Crowdsourcing Smart Home Data

Technical report

Team Leader - Martinho Tavares[1], Frontend Dev - Diogo Monteiro[1], Backend Dev - Camila Fonseca[1], DevOps Master - Rodrigo Lima[1], and Supervisor - Diogo Gomes[1]

[1]DETI, University of Aveiro

June 24, 2022

Abstract

The main objective of this project is to, through a crowdsourcing mechanism integrated in smart homes, gather anonymous information from them. The base of this project is the Open Source Software Home Assistant and its Portuguese community.

The project involves the development of a custom component for Home Assistant to extract data and collect informed consent from the user, a Data Lake to house the smart home data, and a Dashboard for analyzing the platform health and overall data status.

With this, the project aims to generate value by providing the collected data as datasets for further analysis, such as energy consumption statistics. Since we are dealing with information that can be considered sensitive, providing informed decision making (choosing what to share), right to be forgotten, anonymization techniques and data security are important concerns that were taken into account while designing this system.

Various challenges were faced during development, which led to certain decisions being taken in order to manage risk. Despite this, we ended up with a functional system, although with performance issues that need to be taken into consideration, and simple anonymization approaches that require further work for usage in a production environment.

Keywords— Crowdsource, Smart home, Data lake, Anonymization, Home Assistant

Contents

Acronyms

ADL Activities of Daily Living. 3

API Application Programming Interface. vi, 1, 13, 20, 22–25, 28, 29, 35, 37, 41

BASH Bourne Again Shell. 25, 26

CASAS Center for Advanced Studies in Adaptive Systems. iii, 3

CKAN Comprehensive Knowledge Archive Network. 27, 28

CPHA Comunidade Portuguesa de Home Assistant. 9

CSV Comma-separated Values. 28

GDPR General Data Protection Regulation. 1, 20

HA Home Assistant. 19, 22

HACS Home Assistant Community Store. 22

HDFS Hadoop Distributed File System. 23, 25, 26, 37

HTTP Hypertext Transfer Protocol. 24

HTTPS Hypertext Transfer Protocol Secure. 22

ISO International Organization for Standardization. 27, 29

JSON JavaScript Object Notation. 19, 23, 28

MB megabytes. 26

UI User Interface. 42

UTC Coordinated Universal Time. 29

UTF Unicode Transformation Format. 23

UUID Universal Unique Identifier. 19, 20, 22, 23, 26, 29, 35

VCS Version Control System. 13

XML Extensible Markup Language. 28

List of Figures

List of Tables

1 Introduction

Smart homes - homes with automation systems that monitor and control home elements such as lighting, temperature and appliances - are a large market with huge expected growth in the near future[1]. These automation systems have the potential to enhance users' living experience and improve energy saving.

However, despite the large number of already existing smart homes and their foreseen growth, few public statistics and data sets relating to their day-to-day usage exist, with most studies and analytics being focused on smart home adoption and market value instead, as elaborated further ahead in section 2. Due to the absence of large and varied data sets, research on the specifics of smart home usage can be difficult and biased, depending on the data collection methodology.

From this observation came about the "Crowdsourcing Smart Home Data" project, with its goal being the creation of a data set, compiled from the data of real smart home users who volunteer to contribute, which will provide a starting point for future research on the field of home automation.

The solution, as initially planned, would involve a data lake structure to store the collected data, an extension to a smart home system that volunteers may install to collect data, and a set of APIs to store and retrieve data from the data lake.

One important aspect of this project was compliance with the General Data Protection Regulation (GDPR) - the volunteers must remain anonymous, be able to opt-out and request to have their data deleted, among other aspects.

Another vital aspect is a focus on keeping our solution open-source, to be transparent in our collection and treatment of data as well as easily audit-able.

2 State of the art

We conducted a survey on currently available datasets concerning smart home data. Below are 2 examples of the state of the art surrounding this subject, and we analyze them as possible solutions to the problem presented.

2.1 CASAS datasets

The Center for Advanced Studies in Adaptive Systems (CASAS) provides datasets on smart homes, which include information about diverse sensors, such as light switches and temperature sensors. CASAS publicly provides these datasets to fuel investigation on matters such as home automation[2]. CASAS is also focused on Activities of Daily Living (ADL) analysis, with the studied sensors being setup to analyze activities for detecting patterns and making predictions so that smart home systems can take actions to achieve some determined goal, usually of improving the users' living comfort.

People are able to volunteer into the Smart Home in a Box program to provide data from their homes, although it's through specialized equipment provided by CASAS. Our project differs from this approach by letting smart home users use their already installed systems, requiring only the installation of our extensions.

While the number of participants is fairly representative for some of the published datasets (e.g. 400 participants for the Cognitive assessment activity dataset[3]), the user scope is still limited overall.

At the time of writing, the real-time smart home statistics that CASAS also provides is outdated by 1 year[4]. Collecting up-to-date datasets is also dependent on CASAS, as they are the ones that collect the data and publish the datasets. One advantage, however, that has to be noted is that this previous treatment of data by CASAS before the publication of a dataset allows for proper labeling of the data, which is not something we are concerned with in this project.

2.2 UK-Dale dataset

UK-Dale[5] is an example of a very comprehensive dataset concerning home data. The subject of the dataset is electricity usage of various household appliances in 5 homes from the UK in a time series format, sampled approximately once every 6 seconds, and it is available for public study.

The problem with this dataset is that it has a limited region scope, being only applied in the UK, and samples a small number of homes. Also, the data can't be obtained in real time, being only available 3 years after the start of its collection.

Still, this dataset is a great example of the goal of our project: providing detailed statistics of home appliances which can be later analyzed for useful purposes, but not limited to certain regions and number of users covered and allowing data extraction at any time throughout collection.

3 Conceptual Modeling

3.1 Requirements

3.1.1 Functional

These requirements mainly concern privacy and standardization matters. While these are usually considered non-functional requirements, in this specific case they are key features of the project, since user privacy is one of the most important aspects.

- Guarantee anonymity of the data sent:
  - [**COMPLIANCE**][**HIGH**] • Provide a prompt to volunteers in the Lovelace Card to allow a formal consent to data collection
  - [**PRIVACY**] [**HIGH**] • The anonymization procedure should be on the side of the Home Assistant Aggregator, and not after the data has left the Volunteer's Home Assistant installation.
  - [**PRIVACY**] [**HIGH**] • Only provide "aggregated" data, don't work with data that is isolated (for example, only a single user that is providing data in a certain time interval)
  - [**PRIVACY**] [**HIGH**] • Ignore/blacklist certain kinds of information, like location data
  - [**COMPLIANCE**] [**HIGH**] • Guarantee that the system is GDPR compliant, and mention it to the users

  These requirements are extremely important, since if these are not fully achieved then users won't trust our platform, no data will be sent and no-one will be able to extract any value from it.
- Follow standards for data exportation:
  - [**COMPLIANCE**] [**HIGH**] • Follow the CKAN standard for data dumping in the Export API

3.1.2 Non-Functional

- [**SECURITY**] [**LOW**] • Protect against upload malicious/manipulated data (worms, spammers, or other types of malware)
- [**ROBUSTNESS**] [**HIGH**] • If there are irregularities in data upload reliability between volunteers, then fill missing info with AI]
- [**SECURITY**] [**LOW**] • Prevent third parties from abusing the individual record deletion with random/specific IDs
- [**PERFORMANCE**] [**HIGH**] • The data uploading to the Ingest API won't be real-time, it will be done in a batch-processing fashion (to reduce load)
- [**PERFORMANCE**] [**HIGH**] • The Query API has to provide real-time processing/transformation of the data that is sent to the dashboard
- [**USABILITY**] [**HIGH**] • Pagination in the Query API
- [**DEPLOYMENT**] [**MED**] • Both the Home Assistant Aggregator and Lovelace Card should be available on the Home Assistant Community Store (HACS)

(a) Volunteer (b) Researcher (c) Administrator

Figure 1: Actors in this project

- [**OPEN-SOURCE**] [**HIGH**] • All developed code should be open-source, as well as the tools used
- [**DOCUMENTATION**] [**MED**] • Usage of the Home Assistant Aggregator & Lovelace Card should be properly documented

3.2 Actors

For this project, three Actors were identified.

- **Volunteer** Users with the Aggregator installed in their Smart Homes. These users volunteer the data that's collected on the Data Lake.
- **Researcher** Users that want to extract value from the data stored by using it for any purpose, such as analysis work or academic research.
- **Administrator** Users with privileged access to the Data Lake infrastructure, that ensure the system works smoothly.

3.3 Use Cases

The Use Cases can be cleanly divided across the two Actors previously mentioned.

3.3.1 Volunteer

- **Send data** from their Home Assistant installation. The user should be able to choose which data to send, via a whitelist, and be able to change it at any point in time after installation. They should also be able to see what data is going to be sent.
- **Opt-Out** from sending data, as well as request the deletion of all previously sent data.
- **Receive notifications** on sensitive data that has been redacted before being sent.

3.3.2 Researcher

- **Download Data** that has been collected so far for further research, in various CKAN compliant formats.

3.3.3 Administrator

- **Observe and analyze** data about the data collected itself - number of active volunteers, amount of data collected, graphs on the data volume evolution through time - and the data lake status.

3.4 Architecture

The project was built with some key principles in mind, namely scalability, security, and resilience. To accomplish the set goals, every component needed to be created and integrated cautiously.

The project was firstly defined with a simple scaffold to guide future components. This scaffold was a non-comprehensive list of components from a minimal viewpoint of the project as a whole. The list was defined as follows:

- **Data Aggregator** Component to be integrated with the Home Assistant open source project[6] that would then communicate with our project's backend.
- **Dashboard** A visual GUI to enable the user to interact with the project's user-side settings. This could be in the form of a Lovelace card (Home Assistant native component) or a web page. And additionally, a data visualization component that would enable metrics from the Data Lake[7] and other components to be visualized.
- **APIs** Three API endpoints, that would make possible to extract data from the data storage, enable users to have their collected data uploaded, and allow querying the data internally.
- **Data Lake** A data storage component that would be able to store large data volumes.
- **Project website** Publicly available website containing the project's documentation and information.

3.4.1 Logical Model

Three unique actors make up the logical model. As previously mentioned, these actors are the Volunteer, the Researcher, and the Administrator.

By adding the Aggregator Integration[8] to their Home Assistant local setup, with the Home UI from the Home Assistant project, the Volunteer can interact with our project. The Aggregator Integration allows users to define their preferences for sharing their data anonymously with the back end. If everything is set up correctly, the Integration will send data on behalf of the user automatically, on a regular basis.

The Researcher can retrieve and analyze the data stored in the Data Lake via the Export API, which allows the user to filter the data that they wish to obtain by timestamps, and also choose the CKAN[9] compliant format in which the data will be provided.

Interacting with the project's backend dashboard allows the Administrator to view the project health status and system metrics. To access the dashboard, the Administrator must be inside of the project's private network. Once inside, they can either query the metrics or create new dashboards, having the proper admin privileges.

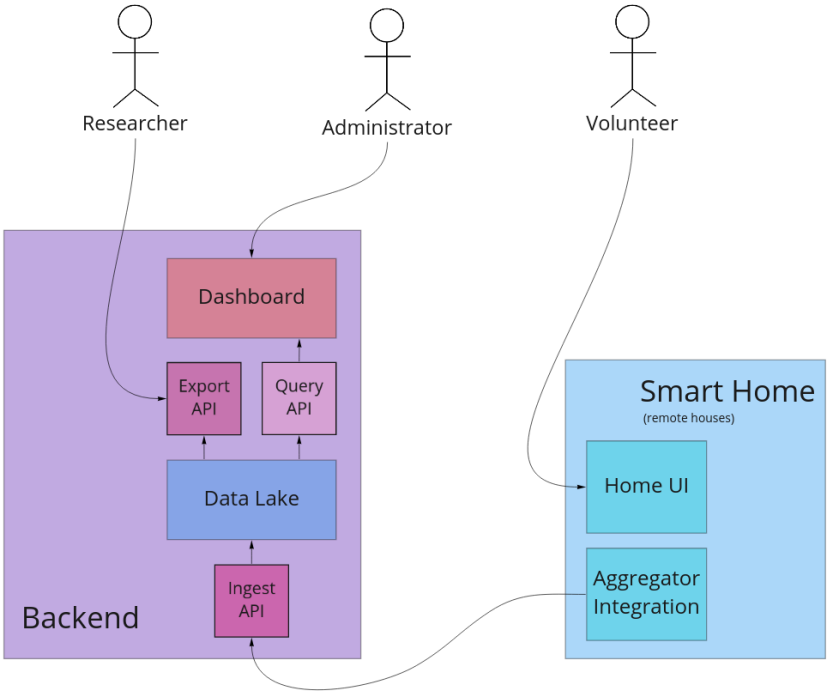


Figure 2: Logical Model

3.4.2 Implementation Model

With the architecture's minimal view of the components in mind, the decision for which technologies to be used needed to be made carefully, for each of the components.

Starting with the data storage component, or data lake, the technologies used were a stack consisting of Hadoop Distributed File System or HDFS[10] for short, YARN[11], Spark[12], and Hudi[13].

The HDFS layer was chosen because it would allow for better scalability, availability, and fault tolerance. HDFS is also deeply documented for data lake usage and although not usually used in bare metal, (As currently, it is usual to use HDFS on the cloud) it is still currently being supported.

With the main file system defined, the choice for the resource management component (Which is used to manage resources for the worker's data, enable file system operations or job scheduling) was simple. The YARN project was the best fitting technology for the job found, since it has splendid support for HDFS, updated and complete documentation and it is currently supported as well.

To schedule operations on the HDFS, or jobs, the Spark technology was used. This layer would enable remote job scheduling and better programmable interaction with HDFS. It was also chosen because it has good support for YARN and it's natively supported by the last layer of our Data Lake component, which is Hudi. Hudi was chosen because it enables the Data Lake to store any type of data while still keeping a coherent scheme and interaction with the data being stored. It also enables one key factor needed for the project, which was to allow users to delete their sent data. The Data Lake component is then fully defined with all of the technology layers previously presented, but a problem still remains, security.

With only these layers of technology in use, the HDFS and other layers were publicly available for all the users in the same private network that's being used to host the system. To solve this problem, other technologies were analyzed in order to offer better security. The technologies researched were Apache Ranger and Kerberised Hadoop[14].

Apache Ranger[15] is a technology made to harden HDFS's security, with a lack of solid implementation examples and dubious support for our use case. (Although HDFS is used on the Data Lake component, the implementation used is far from the commonly used - This will be later documented on the deployment model sub section) For these reasons, the technology was discarded as a viable approach to securing the Data Lake component.

Kerberised Hadoop is an implementation used to harden the HDFS security with the help of Kerberos[16], which is mostly used for Active Directory[17] security. Kerberos is also supported for HDFS, but due to documentation and similar implementations for this technology being nearly nonexistent, the nonstandard implementation of HDFS used in this project, and a better way to approach the problem at cause being found, the technology was discarded as a solution. The better approach to implementation for this problem of security was isolating the Data Lake component from the network. This implementation will be later documented in the deployment model sub-section.

Designed to be a central control hub for smart home devices, Home Assistant is an open-source software for home automation based on Python. It can be installed on multiple platforms, which makes it a very flexible solution, and has garnered a large number of users and communities around it. One such community is the "Comunidade Portuguesa de Home Assistant (CPHA)", i.e. the Portuguese Community of Home

Assistant users.

For the Dashboard component, it was decided to go with a Lovelace card instead of the web page route for user-wise settings interactions. This was due to having an easier implementation, and the native support for HTML[18] components (both the Lovelace card and the web page settings are built with the HTML). For the metrics and system status data visualization part of the dashboard, Grafana[19] was chosen to implement it. It would allow for easy visual representation of the data, and the implementation for it is well documented. The technology was also used due to Hudi, used in the Data Lake component, since it has support for the middleware[20] that would be later used in the query API, which is also natively supported by Grafana.

For the various APIs components (Particularly the query API) we used Prometheus[21], which is supported both by the dashboard's Grafana and the Data Lake's Hudi software. We used both Prometheus software and a Prometheus helper, named PushGateway[22] in this implementation, due to compatibilities in data sharing methods that are later described. Prometheus would enable ease of configuration and ease of use in the back end, being a backbone for data sharing. In both export and the ingest APIs, a technology called Swagger was used, which enables both code and documentation generation, given the right configurations for the APIs. To host these APIs, uWSGI[23] was used, serving Flask[24] with code generated from the previously mentioned Swagger[25]. Due to problems encountered later on, a caching system for the APIs was deemed necessary to use. For this purpose the technology chosen was Redis[26], which is a NoSQL[27] in-memory database that enables easy access to data without having to write it to disk first, on the hosting system.

For the Project's Website component, the static website generator HUGO was used, with the theme Doks[28]. It enabled easy document creation and publishing because it builds web pages using Markdown[29] source code, which was already being used on the project's private documentation, that was being hosted on Notion[30], also a Markdown-based note-taking application. To host the Project's Website Nodejs[31] server hosting is used, after building the website itself with Nodejs.

The Project also has a component that serves both as load balancer[32] and reverse proxy[33]. This component was built with Traefik[34], and it was chosen due to its complete and updated documentation, ease of use, and out-of-the-box compatibility with Let's Encrypt[35]. In this case, the compatibility with Let's Encrypt is crucial because it allows the Home Assistant component to send its data via HTTPS[36], therefore being encrypted while on journey between the user and the Ingest API.

3.4.3 Deployment Model

The deployment model for the project will follow closely the logical and implementation model. The deployment of the project will be relying on three different systems that can be seen in Figure 5.

The first of these systems is the local Home Assistant instance that each user runs. For this, the aggregator and Lovelace card components are hosted in Home Assistant's user-created content repository, HACS[37]. HACS enables the user to install these components on their Home Assistant instance, which then enables communication and data sharing with the other project components.

The second system is one of the two VMs[38] available to use in this project. All the services running within this system were orchestrated using docker-compose[39], which enabled quick and a more intuitive management of the needed services. The project is also be hosted on IT[40]'s network, which forces the communication to be

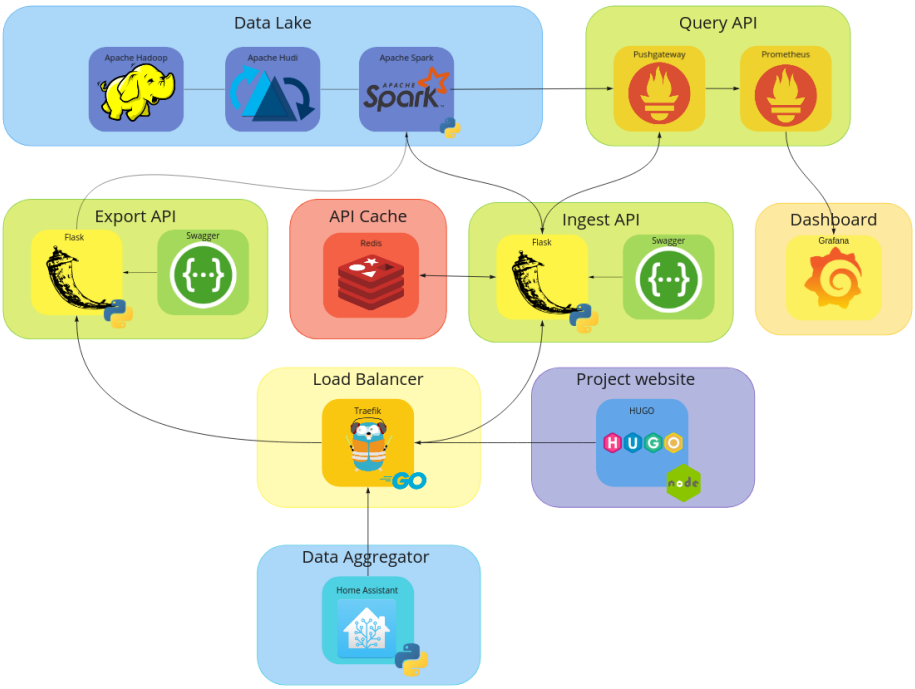


Figure 3: Implementation Model

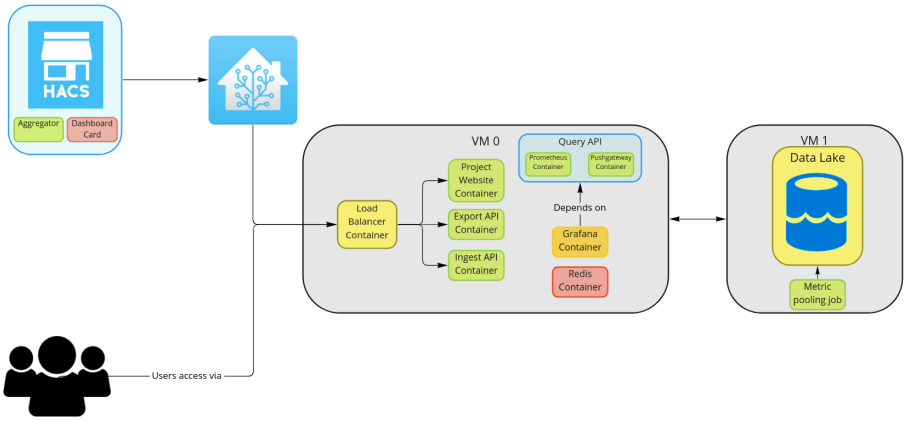


Figure 4: Deployment Model

done either through port 80 or 443, due to the firewall rules applied on the network. To solve this problem, Traefik was used as load balancer and reverse proxy inside a docker[41] container, which enables us to route URL paths to each service running on the system, which allows us to have more than only two services hosted, despite only the two available ports. The use of Traefik also enabled ease of configuration for the HTTPS certificates due to having native support for the Let's Encrypt certificate authority and auto-renewal of the certificates when they eventually expire.

With Traefik routing, the services are publicly exposed so that the users can access the export and ingest APIs and the project's website. Each of the APIs were hosted within a container and on a flask-based server with uWSGI serving the APIs. Although not routed by the Traefik reverse proxy, the Ingest API needs a caching component to be reliable. For this, a Redis service was hosted within a container that could be accessed by the Ingest API. The project website was also be hosted within a container but it was be built and served with the help of Nodejs, due to the static website generator engine Hugo needing to be built with it.

The Query API component has two services running within containers. One of these is the Pushgateway service, which enables sharing of data between the Data Lake component and Prometheus. With these services running, a backbone for data sharing and querying within the system was obtained, which enabled data and metrics to be operated by a visualization service.

The final service running on the first VM was Grafana, also running within a container, which is the data visualization service that enables an Administrator user to query and visualize data and metrics related to the system. This service is not public-facing, which means that Traefik won't route to it, and he user must be inside the project's private network to access it.

The third and final system was in the second VM available for this project, which hosts the Data Lake component and a metric polling job service. Due to I/O limitations, the Data Lake component weren't hosted within a container but directly on the hosting system, obtaining better performance at the cost of reliability. The Data Lake was exposed on a private network between the two VMs given, hardening the Data Lake's security and prevent external access. A deeper look into the Data Lake component can be found in the Storage subsection.

3.5 Mock-ups

The primary means of user interaction with the data collection component is through a custom card for Home Assistant's dashboard system "Lovelace", which enables users to view information on their gathered data, configure collection settings and perform operations such as requesting their data to be deleted. To model the intended user interaction, for the purpose of validating intended features and guide the card development, mock-ups of the card were made.

In figure 5 we see the mock-up for the card's main screen, where the user may see information relating to the sending of their data such as the size and date of sent packages, and navigate to the card's other menus.

Figure 6 shows the collection settings menu, where the user consents to have their data collected, and selects which sensors are allowed to be sent. This menu would remain a mere mock-up however, as later on in development collection settings were moved out of the card, as will be detailed in sections 4.1.1 and 4.1.2.

The data deletion menu, as seen on figure 7, clarifies its function and requests confirmation from the user to perform the operation.

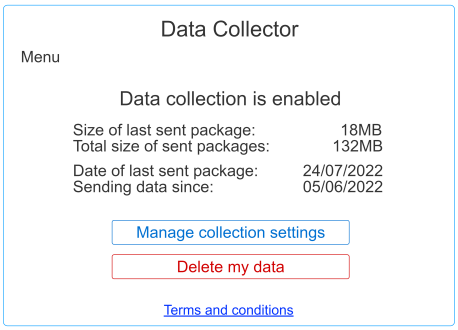


Figure 5: Lovelace card's main screen

This mock-up was later used as reference when developing the card and bears similarity to the end result, with some key changes made, detailed in section 4.1.2.

3.6 Management

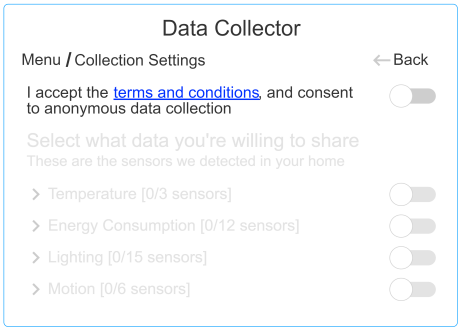
Within the course it was developed in, the project followed the OpenUP framework. Therefore, we utilized an Agile workflow, since almost all of the technologies we had to work with were new and required a learning process, which allowed the project to be more resistant to changes in requirements and any problems that we could face in development.

For task organization, we opted to use an approximation to Kanban Boards with units of work being represented by generic tasks instead of user stories, due to the project being more heavy on building the infrastructure that would allow the data crowdsourcing rather than user interaction with the system. We used weekly sprints, with 3 meetings per week so we could frequently gauge the project's progress throughout the sprint and provide feedback or help in case any team member faced an issue.

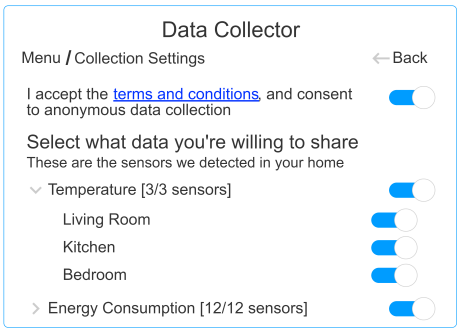
For backlog management, we used Jira[42], and for the Version Control System (VCS) we used GitHub, with an organization where we hosted all of the project's code repositories[43]. All repositories, except for one, did not follow any specific VCS workflow, as they were usually worked on by a single person and were very modular, and so there wasn't the need to divide the work between various development branches. The `crowdsource-smart-home-data` repository was the exception, which followed the Gitflow workflow[44], as it housed various components of the project (such as the APIs and the dashboard) and was worked on by two people, therefore requiring better organization.

In terms of resource/notes management, and for internal documentation, we utilized Notion . For brainstorming and development of diagrams we took advantage of Miro[45]. We also integrated a GitHub hook into Discord, our internal communication tool, which allowed us to easily track the various changes within the organization's repositories.

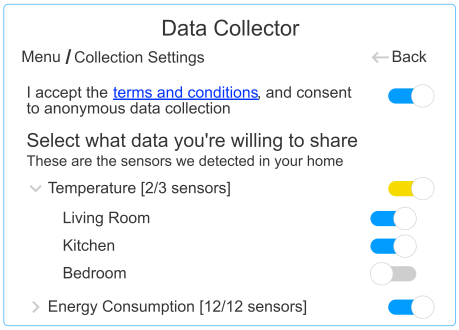
The final project roadmap, after all changes done during development, is presented in figure 8. Details on the decisions and challenges faced that led to the presented roadmap are explained in section 5.



(a) Disabled consent toggle



(b) Enabled consent toggle with all sensor collection enabled



(c) Enabled consent toggle with partial sensor collection enabled

Figure 6: Data collection configuration screen

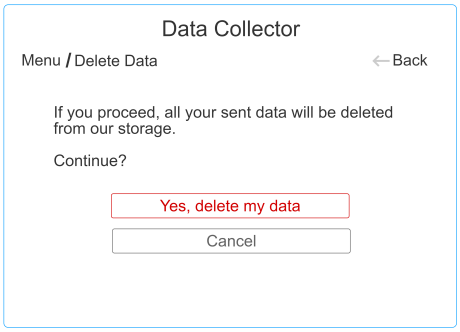


Figure 7: Data deletion screen

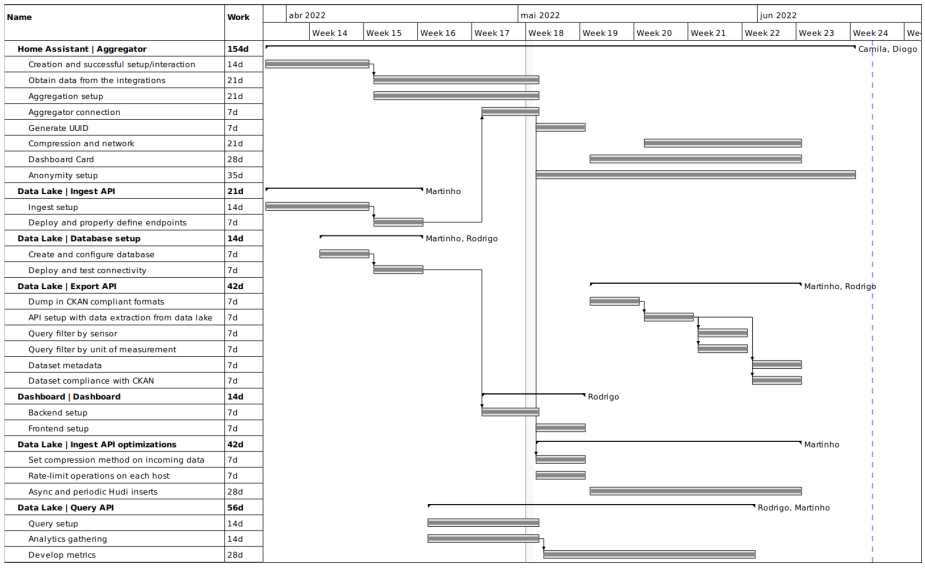


Figure 8: Final roadmap

4 Procedure

4.1 Home Assistant

4.1.1 Integration

Overview

The Aggregator integration is the Home Assistant component that collects and filters the data that the data lake stores. It is written in Python, as it is a Home Assistant Integration.

On installation, it allows the user to customize what data they allow to be collected, and it will then run periodically, collecting, filtering and finally transferring the collected data to the Ingest API.

Since our integration depends on the cloud to work, it is considered a Cloud-Polling integration[**?**] for Home Assistant classification purposes.

Configuration

Making use of Home Assistant's Configuration Flow, the user can restrict the data they want to send, according to the entity group that generates it. Internally, this is done via an initial query, in order to discover all the existing entities (Everything that generates data), which then are split according to their type in order to generate a custom Configuration Schema (The form that is presented to the user).

This might introduce a slight delay, depending on the hardware Home Assistant is running on, and the number of existing entities, but it allows for a dynamic list, without the need for an extensive and immutable list of hard-coded sensors.

On a first install, the user is presented with a form where they can choose which entities to add to the whitelist, either one by one or with blanket options that allow to send data from All or None of the entities.

After submitting this form, the Aggregator completes its initial configuration, generating an UUID that will be permanently assigned to that user's installation and creating an entry in Home Assistant's configuration entries. (An internal mechanism to store persistent data.)

To access this persistent data, the following code snippet can be used:

```
entries = self.hass.config_entries.async_entries()
for entry in entries:
    entry = entry.as_dict()
    if entry["domain"] == "data_collector"
    and entry["title"] == "options":
        # Insert your code here
```

After installation, the user can reconfigure these settings, but not change their UUID. Configuration changes and UUID persist through Home Assistant reboots, but not through reinstalls.

The aggregator uses a single external dependency, **cs-scrubadub**[**?**], which is a custom fork of the data scrubbing package, scrubadub[**?**], altered to our purposes due to conflicts with Home Assistant's own dependencies and slimmed down, to trim unneeded functions and reduce download size for the Volunteers.

Collecting Data

The integration is configured to run once a day, at a fixed time that's randomly picked between 0 and 6AM. The time is randomized to help avoid overloading the Data Lake, and using up volunteers' bandwidth during hours it might be in use.

To collect the data, the Aggregator utilizes Home Assistant's own methods to query its internal database and retrieve all state changes between two specific dates. The first time it runs, it'll fetch data since the start of the day it's being run, but for subsequent runs it'll fetch new data since the last time it ran.

While querying the data, the Aggregator will filter according to the entity whitelist set by the volunteer during the configuration phase. A whitelist was chosen instead of a blacklist so the user has more control over which entities' data is effectively being collected, and data from new sensors isn't automatically sent.

In case the user has chosen to send data from no sensors or no data is found in the time interval, the process is immediately aborted; No empty data is sent to the Data Lake.

Cleaning Data

For data anonymization, different techniques were used, with varying degrees of success. First, and starting off with the data in a key:value pair dictionary format, we filter out values with certain keys that are in a list of banned keys. These are well-known fields and common enough to warrant this kind of blanket cleansing. (Notably, GPU coordinates and user-id)

After, the data is dumped into string format and cleansed with the scrubadub package.

The default scrubbers[**?**] are used, which include:

- Phone numbers
- Social Media handles (Twitter/Instagram format)
- Credit Card numbers
- Emails

...among others. Alongside these, a customized Scrubadub filter was used to filter out keywords present in word lists, that aim to filter:

- Portuguese and English names
- Location names in Portugal (Cities, Districts...)
- Country names

Lastly, Regex-based filters are used to filter out:

- IP Addresses
- Portuguese Postal Codes (Scrubadub recognizes only British Postal Codes)

After the anonymization process, the data is then sanitized in order to remove a number of characters that are incompatible with the Data Lake and cause problems, which are all replaced by underscores, taking precautions to carefully avoid removing characters essential to the data being parsed back into a JSON format.

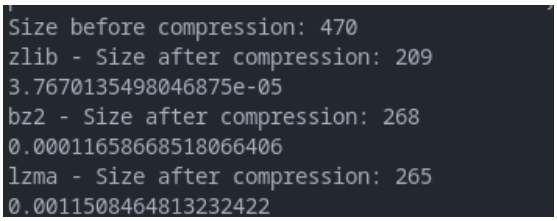


Figure 9: Compression Algorithms Comparison: Speed and final size.

Transferring Data

After filtering, the data is dumped from JSON into a string format, encoded into 'utf-8' bytes format and compressed using Zlib[**?**].

Why Zlib? In order to choose a compression algorithm, three were tested. Compression speed and the final file size were compared. Among the three algorithms (Zlib, bz2 and lzma), zlib outperformed the other two in both parameters, and so, was chosen.

After compression, the data is finally sent to the Data Lake in a POST request, with the user's UUID as a header and the data as an octet-stream. Finally, after obtaining a response, a notification is generated for the volunteer to know that data was sent, and how many bits of data were redacted from it.

4.1.2 Dashboard Card

The dashboard card with which the volunteers can interact was implemented through HA's dashboard system "Lovelace", which features user-customizable layouts where different cards can be used to display information and control sensors in their home.

Internally, Lovelace cards are displayed and controlled through HTML, CSS, and JavaScript, with JavaScript libraries such as Lit[46] to enable simpler scripting, state control and rendering. Additionally, Lovelace cards are typically written in TypeScript[47] to enforce type syntax and encourage cleaner code. Our custom card follows suit, and uses these same technologies in its implementation.

Our custom card offers the following features:

- **View information about sent packages**: Users can see in the main screen information about the size of the packages they send, and the date they are sent, as seen on figure 10.
- **View assigned ID**: The assigned UUID for the current Aggregator installation can be viewed. It is recommend that users save this ID in a safe place, as it is the key to delete their data should they uninstall the Aggregator. See figure 11a.
- **Download last sent data**: The last data to be sent can be downloaded as a JSON file, so that users may view in detail what information is collected and sent. Shown on figure 11b.
- **Delete all previously sent data**: Users may request to have their data deleted from the data lake, and in doing so will receive confirmation on whether the operation was completed or not. Refer to figure 12a.

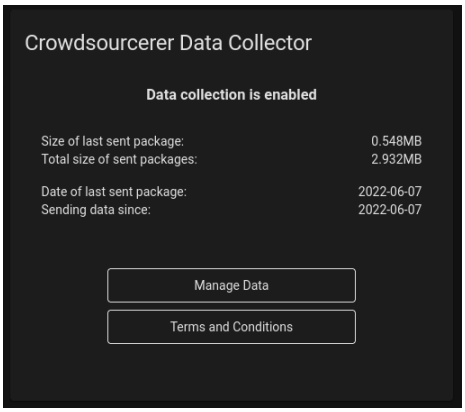
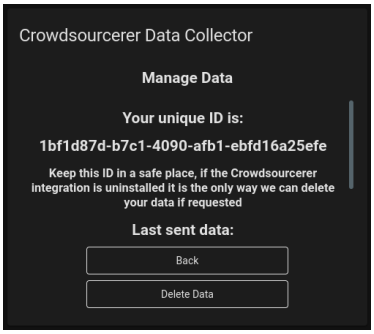
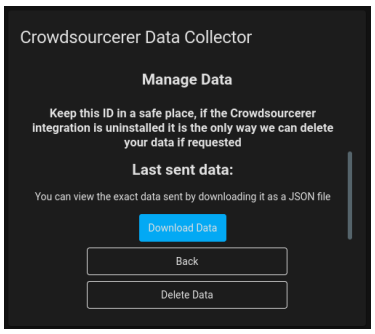


Figure 10: Lovelace card's main screen



(a) Viewing the UUID through the card



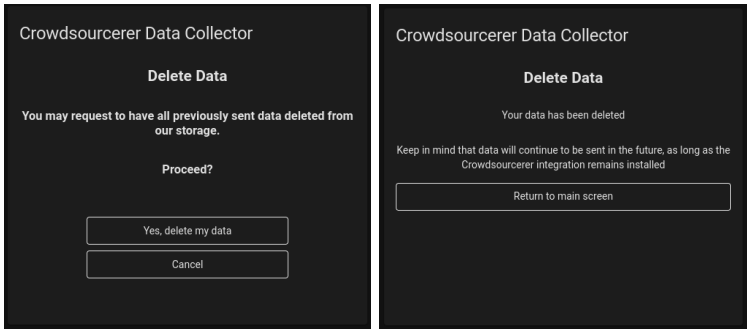
(b) Button to download latest sent data

Figure 11: Lovelace card's data management screen

- **View terms and contact information**: Users can refer to questions and answers to clarify common doubts and concerns, and view contact information such as the Data Protection Officer's email address. This is shown on figure 13.

In terms of design, the card closely follows the initial mock-ups (detailed in section 3.5), while functionality-wise features were added or reworked. Namely, the collection settings from the mock-up were scrapped, as this feature presented technical difficulty and would have involved undesirable workarounds in its implementation. New features like viewing the ID and downloading data were added to address concerns regarding GDPR compliance and situations that weren't initially foreseen, such as how would users request to have their data deleted after uninstalling the Aggregator or somehow losing their original ID.

To display information about their Aggregator installation, the card reads the associated entity's state object, a dictionary-like object that contains data assigned through the integration. The delete operation calls an API endpoint with the user's UUID.



(a) Deletion prompt screen (b) Deletion request was successful

Figure 12: collection configuration screen

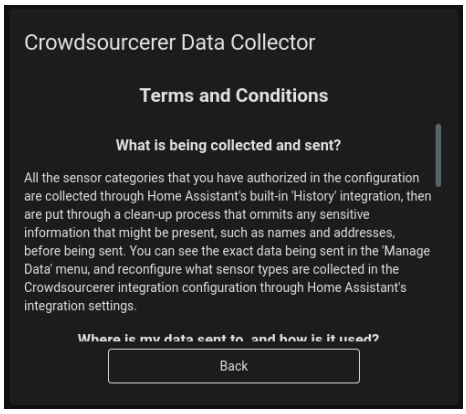


Figure 13: Lovelace card's terms screen

Endpoint	Method	Header parameters	Body
`/data`	POST	Home-UUID (required)	Home data (binary)
`/data`	DELETE	Home-UUID (required)	-

Table 1: Ingest API endpoints and parameters

The card can be installed either manually, by transferring the built JavaScript files to a local HA directory and adding the resources by hand through HA's frontend interface, or by means of HACS by adding the card's repository. The first method was the preferred one during development, while the latter will be the main means of distribution.

4.2 Backend

The backend for the Project is composed of multiple components, these being the backend's dashboard, the Export API, the Query API, the Ingest API, and the Data Lake. All of the backend's components allow the project to process, import, save and check the data being sent from the Volunteers with the Aggregator Integration. A deeper explanation for each of the given backend components will be given in the following subsections.

4.2.1 Ingest API

For data ingestion into the data lake by the users, the Ingest API was built. Swagger Codegen was used to build the Python client SDK and server stub from a documentation file following the OpenAPI specification[48]. Although the client SDKs ended up not being used due to the simplicity of the built API which didn't warrant its usage. Beyond data insertion, the Ingest API also offers an endpoint for data deletion, which is to be used by the opt-out feature of the Home Assistant Aggregator.

This API is ideally only supposed to be used by the Home Assistant Aggregator component, but there are no restrictions on where the requests come from, since those details can easily be forged.

Interface Table 1 shows the available endpoints and the parameters that can be passed to them:

The Home-UUID header parameter is the UUID stored on each Aggregator installation, and it's sent on both endpoints to uniquely identify the home to which the operations will be applied. As the communication channel is secured (using HTTPS) the UUID can't be eavesdropped. No guarantees are made that this UUID is uniquely used by a single home, even though it's very unlikely for conflicts to happen.

The two operations that can be performed in the API are:

- **Data upload**: applied with the POST method, inserts the data sent in the request body directly into the data lake, with no data treatment beyond adding metadata, and therefore it's assumed that it's already anonymized by the Aggregator. The data is compressed in zipped format
- **Data deletion**: applied with the DELETE method, and deletes all records on the data lake that are associated with the supplied UUID. For the response, there is no feedback on whether or not the UUID existed previously on the

platform, but if there were errors during deletion then it is explicitly expressed (such as Internal Server Errors).

Errors Some errors may be encountered, which may be the fault of the client or a problem with the server. Below are the custom errors for this API:

- **Malformed UUID (400)**: the supplied UUID in the header field Home-UUID is not properly formatted. This is due to the string provided in this field not matching the format of a proper UUID, which features 5 fields of hexadecimal characters
- **Bad ingest decoding (400)**: the request body provided, which includes the home data, is not encoded, compressed, or in the format required by the application. Users should make sure that it is a JSON object, is encoded using UTF-8 encoding, and is finally compressed using zlib. The strucure of the JSON object itself does not matter for this error to occur
- **Bad JSON structure (400)**: the decoded JSON data that was provided is not a JSON object. This error is thrown if the decompression and decoding of the data is successful, but the provided JSON primitive at the root level is not a JSON object `{...}` (for example, if it's an array `[...]` or a string `"..."`)

Implementation The server stub was implemented in Python with the Flask framework . In order to communicate with the data lake (HDFS) with Hudi, we used PySpark[49], which is a Spark interface for Python.

A Python module `hudi_utils` was developed containing the initialization of Spark with the Hudi packages, which is called when the API starts, and the operations that are performed on the HDFS, namely the data insertion and deletion. These operations are called on the respective web controller which represents one of the two endpoints.

The Hudi table type chosen for storing the data is Copy on Write, which presents high write amplification and latency with the trade-off of allowing faster querying when compared with Merge on Read tables[50]. One of the main reasons for opting with this table type was the Query API, which had to query the data lake in near real-time in order to provide statistics.

The Hudi table's configuration is set in the Ingest API, since Hudi allows for dynamically creating the Hudi tables when there are none, so there's no need to define its parameters elsewhere. The record key, which is the key that identifies a row within a partition of the table, is the UUID of the smart home that sent the data. The partition key, which uniquely identifies a partition of the Hudi table, is the combination of the year, month and day of the time when the data was inserted. With the record key and the partition key, a particular data upload can be identified by the day the data was uploaded and the smart home that it belongs to.

Due to performance issues on Hudi insertion jobs, we decoupled the insertion tasks from the data upload requests, and made the insertion tasks periodic and asynchronous. For the scheduling of these insertion tasks, the Flask APScheduler Python package was used[51]. On each data upload request, the data sent will be stored temporarily on a password protected Redis database, which will be queried by the periodic Hudi insertion jobs in order to store the data into the data lake. The inserted data is then cleared from Redis. Data deletion requests also clear the data from Redis.

Name	Example	Default
INGEST_BASE_PATH	hdfs://<IP>:<PORT>/	file:///tmp/
INGEST_HUDI_RATE_MINUTES	15	5
INGEST_PUSHGATEWAY_HOST	localhost	localhost
INGEST_PUSHGATEWAY_PORT	9091	9091
INGEST_REDIS_HOST	localhost	localhost
INGEST_REDIS_PASSWORD	6379	6379
PYSPARK_PYTHON	/usr/bin/python3	None

Table 2: Ingest API environment variables

Environment variables Various environment variables can be set to change the API's behavior and configure its connections with other components. Table 2 shows the environment variables that may bet set, with examples and the default values.

The environment variables are described as:

- `INGEST_BASE_PATH`: the path where the application should store the data
- `INGEST_HUDI_RATE_MINUTES`: the number of minutes between each Hudi insertion batch job
- `INGEST_PUSHGATEWAY_HOST`: host of the Pushgateway instance to which metrics will be sent
- `INGEST_PUSHGATEWAY_PORT`: port of the Pushgateway instance to which metrics will be sent
- `INGEST_REDIS_HOST`: host of the Redis instance to which requests will be cached
- `INGEST_REDIS_PORT`: port of the Redis instance to which requests will be cached
- `INGEST_REDIS_PASSWORD`: password of the Redis instance to which requests will be cached
- `PYSPARK_PYTHON`: the Python binary to use for PySpark

The `PYSPARK_PYTHON` environment variable may have to be set if the API is running on a virtual environment.

4.2.2 Query API

The Query API was tasked with providing statistical data about the data lake platform and the data itself, so that it can be monitored in the Metrics Dashboard. We opted for using Prometheus as our time series database.

For providing metrics to Prometheus, the Pushgateway server is used, which allows arbitrary jobs to push any metrics to an aggregator that will be scraped by Prometheus.

This permits various separated instances to provide metrics about the part of the platform that they are working with, instead of using a centralized API to do so. Figure 14 shows the deployment strategy for these metric provider instances.

Therefore, we ended up using 3 different metric providers:

- **Upload counter**: a counter of the number of data upload requests done to the Ingest API which are successful, i.e. returns a response with HTTP OK (200) status code

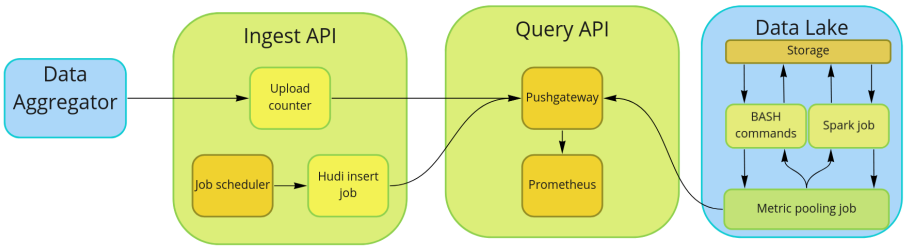


Figure 14: Metric providers deployment architecture

- **Hudi insert job**: Hudi jobs for data insertion into the data lake done at the Ingest API, which are periodic and asynchronous, and therefore independent of the data upload requests. Hudi provides an integrated means to push its batch job metrics to a Pushgateway server, by configuring it to do so.
- **Metric pooling job**: a Bourne Again Shell (BASH) script that queries the data lake for arbitrary metrics, which should be run periodically. It uses two methods for extracting data:
  - **BASH commands**: these are commands that are executed directly in the script, such as HDFS commands for analyzing the folders housing the data
  - **Spark job**: Python file submitted as a Spark job, which directly queries the Hudi table for metrics related to the stored data

Metrics We defined 6 different time series metrics for Prometheus to provide to Grafana:

- `hudi_ingestion_commit_duration`: the commit duration of the Hudi insertion jobs executed in the Ingest API, to understand if there are any errors or abnormal behavior while inserting data (such as very slow inserts)
  - **type**: `gauge`. Since the value of the commit duration can frequently oscillate over time, then the appropriate metric type should be `gauge` as it allows values to decrease and increase
  - **provider**: Hudi insert job
- `ingest_upload_count_total`: the amount of data upload requests done to the Ingest API, which is useful to analyze in comparison with the `hudi_ingestion_commit_duration`
  - **type**: `counter`. Since the number of upload requests can't ever decrease then the appropriate type is `counter`, as it's designed to be a value that only decreases. The only time that this metric decreases, to 0, is when the Ingest API is restarted, since the number of requests done is not persisted. Since this metric will be mainly used to calculate the rate of upload requests (as will be explained in section 4.2.4) its absolute value does not matter.
  - **provider**: Upload counter

- `data_lake_users`: amount of users (distinct UUIDs) currently present on the data lake
  - **type**: `gauge`. Since users can be removed from the data lake (by not opting into the data collection process and removing all data associated with the home's UUID) then the value can increase and decrease
  - **provider**: Spark job submitted by the metric pooling job
- `data_lake_size` (gauge): size of the stored Hudi table containing all smart home data, in megabytes (MB). This is effectively the size of the directory on the HDFS containing this Hudi table
  - **type**: `gauge`. Since users can be removed from the data lake (by not opting into the data collection process and removing all data associated with the home's UUID) then the value can increase and decrease
  - **provider**: command executed on the HDFS (`du`[52]) to obtain the disk usage of the folder hosting the data lake's Hudi table, executed by the metric pooling job
- `data_lake_producers`: amount of unique data producers/entities (e.g. sensors) whose data is currently stored in the data lake (this is equivalent to the number of data columns in the resulting datasets from the Export API)
  - **type**: `counter`. When home data is deleted from the data lake, only rows are explicitly deleted, but column clean up (removing entities that aren't being uploaded by any current home) is not done, so the number of data columns only increases
  - **provider**: Spark job submitted by the metric pooling job
- `data_lake_discontinued`: amount of data uploads that have been discontinued, i.e. the number of users whose data hasn't been updated in the last 30 days
  - **type**: `gauge`. This value can increase with the lack of updates, but can also decrease if the discontinued homes receive updates again, or if the discontinued homes ask to delete their own data
  - **provider**: Spark job submitted by the metric pooling job

Pooling job The periodic metric pooling job was deployed on our machine hosting the data lake, so that the HDFS could be locally queried. Since it was implemented as a BASH script, we simply scheduled the execution of the script to be done once every 15 minutes using the `cron` Unix utility[53].

The script requires some configuration variables to be set. These aren't set as arguments to the script, but in a configuration file `pushgateway_push_metrics.conf` in the same directory as the script, due to the large number of variables to set which would crowd the `cron` job definition. Table 3 shows the required configuration variables.

In order for the script to successfully execute, some environment variables relating to Java, Hadoop and Spark may have to be set. The script for setting these variables is shown in appendix A.1.1.

The `cron` job definition we applied, which runs every 15 minutes and sets the previously mentioned environment variables, is shown below (for the `hduser_` user). This assumes the code containing the metric pooling job is in the `metric-pooling-job` folder in the home directory.

Variable	Meaning	Example
`job_name`	the name of the Prometheus job	`hudi_job`
`instance_name`	the name of the instance that submitted these metrics	`storage`
`hudi_address`	the location where the Hudi table is hosted	`hdfs://localhost:9000`
`hudi_path`	the path appended to `hudi_address`, which points to the Hudi table to analyze	`/hudi_ingestion`
`pushgateway_host`	the host address of the Pushgateway server	`smarthouse.av.it.pt`
`pushgateway_port`	the port of the Pushgateway server	9091

Table 3: Metric pooling job configuration variables

Endpoint	Method	Query parameters
`/dataset`	GET	`formats, date_from, date_to, types, units`
`/formats`	GET	-

Table 4: Export API endpoints and parameters

```
*/15 * * * * cd metric-pooling-job && BASH_ENV=/home/hduser_/metric-pooli
ng-job/environ.sh ./pushgateway_push_metrics.sh
```

The demonstration of the Grafana dashboard, which is the receiver of these metrics, is detailed in section 4.2.4.

4.2.3 Export API

In order to export the stored data into a dataset, the Export API was built. The created datasets follow the CKAN standard, providing metadata that simplifies its upload to a CKAN server and are available in different formats.

In order to develop this API, we followed the same approach as the one used for the Ingest API, using Swagger Codegen to build the server stub and PySpark to extract the data from the data lake. The Pandas Python library[54] was used to work on the Spark dataframes and convert them to the dataset's output formats.

Interface Table 4 shows the available endpoints and the parameters that can be passed to them.

The `/dataset` endpoint is the one that allows exportation of data to a dataset. Below are the various query parameters that can be passed:

- `formats` *(required)*: the case insensitive strings representing the dataset's output formats
- `date_from`: only data from this date forwards will be extracted (UTC+0, in ISO 8601 format), inclusive
- `date_to`: only data from this date backwards will be extracted (UTC+0, in ISO 8601 format), inclusive

- `types`: only data from these types of producer will be extracted (e.g. sensor). An array of types can be provided
- `units`: only data which is represented in the specified units of measurement will be extracted (e.g. GHz). An array of units can be provided

The available formats are JSON, XML and CSV, and they can be accessed in the `/formats` endpoint, which provides this list as a comma-separated string. JSON is the recommended exportation format, due to the fact that Home Assistant sends data with a JSON-like structure, which is naturally embedded in a JSON document. Below is the example of a data column stored in the data lake, which corresponds to a weather integration.

```
{
    'weather_home': [
        {
            'attributes': '{
                friendly_name=Home,
                wind_bearing=334.2,
                temperature=18.6,
                humidity=68,
                wind_speed=19.1,
                pressure=1018.6
            }',
            'entity_id': 'weather_home',
            'last_changed': '{{REDACTED}}',
            'last_updated': '{{REDACTED}}',
            'state': 'partlycloudy'
        }
    ]
}
```

Sensitive data is censored, and the `attributes` key has a string value with arbitrary key-value pairs in a JSON-like structure. This format lends itself well to a JSON representation, which is why we explicitly recommend it for data analysis.

Dataset The data is provided in a zip archive, which when extracted creates the following files:

- `crowdsorcerer_extract_{date}.{format}`: file containing the data extracted. `date` is the date when the file was requested, and `format` is the format of the output file requested. If many formats were specified, then there will be multiple data files with the different formats
- `crowdsorcerer_extract_{date}_metadata.json`: file containing the metadata, which can be supplied in a request to a CKAN server

The metadata is laid out as the description of CKAN datasets and resources, and is based on the data required for the API requests that would have to be done to a CKAN server[55] to create datasets and upload resources (which are the different files that make up a dataset). The metadata we provide, therefore, can be applied to these requests, simplifying the upload process. It includes the license used for the dataset, which is Creative Commons BY-SA[56], and the filters that were applied to the data. Appendix A.2.1 shows an example of an exported dataset's metadata.

Name	Example	Default
`EXPORT_BASE_PATH`	`hdfs://<IP>:<PORT>/`	`file:///tmp/`
`PYSPARK_PYTHON`	`/usr/bin/python3`	None

Table 5: Export API environment variables

The Export API only obtains data from until yesterday, inclusive. This was done so that real-time changes to the data lake can't be monitored by extracting multiple datasets throughout the day, which could allow people to track down the precise time when a data row was inserted. All of the users' UUIDs are mapped to an integer ID, which is a global counter, so that malicious actors can't manipulate the data lake's data by taking advantage of those UUIDs.

Errors Some errors or special results may be encountered, which may be the fault of the client or a problem with the server. Below are the custom errors for this API:

- **Bad date format (400)**: the supplied date parameter (either `date_from` or `date_to`) has an incorrect format, which should be ISO 8601[57] (`yyyy-mm-dd`). The considered timezone is UTC+0, so there may be cases where this consideration may make a difference, but it shouldn't be the cause of this error
- **Unsupported exportation format (400)**: the supplied exportation format is not supported. The format string should be the same as the supported formats listed above, without case sensitivity. The API usually provides feedback on the unsupported string that was presented on the request and the strings that are actually accepted
- **Empty dataset (204)**: the provided query filters produce a dataset without data columns or rows. The filtering process produced a dataset without any data about the homes. This can happen if a query filter restricts the data with constraints that no rows comply with, such as a units filter specifying a single unit of measurement that doesn't exist in the data lake

Implementation The implementation followed a similar organization to the one used for the Ingest API, with a `hudi_utils` Python module housing the Hudi operations for extracting data from the data lake.

Environment variables Like the Ingest API, environment variables can be set to properly configure the Export API. Table 5 shows the environment variables that may be set.

The environment variables are described as:

- `EXPORT_BASE_PATH`: the path from where the application should obtain the data
- `PYSPARK_PYTHON`: the Python binary to use for PySpark

Similar to the Ingest API, the Export API may also require setting `PYSPARK_PYTHON` if the API is running on a virtual environment.

Username	Password
admin	EyJMyvObuDqNxcT*50$

Table 6: Grafana dashboard account credentials

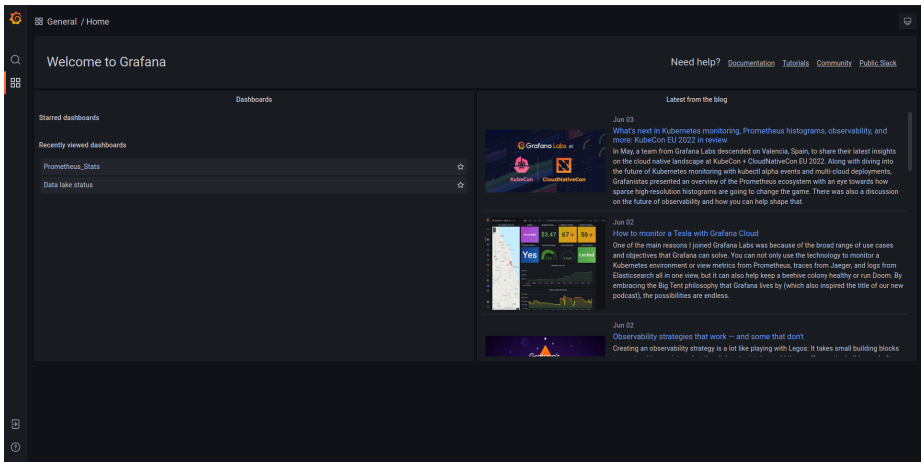


Figure 15: Grafana's Home Page

4.2.4 Metrics dashboard

The Project has a component for health and metrics interactive visualization, which enables an Administrator to quickly check on the system status or visualize system metrics from the backend.

The dashboard component was built with Grafana, which eases dashboard interaction and creation. The dashboard is only available inside the project's private network, for security reasons. The component was built with an admin account configuration and public viewing setup but was deemed too insecure to be a public-facing component. The Administrator account for the Grafana component can be seen in Table 3.

Once inside the private project's network the Administrator can access the component on the following url: https://smarthouse.av.it.pt:3000. On first arrival, the Administrator will be able to see the home page where there are built, as of time of writing, two dashboards as seen on Figure 10.

The Administrator can create their own dashboards. For this task they need to first login with an Administrator account, using the credentials previously given, and then follow Grafana's documentation to create new dashboards.

In case the Administrator selects the dashboard named Prometheus Stats, they will be met with a dashboard composed of several graphs, in which each one represents an unique metric from the Prometheus technology used on the Query API component. For a better understanding of each metrics' meaning, their labelling can been seen on the official Prometheus documentation. A preview of the dashboard can be seen on Figure 11.

In case the Administrator selects the dashboard named Data lake status, they will

Figure 16: Grafana's Prometheus Dashboard

be met with a dashboard composed of curated metrics that represent the current Data Lake component's status. The meaning of each metric can been seen on table 4 and a preview of the dashboard can been seen on Figure 12.

4.2.5 Data Lake

The Data Lake component is a core component of the project, since it is responsible for the storage of the data that is being imported for later analysis. Keeping this importance in mind, the architecture for the Data lake component was thought up having scalability and resilience in mind, because even though the storage is sufficient for the purposes needed right now, it might not be sufficient in the future. For this purpose, the HDFS technology was chosen because it addresses these concerns at its core.

Although the technology is well documented, the HDFS system is, as its name implies, targeted for distributed systems, and the Data Lake component is aimed to be hosted, as of now, on a single host. To address this issue a single node pseudo-distributed install was implemented. This install aims to distribute work, that otherwise would be distributed into other machines, onto other users and Java processes. This approach isn't 100% effective in addressing the resilience problems, because if the host goes down the Data Lake also goes down, but it is a compromise for the limitations at hand. The differences between the standard installation and the single node installation can be seen on Figure 19.

The Data Lake component was also installed on the host machine, this means that it isn't hosted inside a container but instead the HDFS is being run as an user that was created for that specific job on the host machine, the credentials of the user can bee seen on Table 6.

Due to I/O performance prevention, if the Data Lake were to be hosted inside of containers the performance would be worsened. The Data Lake technology stack also creates many listening service ports that were public to other users on the same network. To mitigate this security issue, a private network was created between the

Metric	Meaning	Query
Ingest upload rate	Rate of data uploads being done on the ingest endpoint present on the Ingest API, within an hour	rate(ingest_count_total[1h])
Ingestion time taken	Time taken for data ingestion jobs performed on the Ingest API	hudi_ingestion_commit_duration / 1e3
Data lake size	Current memory being used for the data lake, in megabytes	data_lake_size / 1e6
Data lake users	Current number of unique users present on the data lake	data_lake_users
Data lake producers	Current number of unique entities (e.g. sensors) of anonymous data to be exported on the data lake. Corresponds to the number of data columns in the resulting dataset	data_lake_producers
Data lake discontinued	Number of discontinued data uploads, i.e. home data that wasn't uploaded	data_lake_discontinued

Table 7: Grafana's Data Lake Status Metrics

Username	Password
`hduser_`	`z3eq3RS*a*TdC%QN0lFP`

Table 8: HDFS host machine user account credentials

Figure 17: Grafana's Data Lake Status Dashboard

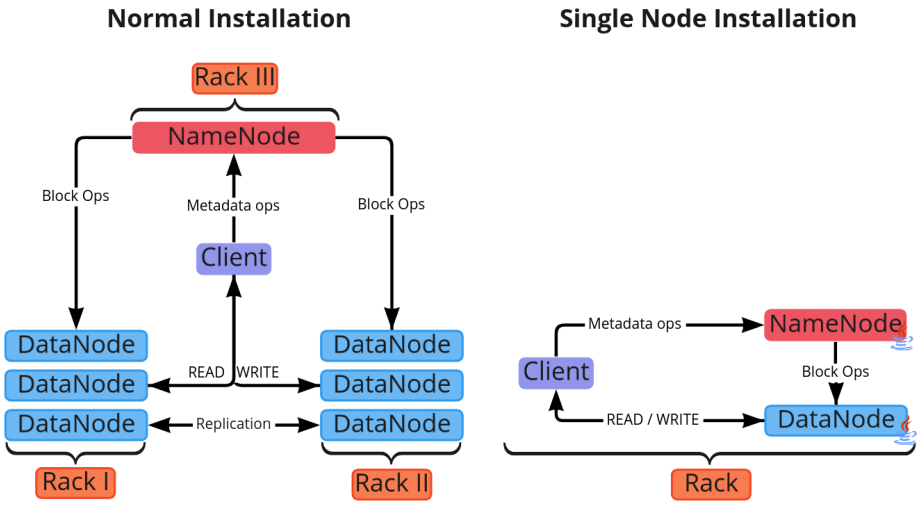


Figure 18: HDFS Installation differences

two VMs given to host the project.

For operations to be done on the Data Lake's data, it is necessary to add a resource manager, a bridge between the APIs, and a data lake CRUD operations enabling platform to bring order to the data that it's being imported to the Data Lake. For these goals, it will be used the following technologies respectively: YARN, Spark, and Hudi.

The installation of YARN was done following the official documentation and after the service startup, it didn't bring any issues to the project. In contrast, the installation of the Spark services were more troublesome. The official supported versions compatible between these two services were mislabelled on the documentation and this compelled a reinstall of the services in question. After installation of the right versions, the Spark service enabled the request of remote operations on the data being hosted on the Data Lake while the Hudi service enabled the use of schemas for the data that would enable the greater organization of the data in question and enable deletion of user-specific data, one of the requirements for the project.

Although built for resilience, the Data Lake component is built with technologies that are far from thoroughly documented for the specific task of building a Data Lake without relying on the Cloud (hosting the data lake with the help of other paid services). This oversight creates a problem of lack of examples and difficult problem-solving in case of errors and misconfiguration, and most of the problem-solving is done relying on non-English speaking, obscure blog posts dating back to several years ago and where their contents might be outdated or dubious.

5 Discussion

In this section we give details on the various key decisions that were made in this project and the reasons behind them. A retrospective analysis is done, and we check if we could have taken a different approach to some of the problems we faced.

We also evaluate the results of the project, presenting its compliance with the requirements set at the beginning, and whether some subjects should be given greater attention in future work.

5.1 Ingest API

Python was chosen as the development language for the Ingest API due to the familiarity that we had with it and its support for libraries such as Pandas. However, Scala would probably have been more appropriate, as Spark is implemented in Java and using PySpark demands the use of a Java interface for Python, where Scala doesn't have the need.

Since we opted for using Copy on Write Hudi tables with batch insertions on each API call, we ended up facing the problem that operations on the data lake took too long to execute, and so the Home Assistant Aggregator had to wait a long time before a response was sent.

Therefore, we first attempted to apply rate-limiting on the endpoints using Flask-Limiter[58], since usually they don't have to be executed more than once everyday for each home (insertion is done every 24 hours, and normally there is no reason to delete the same data again). However, it posed problems, as it made debugging more convoluted (we didn't want to be rate-limited while testing the endpoints) and didn't work well when deployed with uWSGI, due to the multiprocessing it applies.

Ultimately, after speaking with the supervisor, this feature was dropped in favor of other features that were much more important to implement. Fortunately, little development time was lost, as the library used to implement the rate-limiting feature heavily simplified the process. The code developed for this is still present, but commented out.

After we tested frequent insertion payloads from the Aggregator (once every 30 seconds) we noticed that the data lake was struggling to keep up, presenting generic threading issues. This further prompted us, along with the long response times, to come up with a solution to the long insertion times. We couldn't apply many tricks to speed up the responses for deletion operations because the response from the API has to give the guarantee that the supplied UUID is not present in the data lake, and therefore needs to be synchronous (unless we implemented a notification system that would notify the user when the UUID was deleted, but time didn't allow it).

Therefore, we opted for doing the Hudi insertion jobs asynchronously as periodic tasks. After receiving insertion requests, the home data would be stored in a cache, which was later queried by the periodic Hudi insertion task to finally insert it into the data lake in a batch fashion. This brought the advantage that we can tune the Hudi insertion frequency to any desired value, as well as the fact that responses to these requests become much faster. For the cache used, we initially opted for the uWSGI's, which allowed a shared memory space between the multiple processes, but due to its low-level interface we ended up choosing Redis for its simplicity.

If we had more time to explore other solutions to the performance issues, we would have tried using Merge on Read Hudi tables instead of Copy on Write, which are more apt for frequent data ingestion into the data lake and, in hindsight, would probably

be more adequate for our case. The problem of slow querying by the Query API is attenuated since, in our simplified approach, the scraping done by Prometheus is periodic, allowing periodic merging of the tables which reduces the overhead of these operations on future read queries, such as those done by the Export API. But fear of facing other problems due to changing the table type, which could require other considerations when working with it, led us to prioritize other aspects of the project.

There are also other approaches we could have taken that the Hudi documentation recommends we follow for increasing the data lake performance[59], which could also be explored in future work.

5.2 Query API

We were initially thinking of implementing the Query API the same way as the Ingest API, using Swagger Codegen and PySpark for querying, but as we started to investigate possible ways in which we could provide different metrics we found other possible solutions.

Since we were going to implement the Metrics Dashboard with Grafana (detailed in section 4.2.4), we searched for possible data sources, and Prometheus looked simple and good enough for our purposes. Considering as well the fact that Hudi provides a built-in metrics reporter for a Prometheus Pushgateway server , which we could leverage to provide metrics on the ingestion batch jobs, we ended up choosing Prometheus as the metrics provider for Grafana.

This greatly simplified our approach to metric generation, as we simply could make use of the Pushgateway server to push statistics of ephemeral batch jobs, leaving the saving, querying and display of data is left up to Prometheus and Grafana. In order to get statistics other than the ones provided by Hudi jobs, we planned to create a metrics pooling task that would be run periodically and query the data lake for arbitrary statistics, and then push to the Pushgateway server which in turn would be scraped by Prometheus.

However, as we were investigating this solution, we began questioning the role of the Query API. If metrics can be easily queried by Prometheus and provided to the Pushgateway server, then what is the Query API in this case, and how does it fit? After clarifying this issue with the supervisor, we were allowed to follow this simplified pooling approach, even if it wasn't what was initially intended. But since it achieved the overall objective of providing metrics to the Dashboard in a performant fashion, we could opt for this solution, which would allow us to invest time into other important features related to the home data ingestion and exportation.

It has to be noted, however, that this is not the recommended way of using Prometheus' Pushgateway server[60]. The Hudi insertion job use case is valid, as these are indeed ephemeral batch jobs, but there is a better and more reliable alternative to the metrics pooling idea we had, which is to expose a `/metrics` endpoint in an API that Prometheus would scrape directly, instead of using the Pushgateway server as a middle man. We still ended up making use of Pushgateway because:

- Pushgateway simplifies providing the metrics (a simple POST HTTP request), which makes it very flexible. For instance, we made use of the Pushgateway server to easily provide upload request metrics by the Ingest API without requiring anything more than a HTTP request. Therefore, and due to limited time we had available, we decided to follow this compromise
- we were already planning to use the Pushgateway server for the metrics of the

Hudi batch jobs, so we decided to make use of it for arbitrary metric pooling as well instead of building another API just for this purpose

- due to the very small number of instances that we are working with (two machines, with one of them housing Prometheus itself) the issue of Pushgateway being a single point of failure is much less pronounced
- the automatic instance health monitoring that Prometheus provides (through the "up" metric) is not a type of metric we were mainly concerned with including, as the number of instances in our case is very small
- Pushgateway doesn't forget series pushed to it, and so they will remain available to be scraped unless they are manually deleted through its API, but since our case is controlled (the data is only provided by two machines, and none of the series is left to become outdated, as they are all designed to be constantly updated) we believe this didn't prove to be a big issue

Even so, the ideal approach is still to expose a `/metrics` endpoint for scraping, since at the moment the periodic batch job presents the disadvantage that the Spark environment has to be constantly started and ended on every metric pooling task run, as opposed to an ever-present API that only needs to start the Spark application once. This would, in turn, allow for smaller metric scraping intervals, as there wouldn't the overhead of starting/ending Spark.

5.3 Export API

Converting the data queried from the HDFS into a dataset is done by converting the Spark dataframe obtained from the querying operations into a Pandas dataframe, which is very resource intensive since it loads the entire dataframe into memory. Better strategies could be followed so data is converted in a streaming fashion, or smart caching methods could be applied so intensive operations are done less frequently, but due to time constraints a proper performant and scalable solution wasn't explored.

5.4 Project Website

The development of the project website lead to a back and forth development life cycle that introduced wasted time due to rollbacks.

The project required us to develop an official documentation webpage so that other users could see the documentation and obtain a better understanding of the project's inner workings and applications. To meet this requirement it was developed a github page that would host documentation, simple information about the project and instructions for future users of the project.

Alongside the projects development life cycle we were met with more requirements for the website, instead of developing on top of the previous website that wasn't suited for the new requirements, it was decided to create a new website that would meet the new and future requirements. Due to the previous shortsighted approach, the development time used for the first version of the website ended up being wasted.

Having a "blank canvas" to work on, the development of the new website was aiming to shorten the time needed for new documentation creation, content updates and better segregation of contents hosted on it, previously all of the content was hosted on a single page. To accomplish this goals it was decided to use the static website generation engine named Hugo that would make it easier for any developer to create new content due to it using Markdown for page content creation. Alongside this engine

it was decided to use the theme Doks that has solid support for documentation creation and also it was decided to use a CI/CD pipeline with Netlify, enabling developers to update contents of the Website faster while hosting the website for free.

With this approach the new website was being hosted on Netlify instead of github and both users and developers would be met with a better experience while interacting with the website. Although the approach met all of the website requirements, this approach was short-lived. This is due to the fact that while developing this website, we were mistaken on the grafana's component role.

The grafana's component was developed, while aiming for it to be public facing, so the hosting of it was aimed to be used on the root path of the port 443 of the public facing VM, but this approach was in fact a misunderstanding of the requirements and the configuration and hardening that was made to the grafana's component was in the end not needed for the project. With this misunderstanding we disabled public access of the grafana's component and were met with the dilemma of what to host on the root path of the domain.

In the end it was decided to host the webpage, that was already created and hosted on Netlify, in the public facing VM alongside the Ingest and Export API. This decision would lead to more time wasted due to the CI/CD pipeline now being deprecated and the hosting on the Netlify platform being disabled. In the end the final approach was the one that better met all of the requirements even though some time was lost in reconfiguration.

5.5 Reverse Proxy

With multiple public facing services being hosted on this project, the management of the routes their access would require were decided to be handled by a reverse proxy. The goal of this reverse proxy was to enable access of multiple services, in this cage the Project's Website and the Ingest and Export APIs, to users by just requiring the use of different URL paths on the ports 80 or 443 of the VM that is hosting those services.

The first approach was to rely on the Nginx project to route those services, but due to a new project's requirement, it was decided to use the project Traefik instead. This switch of reverse proxies approaches ended up wasting time due to the development of the reverse proxy that used Nginx being done when the new requirement was discovered.

The new discovered requirement was the need of a secure data transmission between the users and the project, to meet this requirement it was decided to use HTTPS so that the data would be natively encrypted between transport.

The use of HTTPS proved to be troublesome even though the decision to change the reverse proxy service was done with meeting that goal in mind. The change of Nginx to Traefik was due to the native support of Traefik to Let's Encrypt, being Let's Encrypt a non-profit certificate authority that would facilitate the creation and renewal of certificates needed for encryption. The problem of this approach originates on the fact that the technology was new and the problem that it tried to solve was complex in nature. This led to further wasted time, but in the end the requirement was met.

5.6 Data Lake

The development of the Data Lake component showed to be troublesome due to several issues. By having multiple complex technologies at it's core, the development of the Data Lake component was expected to be complex and time consuming, this ended up being underestimated.

Before approaching the development of the Data Lake component, multiple solutions where designed, the end result was a stack of Apache technologies that would meet the requirement expected from this project.

Although some time was used to research the technologies needed to be used on the development of the component, due to every technology being new to each developer working on it and due to lack of resources, a lot of time was spent on troubleshooting and deployment.

The first problem arrived due to the lack of documentation on approaches to build a Data Lake with the technology Stack at hand. This problem arises due to the fact that currently companies use mainly already built components from Cloud providers. This leads to documentation being mainly in non English and dubious.

Other roadblocks that were met during development were mislabeling of compatible versions on the Hudi documentation that lead to the re installation of the Spark technology layer and the complexity of securing the access to the HDFS component.

The task of hardening the security of the HDFS component lead to time being spent on reading documentation about Kerberos and also lead to a failed installation of Apache Ranger to deal with security, both approaches were later discarded in favor of a private network between the Data Lake's host machine and the public facing machine.

5.7 Dashboard Card

During initial planning and architecture modeling, two approaches were considered for user interaction with the Home Assistant integration. The first approach was to develop a custom card for Home Assistant's dashboard system "Lovelace". The alternative was to create a website that, when accessed from the user's home network, would connect to the local HA installation. The former approach is more direct, as a dashboard card would be a part of the user's HA environment, same as the integration, and would provide information passively after being setup. Meanwhile, a website would require the user to actively seek information as opposed to it being present at all times, though it could provide additional and more advanced features. After considering both options, the card approach was chosen as we deemed having the user interaction component being a part of smart home system more valuable.

5.8 Aggregator

Due to a lacking and incomplete documentation for Home Assistant Integration development, the Aggregator development was very slow, ending up taking over double the initially planned. Every feature added revealed a whole new slew of issues, such as saving persistent data, how to execute a blocking method call (For network requests, in this case.) and accessing Home Assistant's internal databases, as some pertinent examples.

Problems and errors were incredibly hard to solve, most without an immediately apparent cause and a Google search turning virtually no results. Searching Home

Assistant's Discord server for past conversations and other integrations' code for comments ended up being the only ways to find out how to implement many functionalities.

Testing the Aggregator outside its development environment - on a raspberry pi instead of a relatively capable laptop - also introduced various layers of problems, some of which are explored in subsections ahead. One of which was the total lack of usable logs in case of crashes, which led to a large amount of time blindly debugging and applying - often erroneous - patches. However, this testing uncovered a large amount of edge case scenarios that would not have been considered otherwise, and much less fixed or mitigated.

5.8.1 Performance

Performance issues ended up being critical for the Aggregator operation, with Home Assistant being able to be installed in hardware vastly ranging in capabilities. A slow or intensive operation running on a fully-featured computer, taking a couple of minutes, would easily take up upwards of an hour on hardware such as a Raspberry Pi (A common host for Home Assistant), *while* locking up the entire system, *and* sometimes requiring a cold reboot of the hardware. Testing the Aggregator on different hardware also revealed many issues with what was otherwise considered to be completed features and code, requiring a lot of work redoing parts, often with little error logs, if any at all. Due to serious performance issues found last-minute, some anonymity filters had to be completely removed, since no solution was found in time and taking hours in processing data was not considered acceptable, since we would be using up the Volunteers' resources.

Even if the anonymization was the most intensive operation, by far, querying the internal database itself could also be a time-consuming operation, depending on the size of the Home Assistant installation. Since running the integration more frequently would result in less data being collected and processed on each go this was considered as a possible counter-measure, but ultimately not implemented due to concerns of possibly slowing down Home Assistant during active hours instead of the middle of the night, which is the time it would otherwise run.

A concerning issue was also discovered, where the Aggregator would show significant differences in execution speed in a bigger installation, with more sensors and running processes. The Aggregator was tested both in minimal, small and large-sized Home Assistant installations, but performance on anything larger is something that still demands some improvement.

5.8.2 Anonymity

Data anonymity is an area of the Aggregator that was predicted to be troublesome, and this was confirmed during implementation. The library that was chosen for the project, Scrubadub[**?**], despite remaining the best tool found for the task, ended up falling short of expectations due to a multitude of filters being only usable for data originating from the United Kingdom, and these had to be manually implemented to use with Portuguese formats. Notable example being postal codes. We quickly realized total anonymization would be much harder to implement due to the data being completely unpredictable, so all the filters present are in a Best-Effort situation - We tried to make them as wide-ranging as possible without incurring in too many false positives.

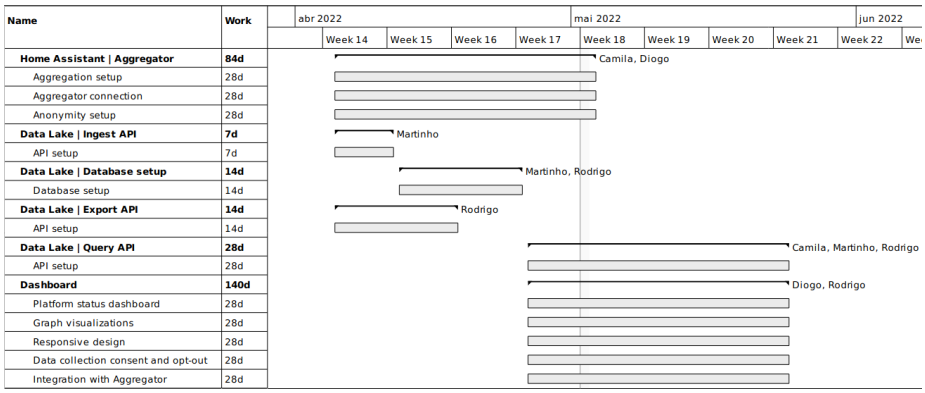


Figure 19: Roadmap at Inception phase

Scrubadub iself was found to have a fatal flaw when testing outside of a development environment, since one of it's own dependencies, numpy, was required in a version incompatible with Home Assistant itself. To solve this, a fork of scrubadub was created, cs-scrubadub, and the necessary alterations were made to make it compatible with Home Assistant. While this was time-consuming, taking it out of the project would have been a great loss.

Additionally, data filling with AI on the data lake was originally a feature of our project, where missing data from users would be filled according to their presented trends. However, it had to be scrapped entirely due to time constraints, losing the extra layer of data anonymization it would've otherwise provided. Even if it decreased the value of the resulting datasets by applying our own AI models for predicting missing values (which in itself can be controversial), it is an important asset to protecting users' privacy when data uploads are interrupted. Since these interruptions can happen for diverse reasons, identification attacks could be performed for extrapolating information based on the cause of these interruptions.

5.9 Project management

Management of this project proved to be difficult. Issues met during development led to delays of the modules' defined delivery dates, which can be evidenced by the altered roadmap in later stages of development. For reference, figure 19 shows the roadmap defined at the Inception OpenUP phase of the project, which is at the beginning.

There are a few noteworthy points to be made about our first version of the roadmap, in comparison with the final one:

- The time taken for the Home Assistant module was severely underestimated, since we couldn't predict the problems faced while working with the Home Assistant environment
- The time taken for the Query API was overestimated, but this was due to the quicker and easier solution that we ended up opting for
- The Export API was misplaced, being put to be worked on in parallel with the Ingest API, even though the exportation function is dependent on the data that the Ingest API inserts

- The UI for interaction with the Aggregator component, from which informed consent and opt-out can be done, was intended to be done within the web-based dashboard where visualization of the platform and overall data's status was also performed, as was explained in section 5.7
- Overall, the work is divided between pairs of developers, with each of the team members ending up working on more than one project module with another different member. This didn't end up happening, as the Home Assistant modules' delivery date extended to the end of the project deadline, and so those that were working in these modules stuck with them until the end of development. Not only that, the collaborative work that we desired for each module wasn't fulfilled, and so each work module ended up being the isolated work of mostly one developer, since we prioritized parallel development of each module, for what we believed would bring faster delivery. This led to knowledge of each module being mostly concentrated on a single developer.

In retrospective, we could have done better risk management, especially in a project of this nature where the team lacked familiarity with the tools and technologies we were going to use. When we started facing various problems with these tools, we began fearing that, by applying pair programming, we would be losing time as less modules would be worked on in parallel. However, pair programming could have proved to be more resilient to these problems as the collaborative nature of this workflow can allow for better reflection on issues being faced, where one person is focused on developing the module and the other is looking for solutions to those issues[61].

If we could have planned this project differently, we would attempt to not isolate each developer to a module of the project, opting for pair programming or rotation of developers, so that more than one person is deeply aware of each project module, even if at first it seems to slow down development. In the long-term, these strategies could have proven to be more efficient.

6 Conclusion

Crowdsourcing potentially sensitive smart home data requires special treatment so that anonymization procedures are applied and data can be minimally secured. To tackle this complex problem, we took advantage of open-source solutions, building a system, which is itself also public, by coupling these different components together. This allowed us to gain experience in autonomously developing with up-to-date community-backed projects, which is also what ended up being the biggest hurdle for us to overcome.

Developing a project with new technologies that we weren't previously familiar with posed a huge challenge in project management in order to deal with blocking issues that we were facing along the way. As was previously shown, this effect can be evidenced by the roadmap changes that the project severely suffered throughout development.

We still ended up meeting the basic project requirements, providing a system where, through a Home Assistant integrated crowdsourcing mechanism, datasets can be extracted containing smart home data from any volunteering home, including a maintenance dashboard for analysis of the platform and data's overall status.

Still, the final system requires further consideration and development to assure that identification attacks by combining outside information can't be reliably executed, so while data anonymization efforts were certainly applied to achieve a minimal level of privacy, there is still further work that needs to be done so that it can be safely deployed in a production environment.

References

[1] M. Armstrong, "The market for smart home devices is expected to boom over the next 5 years," Jun. 2022, [Online; accessed 5. Jun. 2022]. [Online]. Available: https://www.weforum.org/agenda/2022/04/homes-smart-tech-market

[2] "About CASAS," Jun. 2022, [Online; accessed 16. Jun. 2022]. [Online]. Available: http://casas.wsu.edu/about

[3] D. Cook, A. Crandall, B. Thomas, and N. Krishnan, "CASAS: A smart home in a box," 2013. [Online]. Available: http://eecs.wsu.edu/~cook/pubs/computer12.pdf

[4] "CASAS Stats," Feb. 2021, [Online; accessed 16. Jun. 2022]. [Online]. Available: http://casas.wsu.edu/smarthomestats

[5] "UK Domestic Appliance Level Electricity (UK-DALE) - Disaggregated appliance/aggregated house power," 2015, [Online; accessed 5. Jun. 2022]. [Online]. Available: https://ukerc.rl.ac.uk/DC/cgi-bin/edc_search.pl?GoButton=Detail&WantComp=41

[6] Home Assistant, "Home Assistant," Jun. 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://www.home-assistant.io

[7] Contributors to Wikimedia projects, "Data lake - Wikipedia," Apr. 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Data_lake&oldid=1083865142

[8] "Integration Architecture | Home Assistant Developer Docs," Jun. 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://developers.home-assistant.io/docs/architecture_components

[9] "CKAN - The open source data management system," Jun. 2022, [Online; accessed 6. Jun. 2022]. [Online]. Available: https://ckan.org

[10] "What is HDFS? Apache Hadoop Distributed File System | IBM," Jun. 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://www.ibm.com/topics/hdfs

[11] "Apache Hadoop 3.3.3 – Apache Hadoop YARN," May 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[12] "Apache Spark™ - Unified Engine for large-scale data analytics," May 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://spark.apache.org

[13] "Hello from Apache Hudi | Apache Hudi," Jun. 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://hudi.apache.org

[14] S. Loughran, "What is Kerberos? · Hadoop and Kerberos: The Madness Beyond the Gate," Jun. 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://steveloughran.gitbooks.io/kerberos_and_hadoop/content/sections/what_is_kerberos.html

[15] M. O. Ching, "Apache Ranger – Introduction," Nov. 2021, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://ranger.apache.org

[16] Contributors to Wikimedia projects, "Kerberos (protocol) - Wikipedia," Jun. 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Kerberos_(protocol)&oldid=1093303076

[17] iainfoulds, "Active Directory Domain Services Overview," Jun. 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://docs.microsoft.com/en-us/windows-server/identity/ad-ds/get-started/virtual-dc/active-directory-domain-services-overview

[18] Contributors to Wikimedia projects, "HTML - Wikipedia," May 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=HTML&oldid=1090322066

[19] "Grafana: The open observability platform," Jun. 2022, [Online; accessed 6. Jun. 2022]. [Online]. Available: https://grafana.com

[20] Contributors to Wikimedia projects, "Middleware - Wikipedia," Apr. 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Middleware&oldid=1081799492

[21] Prometheus, "Prometheus - Monitoring system & time series database," Jun. 2022, [Online; accessed 6. Jun. 2022]. [Online]. Available: https://prometheus.io

[22] prometheus, "pushgateway," Jun. 2022, [Online; accessed 6. Jun. 2022]. [Online]. Available: https://github.com/prometheus/pushgateway

[23] "The uWSGI project — uWSGI 2.0 documentation," Oct. 2021, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://uwsgi-docs.readthedocs.io/en/latest

[24] "Welcome to Flask — Flask Documentation (2.1.x)," Jun. 2022, [Online; accessed 6. Jun. 2022]. [Online]. Available: https://flask.palletsprojects.com/en/2.1.x

[25] swagger api, "swagger-codegen," Jun. 2022, [Online; accessed 6. Jun. 2022]. [Online]. Available: https://github.com/swagger-api/swagger-codegen

[26] "Redis | The Real-time Data Platform," Jun. 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://redis.com

[27] Contributors to Wikimedia projects, "NoSQL - Wikipedia," Jun. 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=NoSQL&oldid=1093266308

[28] "Modern Documentation Theme," Jun. 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://doks-child-theme.netlify.app

[29] "Markdown Guide," Jun. 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://www.markdownguide.org

[30] "Notion – One workspace. Every team." Jun. 2022, [Online; accessed 6. Jun. 2022]. [Online]. Available: https://www.notion.so/product?fredir=1

[31] Node.js, "Node.js," Jun. 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://nodejs.org/en

[32] Contributors to Wikimedia projects, "Load balancing (computing) - Wikipedia," Jun. 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Load_balancing_(computing)&oldid=1092155455

[33] Wikipedia contributors, "Reverse proxy — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Reverse_proxy&oldid=1089196899, 2022, [Online; accessed 4-June-2022].

[34] "Traefik Labs: Makes Networking Boring," Jun. 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://traefik.io

[35] "Let's Encrypt," Jun. 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://letsencrypt.org

[36] Contributors to Wikimedia projects, "HTTPS - Wikipedia," Jul. 2001, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=HTTPS&oldid=1093171840

[37] "Home Assistant Community Store | HACS," Jun. 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://hacs.xyz

[38] "What Is a Virtual Machine and How Does It Work | Microsoft Azure," Jun. 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://azure.microsoft.com/en-us/overview/what-is-a-virtual-machine

[39] "Overview of Docker Compose," Jun. 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://docs.docker.com/compose

[40] I. T.-I. de Telecomunicações, "IT - website," Jun. 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://it.pt

[41] "Home - Docker," May 2022, [Online; accessed 15. Jun. 2022]. [Online]. Available: https://www.docker.com

[42] Atlassian, "Jira | Issue & Project Tracking Software | Atlassian," Jun. 2022, [Online; accessed 6. Jun. 2022]. [Online]. Available: https://www.atlassian.com/software/jira

[43] "CrowdSorcerer," Jun. 2022, [Online; accessed 6. Jun. 2022]. [Online]. Available: https://github.com/CrowdSorcerer

[44] Atlassian, "Gitflow Workflow | Atlassian Git Tutorial," Jun. 2022, [Online; accessed 6. Jun. 2022]. [Online]. Available: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

[45] "About Miro | Meet the team | Our mission," Jun. 2022, [Online; accessed 7. Jun. 2022]. [Online]. Available: https://miro.com/about

[46] "Lit," Jun. 2022, [Online; accessed 7. Jun. 2022]. [Online]. Available: https://lit.dev/

[47] "Typescript: JavaScript With Syntax For Types," Jun. 2022, [Online; accessed 7. Jun. 2022]. [Online]. Available: https://www.typescriptlang.org/

[48] "Home - OpenAPI Initiative," Apr. 2022, [Online; accessed 6. Jun. 2022]. [Online]. Available: https://www.openapis.org

[49] "PySpark Documentation — PySpark 3.2.1 documentation," May 2022, [Online; accessed 6. Jun. 2022]. [Online]. Available: https://spark.apache.org/docs/latest/api/python

[50] "Table & Query Types | Apache Hudi," Jun. 2022, [Online; accessed 6. Jun. 2022]. [Online]. Available: https://hudi.apache.org/docs/table_types

[51] "Flask APScheduler," Apr. 2022, [Online; accessed 16. Jun. 2022]. [Online]. Available: https://viniciuschiele.github.io/flask-apscheduler

[52] "du(1) - Linux manual page," Jun. 2022, [Online; accessed 7. Jun. 2022]. [Online]. Available: https://man7.org/linux/man-pages/man1/du.1.html

[53] Wikipedia contributors, "Cron — Wikipedia, the free encyclopedia," 2022, [Online; accessed 7-June-2022]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Cron&oldid=1091715941

[54] "pandas - Python Data Analysis Library," Jun. 2022, [Online; accessed 7. Jun. 2022]. [Online]. Available: https://pandas.pydata.org

[55] "API guide — CKAN 2.9.5 documentation," Apr. 2022, [Online; accessed 7. Jun. 2022]. [Online]. Available: http://docs.ckan.org/en/2.9/api/index.html#

[56] "Creative Commons — Attribution-ShareAlike 4.0 International — CC BY-SA 4.0," Jun. 2022, [Online; accessed 10. Jun. 2022]. [Online]. Available: https://creativecommons.org/licenses/by-sa/4.0

[57] "ISO 8601 — Date and time format," Jun. 2022, [Online; accessed 7. Jun. 2022]. [Online]. Available: https://www.iso.org/iso-8601-date-and-time-format.html

[58] "Flask-Limiter," Apr. 2022, [Online; accessed 6. Jun. 2022]. [Online]. Available: https://flask-limiter.readthedocs.io/en/stable

[59] "Use Cases | Apache Hudi," Jun. 2022, [Online; accessed 16. Jun. 2022]. [Online]. Available: https://hudi.apache.org/docs/use_cases#data-lake-performance-optimizations

[60] Prometheus, "When to use the Pushgateway | Prometheus," Jun. 2022, [Online; accessed 6. Jun. 2022]. [Online]. Available: https://prometheus.io/docs/practices/pushing

[61] "On Pair Programming," Jun. 2022, [Online; accessed 16. Jun. 2022]. [Online]. Available: https://martinfowler.com/articles/on-pair-programming.html

A Backend

A.1 Query API

A.1.1 environ.sh

```
#!/bin/bash

# Set HADOOP_HOME
export HADOOP_HOME=~/hadoop
# Set JAVA_HOME
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
# Add bin/ directory of Hadoop to PATH
export PATH=$PATH:$HADOOP_HOME/bin
# Set SPARK_HOME
export SPARK_HOME=~/spark
# Add bin/ directory of Spark to PATH
export PATH=$PATH:$SPARK_HOME/bin
# Set LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$HADOOP_HOME/lib/native:$LD_LIBRARY_PATH
# Custom bins
export PATH=$PATH:~/bin
```

A.2 Export API

A.2.1 Dataset metadata example

```
{
    "name": "crowdsorcerer-extract",
    "title": "CrowdSorcerer extract",
    "author": "CrowdSorcerer",
    "license_id": "cc-by-sa",
    "notes": "Crowdsourced smart home data collected from the CrowdSorcerer open source project. More info
    "resources": [
        {
            "package_id": "crowdsorcerer-extract",
            "url": "upload-json",
            "format": "json"
        },
        {
            "package_id": "crowdsorcerer-extract",
            "url": "upload-csv",
            "format": "csv"
        }
    ],
    "extras": [
        {
            "key": "date_from",
            "value": "2022-04-28"
        },
```

```
        {
            "key": "date_to",
            "value": "2022-06-03"
        },
        {
            "key": "types",
            "value": "['sensor']"
        },
        {
            "key": "units",
            "value": "any"
        }
    ]
}
```

B Code repositories

All code repositories are present under the Crowdsorcerer GitHub organization at: `https://github.com/CrowdSorcerer`

C Website

The website where all project documentation is present can be accessed here: `https://smarthouse.av.it.pt/`.

If the website is down, then the same documentation can be accessed on the website's GitHub repository: `https://github.com/CrowdSorcerer/crowdsource-smart-home-site/tree/main/content/en`

C.1 Installation instructions

For instructions on how to install the Home Assistant Data collector and Dashboard card, follow the following links:

- Data collector
  - Website: `https://smarthouse.av.it.pt/docs/installation/data_collector/`
  - GitHub: `https://github.com/CrowdSorcerer/data_collector/blob/main/README.md`
- Dashboard card
  - Website: `https://smarthouse.av.it.pt/docs/installation/lovelace_card/`
  - GitHub: `https://github.com/CrowdSorcerer/crowdsourcerer-card/blob/master/README.md`