrics.com/vmauth.html` (cit. on p. 21).

Appendix A

ER model for the Metrics and metadata device DB



Appendix B

Scripts

B.1 tables_creation_script.sql

28/10/21, 16:07 databaseCreationScript.sql

```sql
DROP SCHEMA public CASCADE;
CREATE SCHEMA public;

CREATE TYPE energy_class AS ENUM ('A+++','A++','A+','A','B','C','D','E','F','G');
CREATE TYPE noise_class AS ENUM ('A','B','C','D');
CREATE TYPE spin_class AS ENUM ('A','B','C','D','E','F','G');

CREATE TABLE measure_unit (
  measure_unit_id serial,
  measurement_unit_name varchar(50),
  PRIMARY KEY (measure_unit_id)
);

CREATE TABLE device (
  device_id serial,
  brand varchar(50),
  model varchar(50),
  observations text,
  eprel_registration_number varchar(15),
  created_at timestamp,
  PRIMARY KEY (device_id)
);

CREATE TABLE sensor (
  sensor_id serial,
  sensor_type varchar(15) CHECK (sensor_type IN
('accelerometer','temperature','waterflow','current','noise')),
  brand varchar(50),
  model varchar(50),
  observations text,
  device_id int REFERENCES Device,
  PRIMARY KEY (sensor_id)
);

CREATE TABLE device_cycle (
  cycle_id serial,
  power_demand_average numeric,
  power_demand_standard_deviation numeric,
  power_demand_max  numeric,
  power_demand_min numeric,
  waterflow_average numeric,
  waterflow_standard_deviation numeric,
  waterflow_max numeric,
  waterflow_min numeric,
  noise_emission_average numeric,
  noise_emission_standard_deviation numeric,
  noise_emission_min numeric,
  noise_emission_max numeric,
  temperature_average numeric,
  temperature_standard_deviation numeric,
  temperature_max numeric,
  temperature_min numeric,
  start_time timestamp,
  end_time timestamp,
  power_demand_measure_unit varchar(15),
  waterflow_measure_unit varchar(15),
  noise_measure_unit varchar(15),
  temperature_measure_unit varchar(15),
  device_action varchar(50),
  device_id int REFERENCES Device,
  PRIMARY KEY (cycle_id)
);

CREATE TABLE device_failure (
  failure_id serial,
  detection_time timestamp,
  error_message varchar(128),
  observations text,
  device_id int REFERENCES Device,
  PRIMARY KEY (failure_id)
);

CREATE TABLE device_data_source (
  device_id int REFERENCES Device,
  source varchar(15) CHECK (source IN ('EPREL','sensors')),
  created_at timestamp,
  PRIMARY KEY (device_id, source, created_at)
);

CREATE TABLE electronic_display (
  energy_class_sdr energy_class,
  energy_class_hdr energy_class,
  power_demand_sdr numeric,
  power_demand_hdr numeric,
  power_demand_off numeric,
  power_demand_standby numeric,
  power_demand_standby_networked numeric,
  category varchar(15) CHECK (category IN
('TELEVISION','MONITOR','SIGNAGE','OTHER')),
  panel_technology varchar(15) CHECK (panel_technology IN
('LCD','LED_LCD','QLED_LCD','OLED','MicroLED','QDLED','SED','FED','EPD','OTHER')),
  size_ratio_x int,
  size_ratio_y int,
  resolution_h_pixel int,
  resolution_v_pixel int,
  diagonal_cm numeric,
  visible_area numeric
) INHERITS (Device_data_source);

CREATE TABLE refrigerator (
  energy_class energy_class,
  energy_efficiency_index int,
  annual_energy_consumption numeric,
  noise_emissions_class noise_class,
  noise_emissions int,
  low_noise_appliance boolean,
  total_volume int,
  heigth int,
  width int,
  depth int,
  design_type VARCHAR(15) CHECK (design_type IN ('BUILT_IN','FREE_STANDING'))
) INHERITS (Device_data_source);

CREATE TABLE refrigerator_compartment (
  refrigerator_compartment_id serial,
  compartment_type varchar(15) CHECK (compartment_type IN
('PANTRY','WINE_STORAGE','CELLAR','FRESH_FOOD','CHILL','ZERO_STAR','ONE_STAR','TWO_ST
AR','THREE_STAR','FOUR_STAR','TWO_STAR_SECTION','VARIABLE_TEMP')),
  volume numeric,
  device_id int REFERENCES Device,
  PRIMARY KEY (refrigerator_compartment_id)
);

CREATE TABLE untyped_Device (
  power_demand_on numeric,
  power_demand_standy numeric,
  power_demand_standy_network numeric,
  energy_annual_consumption numeric,
  water_consumption numeric,
  sound_power_min int,
  sound_power_max int,
  noise_emissions int
) INHERITS (Device_data_source);

CREATE TABLE dishwasher (
  energy_class energy_class,
  energy_efficiency_index numeric,
  rated_capacity int,
  energy_consumption_cycle numeric,
  energy_consumption_100_cycles int,
  water_consumption_cycle numeric,
  programme_duration int,
  power_demand_off numeric,
  power_demand_standby numeric,
  power_demand_standy_network numeric,
  power_demand_delay_start numeric,
  noise_emissions_class noise_class,
  noise_emissions int,
  heigth int,
  width int,
  depth int,
  cleaning_performance_index numeric,
  drying_performance_index numeric
) INHERITS (Device_data_source);

CREATE TABLE washing_machine (
  energy_class energy_class,
  energy_efficiency_index numeric ,
  rated_capacity numeric,
  energy_consumption_cycle numeric,
  energy_consumption_100_cycles int,
  water_consumption_cycle int,
  washing_efficiency_index numeric,
  rinsing_effectiveness numeric,
  max_temperature_rated int,
  max_temperature_half int,
  max_temperature_quarter int,
  moisture_rated int,
  moisture_half int,
  moisture_quarter int,
  spin_speed_rated int,
  spin_speed_half int,
  spin_speed_quarter int,
  spin_class spin_class,
  programme_duration_rated int,
  programme_duration_half int,
  programme_duration_quarter int,
  power_demand_off numeric,
  power_demand_standby numeric,
  power_demand_standy_network numeric,
  power_demand_delay_start numeric,
  noise_emissions_class noise_class,
  noise_emissions int,
  heigth int,
  width int,
  depth int
) INHERITS (Device_data_source);

CREATE TABLE combined_washer_drier (
  energy_class_w energy_class,
  energy_class_wd energy_class,
  energy_efficiency_index_w numeric,
  energy_efficiency_index_wd numeric,
  rated_capacity_w numeric,
  rated_capacity_wd numeric,
  energy_consumption_cycle_w numeric,
  energy_consumption_cycle_wd numeric,
  energy_consumption_100_cycles_w int,
  energy_consumption_100_cycles_wd int,
  water_consumption_cycle_w int,
  water_consumption_cycle_wd int,
  rinsing_effectiveness_w numeric,
  rinsing_effectiveness_wd numeric,
  max_temperature_rated_w int,
  max_temperature_half_w int,
  max_temperature_quarter_w int,
  max_temperature_rated_wd int,
  max_temperature_half_wd int,
  moisture_rated int,
  moisture_half int,
  moisture_quarter int,
  spin_speed_half int,
  spin_speed_rated int,
  spin_speed_quarter int,
  spin_class spin_class,
  programme_duration_rated_w int,
  programme_duration_half_w int,
  programme_duration_quarter_w int,
  programme_duration_rated_wd int,
  programme_duration_half_wd int,
  power_demand_off numeric,
  power_demand_standby numeric,
  power_demand_standy_network numeric,
  power_demand_delay_start numeric,
  noise_emissions int,
  noise_emissions_class noise_class,
  heigth int,
  width int,
  depth int,
  washing_efficiency_index_w numeric,
  washing_efficiency_index_wd numeric
) INHERITS (Device_data_source);

```

B.2 eprel_kafka_producer.py

```python
import requests
import json
import sys
from kafka import KafkaProducer

eprel_producer = KafkaProducer(bootstrap_servers='194.117.20.229:9092',
value_serializer=lambda v: json.dumps(v).encode('utf-8'))

def get_eprel_data(registration_number, device_group):
  response =
requests.get(f'https://eprel.ec.europa.eu/api/products/{device_group}/{registration_n
umber}')

  response_content = response.json()

  return response_content

def send_to_kafka(eprel_data):
  header = [('source', 'EPREL'.encode())]
  eprel_producer.send('eprel_data', value=eprel_data, headers=header)

def get_registration_number(brand, model):
  data = {
    ('Candy', 'ROW4966DWMCE/1-S'): ('300268'),
    ('Beko', 'TS190030N'): ('341954'),
    ('AOC', '27B1H'): ('419895'),
    ('PRINCESS', 'WM1044CT0'): ('352767'),
    ('Candy', 'X14IX'): ('352774')
  }

  return data[(brand, model)]

if __name__ == "__main__":
  device_type = sys.argv[1]
  brand = sys.argv[2]
  model = sys.argv[3]

  registration_number = get_registration_number(brand, model)

  eprel_data = get_eprel_data(registration_number, device_type)
  send_to_kafka(eprel_data)

  eprel_producer.close()
```

B.3 eprel_kafka_consumer.py

```
from datetime import datetime
import psycopg2
from psycopg2 import Error
import json
from kafka import KafkaConsumer

try:
  conn = psycopg2.connect(dbname='SATO', host='localhost', port='5432',
user='admin', password='admin')
  cursor = conn.cursor()

except (Exception, Error) as error:
  print("Error while connecting to PostgreSQL", error)

device_id = 0
eprel_consumer = KafkaConsumer('eprel_data',
    bootstrap_servers='194.117.20.229:9092',
    value_deserializer=lambda v: json.loads(v.decode('utf-8')))

def insert_into_database(eprel_data):
  device_group = eprel_data['productGroup']
  insert_device(eprel_data)

  if (device_group == "electronicdisplays"):
    insert_electronic_display_device(eprel_data)

  elif (device_group == "refrigeratingappliances2019"):
    insert_refrigerator_device(eprel_data)

  elif (device_group == "dishwashers2019"):
    insert_dishwasher_device(eprel_data)

  elif (device_group == "washingmachines2019"):
    insert_washing_machine_device(eprel_data)

  elif (device_group == "washerdriers2019"):
    insert_washer_dryer_device(eprel_data)

  conn.commit()


def insert_device(eprel_data):
  global device_id

  print(eprel_data)

  query = "INSERT INTO device(brand,model,eprel_registration_number,created_at)
VALUES (%s,%s,%s,%s) RETURNING device_id;"

  query_data = (
    eprel_data['supplierOrTrademark'],
    eprel_data['modelIdentifier'],
    eprel_data['eprelRegistrationNumber'],
    datetime.utcnow()
  )

  cursor.execute(query, query_data)
  device_id = cursor.fetchone()[0]


```

```python
def insert_electronic_display_device(eprel_data):
  query = "INSERT INTO electronic_display VALUES (%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s);"

  query_data = (device_id,
    'EPREL',
    datetime.utcnow(),
    eprel_data['energyClassSDR'],
    eprel_data['energyClassHDR'],
    eprel_data['powerOnModeSDR'],
    eprel_data['powerOnModeHDR'],
    eprel_data['powerOffMode'],
    eprel_data['powerStandby'],
    eprel_data['powerStandbyNetworked'],
    eprel_data['category'],
    eprel_data['panelTechnology'],
    eprel_data['sizeRatioX'],
    eprel_data['sizeRatioY'],
    eprel_data['resolutionHorizontalPixels'],
    eprel_data['resolutionVerticalPixels'],
    eprel_data['diagonalCm'],
    eprel_data['visibleArea'],
  )

  cursor.execute(query, query_data)


def insert_refrigerator_device(eprel_data):
  query = "INSERT INTO refrigerator VALUES (%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s);"

  query_data = (device_id,
    'EPREL',
    datetime.utcnow(),
    eprel_data['energyClass'],
    eprel_data['energyEfficiencyIndex'],
    eprel_data['energyConsAnnual'],
    eprel_data['noiseClass'],
    eprel_data['noise'],
    eprel_data['lowNoiseAppliance'],
    eprel_data['totalVolume'],
    eprel_data['dimensionHeight'],
    eprel_data['dimensionWidth'],
    eprel_data['dimensionDepth'],
    eprel_data['designType']
  )

  cursor.execute(query, query_data)

  for compartment in eprel_data['compartments']:
    query = "INSERT INTO refrigerator_compartment (compartment_type,volume,device_id) VALUES (%s,%s,%s);"

    query_data = (
      compartment['compartmentType'],
      compartment['volume'],
      device_id
    )

    cursor.execute(query, query_data)
```

```python

def insert_dishwasher_device(eprel_data):
  query = "INSERT INTO dishwasher VALUES (%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s);"

  query_data = (device_id,
    'EPREL',
    datetime.utcnow(),
    eprel_data['energyClass'],
    eprel_data['energyEfficiencyIndex'],
    eprel_data['ratedCapacity'],
    eprel_data['energyCons'],
    eprel_data['energyCons100'],
    eprel_data['waterCons'],
    eprel_data['programmeDuration'],
    eprel_data['powerOffMode'],
    eprel_data['powerStandbyMode'],
    eprel_data['powerNetworkStandby'],
    eprel_data['powerDelayStart'],
    eprel_data['noiseClass'],
    eprel_data['noise'],
    eprel_data['dimensionHeight'],
    eprel_data['dimensionWidth'],
    eprel_data['dimensionDepth'],
    eprel_data['cleaningPerformanceIndex'],
    eprel_data['dryingPerformanceIndex'],
  )

  cursor.execute(query, query_data)

def insert_washing_machine_device(eprel_data):
  query = "INSERT INTO washing_machine VALUES (%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s);"

  query_data = (device_id,
    'EPREL',
    datetime.utcnow(),
    eprel_data['energyClass'],
    eprel_data['energyEfficiencyIndex'],
    eprel_data['ratedCapacity'],
    eprel_data['energyCons'],
    eprel_data['energyConsPerCycle'],
    eprel_data['waterCons'],
    eprel_data['washingEfficiencyIndex'],
    eprel_data['rinsingEffectiveness'],
    eprel_data['maxTemperatureRated'],
    eprel_data['maxTemperatureHalf'],
    eprel_data['maxTemperatureQuarter'],
    eprel_data['moistureRated'],
    eprel_data['moistureHalf'],
    eprel_data['moistureQuarter'],
    eprel_data['spinSpeedRated'],
    eprel_data['spinSpeedHalf'],
    eprel_data['spinSpeedQuarter'],
    eprel_data['spinClass'],
    eprel_data['programmeDurationRated'],
    eprel_data['programmeDurationHalf'],
    eprel_data['programmeDurationQuarter'],
    eprel_data['powerOffMode'],
    eprel_data['powerStandbyMode'],
    eprel_data['powerNetworkStandby'],
    eprel_data['powerDelayStart'],
    eprel_data['noiseClass'],
    eprel_data['noise'],
    eprel_data['dimensionHeight'],
    eprel_data['dimensionWidth'],
    eprel_data['dimensionDepth'],
  )

  cursor.execute(query, query_data)

def insert_washer_dryer_device(eprel_data):
  query = "INSERT INTO combined_washer_drier VALUES (%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s);"

  query_data = (device_id,
    'EPREL',
    datetime.utcnow(),
    eprel_data['energyClassWash'],
    eprel_data['energyClassWashAndDry'],
    eprel_data['energyEfficiencyIndexWash'],
    eprel_data['energyEfficiencyIndexWashAndDry'],
    eprel_data['ratedCapacityWash'],
    eprel_data['ratedCapacityWashAndDry'],
    eprel_data['energyConsumptionWash'],
    eprel_data['energyConsumptionWashAndDry'],
    eprel_data['energyConsumption100Wash'],
    eprel_data['energyConsumption100WashAndDry'],
    eprel_data['waterConsumptionWash'],
    eprel_data['waterConsumptionWashAndDry'],
    eprel_data['rinsingEffectivenessWash'],
    eprel_data['rinsingEffectivenessWashAndDry'],
    eprel_data['maxTempRatedWash'],
    eprel_data['maxTempHalfWash'],
    eprel_data['maxTempQuarterWash'],
    eprel_data['maxTempRatedWashAndDry'],
    eprel_data['maxTempHalfWashAndDry'],
    eprel_data['moistureRated'],
    eprel_data['moistureHalf'],
    eprel_data['moistureQuarter'],
    eprel_data['spinSpeedRated'],
    eprel_data['spinSpeedHalf'],
    eprel_data['spinSpeedQuarter'],
    eprel_data['spinClass'],
    eprel_data['programDurationRatedWash'],
    eprel_data['programDurationHalfWash'],
    eprel_data['programDurationQuarterWash'],
    eprel_data['programDurationRatedWashAndDry'],
    eprel_data['programDurationHalfWashAndDry'],
    eprel_data['powerOffMode'],
    eprel_data['powerStandbyMode'],
    eprel_data['powerNetworkedStandby'],
    eprel_data['powerDelayStart'],
    eprel_data['noise'],
    eprel_data['noiseClass'],
    eprel_data['dimensionHeight'],
    eprel_data['dimensionWidth'],
    eprel_data['dimensionDepth'],
    eprel_data['washingEfficiencyIndexWash'],
    eprel_data['washingEfficiencyIndexWashAndDry']
  )

  cursor.execute(query, query_data)

if __name__ == "__main__":

  for msg in eprel_consumer:
    payload = msg.value
    insert_into_database(payload)

  cursor.close()
  conn.close()
  print("PostgreSQL connection is closed")
```

B.4 insert_sensor_data.py

```
import xmltodict,os
from time import sleep
from datetime import datetime,timedelta

from influxdb_client import InfluxDBClient, Point, WritePrecision
from influxdb_client.client.write_api import SYNCHRONOUS

#PostgreSQL conection
import psycopg2
from psycopg2 import Error

try:
  conn = psycopg2.connect(database='SATO', host='localhost', port='5432',
user='admin',  password='admin')
  cursor = conn.cursor()

except (Exception, Error) as error:
  print("Error while connecting to PostgreSQL", error)


def get_device_info(file_name):
  with open(file_name, 'r') as f:
    file_content = f.read()

  file_dict = xmltodict.parse(file_content)

  device_type = file_dict['satoDb']['data']['targetAppliance']['@type']
  device_model = file_dict['satoDb']['data']['targetAppliance']['@model']

  return device_type, f"{device_type}-{device_model}"

def get_device_id(device_name):
  brand = device_name.split("-")[1]
  model = device_name.split("-")[2] + "%"

  query = "SELECT * FROM device WHERE brand LIKE %s and model LIKE %s;"

  query_data = (brand, model)

  cursor.execute(query, query_data)

  return cursor.fetchone()[0]


#influxDB insert
def insert_file(device_name, device_id, device_group, file_name):
  with open(file_name, 'r') as f:
    file_content = f.read()

  file_dict = xmltodict.parse(file_content)
  device_action = file_dict['satoDb']['data']['targetAppliance']['@action']
  snapshots = file_dict['satoDb']['snapshots']['snap']

  device_brand = device_name.split("-")[1]
  device_model = device_name.split("-")[2]

  for snap in snapshots: #convert data into float
    point = format_data_for_influx(snap, device_name, device_id, device_brand,
device_model, device_action, device_group)

```

```python
    write_api.write(bucket, org, point)

    sleep(1)

  print(f"{device_name} inserted")

def format_data_for_influx(snap, device_name, device_id, device_brand, device_model, device_action, device_group):
  curr = float(snap['@curr'])
  eng = float(snap['@eng'])
  freq = float(snap['@freq'])
  pf = float(snap['@pf'])
  pw = float(snap['@pw'])
  temp = float(snap['@temp'])


  point = Point('mem')\
    .tag('_device', device_name)\
    .tag('_device_id', device_id)\
    .tag('_device_brand', device_brand)\
    .tag('_device_model', device_model)\
    .tag('_device_group', device_group)\
    .tag('_device_action', device_action)\
    .field('current', curr)\
    .field('energy', eng)\
    .field('frequency', freq)\
    .field('powerFactor', pf)\
    .field('power', pw)\
    .field('temperature', temp)\
    .time(datetime.utcnow() , WritePrecision.MS)

  return point

if __name__ == '__main__':
  # Set up influxDB connection
  token = 'NipdVNkwbQ0kXiNbXBQ7P--iQZX0WkAbdrDAoFgNFwsRULUpIF6U70_pn-viDVeeaocMmXgUCUcgt-QgCkTZFw=='
  org = 'example-org'
  bucket = 'electrical_signatures'

  client = InfluxDBClient(url='http://localhost:8086', token=token)
  write_api = client.write_api(write_options=SYNCHRONOUS)

  file_name = 'Monitor_AOC-27B1-27_on_1.xml'

  device_group, device_name = get_device_info(file_name)

  device_id = get_device_id(device_name)

  insert_file(device_name, device_id, device_group, file_name)

  #close connections
  write_api.close()
  cursor.close()
  conn.close()
```

B.5 load_device_cycle.flux

```
import "experimental"
import "sql"
import "influxdata/influxdb/secrets"

option task = {
    name: "load_device_cycle",
    every: 30s,
    offset: 0m,
    concurrency: 1,
}

POSTGRES_USER = secrets.get(key: "POSTGRES_USER")
POSTGRES_PASS = secrets.get(key: "POSTGRES_PASS")

get_last_cycle_exists = (device_id) =>
  sql.from(
    driverName: "postgres",
    dataSourceName: "postgres://${POSTGRES_USER}:${POSTGRES_PASS}@localhost/SATO",
    query: "SELECT EXISTS (SELECT * FROM device_cycle where device_id = ${device_id})"
  )
  |> findColumn(fn: (key) => true, column: "exists")

get_last_cycle_end_time = (device_id) =>
  sql.from(
    driverName: "postgres",
    dataSourceName: "postgres://${POSTGRES_USER}:${POSTGRES_PASS}@localhost/SATO",
    query: "SELECT * FROM device_cycle where device_id = ${device_id} ORDER BY end_time DESC"
  )
  |> findColumn(fn: (key) => true, column: "end_time")

get_last_power_value = (device) =>
  from(bucket: "electrical_signatures")
    |> range(start: -1h)
    |> filter(fn: (r) => r["_field"] == "power" and r["_device"] == device)
    |> aggregateWindow(every: 5s, fn: min, createEmpty: false)
    |> last()
    |> findColumn(fn: (key) => true, column: "_value")

get_date_to_search = (tables=<-) =>
  tables
    |> map(fn: (r) => ({ r with
      date_to_search:
        if r.last_power_value != 0 //only execute if device is off
          then experimental.subDuration(d: 1ms, from: now())
        else if r.last_cycle_exists == true
          then experimental.addDuration(d: 1ms, to:  r.last_cycle_end_time)
        else experimental.subDuration(d: 1mo, from: now())  //duration to use when searching for last cycle data
    }))

get_cycle_start_time = (date_to_search, device) =>
  from(bucket: "electrical_signatures")
    |> range(start: date_to_search)
    |> filter(fn: (r) => r["_field"] == "power" and r["_device"] == device and r["_value"] > 0.5)
    |> aggregateWindow(every: 5s, fn: min, createEmpty: false)
    |> first()
    |> findColumn(fn: (key) => true, column: "_time")
```

```

get_cycle_end_time = (date_to_search, device) =>
  from(bucket: "electrical_signatures")
    |> range(start: date_to_search)
    |> filter(fn: (r) => r["_field"] == "power" and r["_device"] == device and
r["_value"] > 0.5)
    |> aggregateWindow(every: 5s, fn: min, createEmpty: false)
    |> last()
    |> findColumn(fn: (key) => true, column: "_time")

get_max_power = (start_time, stop_time, device) =>
  from(bucket: "electrical_signatures")
    |> range(start: start_time, stop: stop_time)
    |> filter(fn: (r) => r["_field"] == "power" and r["_device"] == device and
r["_value"] > 0.5)
    |> max()
    |> rename(columns: {_value: "power_demand_max"})
    |> findColumn(fn: (key) => true, column: "power_demand_max")

get_min_power = (start_time, stop_time, device) =>
  from(bucket: "electrical_signatures")
    |> range(start: start_time, stop: stop_time)
    |> filter(fn: (r) => r["_field"] == "power" and r["_device"] == device and
r["_value"] > 0.5)
    |> min()
    |> findColumn(fn: (key) => true, column: "_value")

get_average_power = (start_time, stop_time, device) =>
  from(bucket: "electrical_signatures")
    |> range(start: start_time, stop: stop_time)
    |> filter(fn: (r) => r["_field"] == "power" and r["_device"] == device and
r["_value"] > 0.5)
    |> mean(column: "_value")
    |> findColumn(fn: (key) => true, column: "_value")

get_std_deviation_power = (start_time, stop_time, device) =>
  from(bucket: "electrical_signatures")
    |> range(start: start_time, stop: stop_time)
    |> filter(fn: (r) => r["_field"] == "power" and r["_device"] == device and
r["_value"] > 0.5)
    |> stddev()
    |> findColumn(fn: (key) => true, column: "_value")

formatTable = (tables=<-) =>
  tables
    |> map(fn: (r) => ({
        r with
            device_id: r._device_id,
            power_demand_measure_unit: "W",
            waterflow_measure_unit: "L",
            noise_measure_unit: "db"
      })
    )
    |> drop(columns: ["_device", "_device_id", "date_to_search", "last_power_value",
"last_cycle_exists", "last_cycle_end_time"])

from(bucket: "electrical_signatures")
  |> range(start: -1w)
  |> keep(columns: ["_device", "_device_id"])
  |> unique(column: "_device_id")
  |> map(fn: (r) => ({
    r with
    last_cycle_exists: get_last_cycle_exists(device_id: r._device_id)[0],
    last_power_value: if length(arr: get_last_power_value(device: r._device)) > 0 then get_last_power_value(device: r._device)[0] else 1.0
  }))
  |> map(fn: (r) => ({
    r with
    last_cycle_end_time:
      if r.last_cycle_exists == true
        then get_last_cycle_end_time(device_id: r._device_id)[0]
      else now()
  }))
  |> filter(fn: (r) => r.last_power_value == 0)
  |> get_date_to_search()
  |> map(fn: (r) => ({
      r with
      start_time: get_cycle_start_time(date_to_search: r.date_to_search, device: r._device)[0],
      end_time: get_cycle_end_time(date_to_search: r.date_to_search, device: r._device)[0]
  }))
  |> map(fn: (r) => ({
      r with
      power_demand_max: get_max_power(start_time: r.start_time, stop_time: r.end_time, device: r._device)[0],
      power_demand_min: get_min_power(start_time: r.start_time, stop_time: r.end_time, device: r._device)[0],
      power_demand_average: get_average_power(start_time: r.start_time, stop_time: r.end_time, device: r._device)[0],
      power_demand_standard_deviation: get_std_deviation_power(start_time: r.start_time, stop_time: r.end_time, device: r._device)[0]
  }))
  |> formatTable()
  |> sql.to(
    driverName: "postgres",
    dataSourceName: "postgres://${POSTGRES_USER}:${POSTGRES_PASS}@localhost/SATO",
    table: "Device_cycle")
```

B.6 external_api.py

```
#!flask/bin/python
from flask import Flask, jsonify, make_response, g, request
import psycopg2
from psycopg2.extras import RealDictCursor
from influxdb_client import InfluxDBClient
import decimal

#API setup

app = Flask(__name__)

def get_db():
  db = getattr(g, '_database', None)
  if db is None:
    db = g._database = psycopg2.connect(dbname='SATO', user='admin',
password='admin', host='localhost', port='5432', cursor_factory=RealDictCursor)
  return db

@app.before_request
def before_request():
  if request.endpoint != "get_device_power_week":
    g.db = get_db()

@app.teardown_appcontext
def close_connection(exception):
  db = getattr(g, '_database', None)
  if db is not None:
    db.close()

@app.route('/')
def hello():
  return "Hello world!"


#API for metrics

@app.route('/metrics/<device_serial_number>/<metric>', methods=['GET'])
def get_indicator(device_serial_number, metric):
  cur = g.db.cursor()

  get_device_query = "SELECT * FROM device WHERE serial_number=(%s);"
  data = (device_serial_number, )
  cur.execute(get_device_query, data)

  get_device_query_result = cur.fetchone()

  if get_device_query_result is not None:
    device_id = get_device_query_result["device_id"]
    device_type = convert_device_type(get_device_query_result["device_type"])

    get_metric_query = "SELECT * FROM " + device_type + " WHERE device_id=(%s) and
source='sensors' ORDER BY created_at DESC LIMIT 1;"

    data = (device_id, )
    cur.execute(get_metric_query, data)

    get_metric_query_result = cur.fetchone()


    if metric in get_metric_query_result.keys():
```

```python
      result_metric = get_metric_query_result[metric]

      r = make_response(str(result_metric))
      r.status_code = 200

    else:
      r = make_response("Metric not found")
      r.status_code

  else:
    r = make_response("Device not found")
    r.status_code = 404

  return r

@app.route('/metrics/<device_serial_number>', methods=['GET'])
def get_all_indicators(device_serial_number):
  cur = g.db.cursor()

  get_device_query = "SELECT * FROM device WHERE serial_number=(%s);"
  data = (device_serial_number, )
  cur.execute(get_device_query, data)

  get_device_query_result = cur.fetchone()

  if get_device_query_result is not None:
    device_id = get_device_query_result["device_id"]
    device_type = convert_device_type(get_device_query_result["device_type"])

    get_metric_query = "SELECT * FROM " + device_type + " WHERE device_id=(%s) and source='sensors' ORDER BY created_at DESC LIMIT 1;"

    data = (device_id, )
    cur.execute(get_metric_query, data)

    get_metric_query_result = cur.fetchone()

    for key, value in get_metric_query_result.items():
      if isinstance(value, decimal.Decimal):
        get_metric_query_result[key] = float(value)

    r = make_response(jsonify(get_metric_query_result))
    r.status_code = 200

  else:
    r = make_response("Device not found")
    r.status_code = 404

  return r

@app.route('/metadata/<device_serial_number>/<metadata>', methods=['GET'])
def get_metadata(device_serial_number, metadata):
  cur = g.db.cursor()

  get_device_query = "SELECT * FROM device WHERE serial_number=(%s);"
  data = (device_serial_number, )
  cur.execute(get_device_query, data)

  query_result = cur.fetchone()

  if query_result is not None:
    result_metadata = query_result[metadata]

    r = make_response(result_metadata)
    r.status_code = 200

  else:
    r = make_response("Device not found")
    r.status_code = 404

  return r

@app.route('/metadata/<device_serial_number>', methods=['GET'])
def get_all_metadata(device_serial_number, ):
  cur = g.db.cursor()

  get_device_query = "SELECT * FROM device WHERE serial_number=(%s);"
  data = (device_serial_number, )
  cur.execute(get_device_query, data)

  query_result = cur.fetchone()

  if query_result is not None:
    r = make_response(jsonify(query_result))
    r.status_code = 200

  else:
    r = make_response("Device not found")
    r.status_code = 404

  return r


def convert_device_type(device_type):
  device_type_table_names = {
    "ELECTRONIC_DISPLAY": "electronic_display",
    "HOUSEHOLD_WASHER_DRYER_2019": "combined_washer_drier",
    "HOUSEHOLD_WASHING_MACHINE_2019": "washing_machine",
    "HOUSEHOLD_REFRIGERATING_APPLIANCE_2019": "refrigerator",
    "HOUSEHOLD_DISHWASHER_2019": "dishwasher"
  }

  return device_type_table_names[device_type]


# API for sensor data

@app.route('/device_power/<brand>/<model>', methods=['GET'])
def get_device_power_week(brand, model):
  token = 'NipdVNkwbQ0kXiNbXBQ7P--iQZX0WkAbdrDAoFgNFwsRULUpIF6U70_pn-
viDVeeaocMmXgUCUcgt-QgCkTZFw=='
  org = 'example-org'

  client = InfluxDBClient(url="http://localhost:8086", token=token, org=org)

  query_api = client.query_api()

  p = {"_device_brand": brand, "_device_model": model}

  records = query_api.query_stream('''
      from(bucket: "electrical_signatures")
        |> range(start: -1w)
        |> filter(fn: (r) => r["_field"] == "power")
        |> filter(fn: (r) => r["_device_model"] == _device_model and r["_device_brand"] == _device_brand)
      ''', params=p)


  result = [point.values for point in records]

  r = make_response(jsonify(result))
  r.status_code = 200

  client.close()

  #r = make_response("Device not found")
  #r.status_code = 404

  return r

if __name__ == '__main__':
  app.run(host='0.0.0.0', port=5000, debug=True)
```