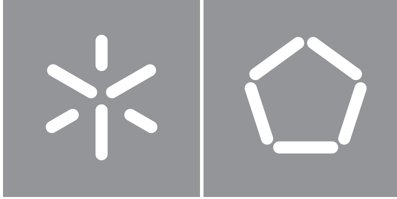
Universidade do Minho

Escola de Engenharia

Pedro Daniel Ferreira Duarte

Implementation of microservices and network management for Cyber-Physical Systems

Fevereiro, 2023



Universidade do Minho
Escola de Engenharia

Pedro Daniel Ferreira Duarte

Implementation of microservices and network management for Cyber-Physical Systems

Dissertação de Mestrado

Mestrado em Engenharia Eletrónica Industrial e Computadores

Sistemas Embebidos e Computadores

Trabalho efetuado sob a orientação do

Doutor Duarte Manuel Azevedo Fernandes

Fevereiro, 2023

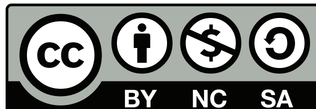
COPYRIGHT AND TERMS OF USE OF THIS WORK BY A THIRD PARTY

This is academic work that can be used by third parties as long as internationally accepted rules and good practices regarding copyright and related rights are respected.

Accordingly, this work may be used under the license provided below.

If the user needs permission to make use of the work under conditions not provided for in the indicated licensing, they should contact the author through the RepositóriUM of Universidade do Minho.

Licença concedida aos utilizadores deste trabalho



Creative Commons Atribuição-NãoComercial-CompartilhaIgual 4.0 Internacional

CC BY-NC-SA 4.0

https://creativecommons.org/licenses/by-nc-sa/4.0/deed.pt

Acknowledgements

First of all, I would like to thank my advisor, Dr. Duarte Fernandes, for his availability in guiding this dissertation and for all the support he provided throughout its development. I would also like to thank the staff at DTx, especially Sofia Paiva and Rui Machado, for accompanying me through the various stages of this project and always being available to help me. To Professor Jorge Cabral, I thank him for the opportunity to carry out this project in his group and for the knowledge he imparted throughout the years of the course.

To my parents, for all their concern, patience, and investment in me, a huge thank you. Without you, this journey would not have been possible.

Not forgetting those who were part of my daily life during these years, to my friends and members of the infamous Mosca da Fruta group, I thank you for all the moments we spent together. These were challenging years, but by your side, they provided many joys. My heartfelt thanks to all.

Finally, to all who directly or indirectly contributed to my academic journey, my sincere thanks.

Project "(Link4S)ustainability - A new generation connectivity system for creation and integration of networks of objects for new sustainability paradigms [POCI-01-0247-FEDER-046122 | LISBOA-01-0247-FEDER-046122]"is financed by the Operational Competitiveness and Internationalization Programmes COMPETE 2020 and LISBOA 2020 under the PORTUGAL 2020 Partnership Agreement, and through the European Structural and Investment Funds in the FEDER component.

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

Abstract

Implementation of microservices and network management for Cyber-Physical Systems

The Internet of Things (IoT) ecosystem is made up of a large number of devices and sensors that capture and collect massive amounts of data before sending it to the cloud for analysis. Traditionally, server-side software development has taken a monolithic approach, in which the application is a single executable. However, the microservices architecture provides an alternative that can be applied to the IoT environment.

This dissertation aims to continue the development of the microservices-based cloud architecture developed as part of the Link4S project, which seeks to create a new generation of connected devices and their platforms. The development involved the integration of the cloud with a Connectivity Management Platform (CMP) and the creation of a fully integrated platform for device management and data analysis.

In the context of this dissertation, new microservices were created to integrate with CMP architecture components and provide application support, as well as a data visualization and device management platform in the form of a Dashboard built with the Dash framework. Furthermore, the development of REST APIs for retrieving and manipulating device data opens the door for other applications to be developed for a variety of purposes. Before being successfully deployed to a production server, the architecture was also tested and validated.

This dissertation concludes with a cloud architecture that is more autonomous, secure, and reconfigurable during runtime via CLI commands. It also highlights the importance of microservices architecture in the IoT ecosystem and shows how the CMP architecture can serve as the foundation for future IoT cloud architectures.

Keywords: cloud, CMP, Dash, dashboard, IoT, microservices, REST API

Resumo

Implementation of microservices and network management for Cyber-Physical Systems

O ecossistema da Internet das Coisas (IoT em inglês) consiste num vasto número de dispositivos e sensores que captam e recolhem enormes quantidades de dados, que são depois enviados para a *cloud* para análise. Tradicionalmente, o desenvolvimento de software do lado do servidor tem adotado uma abordagem monolítica, na qual a aplicação é um único executável. No entanto, a arquitetura dos microserviços fornece uma alternativa que pode ser aplicada ao ecossistema IoT.

Esta dissertação visa continuar o desenvolvimento da arquitetura de *cloud* baseada em microsserviços criada no âmbito do projeto Link4S, que procura criar uma nova geração de dispositivos ligados e as suas plataformas. O desenvolvimento envolveu a integração da *cloud* com uma Plataforma de Gestão da Conectividade (CMP em inglês) e a criação de uma plataforma totalmente integrada para a gestão de dispositivos e análise de dados.

No contexto desta dissertação, foram criados novos microsserviços para se integrarem os componentes da arquitetura CMP e fornecerem suporte às aplicações, bem como uma plataforma de visualização de dados e gestão de dispositivos sob a forma de um dashboard utilizando a *framework* Dash. Além disso, o desenvolvimento de *REST API* para a aquisição e manipulação de dados de dispositivos abre a porta para outras aplicações serem desenvolvidas para uma variedade de fins. Antes de ser implementada com sucesso num servidor de produção, a arquitetura foi também testada e validada.

Esta dissertação conclui com uma arquitetura da *cloud* que é mais autónoma, segura e reconfigurável através de uma interface de linha de comandos em tempo real. Também destaca a importância da arquitetura de microsserviços no ecossistema IoT e demonstra como a arquitetura CMP pode servir de referência para futuras arquiteturas IoT na *cloud*.

Palavras-chave: cloud, CMP, Dash, dashboard, IoT, microsserviços, REST API

Contents

List of Figures x

List of Tables xiii

Listings xiv

Acronyms xv

1 Introduction 1

1.1 Contextualization and Motivation 1

1.2 Goals 3

1.3 Methodology and Methods 3

1.4 Dissertation Structure 5

2 Background and State of the Art 6

2.1 Microservices 6

2.1.1 Microservices architecture 6

2.1.2 Microservices deployment 20

2.2 Dashboard 25

2.2.1 Good design practices 25

2.2.2 Dashboard types 27

2.2.3 Frameworks for dashboard development 30

2.3 Summary 33

3 System Specification & Design 37

3.1 Functional and non-functional requirements 37

3.1.1 Functional requirements 37
3.1.2 Non-functional requirements 38
3.2 Architecture overview 39
3.3 Use cases 42
3.4 Microservices 42
3.4.1 Data parsing 44
3.4.2 Data processing 46
3.4.3 Devices 49
3.4.4 Users 51
3.5 Database 53
3.5.1 Data 53
3.5.2 Application 54
3.6 Dashboard 57
3.6.1 Functional and visual features 57
3.6.2 Framework choice 58
3.6.3 Architecture overview 59
3.6.4 Design prototype 60
3.6.5 Real-time updates 66
3.6.6 Time series cache 67

4 Implementation **71**
4.1 Programming languages and tools 71
4.2 Microservices 72
4.2.1 Data parsing 72
4.2.2 Data processing 74
4.2.3 Devices 75
4.2.4 Users 78
4.3 Dashboard 79
4.3.1 Initialization, routing and layout 79
4.3.2 Pages 81
4.3.3 Real-time updates 93
4.3.4 Time series cache 94

5 Results **96**

5.1 Login 96

5.2 Overview 97

5.3 Metrics 98

5.4 Specifications 100

5.5 Sensors 102

6 Conclusion **105**

6.1 Future work 107

Bibliography **108**

List of Figures

1 System architecture overview 3

2 Monolith vs. Microservices CI/CD pipeline 8

3 The different types of coupling, from loose to tight 10

4 Collaboration styles and common implementation choices for synchronous and asynchronous inter-microservice communication 15

5 Remote Procedure Call flow 16

6 High-level overview of gRPC 17

7 REST API model 18

8 Overview of a Kafka cluster with three brokers 20

9 Comparison between the stack of virtual machines (type 1 hypervisor) and containers . . . 21

10 High-level overview of Docker architecture 23

11 Overview of a Kubernetes cluster 24

12 Taxonomy of dashboard characteristics 28

13 Ubidots dashboard example 33

14 Node-RED dashboard example 33

15 Grafana dashboard example 33

16 Thingsboard dashboard example 33

17 Dash dashboard example 33

18 Architecture overview 40

19 Blackwing message format 41

20 Device use cases 42

21 Device Ecosystem use cases 43

22 Applications use cases 43

23 Generic parsing microservice flowchart 44
24 Cloud payload datagram with example 45
25 Process command flowchart 47
26 Command listener flowchart 48
27 Data listener flowchart 51
28 Data segment database collections 55
29 Application segment database collections 56
30 Dash framework components 58
31 Dashboard MVC diagram 60
32 Dashboard user flow 61
33 Dashboard color palette 61
34 Login page design prototype 62
35 Overview page design prototype 63
36 Metrics page design prototype 64
37 Sensors page design prototype 65
38 Specifications page design prototype 66
39 MongoDB replica set diagram 67
40 Change Streams flowchart 68
41 Time series cache buckets diagram 69
42 Time series cache flowchart 70

43 Login page implementation 82
44 Overview page implementation 83
45 Overview page general status hover 84
46 Overview page device location pin 85
47 Overview page notification 85
48 Metrics page implementation 87
49 Metrics page settings menu 88
50 Metrics page configuration 88
51 Sensors page implementation 90
52 Sensors page configuration 90
53 Specifications page implementation 92

54 Dashboard login page test 97
55 Users microservices authentication request logs 97
56 Dashboard overview page 97
57 Dashboard emergency notification details 98
58 Dashboard device location section 98
59 Dashboard fetch sensor data logs 99
60 Devices microservice metrics request logs 99
61 Dashboard metrics page 99
62 Dashboard metrics exported Excel spreadsheet 100
63 Dashboard specifications set alarm 100
64 Process Data microservice alarm command logs 101
65 Database alarm commands 101
66 Dashboard notifications section 101
67 Dashboard command notification 102
68 Dashboard configuration notification 102
69 Dashboard change streams logs 102
70 Dashboard sensors page parameters update 103
71 Process Data microservice sensor variable command logs 103
72 Dashboard change streams logs 103
73 Dashboard change streams logs 104

List of Tables

1 Comparison between Monolith architecture and Microservices architecture 8
2 Comparison between Microservices and IoT/CPS features 9
3 Comparison between containers and virtual machines 21
4 Guidelines for presentation formats and dashboards 26
5 Design choices of the different types of dashboards 29
6 Analysis of dashboard examples according to design choices 32
7 Comparison of common dashboard platforms available on the market 35
8 Analysis of dashboard examples according to guidelines 36

9 Process Data microservice API specification 46
10 Devices microservice API specification (GET methods) 49
11 Devices microservice API specification (POST methods) 50
12 Users microservice API specification 52

Listings

1 Process data microservice API creation . . . 72
2 Blackwing parsing microservice handler . . . 73
3 Parsing microservice API post method . . . 73
4 Process data microservice API methods . . . 74
5 Process data microservice command listener . . . 75
6 Devices microservice API endpoints . . . 76
7 Devices microservice listeners . . . 77
8 Users microservice authentication endpoint . . . 78
9 Dashboard initialization . . . 79
10 Dashboard layout and routing . . . 80
11 Dashboard sidebar and page content . . . 80
12 Dashboard User class and login callback . . . 81
13 Dashboard device data callback . . . 83
14 Dashboard device location callback . . . 84
15 Dashboard notifications callback . . . 86
16 Dashboard metrics plot generation . . . 89
17 Dashboard sensor variables callback . . . 91
18 Dashboard specifications apply callback . . . 92
19 Dashboard change streams handler . . . 93
20 Dashboard change streams main . . . 94
21 Dashboard WeeklyBucket class . . . 94
22 Dashboard SamplesFetcher class . . . 95

Acronyms

AKS Azure Kubernetes Service

API Application Programming Interface

AWS Amazon Web Services

CD Continuous Delivery

CI Continuous Integration

CLI Command Line Interface

CMP Connectivity Management Platform

CoAP Constrained Application Protocol

CPS Cyber-Physical Systems

CPU Central Processing Unit

CRUD Create, Read, Update and Delete

DDD Domain-Driven Design

DWPS Devices Profile for Web Services

EKS Elastic Kubernetes Service

ESB Enterprise Service Bus

FaaS Function as a Service

FOTA Firmware Over The Air

GKE Google Kubernetes Engine

gRPC Google Remote Procedure Call

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

IoT Internet of Things

IoT-mP Internet of m-Health Things Platform

JSON JavaScript Object Notation

K8s Kubernetes

Link4S (Link4S)ustainability

LTE Long Term Evolution

LXC Linux Containers

MQTT Message Queuing Telemetry Transport

MSA Microservices Architecture

MVC Model View Controller

NB-IoT Narrowband Internet of Things

OS Operating System

PHP Hypertext Preprocessor

REST Representational State Transfer

RPC Remote Procedure Call

RSRP Reference Signal Received Power

RSRQ Reference Signal Received Quality

RSSI Received Signal Strength Indicator

RTOS Real Time Operating System

SOA Service Oriented Architecture

SQL Structured Query Language

TAC Tracking Area Code

TAU Tracking Area Update

TTL Time To Live

UFS Union File System

URI Uniform Resource Identifier

VM Virtual Machine

WSGI Web Server Gateway Interface

Chapter 1

Introduction

This chapter presents the contextualization, motivation, and objectives of the dissertation. First, the contextualization provides an overview of the body of the dissertation. The motive for doing a study on this specific topic is presented next, followed by the study's objectives, research methods, and document structure.

1.1 Contextualization and Motivation

The IoT environment consists of a vast number of devices and sensors that enable the capture and collection of massive volumes of data, which are then sent to the cloud for analysis. Traditionally, server-side software is developed using a monolithic approach, in which the application consists of a single executable. However, there is now an alternative based on microservices approach. When applied in this context, the microservices approach enables "the development of modular and extensible architectures, resulting in significant gains in terms of performance, dynamism, and resilience" [1] as the monolith is broken down into smaller services. Multiple studies have demonstrated that microservices and IoT services are highly compatible [2]–[4], thanks to the massive investments made by companies, the industrial state of practice on microservices is rapidly evolving and has already reached a certain level of maturity [5].

Although it is beyond the scope of this dissertation, a common method for deploying microservices in the cloud is to use lightweight containers, such as Docker containers, which allow microservices to be packaged as isolated and independent units that can then be inserted into a container management platform, such as Kubernetes [6].

The end-devices that communicate with the cloud are monitoring systems with multiple sensors. Each device is responsible for reading sensor values and sending them to the cloud at predetermined intervals. They also communicate with the cloud when values exceed a predetermined threshold, indicating an emergency. The majority of the time, these extremely low-power devices are in a sleep state in an effort to achieve a lengthy battery life.

This dissertation' content was developed in the context of a project in CPS domain, called (LINK4S)ustainability of the Link4S consortium which aims to generate new scientific knowledge regarding the design, development, and testing of a new generation of connectivity devices and their associated platforms (communication and software) with the goal of integrating networks of objects in the context of mobility and energy [7].

In the Masters's thesis titled "Robust Software Services for IoT Embedded Systems" [8] developed in the context of the Link4S project, the purpose was to improve the software stack of the bare-metal embedded devices and enable them for secure communication with the cloud. This meant adding features such as Firmware Over The Air (FOTA) and the ability to change the parameters on the devices through commands received from the cloud. Using a microservices architecture, the cloud implementation is responsible for controlling and receiving data from the devices.

The top-level architecture overview is shown in Figure 1. It is composed of three layers: the device, the network infrastructure and the cloud platform. The device is an ultra-low-power monitoring system with a typical IoT architecture. It spends the majority of its life in a dormant state, awakening only to measure environmental variables using multiple sensors or to signal an anomaly. It uses an Narrow Band Internet of Things (NB-IoT) transceiver to communicate with the cloud, and a microcontroller as the system brain. The network is provided by the project's main sponsor, NOS Comunicações [9], and utilizes NB-IoT technology by implementing a subset of the Long Term Evolution (LTE) standard in its mobile antennas, allowing devices to be deployed in areas with antenna coverage. The cloud is composed of the server and microservices; it receives data from devices, parses the data, and stores it in a database.

One of the main objectives of the Link4S project is to integrate all devices and applications in a Connectivity Management Platform (CMP), that was also used to build an Internet of m-Health Things Platform (IoT-mP) in [10]. This is a generic platform that is responsible for the unification of the IoT services between IoT hardware (e.g., NB-IoT connected devices, such as patient monitoring, hospital records, etc.) and IoT applications (e.g., hospital billings, emerging messages, emerging transportation, health software, etc.).

Additionally, a new IoT application in the form of a dashboard will be developed to provide a way to

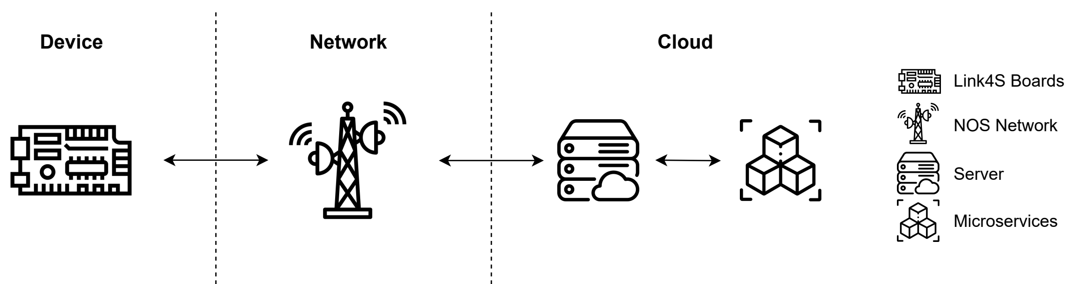


Figure 1: System architecture overview

view and manage the devices and data they generate. This new application will enable users to perform functionalities that previously required the use of complex Python scripts, such as sending commands to devices to change parameters and easily extracting structured sensor data with a single click. The dashboard will facilitate the end-users' access to and utilization of the data generated by the devices, making it more accessible and efficient.

1.2 Goals

The goals of this dissertation are to empower the Link4S prototype with additional functionalities such as control of IoT devices previously done on the cloud platform, and to develop microservices that will allow the cloud to conform to the CMP [10] architecture. This architecture effectively decouples the applications from the ecosystems that produce the data, by splitting the architecture into segments whose only cconnection is the core of CMP, known as the Mediator. This enables applications to consume data from multiple sources by simply modifying the Mediator's configuration, without the need for additional development.

Additionally, the goal is to design and implement microservices to support additional cloud features, such as the Dashboard application for data visualization and device management. The final goal is to test and validate the final cloud architecture, including validation with project partners, and to deploy the cloud architecture on a server that is production-ready.

1.3 Methodology and Methods

To achieve the goals above, the following methodology and methods will be used:

- Literature review - The first step will be to conduct a literature review on microservices architecture and the current state of dashboard development. This will involve studying relevant research articles

and other sources to gain an understanding of the state of the art in these areas. The review will focus on topics such as the benefits and challenges of microservices architecture, best practices for designing and implementing microservices, and the latest trends and innovations in dashboard development.

- Review of the current cloud architecture and CMP architecture - The current cloud architecture and the CMP architecture will be reviewed in order to identify potential improvements and to gain a better understanding of the implemented features. This will involve examining the existing cloud architecture and identifying any areas that could be optimized or enhanced as part of the restructuring process. The review will also involve understanding the features and requirements of the CMP architecture to ensure that the restructured cloud architecture is compatible with it.
- Design and implementation of microservices - Based on the findings of the literature review, and the reviews above, the following microservices will be implemented:
  - Microservices to conform with the CMP architecture - This will require implementing the components of the CMP architecture in advance using the available Command Line Interface (CLI). The Mediator component implementation will not be included because it falls outside the scope of this dissertation. The microservices will utilize the available interfaces to send and receive data between the various components. It will be necessary to understand the communication protocols and data formats used by the CMP architecture and to implement the necessary code to facilitate communication between the components.
  - Microservices to support additional cloud architecture features - These microservices will provide APIs that the dashboard can use to interact with device data, user data, and to send commands to control devices. It is possible that some of these microservices will also be responsible for enabling the restructuring of the cloud according to the CMP architecture.
- Development of the dashboard application - Once the microservices have been implemented, the next step will be to develop the dashboard application. This will involve designing and building an interface that allows users to interact with the devices and view data from the sensors. The application will also include features for analyzing and interpreting the sensor data, such as graphical displays.

- Testing and validation - Before the updated cloud architecture is deployed, it will be thoroughly tested to ensure that it is stable and reliable. This will involve conducting a series of tests to ensure that all the components are working correctly and that the system as a whole meets the requirements.
- Deployment - Once the testing and validation is complete, the final cloud architecture will be deployed on a server that is production-ready. This will involve installing the necessary software and configuring the server to support the cloud and devices.

1.4 Dissertation Structure

This dissertation will consist of six chapters. The first chapter is dedicated to the introduction of the project, its goals and objectives, and the dissertation's position within the project.

After the introductory chapter, a research on the state of the art for some of the topics involved in the implementation is presented, first covering the literature on the microservices architecture, why it is considered a viable alternative to the monolithic architecture, and how to design and implement microservices, and then delving deeper into the best practices for dashboard design and determining which technologies are best suited to achieve its objectives, laying the groundwork for the dashboard.

The third chapter contains the system specification and design, which begins with a review of the cloud architecture as a whole, its systems, and their interactions. Then, each system in the cloud architecture will be discussed in greater depth. In this chapter, there are three main topics: the Microservices that process the device data and provide support to other cloud-based systems, the Database that stores all device and end-user data, and the Dashboard that serves as the user interface.

The fourth chapter describes in depth how all the cloud-based systems designed in the previous chapter are implemented and deployed on a production server.

The tests conducted to ensure that all systems are operating as intended and that all objectives have been met are detailed in the next chapter.

The final chapter concludes the dissertation, summarizes the developed work, and provides recommendations for potential future improvements.

Chapter 2

Background and State of the Art

This chapter addresses the literature review on the two main topics of this dissertation: microservices and dashboard design. The section on microservices covers the main concepts and provides a comprehensive understanding of microservices architecture, from its origins to the most recent implementations based on containerization technologies. The section on dashboard design discusses best practices and key considerations for designing effective dashboard platforms.

Multiple publications were collected and analyzed to determine the most recent advancements in the above-mentioned research fields. This research provides the foundation for the cloud's architecture development.

2.1 Microservices

2.1.1 Microservices architecture

The development of server-side applications consists on the design and creation of single executable artifacts, also called *monoliths*, whose modularisation abstractions rely on the sharing of resources of the same machine (memory, databases, files). The monoliths, usually developed in languages like C/C++, Java, and Python are not independently executable, since their modules depend on shared resources [11].

The inability to break down the complexity of monoliths into independent modules presented several problems, as presented in [11]. Some of the most crucial being:

1. difficult to maintain and evolve due to complexity (large code base);
2. dependency issues due to libraries incompatibilities, resulting in systems that don't compile/run or misbehave;
3. updates, even small ones, significantly increase downtime due to rebooting the whole application;
4. limited scalability. The allocation of new resources has to go towards the whole application and not the modules whose traffic justifies more resources since modules are not independent;
5. technology lock-in for developers. The application requires the use of the same languages and frameworks over time.

Some of the issues presented above such as maintainability and the update of systems have been mitigated using different Service Oriented Architecture (SOA) approaches, where an application is divided into a set of business applications offering services to others through different protocols. SOA implementations appeared to be the solution for many companies, but the scaling of SOA became an issue due to the usage of communication systems like Enterprise Service Bus (ESB) [12] that weren't designed for the cloud [13].

The microservices architectural style (MSA) emerged in 2014 as a way to avoid the problems of monolithic applications while taking advantage of the best aspects of the SOA architecture. It was initially defined as an approach to develop an application as a set of small services, each with a single and clearly defined purpose, and running in its process and communicating over lightweight mechanisms, such as an Hypertext Transfer Protocol (HTTP) based API or Remote Procedure Call (RPC) [14].

The decomposition of applications into smaller independent services allows each service to be developed, deployed and operated by different development teams using different technology stacks [15]. This gives the ability for teams to set their own rhythm of development, meaning that features from a certain team can be released and go into production whenever they are ready, instead of having to wait for the others to be ready for release, as illustrated in Figure 2.

The comparison of microservices vs. monolithic architecture has been condensed into the categories shown in Table 1. Although MSA is the most convenient option in most of the categories, there are still drawbacks in the Communication and Integration categories when compared to the monolithic approach. The usage of mechanisms such as HTTP introduces overhead that can present a problem towards IoT devices, which are in many cases resource and time-constrained [16].

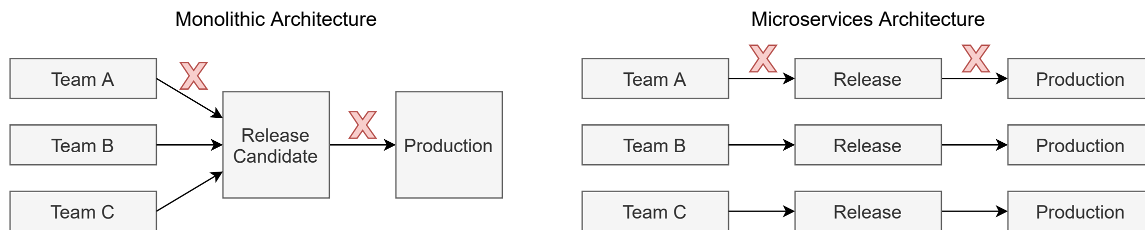


Figure 2: Monolith vs. Microservices CI/CD pipeline

Table 1: Comparison between Monolith architecture and Microservices architecture [17]

Category	Monolith architecture	Microservices architecture
Development	hard to develop due to large code base, while restricted to the languages and frameworks already in use	each microservice implements a simple task; different microservices can use different languages or frameworks
Scalability	scaling requires replicating the whole system	the smaller sized instances of microservices are easily scalable
Maintenance	hard to maintain and upgrade due to single source code	easy to maintain and upgrade due to small code size
Independence	modules share resources, and thus are not independent	modules should be completely independent
Deployment	all at once	services can be deployed independently
Size	large code base	services are fine-grained, i.e. small
Communication	in-memory	lightweight mechanisms (HTTP, RPC)
Integration	challenging due to a large number of independent modules	straightforward, since modules are placed together

2.1.1.1 Microservices in the Internet of Things

The IoT architecture is comparable to MSA in that both are modular and capable of constructing applications that unify a large number of services. Due to the restricted nature of IoT devices, the service oriented architecture that inspired microservices architecture is already actively employed for many CPS implementations, with its solutions focusing on RPC and Representational State Transfer (REST) lighter versions.

The main obstacle for MSA adoption in the IoT space is the fact that they come from different directions and have conceptual differences. Microservices arose from the necessity of breaking large monoliths into smaller, modular components, whereas IoT applications are already small to run in resource-constrained

embedded devices.

Moreover, as analysed in [18], where both approaches are compared in regards to several features, these being: self-containment, monitoring and fault handling, choreography and orchestration, container technologies and the handling of different service versions, with the conclusions in Table 2.

Table 2: Comparison between Microservices and IoT/CPS features [18]

Feature	**Description**	**Microservices**	**IoT/CPS**
self-containment	service completely independent and contains all the necessary dependencies to run on its own	services contain business logic, front-end and back-end, as well as the required libraries	services built around device capabilities, libraries might not be packed with application
orchestration vs. choreography	orchestration: the services are controlled by an orchestrator in a centralized fashion; choreography: the services are independent and event driven	choreography preferred	usually uses orchestration unless based on Message Queuing Telemetry Transport (MQTT) protocol
OS-level virtualization	container includes all the necessary dependencies to run the application	yes, mostly using Docker	no
continuous integration	development changes often, followed by builds and tests for validation	yes	partly (single vendor scenarios)
continuous delivery	deploying changes often in a predicable and routine manner	yes	no
protocols	communication between services	HTTP, RPC	HTTP, MQTT, Constrained Application Protocol (CoAP), Devices Profile for Web Services (DPWS)

Research study [18] claims that the microservice approach is presented as a viable approach for IoT. The differences, presented in Table 2, between microservices and IoT/CPS implementations are easily bridged, as they are already mainly based on SOA architectures. The distributed nature and scalability of

microservices enables IoT implementations to achieve a level of interoperability that was not possible so far.

2.1.1.2 Microservices design

The process of building a microservices architecture requires well-defined boundaries for different services, and because there is no standard process for doing so, numerous aspects, such as the domain model, bounded contexts, entities, aggregates, services, and non-functional requirements should all be considered.

When creating the boundaries for the services, two concepts must be properly implemented: loose coupling and high cohesion. Services in a loosely coupled system are as independent of one another as possible, which implies that changes to one service have little effect on other services. Then, to achieve high cohesion, related logic should be contained in a single service, reducing communication between services and overall system latency.

Cohesion and coupling are closely related concepts because cohesion describes the relationship between things inside a boundary (microservice), whereas coupling describes the relationship between things across a boundary. Given the context and problem at hand, the goal is to strike the right balance between the two ideas [19].

Not all coupling is bad, and having some coupling is ultimately unavoidable, so the goal is to reduce it as much as possible. In the context of a modular architecture like microservices-based systems, the different types of coupling can be classified from loose coupling to tight coupling as illustrated in Figure 3.

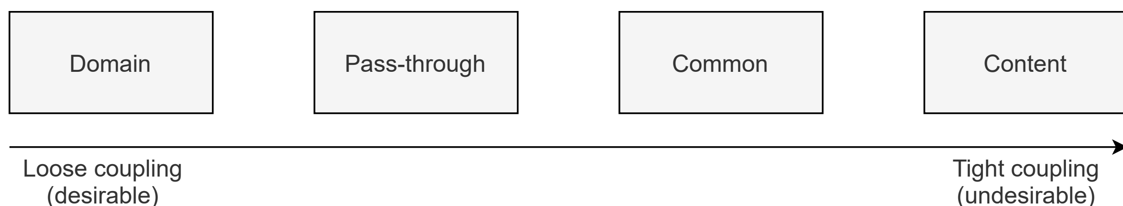


Figure 3: The different types of coupling, from loose to tight [19]

The different types of coupling are defined in [19] as follows:

- **Domain coupling** - when a microservice needs to communicate with another microservice in order to use a feature provided by the other microservice. Because a microservice-based system requires the collaboration of multiple microservices to perform its function, this type of interaction

is unavoidable. However, a service should not be overly reliant on other services, as this may imply that a microservice is doing too much.

- **Pass-through coupling** - when one microservice needs an intermediary microservice to pass data further downstream to another microservice. To perform this coupling, the caller must understand that the microservice it is invoking is simply an intermediary, as well as how it operates. This creates a problem whenever the required data is changed downstream, because it also implies a change upstream.

- **Common coupling** - when two or more microservices access a common set of data, such as a shared database. Any changes to the data structure necessitate changes to the microservices that rely on the data. This problem can worsen if multiple microservices are both reading and writing to the same data, possibly leading to an overload of the shared resource, making it slow or even unavailable in many cases.

- **Content coupling** - when an upstream service reaches into a downstream service and changes its internal state, such as an external service directly accessing and changing the database of another service. Doing this bypasses the logic in the service which *owns* the database, making the lines of ownership less clear and increasing the difficulty for developers to change the system.

According to the findings in [20], the primary mechanism for determining microservices boundaries is through a combination of domain-driven design (DDD) and business capabilities. It is also stated that 70.2% of practitioners use DDD alone or in conjunction with business capabilities.

Domain-driven design

Domain-driven design is a methodology that can assist in the creation of properly designed microservices in two distinct phases: strategic and tactical. Strategic DDD focuses on the big picture of the system while keeping the business capabilities in mind. Tactical DDD provides a set of design patterns used to develop the domain model, which will aid in the creation of loosely coupled and cohesive microservices [21].

To properly implement DDD for MSA, the following approach should be taken:

Analyse domain:

A domain refers to the practical aspects of a solution (e.g. Healthcare, Aviation, Finance, Retail, etc). The domain informs the requirements and acceptance criteria for the system.

All business functions and their connections should be mapped in order to model the business domain. Domain experts, software architects, and other stakeholders are all involved in this collaborative effort. This can be done in a simple, non-formal way, such as a diagram sketch or a whiteboard drawing [21].

As the domain model is being created, subdomains can also be identified. Subdomains typically reflect some organizational structure in which some users use a specific ubiquitous language; for example, the automobile (domain) can be broken down into logistics, R&D, traders, production, and marketing (subdomains). Subdomains can communicate with one another, and each subdomain can have its own subdomains [22].

Define bounded contexts:

A bounded context is the region of a domain where a particular domain model applies. Because the entire business model is too large and complex to comprehend as a whole, and it is impractical to maintain a unified model, boundaries and relationships between different models must be established. Having bounded contexts aids in the formalization of interactions between different services and the construction of interfaces between them [22].

It is essential to understand the difference between subdomains and bounded contexts. A subdomain belongs in the problem space, which is how the business perceives the problem, whereas bounded contexts belong in the solution space, which is how the solution will be implemented. Each subdomain may have multiple bounded contexts, but the goal should be one per subdomain [23].

Define entities, aggregates, and services:

Tactical DDD patterns are used in this section to define domain models with greater precision, and they are applied within a single bounded context. The entity and aggregate patterns are the most relevant; as a general rule, a microservice should be no smaller than an aggregate and no larger than a bounded context [24]. According to [24], [25], the tactical patterns are:

- **Entities** - entities are an object with a unique identity that persists over time. Customers and accounts, for example, are entities in a banking application. The unique identity may span many bounded contexts and may persist beyond the lifetime of the application. It can hold references to other entities, and its attributes can change over time.

- **Value objects** - value objects are objects with no identity and are defined solely by the value of their attributes. These objects cannot be modified after they are created, and in order to update

them, a new instance must be created to replace the old one. Colors, dates and times, and currency values are examples of value objects.

- **Aggregates** - aggregates describes a cluster or group of entities and behaviors that can be treated as a cohesive unit with regard to data changes. In an aggregate, one single entity is accessible from the outside, the root, and any other entities in the same aggregate are children of the root, and can be referenced by following pointers from the root. The main purpose of the aggregate root is to ensure consistency in the aggregate. If the root is removed from the aggregate, so are the other objects in the aggregate. Aggregates are usually identified from the transaction operations within a domain.
- **Domain and application services** - services are objects that implement some logic and don't maintain any state. They are frequently used to model behavior that spans multiple entities and are classified as either application services, which provide technical functionality, or domain services, which encapsulate domain logic.
- **Domain events** - domain events are used to notify other parts within the same domain when something happens. "A purchase was made"is an example of a domain event. Domain events are extremely useful in MSA since they provide a way for microservices to coordinate with each other since microservices are distributed and don't share data stores.

Identify microservices:

When deriving microservices from a domain model, bounded contexts, entities, aggregates, services, and non-functional requirements should all be considered. The approach outlined in [26] is a viable option for this operation.

1. Begin with a bounded context because a microservices' functionality should not span more than one bounded context.
2. Consider the aggregates from the domain model, which are frequently good candidates for microservices. A well-designed aggregate should be loosely coupled and have high cohesion, both of which are crucial concepts for well-designed microservices.
3. Domain services are another candidate for microservices. A workflow involving multiple microservices is an example of domain service. They are stateless operations that can be performed across multiple aggregates.

4. Non-functional requirements such as technologies, security, reliability, performance, maintainability, scalability, and usability should be taken into account. Taking these into consideration may result in the decomposition of a microservice into several smaller services, and vice-versa.

Finally, the microservices should be validated using the criteria listed below:

- Each service has a unique responsibility;
- Services should not communicate excessively with each other to perform their functions;
- Services are small enough to be developed by an independent small team;
- The deployment of a service should not require the redeployment of other services;
- Services can be updated and evolved independently of one another;
- The service boundaries should not cause issues with data consistency or integrity. To maintain data consistency functionality can be combined into a single microservice, but in some cases, the benefits of decomposing services can outweigh the challenges of managing eventual consistency as strong consistency is not always needed.

As [26] states, domain-driven design is an iterative process, and when in doubt, it can be beneficial to start with fewer and larger microservices and then try to split those into smaller services, as this is a simpler process than attempting to refactor functionality from poorly designed smaller services.

2.1.1.3 Communication between services

Although methods such as DDD assist the developer in deriving and identifying the boundaries of microservices, they do not assist the developer in determining the best method for microservice communication. As an application is divided into microservices, one of the most important aspects is determining how these services will communicate with one another, as communication between services is no longer as straightforward as making a function call and is primarily accomplished via the network.

In Figure 4 the different collaboration styles of inter-microservice communication are presented, along with the common implementation choices according to [19].

Within this model, the collaboration style and its respective implementation choices can be synchronous or asynchronous:

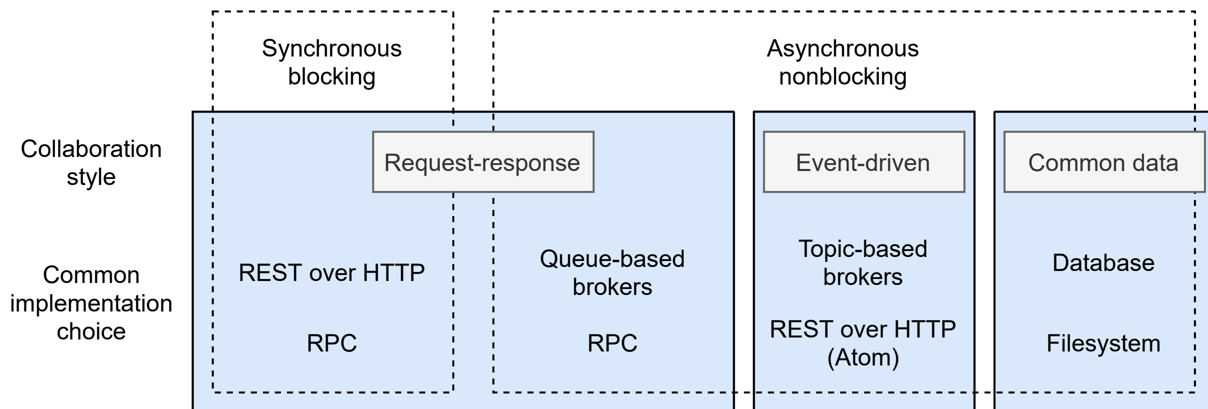


Figure 4: Collaboration styles and common implementation choices for synchronous and asynchronous inter-microservice communication [19]

- **Synchronous blocking** - a microservice makes a call to another microservice and blocks operation while waiting for a response.

- **Asynchronous nonblocking** - the microservice is able to continue processing after emitting a call, even if the call is not received.

There are several collaboration styles, each providing different implementation choices:

- **Request-response (Synchronous or Asynchronous)** - a microservice sends a request to another microservice to perform some action. The microservice sending the request expects a response with the result. Depending on the implementation, may be either synchronous or asynchronous.

- **Event-driven (Asynchronous)** - Microservices emit events that are detected and consumed by other microservices. The microservice emitting the event is unaware of any microservices who consume it. Typically used in microservice choreography, which follows a decentralized approach.

- **Common data (Asynchronous)** - microservices can collaborate by using a common data source.

Several factors, such as communication reliability, acceptable latency, and communication volume, are considered when determining the optimal inter-microservice communication strategy. The first decision that must be made is whether the communication should be synchronous or asynchronous, as this will dictate the collaboration styles and implementation choices that will be utilized. Choosing asynchronous communication, for instance, can enable more collaboration styles and implementation options. However,

asynchronous implementations are typically more complex than synchronous implementations, which may not be desired in many application use cases [19].

In any case, it is important to notice that in a microservices architecture a mix of collaboration styles may be implemented, as some interactions between microservices make more sense in a specific style than the other, and even one microservice may implement more than one style of collaboration and implementation choices [19].

Technology choices

There are many choices when it comes to communication technology in microservices, but the most commonly used are RPC, REST and message brokers.

Remote Procedure Call:

RPC is a protocol based on the client-server model designed specifically for the support of network applications, shown in Figure 5. It enables users to work with remote procedures as if the procedures were local. The remote procedure calls are defined through routines in the RPC protocol and each call message is matched with a reply message [27].

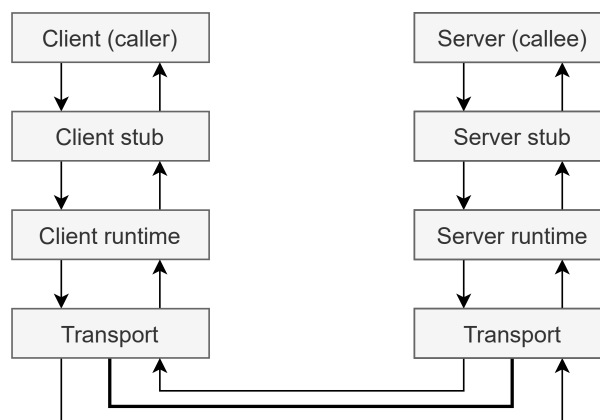


Figure 5: Remote Procedure Call flow

Each server provides a set of remote service procedures comprised of a host address, program number, and procedure number. The RPC model requires the client to make a procedure call in order to send a data packet to the server. When a packet arrives at the server, it invokes a dispatch routine, performs the requested service, and returns a response to the client. The procedure call then returns to the client [27].

gRPC, also known as Google Remote Procedure Call, is an open source RCP architecture variant. One of the primary advantages of gRPC is its performance, as it uses HTTP 2.0 as its underlying transport protocol, a version that dramatically increases network efficiency and enables real-time communication. On the server side, gRPC implements a service by specifying the methods that can be called remotely, as

well as their parameters and return types, and it runs a gRPC server to handle client calls. On the client side, the client is comprised of a stub providing the same methods as the server [28], as can be seen in Figure 6.

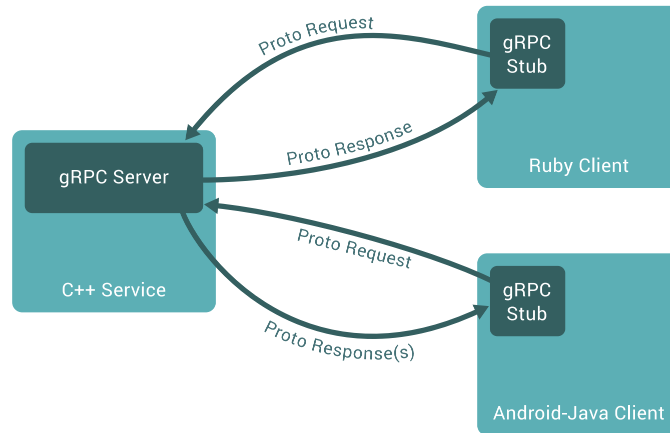


Figure 6: High-level overview of gRPC [28]

To serialize data between services, gRPC uses protocol buffers by default, an open source mechanism developed by Google with a focus on performance (faster than JSON), small size, and simplicity. It also supports the most popular programming languages, including Python, Java, C++, C#, and others. Given this, gRPC may be a good choice for resource-constrained systems, such as many IoT environments, that require lightweight messaging mechanisms.

REST:

REST is a set of architectural constraints that are commonly used to build RESTful APIs that interact with RESTful web services. When a client makes a request via a RESTful API, it transfers a representation of the resource's state. This data is delivered via HTTP in one of several formats, including JSON, HTML, PHP, or plain text, with JSON being the most popular [29].

For an API to be considered RESTful, it must comply with the following criteria [29]:

- client-server architecture comprised of clients, servers, and resources, with requests managed through HTTP.
- statelessness, as each request needs to include all information necessary for processing it, which means that no client information is stored between get requests and that each request is distinct.
- a uniform interface between components so that information is transferred in a standard form. The same piece of data, such as the address of a user, belongs to only one uniform resource identifier (URI).

- cache data when possible on the client and server side to improve performance on the client side, and increase scalability on the server side.

- a layered system architecture that organizes each type of server involved in the retrieval of requested information into many layers. Neither the client nor the server should be able to tell whether they communicate with the end application or an intermediary.

- code-on-demand (optional): REST APIs usually only send static resources, but when requested, they should provide the ability to send executable code from the server to the client.

REST APIs employ CRUD operations, which stand for Create, Read, Update, and Delete. When a client sends a request to a server, he performs one of these operations, represented by the standard HTTP methods of GET, POST, PUT, UPDATE, and DELETE, as shown in Figure 7. REST API calls also contain request headers and parameters, which include identifier information such as metadata, authorizations, uniform resource identifiers (URIs), caching, cookies, and more.

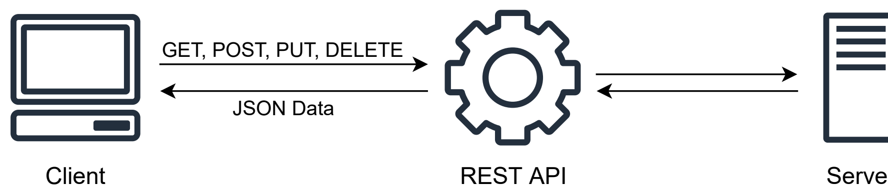


Figure 7: REST API model

REST APIs' ease of use and scalability make them suitable for microservices, particularly because REST services can communicate without requiring internal knowledge of one another; however, tight coupling problems may arise over time as applications can become tightly coupled to the REST API services they use.

Message brokers:

A Message broker allows applications, systems and services to communicate with one another by translating messages between formal messaging protocols. This enables services to communicate even when written in different languages or implemented on different platforms, making data exchange simple and reliable [30].

The usage of message brokers enables the decoupling of processes and services within systems by allowing senders to send messages without knowing the location of receivers, whether they are active, or how many there are [30]. To accomplish this, they typically rely on a component known as a message

queue, widely used in asynchronous systems, that stores and orders messages until the consuming applications can process them. This frees the client from waiting for a task to finish, and it also allows the server to process jobs in the order it prefers.

Message brokers typically offer two messaging distribution patterns: point-to-point messaging, used in message queues where there is a one-to-one relationship between the senders and receivers, and publish/subscribe messaging, used in message queues with a one-to-many relationship between senders and receivers. In point-to-point messaging, a sender produces a message to a queue that is consumed by only one receiver. In publish/subscribe messaging, the sender publishes to a topic, and the receiver subscribes to one or more topics from which it wishes to receive data. When a sender publishes to a topic, the message gets delivered to every consumer who has subscribed to it [30].

Apache Kafka is a popular example of a platform which provides messages queues. It was originally conceived as a messaging queue, but now provides many more features as a distributed event streaming platform. It can be used as a publish/subscribe messaging system, a storage system for streams of events and a real-time data processing system [31].

A Kafka system is composed of clients and servers communicating via the TCP network protocol. Kafka can run as a cluster of one or more servers, with some forming the storage layer, also known as brokers, and the other servers running the tool Kafka Connect to continuously import and export data as event streams, integrating Kafka with existing systems such as databases and other Kafka clusters, illustrated in Figure 8. Kafka clusters also provide fault tolerance because if one server fails, the others will take over and ensure that no data is lost. Kafka clients can be written in a variety of languages, including Java, C/C++, Python, and others. The clients enable the development of distributed applications and microservices that read, write, and process streams of events in parallel, at scale and in a fault-tolerant manner [31].

Events are used for reading and writing to Kafka, with each event containing a key, value, timestamp, and optional metadata headers. Events are organized and stored in multi-producer and multi-subscriber topics because many producers can write to a topic, and many consumers can subscribe to a topic. Unlike other messaging systems, events are not deleted after consumption, and the user decides how long they should be kept alive [31].

Kafka achieves high scalability because producers and consumers are decoupled from one another, and then, to ensure fault tolerance and high availability, Kafka replicates topics across multiple brokers. These concepts are critical in the implementation of microservices, which is why Kafka's a choice in many of them.

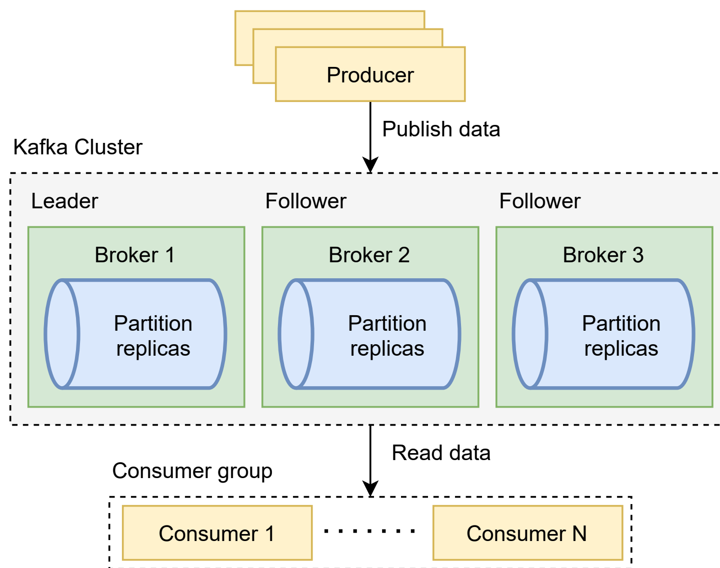


Figure 8: Overview of a Kafka cluster with three brokers

2.1.2 Microservices deployment

After developing microservices, one must find a suitable infrastructure for them to run on. Virtual machines and containers are the two most common solutions for this purpose. Until a few years ago, the majority of server-side software ran directly on virtual machines, but containers have since gained popularity and, more importantly, have proven to be a suitable fit for microservices architecture.

Containers are executable software units that package code and its dependencies into a single object that can run as a resource-isolated process in different computing environments. A container unit contains all of the components required to run an application, such as code, runtime, system tools, system libraries, and settings.

2.1.2.1 Containers vs. virtual machines

Container adoption in microservices has grown exponentially since their introduction, with containers becoming the de facto choice for many for packaging and running microservices [19]. Before the popularity of containers, most implementations were based on virtual machines, which share many benefits with containers in terms of resource isolation and allocation but operate differently.

The main difference between virtual machines and containers, as illustrated in Figure 9, is that virtualization in containers occurs at the operating system level, whereas virtualization in virtual machines occurs at the hardware level (type 1 hypervisor, i.e. bare metal). This means that a single machine can run multiple containers and achieve isolation, whereas virtual machines, due to including the guest OS,

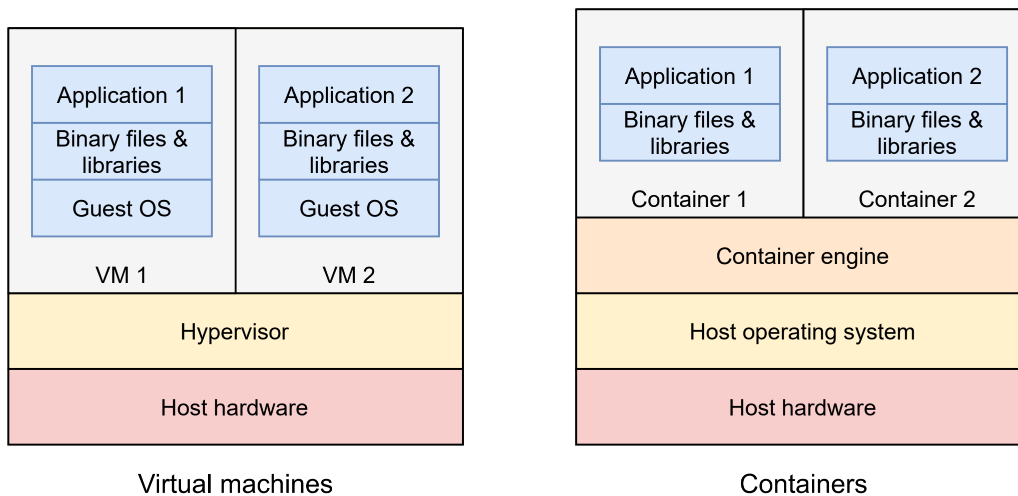


Figure 9: Comparison between the stack of virtual machines (type 1 hypervisor) and containers

are much heavier and thus not as easily scalable. There are also a few other differences, listed in Table 3.

Table 3: Comparison between containers and virtual machines

Description	**Virtual machines**	**Containers**
Where virtualization occurs	at hardware level	at operating system level
Type of isolation achieved	isolation of machines	isolation of processes
How resources are accessed	via hypervisor	via kernel features such as namespace and cgroups
Flexibility and portability	great flexibility of hardware, as many machines can be created with specific resources	great portability because the same container can run on a variety of machines
Security	fully isolated hence more secure	process-level isolation hence less secure
Size	can take up to tens of GBs	typically tens of MBs

It is possible to conclude that containers are more portable and efficient than virtual machines and thus more suitable for microservices. Running a microservice inside a container provides isolation and hides the underlying technology used for the application, enabling developers to use different technologies for different services [19].

However, these two technologies do not have to compete because virtual machines can still be very useful in a microservices environment. Virtual machines composed of type 2 hypervisors, where the hypervisor runs on top of the operating system, can run multiple containers, combining the flexibility of virtual

machines and the portability of containers.

2.1.2.2 Containerization

Containerization is the process of creating containers through the use of virtualization technology. Although Docker made containers much more popular in recent years, the technology has been present in UNIX-style operating systems such as Linux for many years.

With the rise of Linux as the dominant open platform, the technology was eventually incorporated into the standard distribution LXC, which combined the cgroups and namespaces kernel features to provide an isolated environment for applications. Docker's early iterations used LXC under the hood, though LXC was dropped in later versions [32].

Docker

Docker is an open-source containerization platform that enables developers to package applications and their dependencies into lightweight containers and run them in any environment using virtualization technology.

Every docker container is created using base images. A docker image contains the application source code, as well as all the libraries and dependencies that the application needs to run as a container. An image is created based on a Dockerfile script which contains the instructions and arguments listed in succession to automate the creation process [32].

Docker architecture:

The docker architecture illustrated in Figure 10 uses a client-server model divided into several components [34]:

- **Docker host** - represents the physical machine or VM where the Docker daemon and containers are deployed. The Docker daemon is in charge of building and storing images, as well as creating, running, and monitoring containers. Normally, the host OS handles the Docker daemon launch.
- **Docker client** - it is the main interface for users to interact with Docker through commands. It controls the host, creates images, publishes, executes/manages containers, and can communicate with multiple Docker daemons. The communication with the Docker daemon occurs via sockets through a RESTful API. The Docker client and Docker daemon compose the Docker engine.
- **Docker registry** - it stores and distributes Docker images. The Docker Hub is used by default, and it can host both private and public repositories, but a private registry server can also be configured.

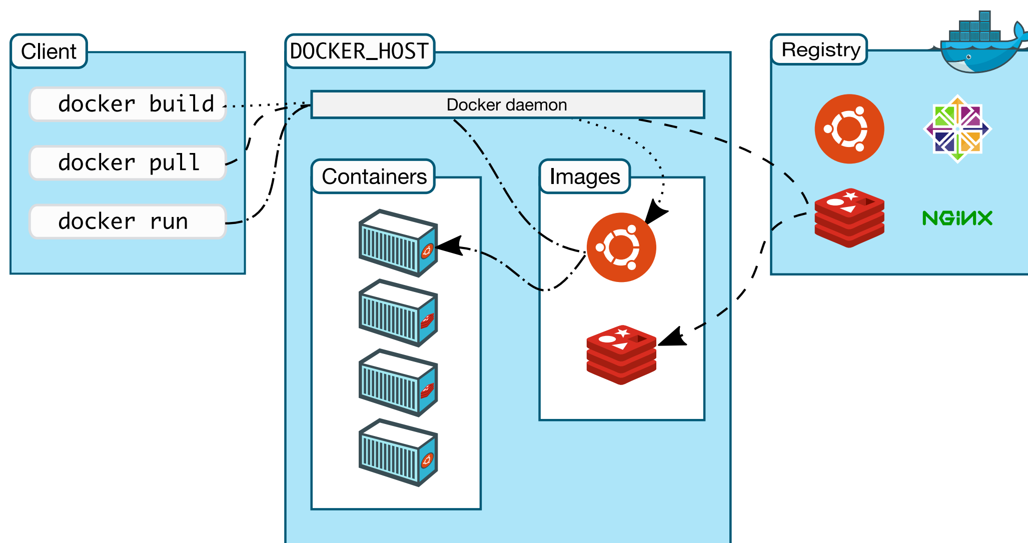


Figure 10: High-level overview of Docker architecture [33]

Underlying technologies:

Docker builds on top of existing Linux container technology such as cgroups and namespaces, additionally it also uses union file systems (UFS) for added benefits to the container development process. These features are integrated into a low-level component known as the runC tool, a portable container runtime [34]. The key technologies of Docker are listed below:

- *cgroups*, which provide a way for the Linux OS to manage and monitor resource allocation for a given process and set resource limits (e.g., CPU, memory, and network).
- *namespaces* help isolate containers by making sure that each container will have its own namespace and that processes running inside that namespace will not have access to the rest of the system.
- the *union file system* is used to create and layer Docker images. All images are built on top of a base image, and every change adds a new layer to the base image. Docker caches the layers the first time they are built, so if a new layer is created and Docker detects it already exists in cache, it will re-use the existing layer, avoiding the duplication of a complete set of files. This feature is what enables Docker containers to be small in size and have quick start times.

2.1.2.3 Container orchestration

Container orchestration automates the deployment, networking, scaling, availability, and lifecycle management of containers. When used in small numbers, containers are easy to deploy and manage manually,

but as the number of containerized applications grows, so does the need of an orchestration platform like Kubernetes [35].

The main benefit of container orchestration is automation, which reduces the effort and complexity of managing a large number of containers, allowing developers to create, deploy, and release new features faster. Additionally, container orchestration platforms typically include features like load balancing and scaling, which extend the benefits of containerization [35].

Kubernetes

Kubernetes, also known as K8s, is an open-source container management solution first announced by Google in 2014 and released in 2015. It provides a managed execution environment for deploying, running, managing, and orchestrating containers across clusters or clusters of hosts, making it extremely useful for developers. Kubernetes clusters can be deployed on a variety of public clouds (AWS, Google Cloud, Azure) and bare metal servers due to Kubernetes being infrastructure agnostic [36].

Kubernetes architecture:

As shown in Figure 11, the Kubernetes cluster is the largest entity in Kubernetes, and it is made up of one or more machines, also known as nodes. In a Kubernetes cluster, there are two types of nodes: the master node, which hosts the Kubernetes control plane and manages the cluster, and the worker nodes, which run the containerized applications.

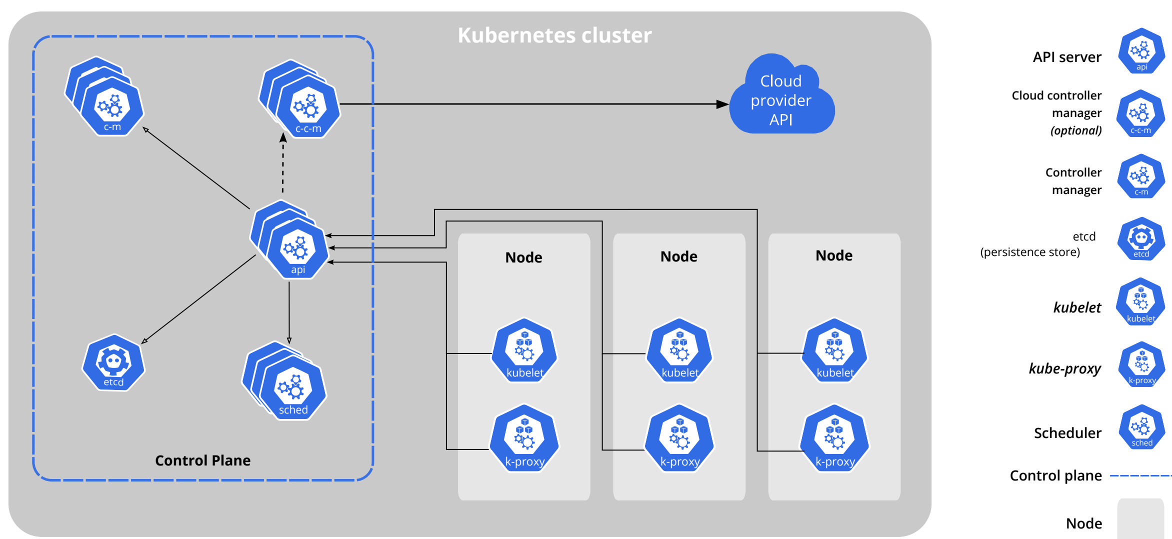


Figure 11: Overview of a Kubernetes cluster [37]

The worker nodes host the pods, which are the components of the application workload. A pod is the smallest deployable unit on a cluster that can be created and managed. Multiple containers can run within

a pod, sharing the same network and storage space, but it is best practice to run as few containers as possible per pod [37].

The control plane, which is located in the master node, manages the cluster's worker nodes and pods. It is in charge of making global cluster decisions as well as detecting and responding to cluster events via its many components.

Kubernetes also provides several addons for more features, such as DNS and a web-based dashboard. The dashboard provides an interface to manage the Kubernetes cluster, which can be used as an alternative to the kubectl command-line tool, although with limited functionality.

2.2 Dashboard

The term dashboard, in its broad meaning, is simply a visual display of the relevant information required for a certain purpose consolidated and arranged on one or multiple screens [38]. However, nowadays it is seen as a nuclear tool for both individuals and organizations, as it can function as an interactive management tool to monitor and act on critical information to achieve one or more individual and/or organizational goals [39]. To that end, dashboards should not be be deemed as single-purpose visualization tools since "their design and contexts of use are considerably different from exploratory visualization tools" [40].

Dashboards can also benefit from a cloud architecture based on microservices in terms of scalability and reliability. Microservices allow the dashboard to be independently scalable, making it easier to handle traffic or data volume increases. Moreover, if the dashboard is replicated and one instance fails, the remaining instances can continue to function, resulting in a more resilient system overall. These advantages make a microservices-based approach an attractive option for building and maintaining cloud-based dashboards.

2.2.1 Good design practices

In the literature surrounding the subject of dashboards there are some guidelines that can be considered as best practices, as is presented in [39], [41]. We leverage these studies to highlight the most relevant characteristics, summarized in Table 4, in regards to the presentation and format of dashboards in the context of operational dashboards.

The first step in dashboard design should be identifying the dashboard's purpose, since the features in a dashboard should always align with its purpose. A poor fit between purpose and features might result

Table 4: Guidelines for presentation formats and dashboards

Year	References	Description
2000	[41]	Define dashboard purposes
2006	[38]	The dashboard should fit on a single screen but allow the user to drill-down to break up large data sets
2006	[42]	Use colors to increase perception
1993	[43]	The use of Gestalt principles (similarity, continuity, closure, proximity, figure/ground, and symmetry & order) to improve perception
2005	[44]	The ability to alter the display format can help focus on more relevant information
1991	[45]	Tabular information leads to better decisions when monitoring specific values
1991, 1991, 1994	[46]–[48]	Graphs are more suitable for comparing a set of values
2007	[48], [49]	Graphs reduce information overload

in sub-optimal decision making by the end-user. Features such as drill-down and presentation flexibility are crucial as they can serve all the purposes of the dashboard. The number of features should also be carefully considered, as excessive or unnecessary feature usage can complicate the dashboard and detract from its purpose, while too few features can also hinder the dashboard's goals.

The features in a dashboard can be divided into functional features and visual features [39]. Functional features refer indirectly to visualization and describe the capabilities of the dashboard:

- **Format presentation type** - provide the ability to choose between graph or tabular;
- **Format presentation flexibility** - display the data at different levels of aggregation;
- **Drill-down and drill-up** - move between levels of information to get a more detailed view or a broader view;
- **Scenario analysis** - predict scenarios that can take place in the future with the available data;
- **Automated alerts** - notify the user when an event or error condition occurs.

Visual features refer to the essence of data visualization, i.e. how the data is effectively presented to the user:

- **Single page** - keep the number of pages to a minimum;
- **Frugal use of colors** - use a color palette throughout the dashboard to keep colors consistent and non distracting;
- **High data-ink ratio** - remove elements that do not add information to graphs;
- **Grid lines in graphs** - use grid lines to help detect differences between quantitative values.

Typically, unstructured data is preferred over aggregated and structured data, which may not be sufficient for proper analysis. The drill-down feature can be used for this purpose, allowing the user to examine the data further if necessary. And to compare data, the cognitive fit thesis suggests graphs, but the user should have the option to select another format, such as tabular, due to varying user preferences.

To enhance visual perception, the choice of colors and style of graphs should adhere to the principles of visualization, i.e., simple solutions that correctly guide the user, as opposed to flashy graphs that diminish perception. The dashboard should be user-friendly and straightforward, allowing the user to concentrate on the most pertinent data. In light of this, it is advisable to reduce the dashboard size to a minimum number of pages, if possible a single page, and to enable additional data analysis via dashboard features.

In brief, given the versatility in functionality and applications of dashboards, there is no unique approach to building dashboards. The best that can be achieved is a set of guidelines which can serve as a base for dashboard design, as there is no guarantee that the design principles of one dashboard type will systematically transfer to another. In addition, dashboard objectives and goals are not always clear at the start, so it is practical to choose dashboard solutions that are more flexible and easier to upgrade.

2.2.2 Dashboard types

In [40] a survey on dashboard design was conducted to ”construct a design space for dashboards, and identify major dashboard types”. The authors analysed 83 different dashboards and, in an initial phase, derived the design space categories and codes to be later used to cluster dashboards into different types. As shown in Figure 12, there are 15 distinguishing factors organized into 5 major groups: purpose, audience, visual & interactive features, and data semantics.

The characteristics in Figure 12 establish a dashboard taxonomy, which the authors used to classify dashboards based on ”their design goals, levels of interaction, and the practices around them”. As shown

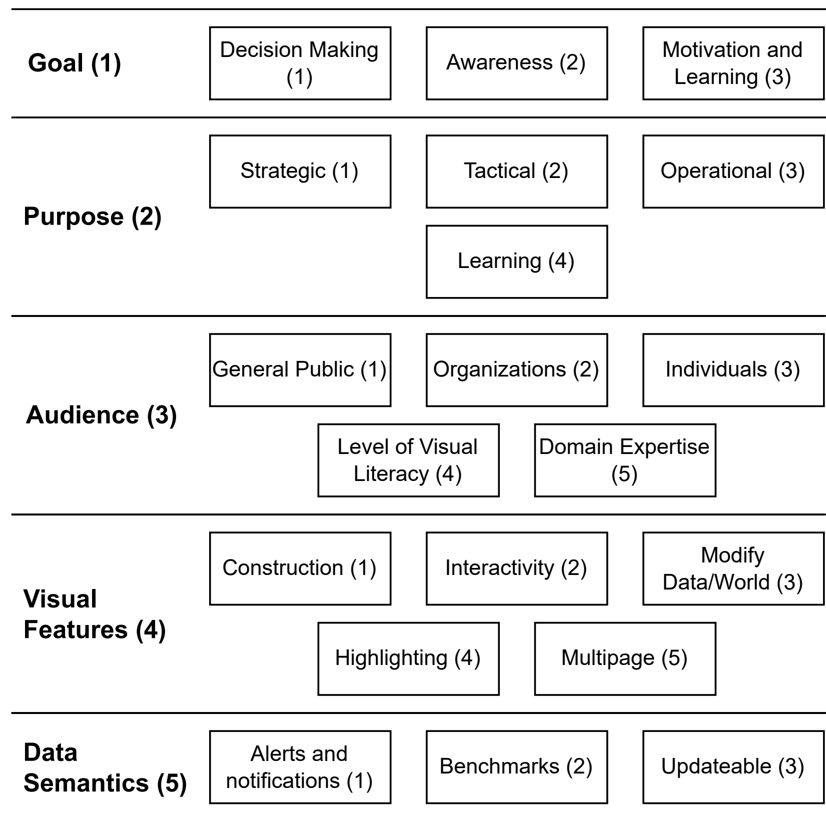


Figure 12: Taxonomy of dashboard characteristics

in Table 5, the authors defined 7 separate groups of dashboards based on their shared qualities. These categories are elaborated upon below:

- **Dashboards for Decision-Making** - Strategic Decision-Making and Operational Decision-Making. They primarily target organizations and provide views that allow the user to interact with the most important data and identify areas of concern or opportunity for the organization. In business contexts, they can be used to view metrics in real-time (operational) or on a temporal scale (strategic).

- **Static Dashboards for Awareness** - Static Operational and Static Organizational. These dashboards are considered "static"because they are designed for minimal interactivity and are primarily used to provide awareness of a particular aspect of an operation or organization. Although they may be designed with low visualization literacy in mind, domain knowledge is required for the user to comprehend the significance of data trends and distributions.

- **Dashboards for Motivation and Learning** - Quantified Self and Communication. These dashboards fall out of the organizational context, and are either targeted towards individuals, or the general public. Dashboards for personal use typically feature interactive user interfaces and alerts;

examples include finances, exercise, and dieting, whereas dashboards for the general public appear to be designed for communication and education of the consumer, as such they present the data in a simple manner that can be independently interpreted by the end-user; examples include dashboards for public health, crime rates, and other civic data.

- **Dashboards Evolved** - Composed of a singular cluster. This category identifies dashboards that do not fit into the previously stated clusters, even if they share characteristics with multiple clusters.

Table 5: Design choices of the different types of dashboards

Category	Goal	Purpose	Audience	Visual Features	Data Semantics
Strategic Decision-Making	1.1	2.1 & 2.2	3.2	4.2 & 4.5	5.3
Operational Decision-Making	1.1	2.2 & 2.3	3.2	4.2 & 4.5	5.2 & 5.3
Static Operational	1.2	2.3	3.2 & 3.4 (low)	———	5.2 & 5.3
Static Organizational	1.2	———	3.2 & 3.4 (medium)	———	5.3
Quantified Self	1.3	2.3	3.3 & 3.4 (high)	4.2 & 4.5	5.3
Communication	1.3	2.4	3.1 & 3.4 (medium)	———	5.3
Dashboards Evolved	———	2.4	3.1 & 3.4 (high)	———	5.3

Although there are numerous types of dashboards, they do not utilize all of the characteristics identified in Figure 12, either because the characteristic is not dominant for the observed type of category, or because it should not be present in that category. This demonstrates how challenging it is to properly categorize dashboards, as they may exhibit characteristics from multiple clusters and there is no one-size-fits-all solution. This is implied by the "Dashboards Evolved"category, which encompasses an expanding universe of dashboards that do not neatly fit the previous assumptions regarding dashboards' appearance and functionality. As exemplars circulate and tastes and fashions vary, dashboards may also undergo substantial conceptual and graphical changes over time.

2.2.3 Frameworks for dashboard development

Most of the development in dashboards has been driven by practice and software providers, as opposed to academic research. This means that a large range of solutions are available on the market, both as a paid service and as a free service. This section compares the capabilities of some of these solutions based on their market appeal and ubiquity in IoT-based systems, in order to locate the best-suited frameworks. We also analyse the following properties of the frameworks available in the market to properly compare the available solutions.

- **License** - describes the type of license of the software. The most common licenses are open-source, which is available for anyone to use and modify for free, and proprietary software, which is software that legally remains the property of the organisation that created it, and requires buying a special license to use it.
- **Codeless** - codeless platforms, also known as low-code/no-code development platforms, allow developers to quickly build applications by using drag-and-drop and other similar tools. The use of codeless platforms lowers the development complexity and time to market, however, they may limit the developer in terms of customization, i.e., a codeless application might not be able to fulfill all the business needs.
- **Write-back capacity** - to provide write-back capacity, a platform must allow the user to interact and change the state of the devices in the dashboard (e.g. a button to turn a LED on/off). This can achieved through a variety of different communication protocols, or by simply writing to a database from which the device can read periodically.
- **Database support** - the dashboards that the platform supports for reading/writing data to and from the dashboard/devices. Some platforms extend their official support by allowing the community to contribute (i.e. community created plugins) and thus provide support for a wide array of databases.
- **Multi-page application** - describes if the platform provides the ability to create dashboard applications with more than one page.
- **Design flexibility** - a flexible design allows for functional modifications, such as new features or upgrades to existing ones, without requiring structural changes. Platforms with greater flexibility can adapt to new requirements considerably more quickly, albeit at the expense of increasing complexity.

- **Ease of deployment** - software deployment can be generally divided into three phases: development, testing and monitoring. A platform with a higher ease of deployment covers the three phases and produces an application which is ready for market in less time than others. Depending on the market and the business needs, some businesses might prioritize less time-to-market over expending more time to create a more expensive and perfected solution.

- **Optimal use case** - describes one or several attributes which can be seen as the main "selling points"of the respective platform. If the requirements for an application match the optimal use case of a platform, then the platform might be a good choice for its development.

The findings in Table 7 were drawn from the official documentation provided by the platforms Thingsboard [50], Ubidots [51], Node-RED [52], Grafana [53], and Dash [54]. To better understand these platforms a few examples were analysed in respect to the topics that were approached in this chapter, such as good design practices and dashboard categorization. These examples are drawn from literature, mostly focusing on IoT applications.

The initial step in this analysis is to determine whether the examples adhere to the good design practices defined before. As the majority of guidelines and functional/visual features are intertwined, a mixture of them was chosen and analysed in Table 8.

According to Table 8, the examples which are illustrated in Figures 13-17, use graphs as a presentation format and do not offer tabular as an alternative, that could assist in decision-making. Some of these examples, such as Mataloto *et al.* [55] could be improved by implementing scenario analysis, as predictive analysis is already one of their stated goals for the platform; this would relieve the end-user of manually performing this task.

In general, the examples presented are capable of achieving their purpose in accordance with the guidelines, however there is room for improvement; only Chetty *et al.* [56] failed to adequately define its purpose, and therefore its quality cannot be assumed.

The next step is to analyse the examples based on their design choices. Each example is assigned the category that best matches its description, and then it is evaluated based on whether or not it satisfies the category's characteristics. This is accomplished using Table 5, and the results are presented in Table 6.

All of the examples meet the design choices for the proposed category, demonstrating that the proposed taxonomy can successfully classify dashboards according to their characteristics. This may be partly

Table 6: Analysis of dashboard examples according to design choices

Example	Chetty *et al.* [56]	Mataloto *et al.* [55]	Ali *et al.* [57]	Aghenta and Iqbal [58]	Diaz *et al.* [59]
Platform	Ubidots	Node-RED	Grafana	Thingsboard	Dash
Category	Static Operational	Operational Decision-Making	Static Operational	Static Operational	Static Organizational
Goal	Yes	Yes	Yes	Yes	Yes
Purpose	Yes	Yes	Yes	Yes	Yes
Audience	Yes	Yes	Yes	Yes	Yes
Visual Features	Yes	Yes	Yes	Yes	Yes
Data Semantics	Yes	Yes	Yes	Yes	Yes

attributable to the selected platforms, as they permit customization but also direct the developer toward common design interfaces and metrics, resulting in dashboards with consistent characteristics.

From this analysis, and with the data from Table 7 it is possible to conclude that Thingsboard and Ubidots are complete solutions that attempt to address every business need in the IoT domain. As a result, organizations should fully adhere to them in order to get the most out of them, rather than just use them for data visualization.

Organizations who have already implemented some sort of IoT infrastructure and wish to expand it to include data visualization and monitoring should look into more specialized platforms such as Node-RED, Grafana, and Dash. These platforms are attractive alternatives since they are open-source and have been enhanced by community contributions, making them more compatible with existing IoT infrastructure.

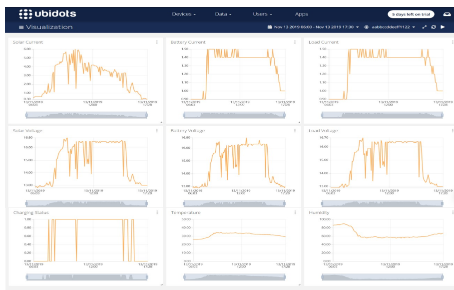


Figure 13: Ubidots dashboard example [56]

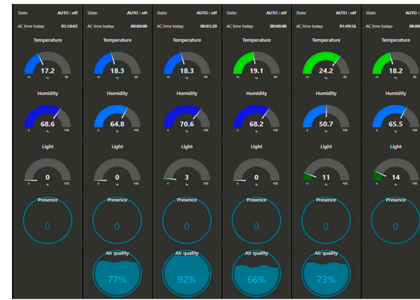


Figure 14: Node-RED dashboard example [55]

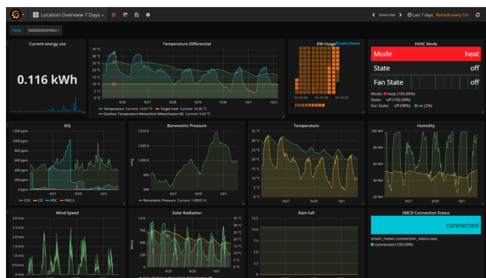


Figure 15: Grafana dashboard example [57]

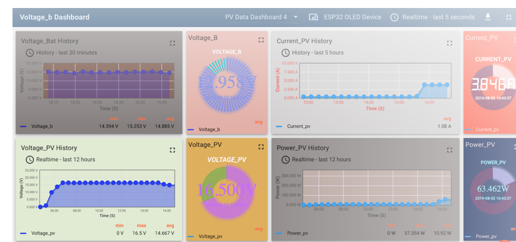


Figure 16: Thingsboard dashboard example [58]

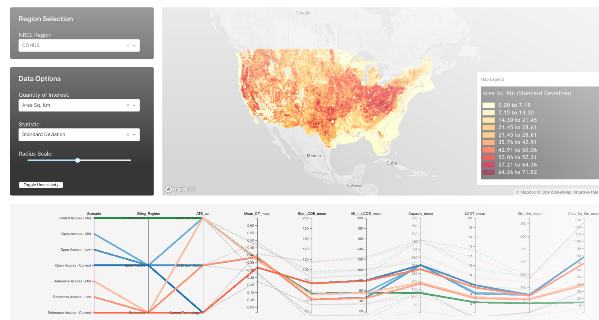


Figure 17: Dash dashboard example [59]

2.3 Summary

The information presented in this chapter provides some valuable insight: (1) Microservices architecture adheres to DDD principles by decomposing an application into small, focused services with a clear and consistent domain model in order to maintain cohesion and loose coupling. The design process requires a comprehensive knowledge of business processes and the application of DDD techniques such as bounded contexts, entities, and aggregates. (2) Containers package applications with their dependencies and can be deployed on most computing platforms and thus are ideal for microservices. Containers also provide isolation and conceal the underlying technology of the applications, which are core concepts in the

microservices architecture. (3) The collaboration style and implementation choice for inter-microservice communication depends on the requirements imposed by the microservices solution, such as communication reliability and maximum allowed latency. A combination of technologies may also constitute a viable strategy. (4) Dashboards are not limited to data visualization, and can be used as interactive management tools. (5) There are guidelines and design choices that can be used to create a dashboard that is suitable for the intended purpose.

Table 7: Comparison of common dashboard platforms available on the market

Platform	Ubidots [51] - 2013	Node-RED [52] - 2013	Grafana [53] - 2014	Things-board [50] - 2016	Dash [54] - 2017
License	Paid Professional/Enterprise versions	Open-source	Open-source & Paid Professional/Enterprise versions	Open-source & Paid Professional/Enterprise versions	Open-source & Paid Enterprise version
Codeless	Supported, using point-and-click visual tools	Supported, using flow-based programming	Supported, using point-and-click visual tools	Supported, using flow-based programming	Not supported, requires programming knowledge of either Python, R, Julia, or F#
Write-back capability	Supported, using HTTP, MQTT or TCP/UDP	Supported, using communication protocols available in the Node-RED Library	Not supported	Supported, in the form of RPC commands	Supported, using communication protocols available in Python
Database support	Ubidots' own time-series backend solution	Support for the most common databases is provided through the Node-RED Library	MySQL, PostgreSQL and SQLite are supported by default, with plugins available to extend support to other databases	PostgreSQL is recommended, but Cassandra and TimescaleDB are also supported	Any database available for Python
Multi-page application	Not supported	Supported, using community created nodes	Not supported	Not supported	Supported
Design flexibility	Low	Average	Average	Low	High
Ease of deployment	Easy	Easy/Moderate	Easy/Moderate	Easy	Moderate
Optimal use case	Out-of-the-box IoT cloud solution for device management, data collection, processing, and visualization	Visual programming tool with a broad range of applications, including IoT workflows	Cloud or on-premises framework for data analysis and monitorization	Out-of-the-box IoT cloud or on-premises solution for device management, data collection, processing, and visualization	Cloud or on-premises framework for interactive web applications, with great scalability and design flexibility

Table 8: Analysis of dashboard examples according to guidelines

Example	Chetty *et al.* [56]	Mataloto *et al.* [55]	Ali *et al.* [57]	Aghenta and Iqbal [58]	Diaz *et al.* [59]
Defined dashboard purpose	No	Yes	Yes	Yes	Yes
Fits on a single screen	Yes	No	Yes	Yes	Yes
Format presentation type	Graphs	Graphs	Graphs	Graphs	Graphs
Format presentation flexibility	No	Yes	No	No	Yes
Drill-down and drill-up features	No	Yes	No	Yes	No
Scenario analysis	No	No	No	No	No
Automated alerts	No	Yes	No	No	No
Use of Gestalt principles	Average, lacks continuity	Great	Poor, the layout is confusing and lacks symmetry	Great	Great
Use of colors	Poor, should use more colors for different graphs	Great	Great	Poor, excessive use of colors without consistency	Great
Data-ink ratio	High	High	Average, some graphs are hard to read due to overlapping	Low, too much color and overlapping	High

Chapter 3

System Specification & Design

This chapter offers a comprehensive examination of the inner workings of the cloud, and the changes that are necessary for achieving the final architecture based on the CMP architecture. The chapter begins by addressing the system's requirements, architecture, and use cases, and then delves deeper into the three key components of the system: microservices, database, and dashboard.

3.1 Functional and non-functional requirements

Since this dissertation is a continuation of [8], it must satisfy some of Link4S former requirements in addition to new ones. To be successful, every project of this scope must define both functional and non-functional requirements. Non-functional requirements, also known as non-behavioral requirements, are the quality constraints that a system must meet. Functional requirements are the basic capabilities that a system must provide. The project's requirements, which influence the development of this dissertation, are related to the cloud and data visualization:

3.1.1 Functional requirements

- Receive and parse data from devices in MessagePack format
- Send commands to devices in a format that can be understood by them
- Visualize the latest sensor data in real-time using a dashboard

- Control access to users by implementing a login system
- Allow for the ability to change end-device configuration remotely
- View the evolution of sensor data over time using a graph or chart
- Visualize system overview in real-time using a dashboard
- Implement in the microservices the components of the CMP architecture
- Use the interfaces of the CMP architecture to enable seamless communication between the components
- Store the parsed data in a database for future querying and analysis

3.1.2 Non-functional requirements

- A user interface that is easy to understand and navigate for all users, as the user that interacts with the dashboard might have low level of technological literacy and the amount of data available for query is extensive. This includes providing clear labels and implementing a search function.
- Ensure the security of sensitive data by implementing encryption for data transmission between the cloud and devices, hashing for data storage, and a robust access control system to protect against unauthorized access.
- Scalability to accommodate new devices and sensors, with minimal interruption to the system's operation. This will be achieved by using the CMP architecture that allows for easy addition of new devices and sensors with minimal interruption to the system's operation.
- High reliability and availability, meaning the system should be able to operate consistently and without interruption for prolonged periods of time.
- Performance must be efficient enough to handle high volume of data.
- The system will be designed to handle devices with different protocols and communication standards, while maintaining compatibility with existing system.

3.2 Architecture overview

Figure 18 provides an overview of the final cloud architecture, displaying the components and interfaces of the CMP architecture on colored backgrounds. In addition, the original cloud implementation is highlighted and included in the Device Ecosystem.

The Device Ecosystem is a key component of the CMP architecture as it provides the "infrastructure for the devices to operate using their own protocols and data formats" [10]. The infrastructure in this instance consists of the Blackwing server, parsing microservices, Process Data microservice, and MongoDB database. The Blackwing server communicates directly with the devices and reroutes the messages to the appropriate microservice for parsing.

The server and parsing microservices utilize the Blackwing [60] framework, a Python framework that handles all communication between microservices within the framework, freeing the developer to only implement the code for the parsing microservices. TCP/IP is used for communication between the server and microservices, and configuration files contain all required network settings.

The Blackwing framework uses the specific packet format illustrated in Figure 19. This format consists of a header and a payload, with the header itself separated into four sections. The header's first byte identifies the Message Type, which indicates whether or not the header is encrypted. If the header is not encrypted, it will only contain the Microservice Opcode which identifies the parsing microservice; if it is encrypted, it will be encrypted using the RSA algorithm and will also contain the AES Key and AES IV.

When data is encrypted, decryption is the responsibility of the server. To accomplish this, it will first use its own private RSA key to decrypt header data encrypted with a public RSA key. After the header data is decrypted, the AES key and AES IV are used to decrypt the payload, which is then sent to the microservice with the matching Opcode. If a response needs to be sent from the parsing microservice to the respective device, the server must also encrypt the data.

For the final implementation, the parsing microservices require modifications in order to make API calls to the Process Data microservice. In addition, direct database interactions are no longer supported, as they are now handled by the Process Data microservice. Although not highlighted in the figure, the MongoDB database was also present in the original implementation, but was instead connected to all the parsing microservices in order to store device messages and configuration data.

Using a REST API, the parsing microservices communicate with the Process Data microservice. When data that has been parsed arrives at the Process Data microservice, it is sent to the Mediator via the Device Interface. As the foundation of the Device Ecosystem, the Process Data microservice uses the

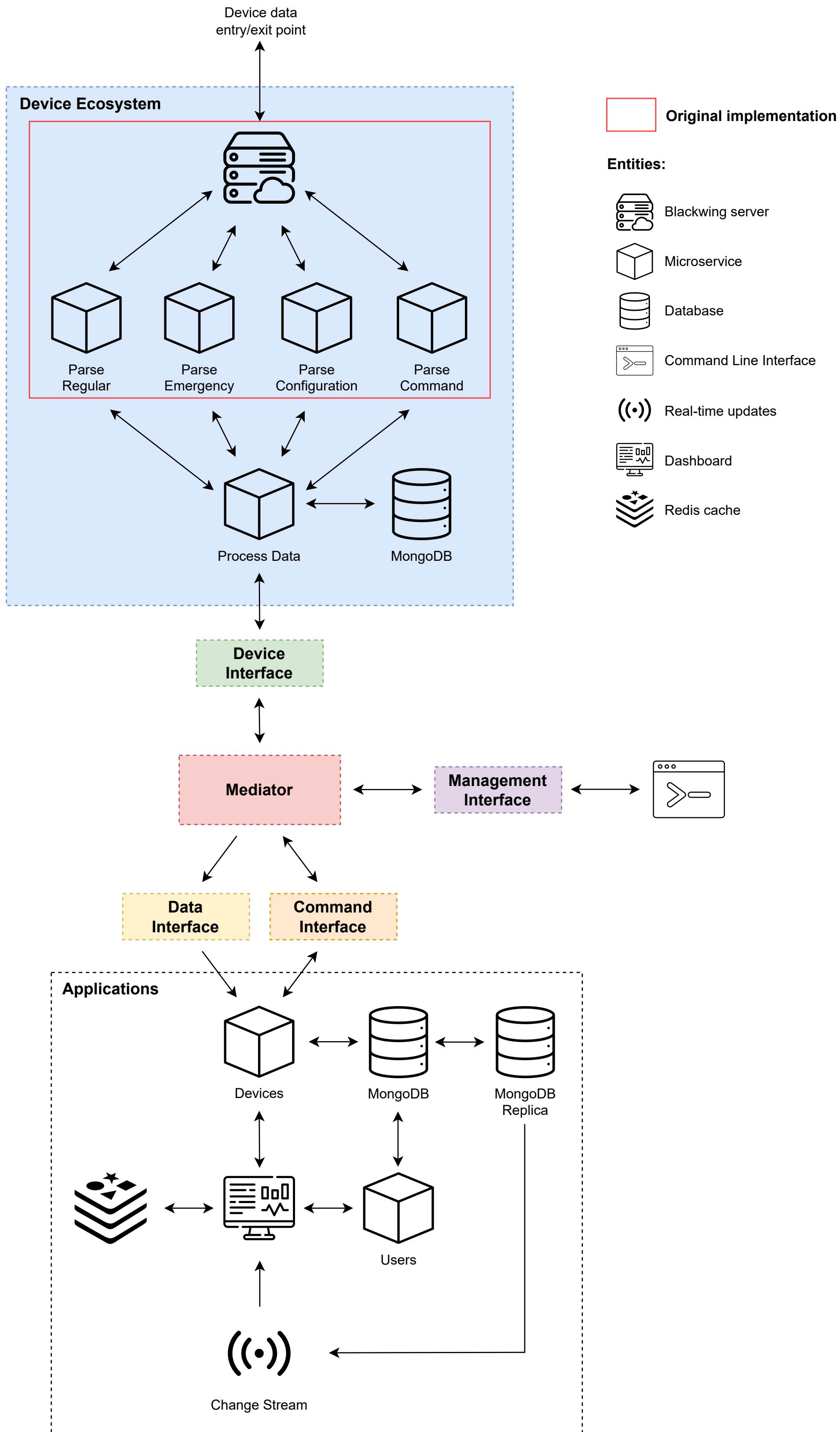


Figure 18: Architecture overview

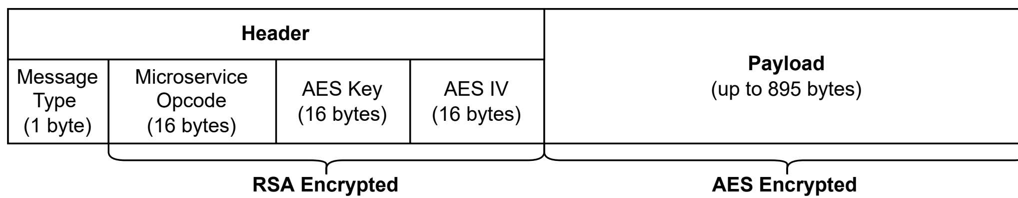


Figure 19: Blackwing message format

Device interface to communicate with the Mediator via gRPC and can forward data from devices to the Mediator as well as process responses to commands received from the Mediator.

The Mediator is the central component of the CMP architecture and is responsible for routing messages to the architecture's other components via the available Interfaces. The implementation of interfaces and the Mediator are beyond the scope of this dissertation. Instead, the microservices interact with the interfaces via Python libraries that provide a high level of abstraction, and in the case of the Mediator, its configurations are performed via a CLI so that it is aware of the correct data paths.

In the Applications section the Dashboard application and its supporting entities are present. The Devices microservice implements two CMP architecture components: the Data Consumer and the Command Producer. The first is responsible for receiving messages sent by devices, while the second is responsible for sending commands to devices and receiving the corresponding response (if sent). Each of these components has a corresponding Interface for communicating with the Mediator.

Alongside the CMP components, the Process Data microservice interacts with the MongoDB database, where it stores device data, and provides a REST API with device-specific endpoints. An API endpoint is a URL that enables communication with a microservice to retrieve or modify data. The endpoints are accessible via HTTP requests with common methods such as GET and POST, enabling applications to access sensor data, device configurations, and device commands.

The Users microservice, unlike the Process Data microservice, does not implement any of the CMP architecture's components and is solely responsible for managing users; it also provides a REST API with endpoints for user creation, management, and authentication.

The Dashboard is the primary component of the Application segment; it enables end-user interaction with the devices via an intuitive interface. The user can view the state of the devices, sensor data, and configurations, in addition to sending commands to the devices, which was previously only possible using Python scripts. Using a WebSocket to communicate with the Dashboard, the Change Stream process provides real-time updates.

The MongoDB Replica duplicates the data from the primary MongoDB instance, ensuring data availability and redundancy. It is also a requirement for the Change Stream service, as it is dependent on the same-named MongoDB feature and cannot be implemented on a non-replicated database set.

3.3 Use cases

The use case diagrams in Figure 20, Figure 21, and Figure 22 illustrate how entities, also known as actors, in a system interact to achieve specific goals. Three systems were identified for this exercise: the end-device, the Device Ecosystem, and the Applications. The majority of the entities in the identified systems were used as actors; only the database was omitted because interaction with the database occurs solely via the respective microservices and can therefore be simplified.

Include and exclude relationships are used to establish relations between use cases. An include relationship is used to include one use case within the behavior sequence of another use case, this relationship implies that the included use case must be executed when the base use case is executed. This is not true for an exclude relationship, as most of the time, an extend relationship has a condition attached to it and it only executes when the condition is true.

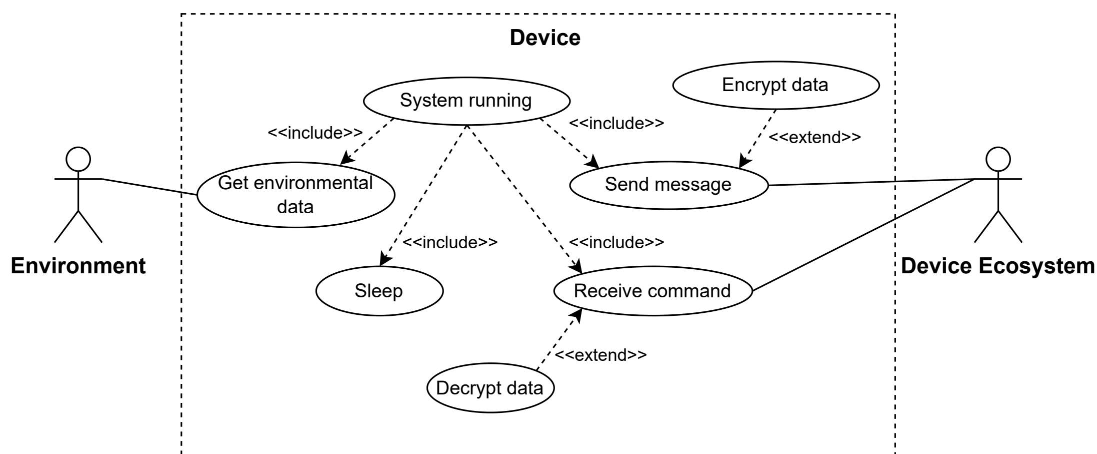


Figure 20: Device use cases

3.4 Microservices

The cloud architecture contains a number of microservices that serve a variety of purposes and must communicate using different protocols. This section will delve deeper into the design of the microservices', their features and how they communicate with each other.

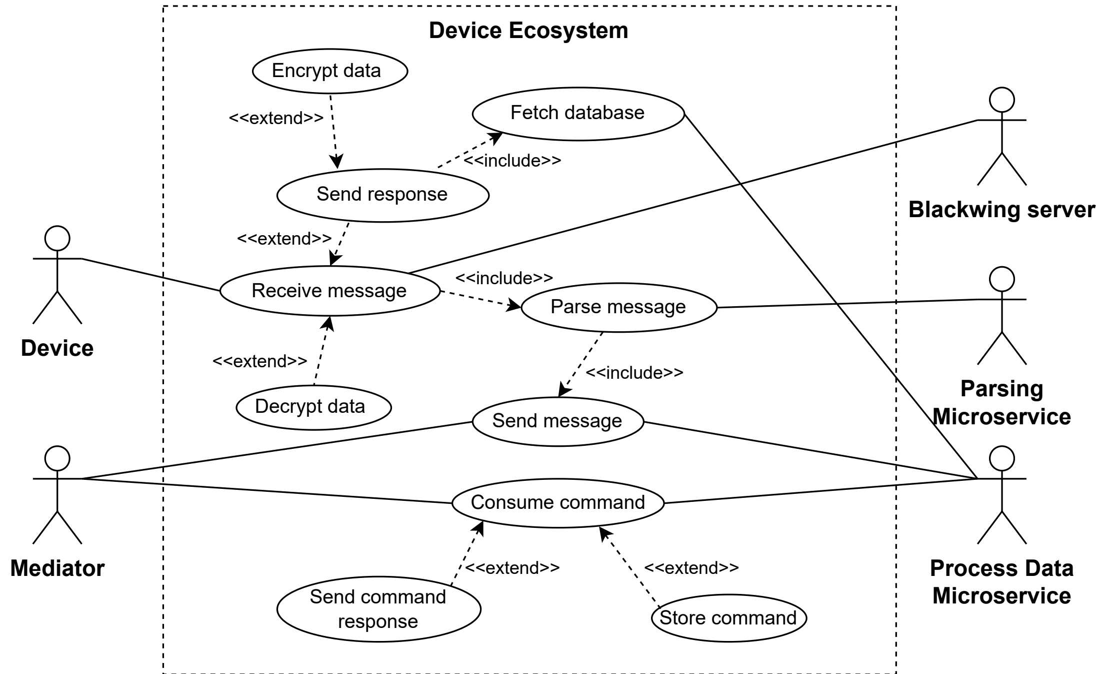


Figure 21: Device Ecosystem use cases

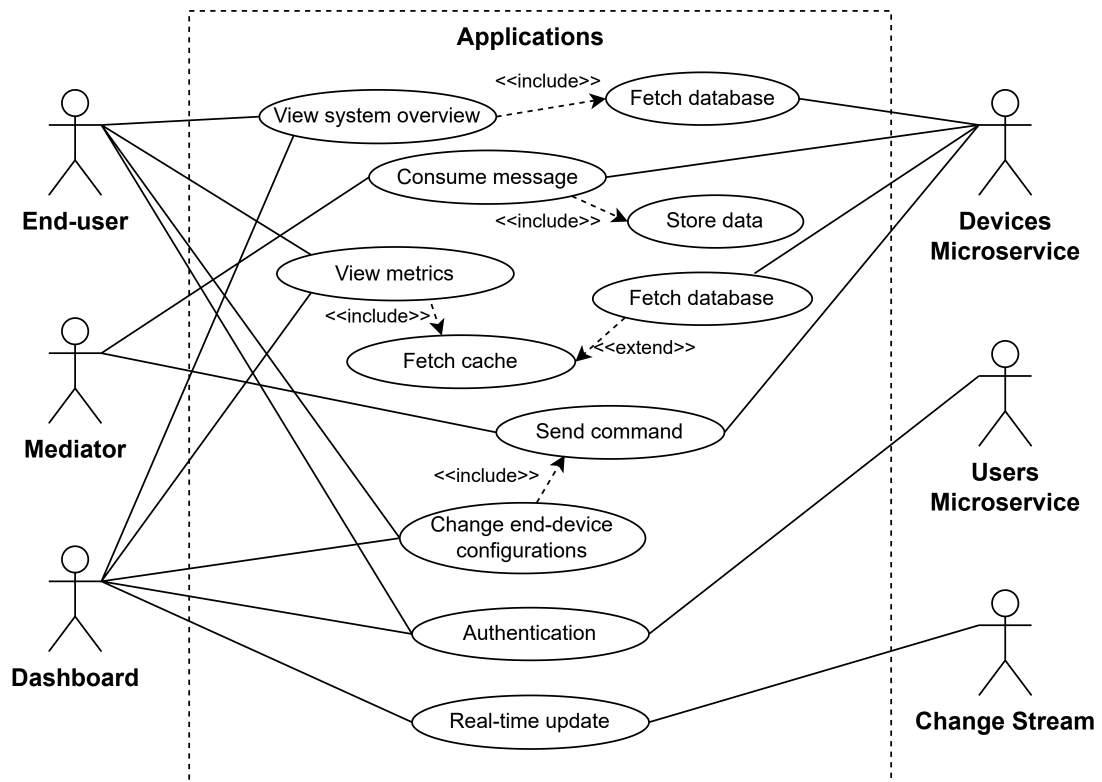


Figure 22: Applications use cases

3.4.1 Data parsing

The data parsing microservices are responsible for receiving the server's raw data and transforming it into a format that can be easily stored in a database and transmitted over the network. There are four microservices that parse data: Regular, Emergency, Configuration, and Command. Due to the fact that device communication is generic, all microservices adhere to the general structure depicted in Figure 23.

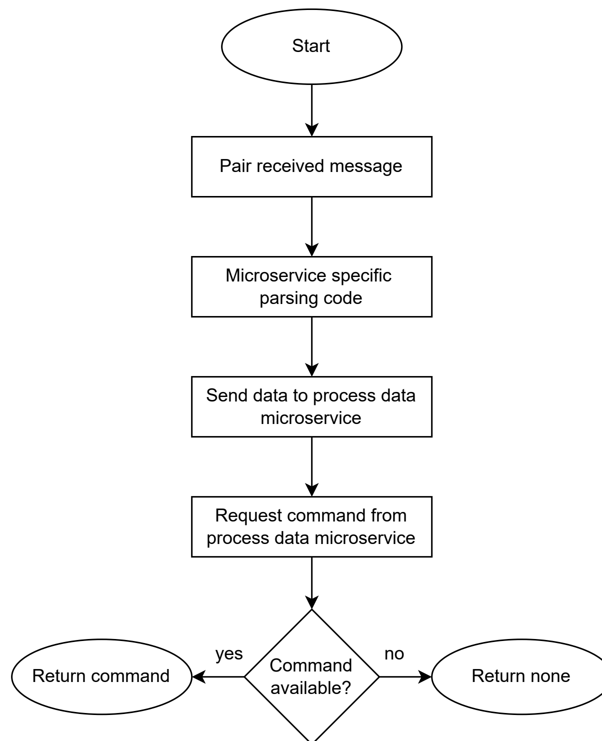


Figure 23: Generic parsing microservice flowchart

Figure 24 illustrates the datagram format that the parsing microservice receives from the server. This format is composed of datagram objects and is not constrained by the number of objects or the order in which they appear. Also provided is an example of a datagram, in which each object includes a string identifier and data that can consist of a single value or multiple values in a list. Certain objects within the datagrams are included in every transmission because they provide essential metadata to the cloud, as described below. The remaining data in the datagram is optional, and is typically sensor data or device configuration data. The datagram displayed is only for a temperature (sTMP) and humidity (sHUM) sensor, although it is scalable to many more sensors.

- **P** - the parts identifier specifies the current datagram part and the total number of parts. When a message exceeds the 1024-byte modem transmission limit, it is broken up and sent in multiple

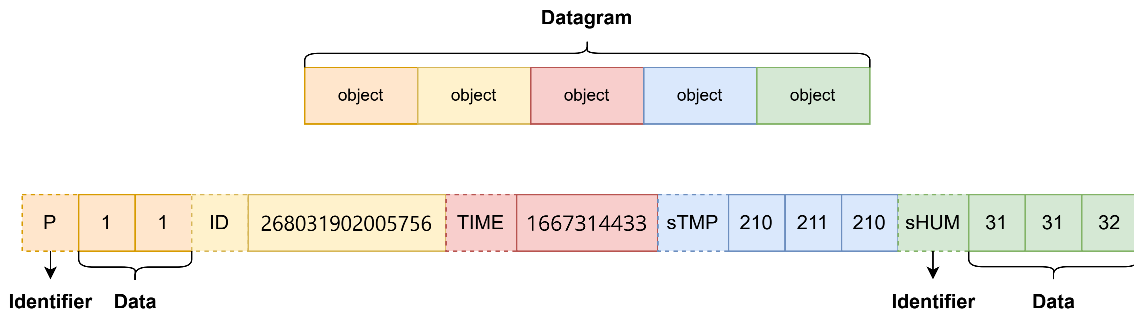


Figure 24: Cloud payload datagram with example

transmissions. If the values are 1 and 2, for instance, it indicates that this datagram corresponds to the first of two total parts. Packages can also be received out of sequence, so receiving parts with matching numbers does not necessarily indicate that all parts have been processed.

- **ID** - the board id identifying the board that transmitted the message.
- **TIME** - the transmission's timestamp in Unix epoch format as determined by the device.

To process the message, the microservice will first form pairs from the received datagram consisting of the string identifier and the corresponding data values. Then, the parsing code for the generated pairs is executed, and the finalized data is sent to the Process Data microservice via an HTTP POST request. After transmitting the data, the microservice makes a second request, this time using the HTTP GET method to retrieve a command awaiting transmission to the respective device. If a command is returned, it is included in the response and sent to the server, which will encrypt the response if encryption is enabled and communicate with the device, otherwise a null value is returned.

In addition to the required datagram objects, the regular microservice will also parse the signal quality with the respective Received Signal Strength Indicator (RSSI), Reference Signal Received Power (RSRP), and Reference Signal Received Quality (RSRQ) and the sensor variables samples if they are present. In the case of the configuration microservice, parsing is performed in various levels, as modem configuration, sensor variables, and transmission interval configurations require additional parsing. The modem configuration parsing will separate the cell id, Tracking Area Code (TAC), and Tracking Area Update (TAU); the sensors variable parsing will unify all sensor configurations; for example, if it receives the variable sTMP corresponding to the temperature sensor, it will group the sensor's period, interrupt id, interrupt status, and state; and the transmission interval configurations parsing will parse the alarm or time interval in accordance with the transmission mode. The remaining microservices necessitate no further processing.

3.4.2 Data processing

The Process Data microservice is responsible for receiving the parsed data from the parsing microservices in JSON format through the available REST API endpoints. When data is received it is immediately sent to the Mediator. Since it implements the Device Ecosystem, it also has the ability to receive commands from the Mediator and store them in the database to be processed at a later stage, when a request is received from a parsing microservice the command is included in the response. To communicate with the Mediator, the Device Interface is used, which uses gRPC as the communication protocol. The use of gRPC is a constraint for this microservice, as it was defined outside the scope of this dissertation.

The REST API consists of 6 endpoints, whose specification is provided in Table 9. The first 5 endpoints are POST methods whose only purpose is to receive data from the parsing microservices and send it to the Mediator using the Device Interface. Since this operation should always be successful, the HTTP response status code should be "200 OK". If the operation is not successful, that means that the Mediator is not functioning as intended, and thus a status code of "500 Internal Server Error"is returned. The other available endpoint is a GET method which receives two parameters, the board id, and the microservice. These parameters are then used to query the database for commands with the matching board id and microservice.

Table 9: Process Data microservice API specification

HTTP Method	URI	Description	Status code
POST	/v1/regular	Send a regular message to the Mediator	- 200 OK - 500 Internal Server Error
POST	/v1/emergency	Send an emergency message to the Mediator	- 200 OK - 500 Internal Server Error
POST	/v1/configuration	Send a configuration message to the Mediator	- 200 OK - 500 Internal Server Error
POST	/v1/command	Send a command result message to the Mediator	- 200 OK - 500 Internal Server Error
GET	/v1/command	Retrieve and process a command from the database	- 200 OK - 204 No Content

The GET endpoint requires additional processing to be in a format that can be processed by the device. Figure 25 illustrates the flowchart that describes this process in greater detail. Using the provided

parameters, board id and microservice, the first step is to locate a match in the database. If a microservice-specific command cannot be located, the search scope is expanded to all microservices, also known as a global command, to determine if there is a command available for the specified board id. If no command is found, processing ends and a null value is returned; if multiple commands are found, a function is executed to find the most recent temporary command, as they have a higher priority. The command will then be removed from the database if it is temporary. Then, if the opcode of the command matches the firmware over-the-air update opcode, the delta package containing the binary data for the device application is retrieved from the database and appended to the response. The final steps involve serializing the data using the MessagePack format and, if enabled, appending an MD5 hash to the response for checksum purposes.

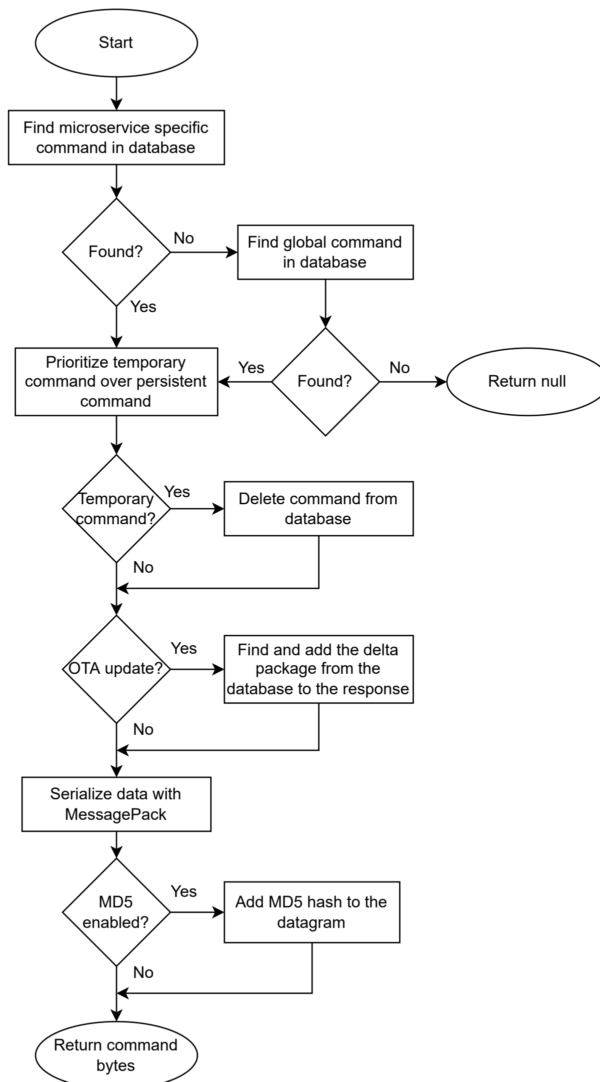


Figure 25: Process command flowchart

To receive commands sent by an application to the Mediator, a command listener is implemented as a part of the Device Ecosystem component. This command listener supports four distinct command types, and the execution pattern is similar for the majority of them. In Figure 26, the command listener begins by decoding the received command and then identifies the command type. If a command is a sensors variable command, alarm command, or FOTA update command, it is processed by Borges [8] commands' module, which generates command binary data that can be processed by devices and is then stored in the database. After processing these commands, no result is generated, as this occurs only after the command has been processed by the device. In the case of the FOTA versions command, it is not necessary to send the command to the device because it only applies to cloud-based applications. In response to this command, the database is searched for distinct firmware versions, which are then sent using the API response method.

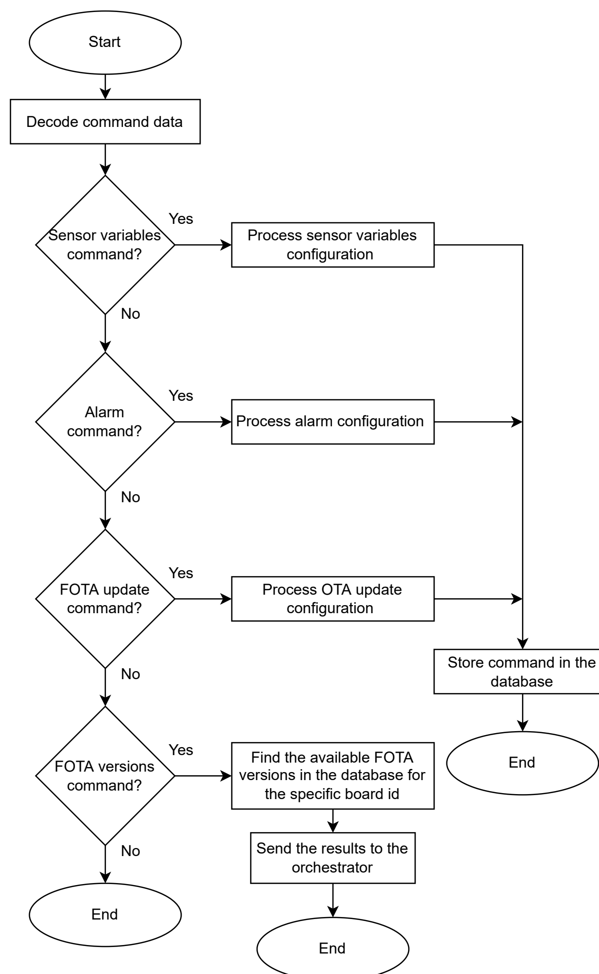


Figure 26: Command listener flowchart

3.4.3 Devices

The devices microservice is the primary interface for applications, such as the dashboard, that wish to interact with devices and their data. This is accomplished via a REST API with multiple endpoints that meet the needs of the applications. These endpoints are defined in Table 10 and Table 11, and they allow applications to retrieve sensor data and device configurations, as well as send commands to devices. In addition, this microservice implements a Data Consumer and a Command Producer to receive data from devices, send commands to devices, and receive command responses.

Table 10: Devices microservice API specification (GET methods)

URI	**Description**	**Status code**
/v1/devices	Retrieve the available device ids	- 200 OK
/v1/devices/state	Retrieve the current state of all devices	- 200 OK
/v1/devices/location	Retrieve the location of all devices	- 200 OK
/v1/devices/emergencies	Retrieve the emergencies of all devices	- 200 OK
/v1/devices/configurations	Retrieve the configuration history of all devices	- 200 OK
/v1/devices/commands	Retrieve the command results of all devices	- 200 OK
/v1/devices/{device_id}/ emergencies	Retrieve the emergencies of a device	- 200 OK - 400 Bad Request
/v1/devices/{device_id}/ configurations	Retrieve the configuration history of a device	- 200 OK - 400 Bad Request
/v1/devices/{device_id}/ configurations/latest	Retrieve the latest configuration of a device	- 200 OK - 400 Bad Request
/v1/devices/{device_id}/ samples	Retrieve the sensor samples of a device	- 200 OK - 400 Bad Request
/v1/devices/{device_id}/ samples/latest	Retrieve the latest sensor samples of a device	- 200 OK - 400 Bad Request
/v1/devices/{device_id}/ firmware	Retrieve the available firmware versions of a device	- 200 OK - 202 Accepted - 400 Bad Request - 500 Internal Server Error

Most endpoints provided in Table 10 can interact with the data from multiple devices, or a single device when the ”device_id”field is included in the URI. If an operation is successful, the status code ”200 OK”must be returned for each and every endpoint. The endpoints whose queries must contain the ”device_id”field return the status code ”400 Bad Request”if the field is absent.

The last endpoint in Table 10 differs from the others in that it will send a command to the Mediator to request the firmware versions available for a device if this information is not present in the database. When this occurs, a ”202 Accepted”status code is returned, as a request has been made but cannot be fulfilled until later. If this command cannot be sent to the Mediator, the status code ”500 Internal Server Error”is returned.

The remaining API endpoints are HTTP POST methods, defined in Table 10. There are three endpoints available, allowing applications to send commands to a specific device to change sensor configuration, such as altering a sensor’s sampling period; altering the alarm configuration; and updating a device’s firmware over the air. When the command is processed and sent correctly to the Mediator, a status code of ”200 OK”is returned; if the command is improperly formatted, a status code of ”400 Bad Request”is returned; and if the Mediator fails to receive the command, a status code of ”500 Internal Server Error”is returned.

Table 11: Devices microservice API specification (POST methods)

URI	**Description**	**Status code**
/v1/devices/{device_id}/ commands/sensors	Send a sensor configuration command	- 200 OK - 400 Bad Request - 500 Internal Server Error
/v1/devices/{device_id}/ commands/alarm	Send an alarm configuration command	- 200 OK - 400 Bad Request - 500 Internal Server Error
/v1/devices/{device_id}/ commands/firmware	Send a FOTA update command	- 200 OK - 400 Bad Request - 500 Internal Server Error

In a manner comparable to that of the Process Data microservice, a connection is established with the Mediator. After establishing a connection to the Mediator, the microservice uses the Data Interface to configure a Data Consumer to receive messages and a Command Producer to send commands and receive command results. The Data Consumer implements a data listener that allows the microservice to receive

messages asynchronously. The data listener, depicted in Figure 27, begins by decoding the received data and then verifies the type of message received in order to store it in the appropriate database collection.

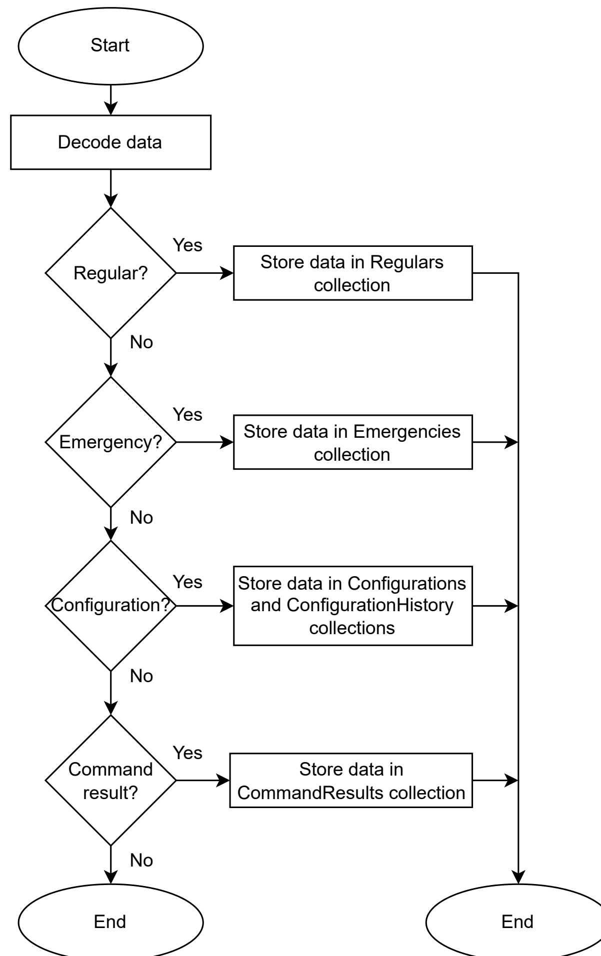


Figure 27: Data listener flowchart

To receive command results, a separate listener associated with the Command Producer is utilized. The command result listener operates similarly to the data listener depicted in Figure 27, with the exception that it only checks for the FOTA versions command. This command's output contains the available firmware versions that are then stored in the configurations collection in the database.

3.4.4 Users

The Users microservice enables applications to effectively manage users, allowing only those with the proper credentials to access applications rather than granting access to everyone. This REST APIs endpoints are defined in Table 12.

Users can be created using the POST method for registering a new user, which, when successful, should return the status code "201 Created."The remaining status codes indicate that a user could not be created: "400 Bad Request"indicates a syntax error; "409 Conflict"indicates that the user already exists; and "500 Internal Server Error"indicates a server error during request processing.

In addition to user registration, one of the most crucial aspects of user management is authentication, which is also supported by this API. The authentication endpoint spends the majority of its time securing the password to avoid storing it in plain text. The first step is to concatenate a Pepper [61] to the received password, which is a secret random value stored on the server but not in the database. The concatenated password is then hashed using the Argon2 [62] algorithm, which converts the data into a completely different, unrecognizable set of characters. If the hash matches the hash stored in the database, the authentication is successful and the status code "200 OK"is returned; otherwise, the status code "403 Forbidden"is returned instead.

Table 12: Users microservice API specification

HTTP Method	**URI**	**Description**	**Status code**
GET	/v1/users	Retrieve the username of all registered users	- 200 OK
POST	/v1/users/ authenticate	Send authentication parameters and check the database for a matching user	- 200 OK - 400 Bad Request - 403 Forbidden - 404 Not Found
POST	/v1/users/ register	Create a new user	- 201 Created - 400 Bad Request - 409 Conflict - 500 Internal Server Error
GET	/v1/users/ {username}	Retrieve a user's information	- 200 OK - 404 Not Found
DELETE	/v1/users/ {username}	Delete a specific user	- 200 OK - 204 No Content - 500 Internal Server Error

3.5 Database

In the previous version of the cloud, there was a single database for all data. However, since the new architecture divides the cloud into two segments, the database is also divided. The database in use is the NoSQL database MongoDB [63]. As an alternative to relational databases, it employs a non-relational, document-oriented data model and a non-structured query language that is quite useful for working with large sets of distributed data. Since it stores data in JSON-like documents, it works well with REST APIs that also store their data in the same format. To avoid creating interdependencies and thus violating one of the microservices architecture principles, only the microservices interact directly with the database, and a single database collection can only be accessed by a single microservice.

Before delving deeper into each database and its collections, it is necessary to explain several parameters found in multiple collections, including ”p”, ”board id”, ”board timestamp”, and ”system timestamp.”Since transmission packages can be sent in multiple packages, the ”p”arrays contain the current package’s number and the total number of packages. Universally Unique Identifier (UUID) used for device identification is ”board id.”Both the board and system timestamps are in Unix epoch format. The board timestamp corresponds to the moment the board transmits, while the system timestamp corresponds to the moment the server receives the message. The remaining parameters are each collection-specific.

3.5.1 Data

The data segment database stores all the data necessary for the correct processing of command responses, which requires four database collections, which are described below: Configurations, Commands, Deltas, and Firmwares. In Figure 28, the fields of collections and their respective data types are illustrated.

- **Configurations** - stores the latest configuration for each endpoint device. These configurations include: the alarms, which indicates the time interval between regular transmissions or the alarms throughout the day at which it will transmit; the modem configuration with the cell id, tac, and tau of the modem; the sensors variables configuration for each available sensor such as temperature (sTMP), and humidity (sHUM); and the ota object, which specifies the addresses of application 1 and application 2 since the device memory is partitioned into two sections. In addition, it contains the current index of the over-the-air update package, as updates are processed one package at a time, as well as the total flash memory size of the device.

- **Commands** - stores the commands awaiting transmission to the target device. Each command document contains the command opcode, the microservice to respond to, the MD5 hash status, the command datagram to be transmitted to the device, and the command description. When a non-persistent command is processed, its entry in this collection is deleted.

- **Deltas** - stores the deltas used for FOTA updates. The document's packages range from 1 to the total number of packages specified in the "total packages"field. If a delta is designed as true, it indicates that it was generated from the difference between two versions, and as a result, its size is smaller; otherwise, it will include all packages pertinent to the new version, and thus is not a true delta.

- **Firmwares** - stores the firmwares used to produce deltas. Each firmware includes a version string and objects for each microcontroller unit that supports that version. Each version of microcontroller unit stores two objects, one for each application address, with binary and Intel hexadecimal data formats in each object.

3.5.2 Application

The database for the application segment stores data sent from devices as well as other data that may be relevant to applications, such as user data. This database includes the Configurations, Configuration History, Command Results, Regulars, and Emergencies collections. The listing that follows describes the collections in greater detail, and Figure 29 shows the fields and their respective data types.

- **Configurations** - stores the most up-to-date configuration for each endpoint. This collection operates identically to the Configurations collection described in the preceding section; the only difference is the addition of a new field, "available_sw_versions,"which stores the results of the FOTA versions command. This field contains an array of strings containing all possible firmware versions for the device, whereas "sw_version"specifies the current version.

- **Configuration History** - stores all configuration messages. This collection employs the exact same structure as the Configurations collection, with the exception that its data is never updated or removed, as a new document is inserted whenever a new configuration message arrives.

- **Command Results** - stores the command result messages. This collection, like the Commands collection from the previous section, stores the command opcode, but it also includes a description

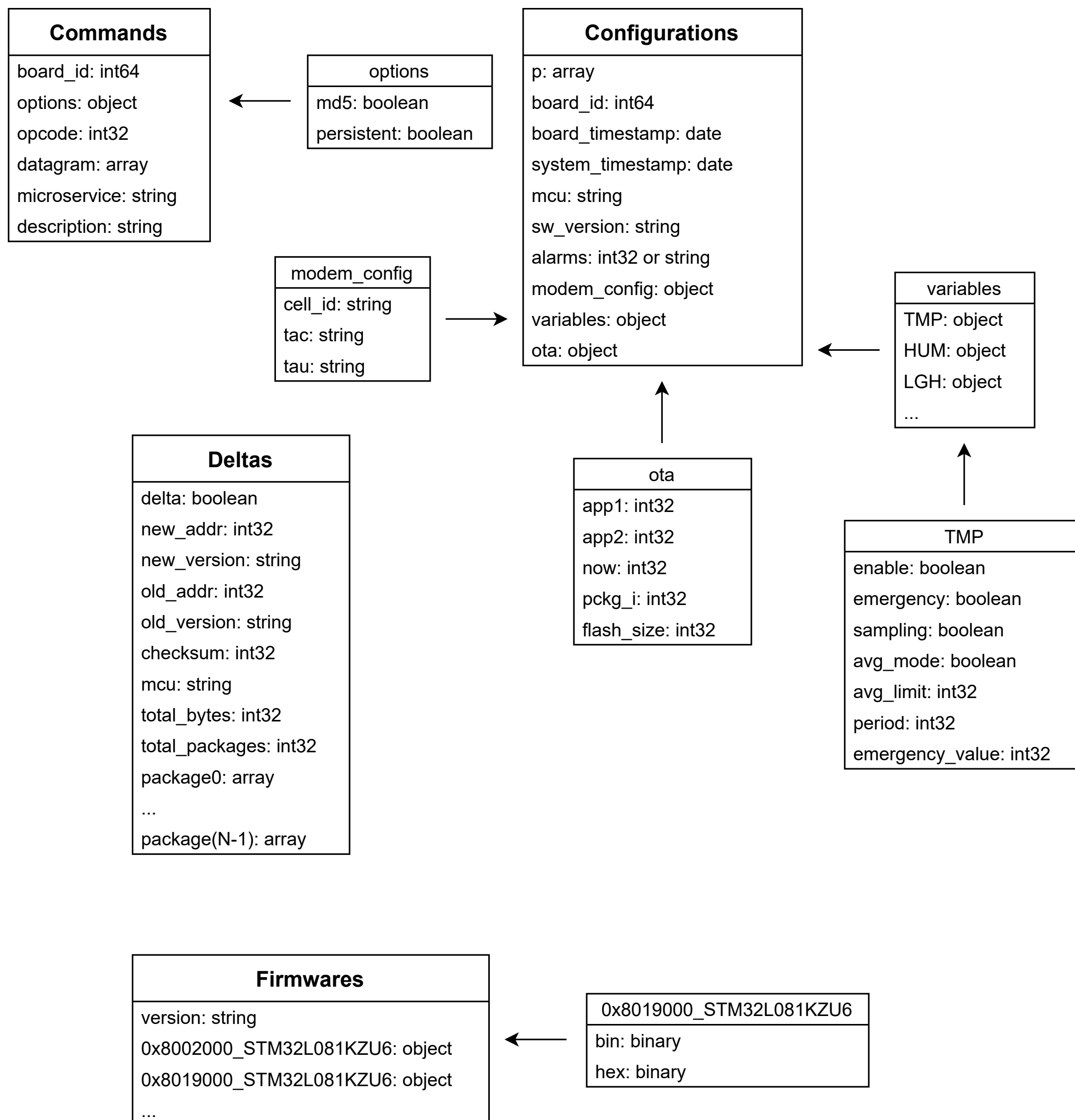


Figure 28: Data segment database collections

of the command opcode to provide context for applications. The command result is an object with boolean values for set and get (a successful command displays both as true). There is an undefined field at the end that varies based on the command type, such as variables for a sensor variables configuration command.

- **Regulars** - stores regular messages. Each message includes the common fields, as well as samples of the sensors and modem signal quality. The signal quality and samples fields in each message are not guaranteed.
- **Emergencies** - stores emergency messages. Each emergency message contains the emergency flags that were triggered and all the sensor variable samples that were captured at the time of the

emergency. However, not all variables samples may warrant an emergency, only those specified in the flags array.

- **Users** - stores relevant user data. There exists a unique document with a distinct username for each registered user. Other user data, including name and email, is also stored. The password field contains a hashed string that cannot be easily reversed to the original password for security purposes.

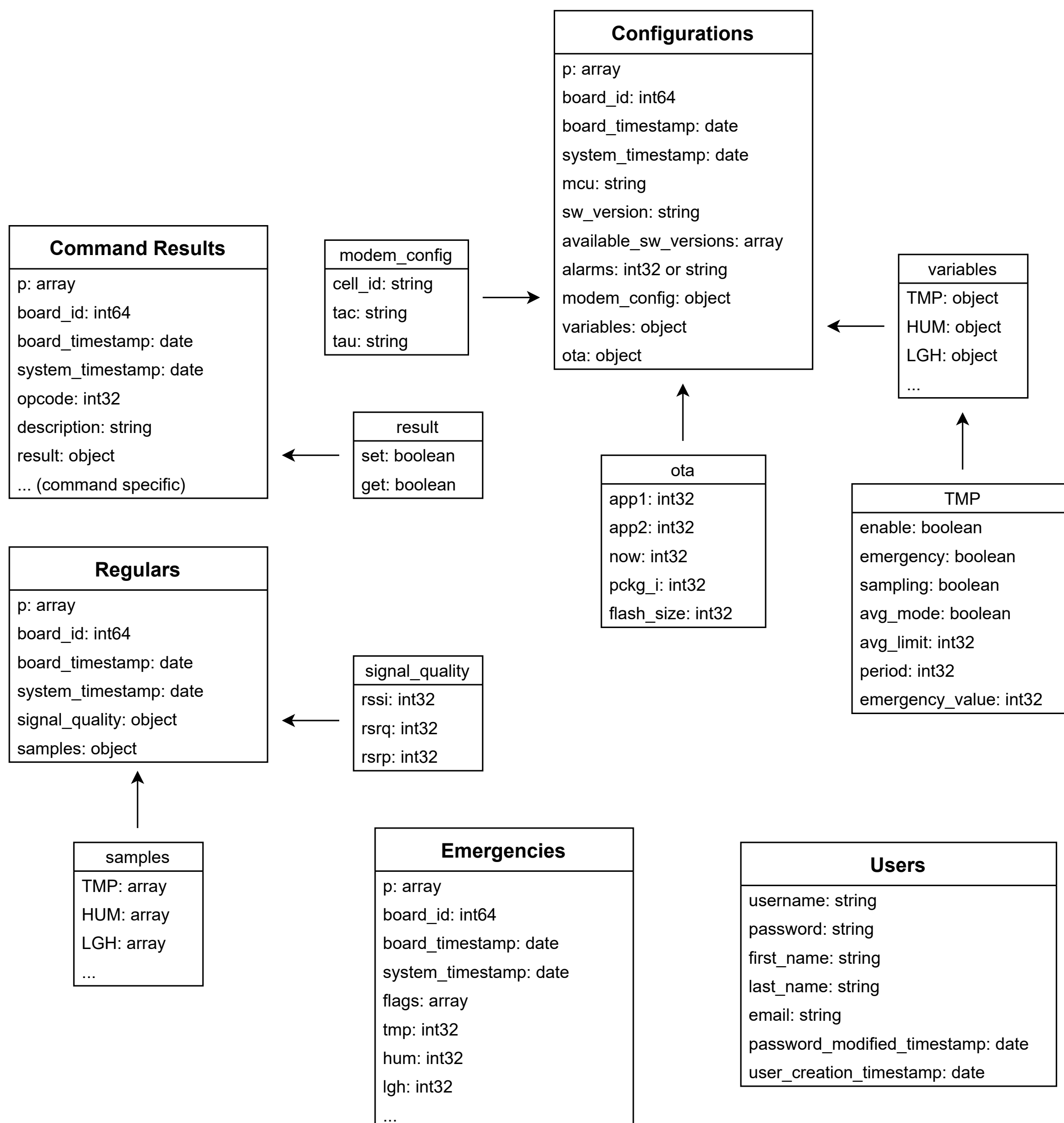


Figure 29: Application segment database collections

3.6 Dashboard

The dashboard is the only application available to interact with the end-user in the Link4S project and is part of the application segment. This section aims to provide a dashboard design that incorporates the good design practices and market research presented in the preceding chapter, as well as develop any additional software required to achieve the dashboard's purpose.

3.6.1 Functional and visual features

As described in the previous chapter, a dashboard contains features that can be categorized as either functional or visual in order to achieve its objectives. The following features were carefully considered to ensure proper alignment with the dashboard's purpose and to enhance the end-user decision-making process.

3.6.1.1 Functional features

- Real-time notifications
- Graph and tabular format
- Show/hide graphs
- Global graph filters
- Time interval widget

3.6.1.2 Visual features

- Multipage
- Frugal use of colors
- High data-ink ratio
- Grid lines for graphs

3.6.2 Framework choice

The preceding chapter introduced several dashboard development frameworks, each with their own advantages and disadvantages. However, only one of these frameworks is required for this project. Identifying the dashboard type according to the taxonomy established in Figure 12 will assist in selecting the most appropriate framework.

Since the dashboard is intended for an organization and requires user-interaction views, it appears to fit best within the Dashboards for Decision-Making. Within these dashboards, there are two types: Strategic Decision-Making and Operational Decision-Making. However, according to the design choices in Table 5, these types of dashboards serve different purposes. In light of this, the dashboard can be categorized under Operational Decision-Making.

Among the examples examined in Table 6, the Node-RED platform is the only one that corresponds to the Operational Decision-Making dashboard category. The Node-RED platform could be an appropriate option for developing a dashboard, and it is adequate for a project that has already implemented the majority of its infrastructure and is seeking to supplement it with a dashboard application. By examining Table 7, it is possible to see that it does not offer the greatest design flexibility, and that there is a framework, Dash, that offers the same capabilities as Node-RED. The Dash framework, despite requiring more development effort and time, provides all the necessary features and allows development in Python, which is the language already in use for the remainder of the Cloud. Due to its stateless design, Dash is also fault tolerant and extremely scalable, making it the most suitable framework.

As shown in Figure 30, the Dash framework consists of a variety of individual components. It employs Flask [64], a well-known Python microframework, for the backend. React [65] is used to manage all components because a Dash application is rendered as a single-page React application. This compatibility with React enables the Dash feature set to be expanded by importing React components that were originally designed to function in JavaScript. It uses Plotly [66], the most widely supported package for data visualization, to generate charts, although this is optional. Plotly will be used in the dashboard whenever graphing is required.

Figure 30: Dash framework components

3.6.3 Architecture overview

The architecture overview illustrated in Figure 18 has already established the connections between the dashboard and other entities. The focus is now on understanding how the selected Dash framework for the dashboard can interact with the other entities to achieve its goals and provide the best user experience.

3.6.3.1 Design pattern

The design pattern for the dashboard application is the MVC (Model View Controller) [67] design pattern, which is commonly used to implement user interfaces, data, and controlling logic. It provides a clear separation of data, business logic, and the user interface at its core. Following is a description of the MVC design pattern's three components:

- **Model** - manages data and business logic
- **View** - handles layout and display
- **Controller** - interprets inputs and commands the model and view parts accordingly

In the case of the Dash framework, the MVC pattern can be contained entirely within the framework, eliminating the need to implement multiple frontend and backend technologies. The MVC pattern is depicted in Figure 31, which also contains all the entities that interact with the dashboard.

In Figure 31, it is evident that the User initiates the flow of information the majority of the time, as it interacts with the dashboard and provides inputs to the Controller. In the case of a dashboard constructed using the Dash framework, the controller is a callback, which are functions that are automatically invoked whenever the property of an input component changes [68]. The controllers can also receive input from other sources, such as the Change Stream, which is responsible for real-time database change detection and sending the relevant data to the controller.

Although the information flow dictates that the Controller must interact with the model, there are times when it can interact directly with the view, such as when the input relates to an action that does not require data manipulation. If not, the controller will manipulate the model to obtain data from the microservices or Redis cache. After processing the data, the View is updated and the User is presented with an interactive layout.

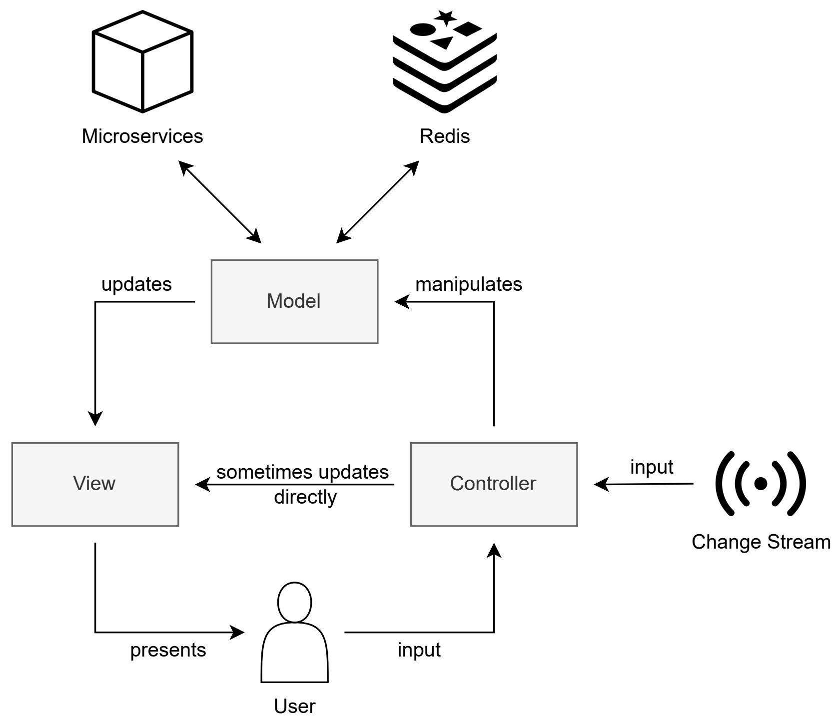


Figure 31: Dashboard MVC diagram

3.6.3.2 User flow

The user interaction is one of the most important aspects of the design of a user interface, such as a dashboard. As depicted in Figure 32, a user will follow certain paths to reach the desired information when interacting with an application. The paths contain only information pertinent from the user's perspective; the interaction between internal systems, such as microservices and databases, is not included.

As shown in Figure 32, the user begins by interacting with the login page and is then redirected to the overview page, which serves as the dashboard's homepage. The user has access to a sidebar containing multiple links, one for each page, to which he is redirected when he clicks. The user may also click the logout button in the sidebar to return to the login page.

3.6.4 Design prototype

To properly develop a dashboard application, a design prototype that incorporates all previously defined requirements and features must be created as a reference for the implementation. The following page prototypes were all developed using the UI design and prototyping tool UXPin [69]. The prototypes also make use of color, and the color scheme depicted in Figure 33 serves as a reference for the implementation. The colors were chosen based on a darker theme that improves visibility and does not strain the eyes of

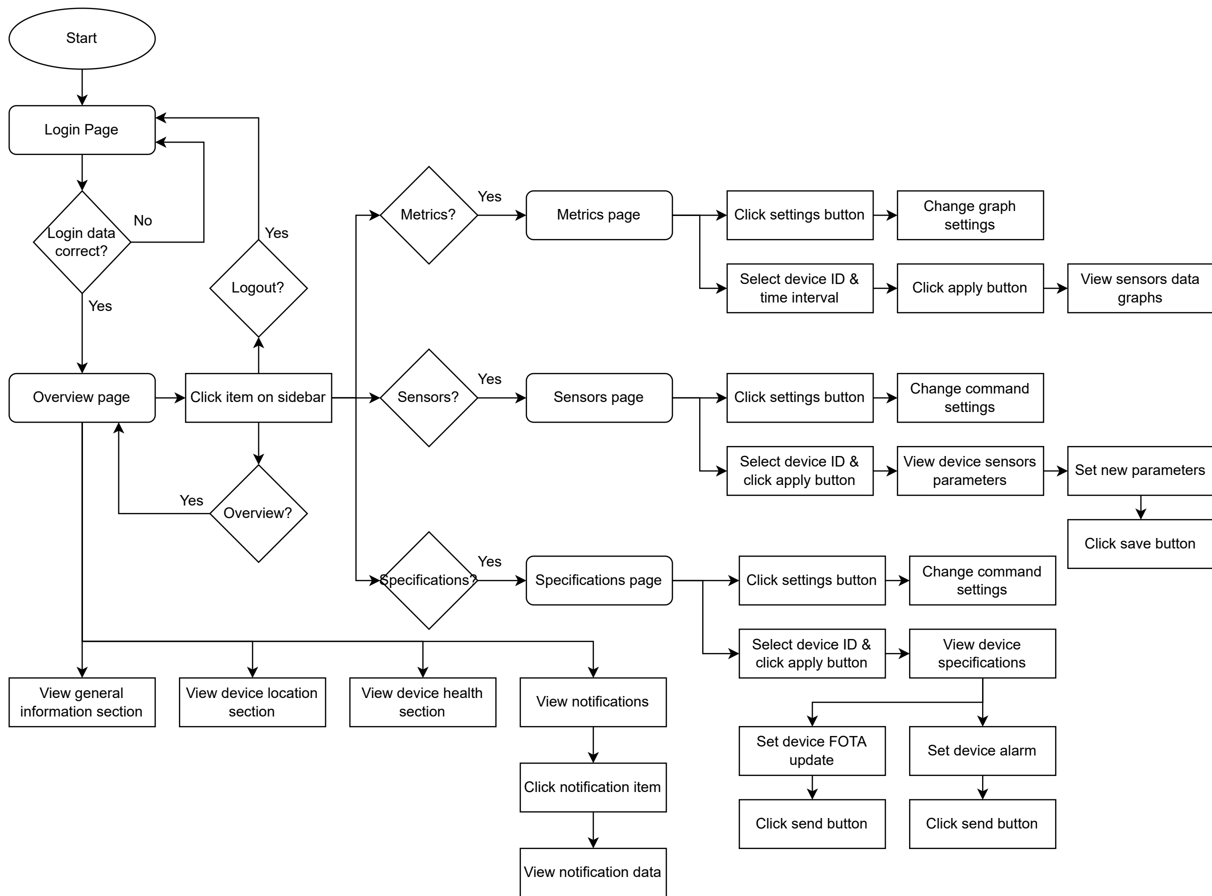


Figure 32: Dashboard user flow

the end-user.

Figure 33: Dashboard color palette

3.6.4.1 Login

The initial page with which the user interacts is the login page, which enables user authentication. This page's design is extremely straightforward, as its sole purpose is to receive two text parameters, the username and password, and to provide a login button. The primary layout is centered on the display, and the logo is present. The dashboard does not provide a method for user registration because this is outside of

its scope and is handled by the organization. As a security measure, the text entered into the password text box will be unreadable.

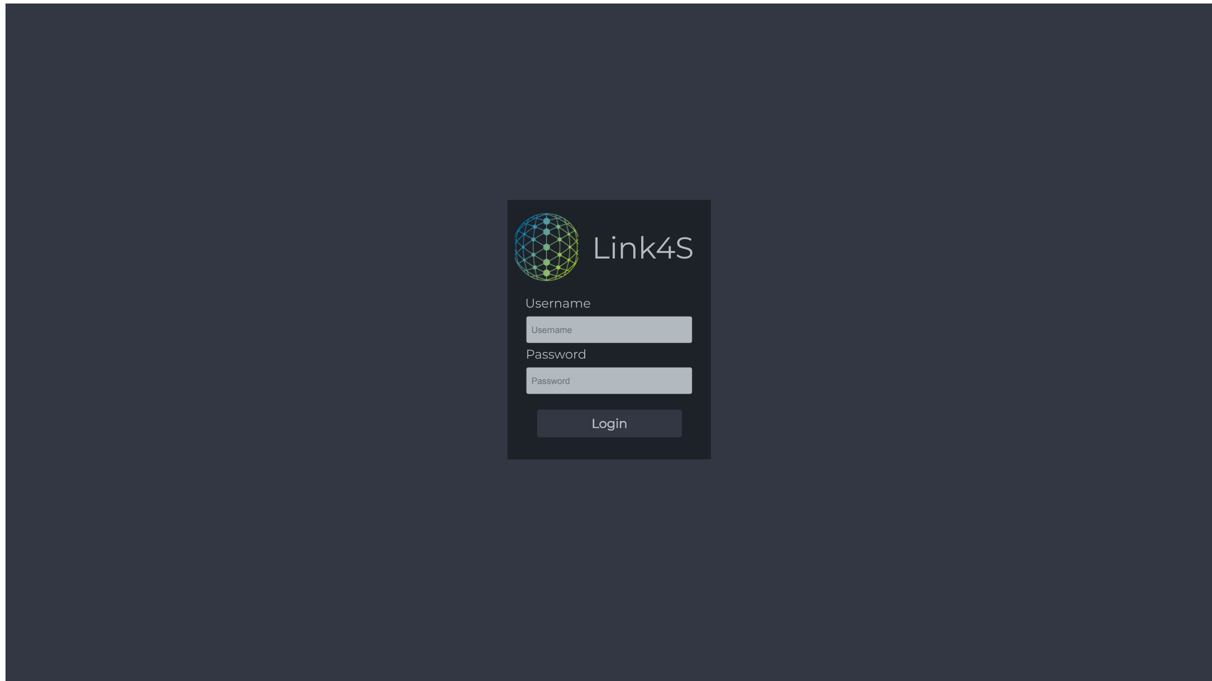


Figure 34: Login page design prototype

3.6.4.2 Overview

After logging in, the user is redirected to the overview page, which serves as the dashboard's homepage. This page is divided into the following four sections: general, device location, device state, and notifications, which are explained below:

- **General** - displays a global overview of the current state of the devices, beginning with a circular gauge indicating the fraction of active devices. The remaining four rectangular items indicate the number of devices in each of the following states: online, faulty (did not communicate in the expected interval plus a grace period), and inactive (did not communicate in the expected interval plus 24 hours), as well as the number of unresolved emergencies.

- **Device location** - displays the current location of all devices that provide location information. Each device is assigned a single pin that, when clicked, provides additional information.

- **Device state** - displays a list of items containing the current state of all devices. Each item displays the device's activity state, signal strength, and battery life.

- **Notifications** - displays a list of items containing three types of notifications: configuration, command result, and emergency. The user can click on these items to view additional information about the specific notification.

Some of the information between the sections is repeated on purpose, as this allows the end-user to obtain the relevant information at multiple levels, as the user may sometimes want to see only how many devices are online without delving further into the data.

Additionally, it is one of the pages that receives real-time updates, so it can be used to monitor devices without the need to frequently refresh the page. Real-time updates are available for all the sections in the page.

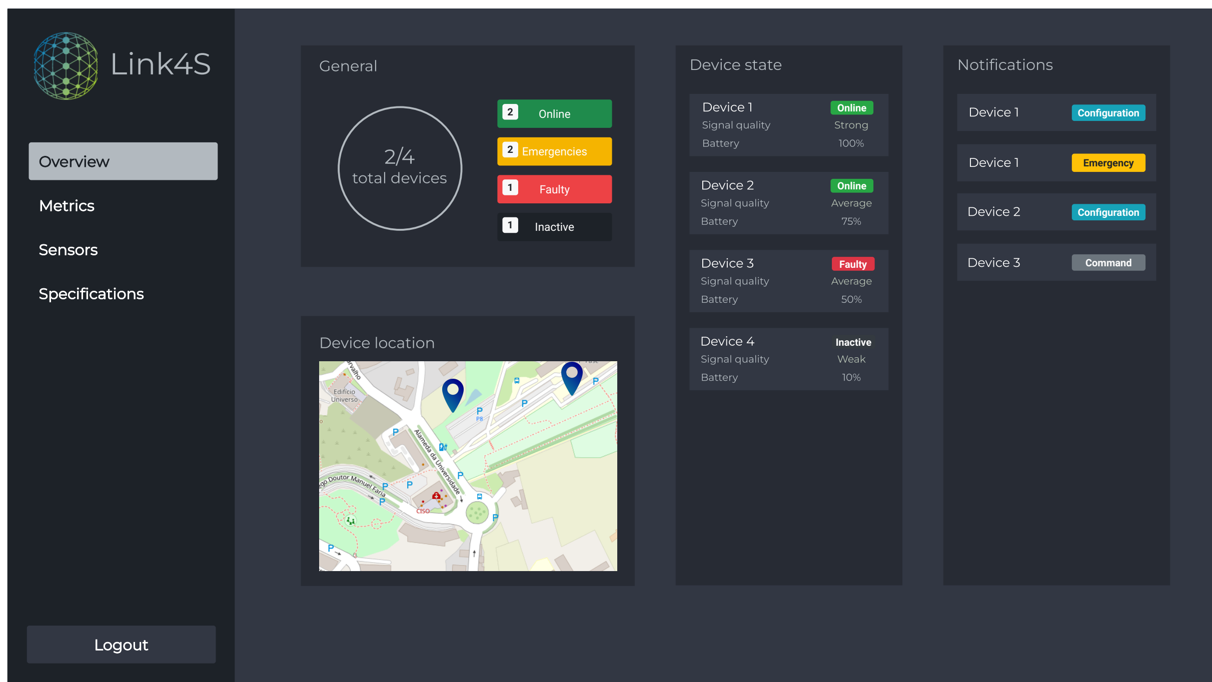


Figure 35: Overview page design prototype

3.6.4.3 Metrics

The metrics page enables the end-user to interact with the sensor data transmitted by the devices. The page includes views for selecting the desired device and the sensor data time interval. When the "Apply"button is clicked, multiple graphs for all sensors with available data are displayed.

Each sensor variable view includes two gauges, one for the most recent value and the other for the average or mean value, depending on the dashboard configuration. While the average or mean will only have an effect on the values from the selected time interval, the latest value is independent of the time

interval and will always be displayed if there is data for the sensor variable in question; it can also change its value through real-time updates. In the instances where there is no data available for the selected time interval, but there is data available outside the time interval, only the latest value gauge will be displayed, without the graph or average/mean gauges. The primary graph displays points on a time scale, always within the selected time interval.

A "Settings"button is also present, which, when clicked, reveals another small window. The settings are used to modify the appearance of the graphs, such as the plot type (lines, markers, or lines and markers) and to enable/disable sensor views and the type of data to display (regular messages or regular messages and emergencies). If multiple sensor variables are available, the graph will display multiple lines and adjust its scale accordingly.

Figure 36: Metrics page design prototype

3.6.4.4 Sensors

The sensors page enables the end-user to interact with the devices, specifically to modify the configurations of device sensor variables. To accomplish this, the user must first select a device from the drop-down menu; then, after clicking the "Apply"button, all of the device's current sensor configurations appear in separate containers.

Each sensor variable container contains switches and input components that correspond to the fields available in the database, as seen in the previous section. The user is free to modify these parameters, and only after clicking the "Save"button are the changes transmitted to the device.

This page also contains a "Settings"button, which contains the settings for the command that is sent whenever a save occurs. These settings include the MD5 hash status, the command's persistence in the database, and the parsing microservice used to consume the command.

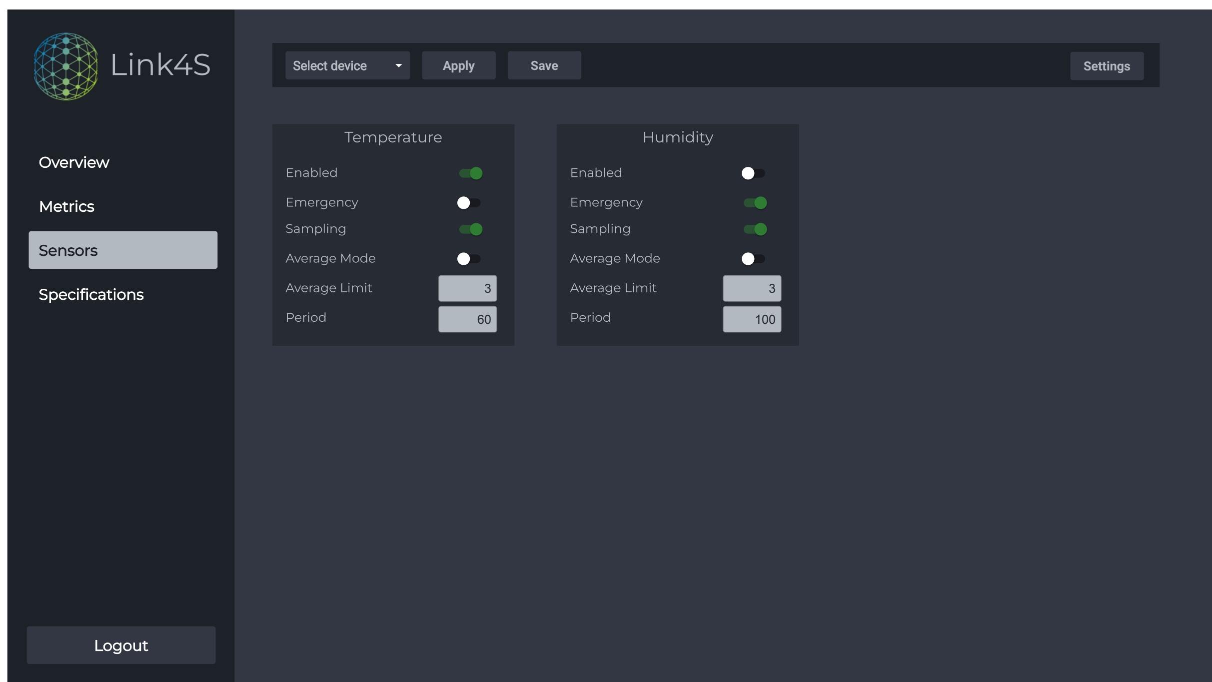


Figure 37: Sensors page design prototype

3.6.4.5 Specifications

The last page of the dashboard, the specifications page provides an overview of the selected device's settings as well as useful views for frequently used commands. Similar to the other pages, the user must select a device from the drop-down menu and then click "Apply."Then, the most pertinent settings for the device are displayed, which correspond to the fields in the configuration collection that were displayed in Figure 29.

Along with the device settings, two additional containers are provided: one for changing the device alarm, which provides the user with containers for the hours, minutes, and seconds to make it easier to select a time value, and another for FOTA updates, which includes a delta switch for determining whether a delta should be used.

Similar to the sensors page, there is a ”Settings”button with the same settings: the MD5 hash status, the command’s persistence in the database, and the parsing microservice used to consume the command.

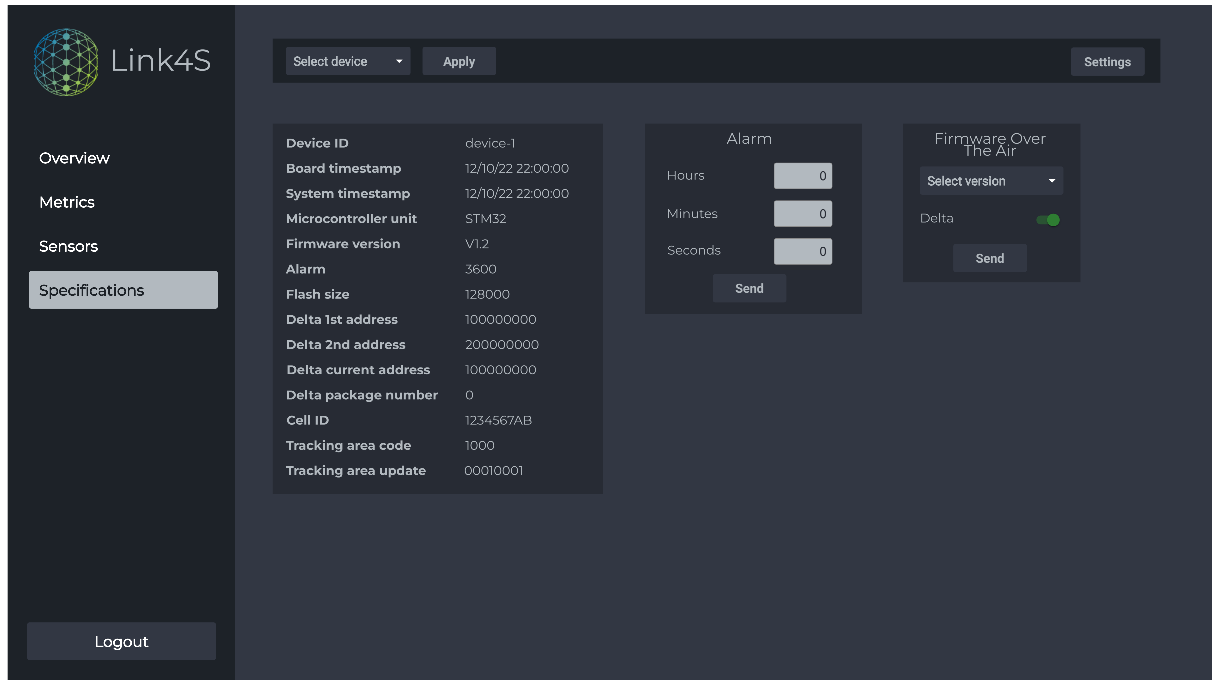


Figure 38: Specifications page design prototype

3.6.5 Real-time updates

The Dash framework requires assistance when real-time data is required. The framework attempts to address this issue by providing components for live updates that enable updates at a predefined interval [70]. When this interval is set to a small value, the behavior is similar to receiving updates in real-time; however, using this approach, which is similar to polling, would present a larger problem, as thousands of requests would be made to the microservices to check if new data was available, resulting in an unreasonable increase in network load.

Change Streams [71] is a feature of MongoDB that provides a better solution. This feature enables applications to subscribe to all data changes on a single or multiple collections of a database and react to those changes immediately. Change Streams also enable the data to be filtered using the same aggregation framework as standard MongoDB database queries, thereby enhancing the overall efficiency of the process.

Change Streams is only available for replica sets, which is one of its few restrictions. A replica set in MongoDB is a collection of processes that interact to ensure the consistency of a data set. Because they offer both redundancy and a high level of availability, these sets are frequently used in production

systems. In this instance, the replica set is only used for Change Streams, so only one replica is required. As depicted in Figure 39, the primary node receives all write operations and logs all modifications to its data sets in its operation log, or oplog. The secondaries replicate the primary's oplog and apply the operations asynchronously to their respective data sets. In addition, a heartbeat signal, also known as a ping, is exchanged every few seconds to ensure that all sets are operational.

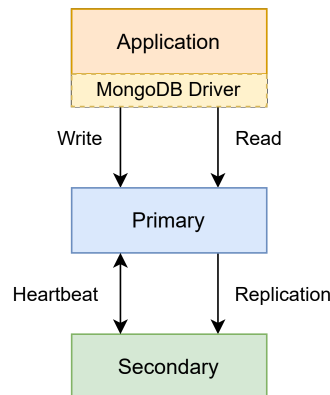


Figure 39: MongoDB replica set diagram

The application flowchart for Change Streams is depicted in Figure 40. The first step in building this application is to create the MongoDB client that will communicate with the database. The web application responsible for communicating with clients via WebSockets is then developed. WebSockets are used for the connections because they offer persistent connections and are event-driven, a requirement for true real-time.

The changes handler, which is responsible for handling the clients and the messages sent to them, is created after the web application has been created and assigned to it. The changes handler will be invoked whenever an update occurs. After assigning the handler, the port where the application will listen for clients is specified and the database collection callbacks are added.

The final action is to initiate the execution loop. After being initiated, the execution loop will jump to the defined callback method whenever a database update occurs on database collections with assigned callbacks. The callback method will then invoke the changes handler to process the data and send it to connected clients.

3.6.6 Time series cache

The time series cache provides a caching solution for the dashboard to store sensor data in a local cache which reduces the need to make frequent API calls for the same chunks of data. The creation of such

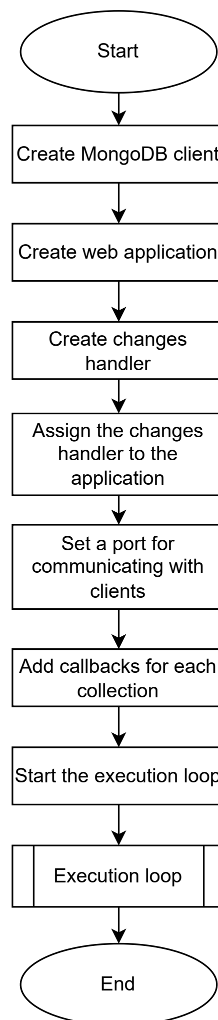


Figure 40: Change Streams flowchart

a cache is challenging due to the fact that time series data can span multiple timeframes and contain millions of samples, making classic caches inefficient. Popular caches such as Flask-Caching [72] use memoization techniques in which the inputs of a function are used to create a cache key and the function results are stored as cache values. When dealing with time series data, the inputs include the start and end dates; therefore, the number of possible variations for the inputs is too high, and the cache would consume an excessive amount of memory with a very low percentage of cache hits.

To address this issue, a more flexible approach is required, in which data collected and stored for specific time periods can be reused. A viable solution is to divide the time series data into blocks and use an approach similar to page caching in operating systems that read data from disk, where they read and store entire chunks of data in cache pages as opposed to reading individual bytes [73]. To apply this method to the sensor data, the time series data for each sensor variable can be divided into buckets with

the same period, such as one week, as depicted in Figure 41. When a request is made to the cache, all buckets that fall within the start and end dates are returned, and if a bucket is not available in cache, a request is sent upstream to the Devices microservice to retrieve the missing data.

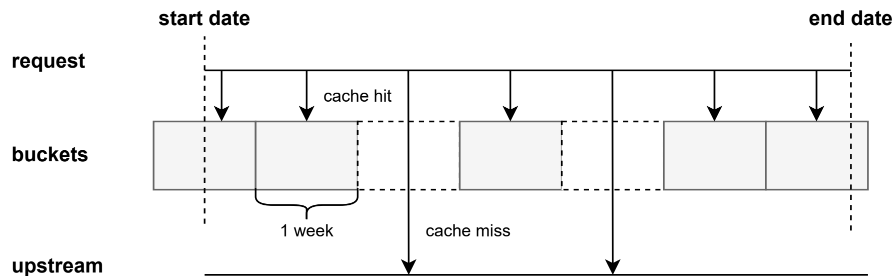


Figure 41: Time series cache diagram [74]

The caching strategy, which describes how data will be loaded into the cache, is another aspect of the cache that requires analysis. There are numerous caching strategies, but for simplicity's sake, the lazy loading strategy will be employed.

Lazy loading is a caching strategy that loads data into the cache only when it is requested, i.e. only when there is a cache miss [75]. Thus, it can be retrieved more quickly the next time it is requested. It is also a good way to prevent the cache from becoming bloated with unnecessary data. This strategy can also be enhanced by adding a TTL (Time to live), which sets an expiration date for cache keys and prevents data from becoming stale.

Redis [76], an open source, in-memory data store commonly used for caching purposes and containing all the necessary features for the metrics cache, will be used as the cache client.

The flowchart in Figure 42 depicts the retrieval and transmission of data to the cache. It begins by creating weekly buckets with the start and end dates the user provided. It's important to note that the buckets will always be aligned to the same day, and it doesn't matter if they span a longer period of time than the specified dates. A cache key is then generated for each bucket containing the bucket's start and end dates, as well as the ID of the device and sensor variable. Then, the keys are utilized to retrieve the cached results, and a loop is initiated to determine if each cached key result contains sample values. If there are no available values for a given key, a request is sent upstream for data, which is then cached with a TTL. After all results have been processed, samples that fall outside of the user-specified start and end dates are removed and the remaining results returned.

Moreover, it can be beneficial to populate the cache in the background with frequently accessed metrics. For this purpose, a task can be scheduled to run sporadically in order to send data to the cache,

thereby enhancing the dashboard user experience.

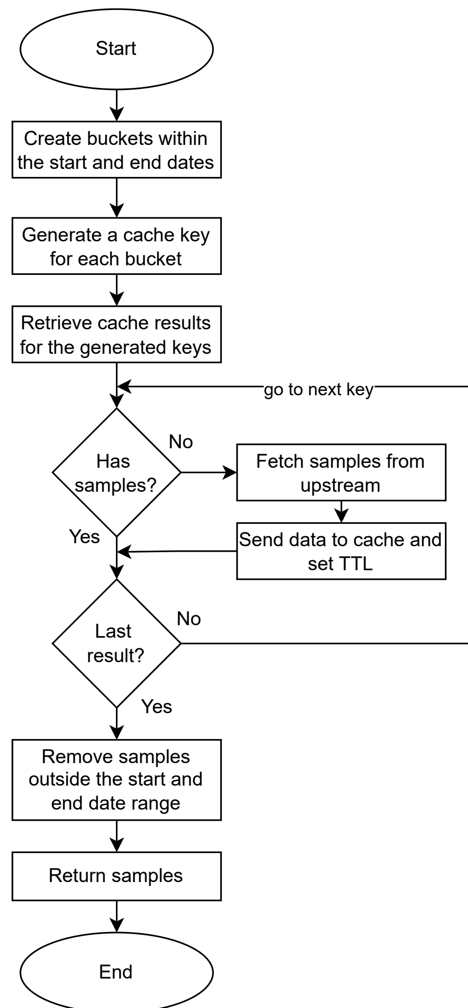


Figure 42: Time series cache flowchart

Chapter 4

Implementation

The implementation phase carries out the previously established design. This chapter details the implementation of the cloud architecture, from the microservices to the dashboard, although not all code is documented due to the large codebase size.

4.1 Programming languages and tools

The main programming language used for the cloud was Python. The cloud components, such as the dashboard, utilized a Python version of the Dash framework, while the microservices utilized either the Blackwing or Falcon web frameworks, minimalist frameworks for building REST APIs and microservices. In addition, CSS and JavaScript were used to improve the functionality of the dashboard.

The Black code formatter, which automatically styles the code in a PEP-8 compliant format, was used to keep the Python code more organized and consistent across the multiple files. In addition to Black, flake8 was used as a linter that analyzed warnings, PEP-8 violations, and code complexity.

Visual Studio Code is an open-source code editor created by Microsoft that includes support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git. The entire coding process was performed in this editor.

The official MongoDB Compass tool was used to interact with the MongoDB database, enabling the analysis of dashboard data and simplifying the testing process.

4.2 Microservices

The implementation of the data parsing microservices only reflects the newly added code, as the majority of the code was already implemented and documented in [8]. Everything had to be implemented from scratch for the remaining microservices, but not all code is displayed because in some cases it is irrelevant or repetitive.

For the Process Data microservice, Listing 1 demonstrates the creation of a REST API. The microservice's primary method, *create_api*, begins by creating a Falcon app object and then adds the endpoints to the app object. Each endpoint is assigned a route in the form of a string and an object of the resource class ProcessData for handling requests. The suffix assists in identifying the resource class's methods.

```
1  def create_api(log=True):
2      ...
3      db.init((db_config["db_url"], db_config["db_port"]), db_config["db_name"])
4  
5      resource = ProcessData()
6  
7      _app = falcon.App()
8      _app.add_route("/v1/regular", resource, suffix="regular")
9      _app.add_route("/v1/emergency", resource, suffix="emergency")
10     _app.add_route("/v1/configuration", resource, suffix="configuration")
11     _app.add_route("/v1/command", resource, suffix="command")
12 
13     return _app
14 
15 app = application = create_api()
```

Listing 1: Process data microservice API creation

The resource class is the primary difference for the other microservices that implement a REST API, namely the devices and users microservices. Every API endpoint was implemented according to the specification.

4.2.1 Data parsing

When a message is received by the Blackwing server, it is routed to one of the four available parsing microservices: Regular, Emergency, Configuration, and Command. Each of these microservices includes a handler derived from BlackwingHandler. This handler, whose code is displayed in Listing 2, is invoked to process the message. It begins by loading the configuration and initiating the NB-IoT database, then calls the appropriate parsing function and API post function to send the parsed data to the Process Data microservice.

Then, a condition is evaluated to determine if the current package corresponds to the previous package. If this condition is met, a GET request is sent to the Process Data microservice, specifically the commands endpoint, along with the board ID and microservice identifier. If the response code is 200, the request was

successful and the data is encoded and assigned to the response; otherwise, a null value is assigned to the response.

```
class ParseRegularHandler(BlackWingHandler):
    def attendRequest(self):
        try:
            self.config = yaml.load(os.path.join(dirname, "../../config.yaml"))
            db.init((self.config["db_url"], self.config["db_port"]), "NB-IoT")
            payload_dict = regular.parse(self.letter)
            regular._api_post(payload_dict)
        except Exception as e:
            print("Error parsing regular.. ", e)
            print("Received: ", self.letter)

        try:
            self.response = None
            if payload_dict["p"][0] == payload_dict["p"][1]:
                resp = requests.get(
                    self.COMMAND_API_URL,
                    params={"board-id": regular.board_id, "microservice": "regulars"},
                )
                if resp.status_code == 200:
                    data = resp.json()
                    self.response = data.encode("latin-1")

        except Exception as e:
            print(f"Error calling response function.. {e}")
            self.response = None
```

Listing 2: Blackwing parsing microservice handler

One of the few differences between this handler and its previous implementation in [8] is the addition of the API post method, illustrated in Listing 3, which is responsible for sending data to the Process Data microservice using the appropriate HTTP method. It also performs a check to determine if a FOTA update is available, as this also requires a configuration update. This method is similar for all four parsing microservices, despite not being shown.

```
class Regular:
    collection = "Regulars"
    board_id = None
    ota_update = False
    headers = {"Content-Type": "application/json", "Accept": "application/json"}

    ...
    @classmethod
    def _api_post(cls, payload_dict):
        data = payload_dict

        if "_id" in data:
            del data["_id"]

        if cls.ota_update:
            requests.post(
                f"{common.API_URL}/configuration",
                data=json.dumps(data),
                headers=cls.headers,
            )

        requests.post(
            f"{common.API_URL}/regular",
            data=json.dumps(data),
            headers=cls.headers,
        )
```

Listing 3: Parsing microservice API post method

4.2.2 Data processing

The Process Data microservice utilizes a REST API to communicate with the parsing microservices and the Python libraries for the Device Interface to communicate with the Mediator.

The resource class is ProcessData, as described in in Listing 4. This class implements all API methods as functions that are invoked when a request is received. There are four similar POST methods, but only the method *on_post_regular* for regular messages is displayed as the others are very similar. Each of these methods receives data from the microservice performing the parsing, converts it to JSON format, and sends it to the Mediator. The response status of the POST method will depend on whether the Mediator received the messages successfully.

To retrieve commands from the database, the *on_get_command* method is implemented, which receives the parameters sent by the parsing microservices: board ID and microservice identification, before calling the response function, which was ported from the commands module in [8]. If there are no commands in the database, the 204 (No Content) HTTP status code is returned; otherwise, the 200 (OK) code along with the command data is returned.

```
1  class ProcessData:
2     def on_post_regular(self, req, resp):
3        logging.log(logging.INFO, req)
4
5        data = req.media
6
7        output = {"type": "regular", "content": data}
8
9        output_bytes = json.dumps(output).encode("utf-8")
10       status = orch_api.sendData(str(data["board_id"]), output_bytes)
11
12       if status:
13          resp.status = falcon.HTTP_200
14       else:
15          resp.status = falcon.HTTP_500
16    ...
17    def on_get_command(self, req, resp):
18       logging.log(logging.INFO, str(req))
19
20       board_id = req.get_param_as_int("board-id", required=True)
21       microservice = req.get_param("microservice", required=True)
22
23       data = response(board_id, microservice)
24
25       if data is None:
26          resp.status = falcon.HTTP_204
27          return
28
29       resp.text = json.dumps(data.decode("latin-1"), ensure_ascii=False)
30
31       resp.status = falcon.HTTP_200
```

Listing 4: Process data microservice API methods

In addition to the REST API, this microservice implements a command listener to receive commands

from the Mediator, which were originally sent by an application, such as the dashboard. The code in Listing 5 shows the function *onSendCommand* that overrides the corresponding method on the *SendCommandListener* class. This function receives the device ID, the command data in bytes, and the context to be returned with the command response.

The first example is the FOTA update command, which uses the commands module to process the command data and store it in the database. As the command can only be executed later, the command's result is irrelevant, so the response returns the same data that was received. The other command is a FOTA versions command, which is cloud-only, meaning it does not reach a device. This command queries a database for the various available firmware versions and returns this information to the Mediator.

```
class _SendCommandListener(SendCommandListener):
   def onSendCommand(self, device_id: str, cmd: bytes, context: bytes) -> None:
      command = json.loads(cmd.decode("utf-8"))

      logging.log(
         logging.INFO,
         f"Command received from orchestrator: {device_id=}, {command=}",
      )

      commands = Commands(
         db,
         md5=command["md5"],
         persistent=command["persistance"],
         microservice=command["consumer"],
      )

      match command["type"]:
         ...
         case "fota":
            commands.setOTAUpdate(
               int(device_id),
               command["fota"]["version"],
               command["fota"]["delta_mode"],
            )
            orch_api.sendCommandResult(device_id, cmd, context)
         case "fota_versions":
            result = db.distinct("Firmwares", "version", {"version": {"$ne": None}})

            output_dict = {
               "board_id": int(device_id),
               "available_sw_versions": sorted(result),
            }

            output_bytes = json.dumps(output_dict).encode("utf-8")
            orch_api.sendCommandResult(device_id, output_bytes, context)
```

Listing 5: Process data microservice command listener

4.2.3 Devices

The devices microservice communicates with the Mediator and provides the sole interface through which applications can interact with devices.

The majority of implemented API endpoints are GET methods, which are primarily used to retrieve data from the database. The *on_get_dev_samples* shown in Listing 6 is particularly relevant, as it is used to

retrieve sensor data. This endpoint obtains the device ID from the URI, which must be an integer. Other parameters, such as the sample ID and the timestamps used to create the sample's time interval, are also provided in the query. The only required parameter among these is the sample ID. The timestamps are optional because the database search can go as far forward or backward in time as needed, as long as there is available data. If all parameters are specified correctly, the search will return a 200 (OK) status code even if no samples are located.

The implementation of the POST methods is also very similar, and they are all used to receive commands from the dashboard and send them to the Mediator so they can reach the devices. The method *on_post_dev_commands_sensors* displayed in Listing 6 receives device sensor configurations and verifies that all required data is present and of the appropriate data type. If the data is in the proper format, it is converted to JSON and transmitted to the Mediator.

```
1  class DeviceResource(object):
2      ...
3      def on_get_dev_samples(self, req, resp, device_id):
4          logging.log(logging.INFO, req)
5
6          device_id = convert_to_int(device_id)
7
8          if not device_id:
9              resp.status = falcon.HTTP_400
10             return
11
12         sample_id = req.get_param_as_list("sample-id", required=True)
13         lower_ts = req.get_param_as_int("lower-ts")
14         higher_ts = req.get_param_as_int("higher-ts")
15
16         data = acq.get_dev_samples(device_id, sample_id, lower_ts, higher_ts)
17
18         resp.text = json.dumps(data, ensure_ascii=False)
19
20         resp.status = falcon.HTTP_200
21     ...
22     def on_post_dev_commands_sensors(self, req, resp, device_id):
23         logging.log(logging.INFO, req)
24
25         device_id = convert_to_int(device_id)
26
27         if not device_id:
28             resp.status = falcon.HTTP_400
29             return
30
31         data = req.media
32
33         required_data = ["md5", "persistance", "consumer", "variables"]
34
35         if not all(key in data for key in required_data):
36             resp.status = falcon.HTTP_400
37             return
38
39         if (type(data["md5"]) is not bool or type(data["persistance"]) is not bool or type(data["consumer"]) is not str
               ↪ or type(data["variables"]) is not list):
40             resp.status = falcon.HTTP_400
41             return
42
43         data["type"] = "variables"
44
45         try:
46             for d in data["variables"]:
47                 process_sensor_command(d)
48         except Exception:
49             resp.status = falcon.HTTP_400
```

```
50            return
51
52        data_bytes = json.dumps(data).encode("utf-8")
53        status = orch_api.sendCommand(str(device_id), data_bytes, b"variables")
54
55        if status:
56            resp.text = json.dumps(data, ensure_ascii=False)
57            resp.status = falcon.HTTP_200
58        else:
59            resp.status = falcon.HTTP_500
```

Listing 6: Devices microservice API endpoints

Two classes of listeners are used to communicate with the Mediator: the *SendDataListener* and the *CommandResultListener*. The first, as its name suggests, is invoked when data is received from the Mediator. In this instance, the only relevant data are the messages sent by the devices, which can be one of four types: regular, emergency, configuration, or command. After determining the type of message, its data is stored in the appropriate database collection. The second listener receives the command's output. This listener has limited utility because the majority of commands do not provide instant results, only those contained in the cloud, as is the case with the only command displayed, the FOTA versions command. The remaining command results will be processed by the standard data listener, as they are treated as any other device-sent message.

```
1  class _SendDataListener(SendDataListener):
2      def onSendData(self, device_id: str, data: bytes):
3          data_dict = json.loads(data.decode("utf-8"))
4
5          logging.log(
6              logging.INFO, f"Data received from orchestrator: {device_id=}, {data_dict=}"
7          )
8
9          device_id = convert_to_int(device_id)
10
11         match data_dict["type"]:
12             case "regular":
13                 acq.post_regular(data_dict["content"])
14             case "emergency":
15                 acq.post_emergency(data_dict["content"])
16             case "configuration":
17                 acq.post_configuration(device_id, data_dict["content"])
18                 acq.post_configuration_hist(device_id)
19             case "command_result":
20                 acq.post_command_result(data_dict["content"])
21
22 class _CommandResultListener(CommandResultListener):
23     def onCommandResult(self, device_id: str, result: bytes, context: bytes, accessToken: str):
24         result_dict = json.loads(result.decode("utf-8"))
25         context = context.decode("utf-8")
26
27         logging.log(
28             logging.INFO,
29             f"Command result received from orchestrator: {device_id=}, {result_dict=}, {context=}",
30         )
31
32         device_id = convert_to_int(device_id)
33
34         if context == "fota_versions":
35             acq.post_firmware_versions(device_id, result_dict)
```

Listing 7: Devices microservice listeners

4.2.4 Users

The users microservice only implements an API for authentication and user management.

This API's most important endpoint is the one used for user authentication, as it will be called more frequently. This endpoint is responsible not only for validating the user name and password, but also for ensuring the security of the user's data. The *on_post_authenticate* method is a POST method that begins by determining whether or not the required parameters are present, and then searches the database for data corresponding to the specified username. If a user is located, the only remaining step is to compare their passwords.

Before comparing passwords, the pepper is added to the user-supplied password. The pepper is a secret 16-character value stored on the server to increase the complexity and length of passwords. After the pepper is added, the password is hashed and compared to the database password using the Argon2 Python bindings. If the passwords match, the password is checked for rehashing, as the Argon2 configuration may be out of date, requiring the password to be hashed to a more recent configuration. The data is then returned to the user with a status code of 200, excluding the password (OK). If the password is invalid, an exception is thrown and the status code 403 (Forbidden) is returned.

```
1  class UserResource(object):
2     ...
3     def on_post_authenticate(self, req, resp):
4        logging.log(logging.INFO, str(req))
5
6        data = req.media
7
8        if "username" not in data or "password" not in data:
9           resp.status = falcon.HTTP_400
10          return
11
12       user_data = acq.get_user_data(data["username"])
13
14       if user_data is None:
15          resp.status = falcon.HTTP_404
16          return
17
18       concatenated_pword = data["password"] + PEPPER
19
20       try:
21          hasher.verify(user_data["password"], concatenated_pword)
22
23          if hasher.check_needs_rehash(user_data["password"]):
24             acq.update_password(
25                user_data["username"], hasher.hash(concatenated_pword)
26             )
27
28          del user_data["password"]
29
30          resp.text = json.dumps(user_data, ensure_ascii=False)
31          resp.status = falcon.HTTP_200
32       except argon2.exceptions.VerificationError:
33          resp.status = falcon.HTTP_403
```

Listing 8: Users microservice authentication endpoint

4.3 Dashboard

The implementation of the dashboard was accomplished using the Python Dash framework. The framework provides pure Python abstraction around HTML, CSS, and JavaScript and allows developers to extend framework use cases with CSS and JavaScript code, which was done in some instances to improve the dashboard's design and/or functionality.

This implementation strives to adhere as closely as possible to the design layout and color scheme established in the previous chapter, while also providing all the specified features. In some instances, the implementation exceeds the design, as additional small features were added to enhance the user experience without violating the established design principles.

Due to the size of the codebase, only the most pertinent callbacks and layouts are described here, along with the dashboard's final appearance.

4.3.1 Initialization, routing and layout

As shown in Listing 9, a Dash app object is created to initialize the dashboard. The Dash object receives as arguments a Flask server that is used for the backend, as well as some stylesheets that improve the components' basic design and layout. Additionally, a server object is created for later use in the server deployment.

```
1  flask_server = flask.Flask(__name__)
2
3  app = dash.Dash(
4      __name__,
5      server=flask_server,
6      external_stylesheets=[
7          dbc.themes.BOOTSTRAP,
8          dbc.icons.BOOTSTRAP,
9          dbc.icons.FONT_AWESOME,
10         "https://fonts.googleapis.com/css?family=Montserrat&display=swap",
11     ],
12     title="Link4S Dashboard",
13 )
14 ...
15 login_manager = LoginManager()
16 login_manager.init_app(flask_server)
17 login_manager.login_view = "/"
18 ...
19 server = app.server
20 ...
```

Listing 9: Dashboard initialization

The main layout is assigned to the app object in Listing 10, and the HTML div contains the sidebar and page content. The sidebar and main content of the page are separated so that one can be updated independently of the other and to provide greater design flexibility for the main page.

The callback displayed in Listing 10 offers routing functionality and updates the page's layout. This callback is triggered whenever a user interacts with the URL. The callback begins by generating a session token if none is already present, and then verifies that the user has successfully logged in. If the user is authenticated and the path matches one of the available paths, he will be redirected to the appropriate page and the page content will be rendered; otherwise, the user will be returned to the login page. The session token is also stored so that the user can use the dashboard for up to an hour without logging in again, even if the dashboard is closed and reopened.

```
1  app.layout = html.Div(
2     [
3        dcc.Location(id="url", refresh=True),
4        sidebar,
5        content,
6        dcc.Store(id="session-token-id", storage_type="local"),
7        WebSocket(id="changestreams-ws", url=WS_URL),
8     ]
9  )
10
11 @app.callback(
12    Output("url", "pathname"),
13    Output("sidebar", "style"),
14    Output("page-content", "children"),
15    Output("session-token-id", "data"),
16    Input("url", "pathname"),
17    State("session-token-id", "data"),
18 )
19 def render_page_content(pathname, session_token):
20    if session_token is None:
21       session_token = str(uuid.uuid1())
22
23    if not current_user.is_authenticated:
24       return (
25          no_update,
26          HIDE_SIDEBAR,
27          authentication.serve_layout(),
28          session_token,
29       )
30
31    if pathname == "/":
32       return "/overview", SHOW_SIDEBAR, no_update, session_token
33    elif pathname == "/overview":
34       return no_update, SHOW_SIDEBAR, overview.serve_layout(), session_token
35    ...
36    elif pathname == "/logout":
37       logout_user()
38       return "/", SHOW_SIDEBAR, no_update, session_token
```

Listing 10: Dashboard layout and routing

In Listing 11, the sidebar and page content are defined, with the sidebar consisting of multiple navigation links, one for each page, and a logout button. The HTML div for the main page's content is generated without any children, as this is handled by the callback in Listing 10.

```
1  sidebar = html.Div(
2     [
3        html.Div(
4           [
5              ...
6              dbc.Nav(
7                 [
8                    dbc.NavLink(
9                       [
10                         ...
```

```
                        "Overview",
                     ],
                     href="/overview",
                     active="exact",
                  ),
                  ...
               ],
               vertical=True,
               pills=True,
            ),
         ]
      ),
      dbc.Button(
         ...
         href="/logout",
         className="sidebar-button",
         id="btn-logout",
         outline=True,
         active=False,
      ),
   ],
   id="sidebar",
   ...
)

content = html.Div(id="page-content", className="content")
```

Listing 11: Dashboard sidebar and page content

4.3.2 Pages

4.3.2.1 Login

The implementation of the login page is illustrated in Figure 43. The authentication of users was implemented with the Python module Flask-Login, which provides user session management for Flask. This greatly simplifies common tasks like logging in, logging out, and managing user sessions, as they are mostly abstracted.

The code in Listing 12 demonstrates the implementation of the login using Flask-Login, which has a few prerequisites. As demonstrated in the initialization of the Dash app 9, a *LoginManager* class object is initially created. The *LoginManager* enables the creation of an *user_loader* callback, which is used to reload the user object using the session-stored user ID. This callback returns an object of the User class that has been declared and is derived from the Flask-Login *UserMixin* class. The *User* class provides numerous authentication properties, including the *is_authenticated* property used in Listing 10.

The login callback is invoked when the login button is clicked. The callback begins by determining whether both text fields are empty; if they are, the callback is terminated. If all required data is present, an HTTP POST request is sent to the users microservice to verify that the user exists in the database. If the request is successful, the user is redirected to the overview page. The user is presented with a toast error message if the login attempt fails.

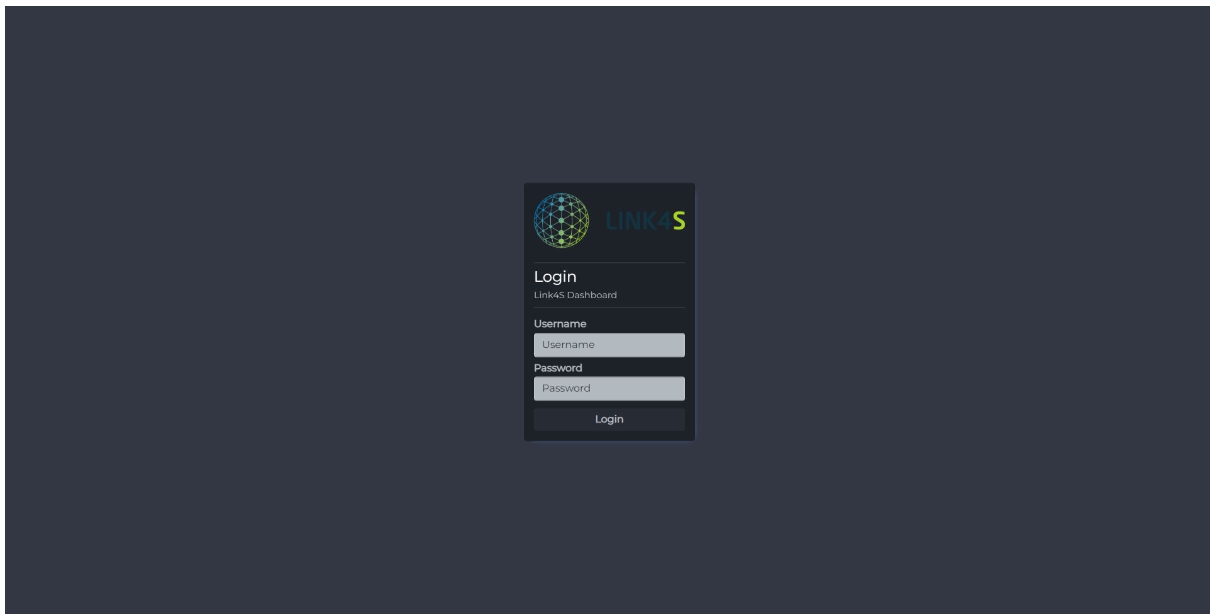


Figure 43: Login page implementation

```
class User(UserMixin):
    def __init__(self, username):
        self.id = username

@login_manager.user_loader
def user_loader(username):
    return User(username)

...

@app.callback(
    Output("login-url", "pathname"),
    Output("login-toast", "is_open"),
    Input("btn-login", "n_clicks"),
    State("uname-field", "value"),
    State("pword-field", "value"),
    prevent_initial_call=True,
)
def login_callback(n_clicks, uname, pword):
    if uname is None or pword is None:
        raise PreventUpdate

    post_data = {"username": uname, "password": pword}

    response = requests.post(f"{USERS_API_URL}/users/authenticate", data=post_data)

    if response.status_code == LOGIN_SUCCESS:
        user_obj = User(uname)
        login_user(user_obj)
        return "/overview", no_update

    return "/", True
```

Listing 12: Dashboard User class and login callback

4.3.2.2 Overview

The overview page implementation is depicted in Figure 44; it includes all of the features proposed during the design phase, as well as a few additional features to enhance the user experience, such as the search

features.

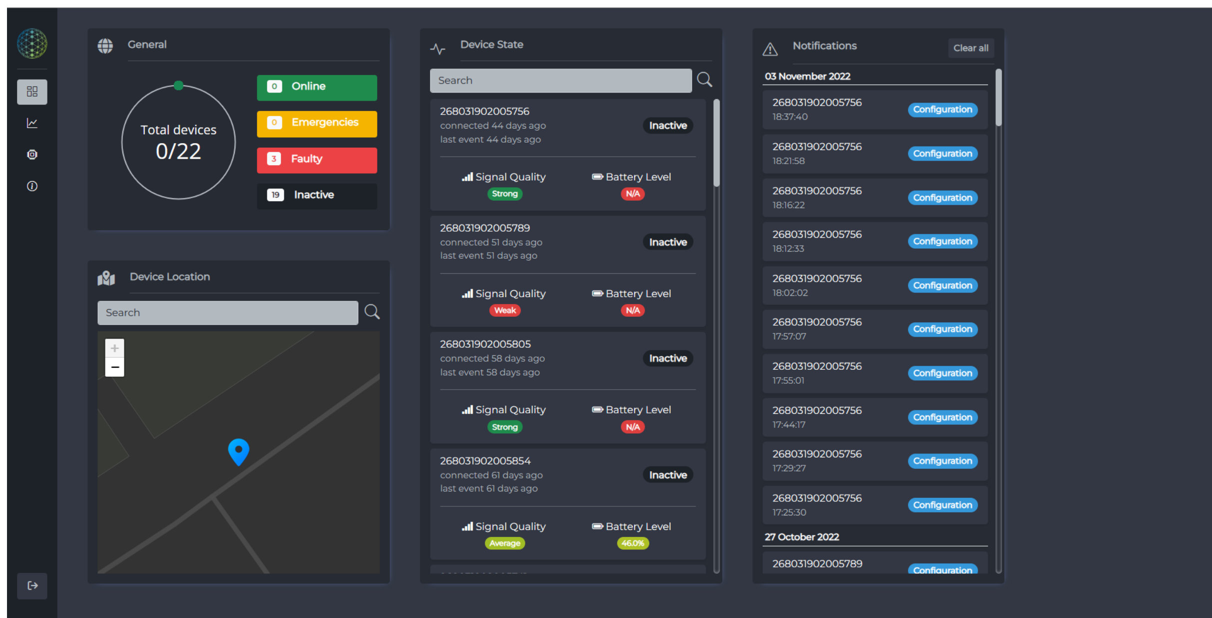


Figure 44: Overview page implementation

The majority of the data used in this section is generated at the time the callback in Listing 13 is invoked. This callback can be triggered in one of three situations: when the page is initially loaded, when the page is already loaded and the database is updated, or when the standard refresh interval of five minutes has passed. In either case, a request is made to the devices API to obtain the current state of the devices, followed by some processing to determine the status of the devices based on the amount of time that has passed since they last connected or sent a message, and finally, the data is converted into JSON format and stored locally in the browser.

```
@app.callback(
    Output("device-data", "data"),
    Input("changestreams-ws", "message"),
    Input("refresh-interval", "n_intervals"),
    State("device-data", "data"),
)
def compute_device_data(ws_message, n_intervals, json_dev_data):
    if json_dev_data is None:
        dev_data = list()
    else:
        dev_data = json.loads(json_dev_data)

    if ctx.triggered_id == "changestreams-ws":
        ...

    dev_data = requests.get(f"{utils.DEVICES_API_URL}/devices/state").json()

    for dev in dev_data:
        ...

        grace_period = dev["alarm"] * 0.5

        cur_ts = datetime.now().astimezone().timestamp()

        if dev["last_event_timestamp"]:
```

```
26            last_event_ts = date_parser.isoparse(dev["last_event_timestamp"]).astimezone().timestamp()
27        else:
28            last_event_ts = None
29
30        last_conn_ts = date_parser.isoparse(dev["last_connect_timestamp"]).astimezone().timestamp()
31
32        if last_event_ts and last_event_ts > last_conn_ts:
33            ts_delta = cur_ts - last_event_ts
34        else:
35            ts_delta = cur_ts - last_conn_ts
36
37        if ts_delta <= dev["alarm"] + grace_period:
38            dev["status"] = "online"
39        elif ts_delta <= dev["alarm"] + grace_period + utils.UNIX_24H:
40            dev["status"] = "faulty"
41        else:
42            dev["status"] = "inactive"
43
44    return json.dumps(dev_data)
```

Listing 13: Dashboard device data callback

After storing the device data, the general section and device status section are updated using two distinct callbacks that receive the JSON data as input and return the updated layouts for each section as output.

By hovering over the state's layouts, as shown in Figure 45, the user can view the ID of all the devices in each state as well as information about that state in the general section.

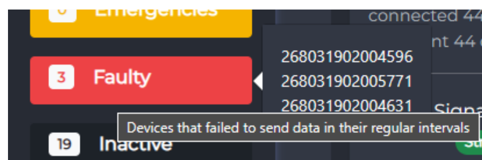


Figure 45: Overview page general status hover

The data for the location of the devices is provided through a different endpoint of the devices API, as shown in Listing 14. Consequently, the computation of the location of the devices is performed differently. This callback is invoked in the same circumstances as the device status callback, but it is also invoked when the user interacts with the search box to only display the results area.

To generate the maps, the open-source Leaflet library was employed, which provides many features that enable the user interaction with the map and device pins.

```
1  @app.callback(
2      Output("geo-json", "data"),
3      Output("map", "center"),
4      Output("map", "zoom"),
5      Input("changestreams-ws", "message"),
6      Input("refresh-interval", "n_intervals"),
7      Input("location-search", "value"),
8      State("geo-json", "data"),
9  )
10 def compute_location_data(ws_message, n_intervals, search_term, geo_data):
11     if ctx.triggered_id == "changestreams-ws":
12         ...
13     elif ctx.triggered_id == "location-search":
14         ...
```

```
15
16    devices_location = requests.get(f"{utils.DEVICES_API_URL}/devices/location").json()
17
18    for device in devices_location:
19        board_ts_str = (date_parser.isoparse(device["board_timestamp"]).astimezone().strftime("%d-%m-%Y %H:%M:%S"))
20        system_ts_str = (date_parser.isoparse(device["system_timestamp"]).astimezone().strftime("%d-%m-%Y %H:%M:%S"))
21
22        device["popup"] = f'Device ID: {device["device_id"]}<br>\
23                            Board timestamp: {board_ts_str}<br>\
24                            System timestamp: {system_ts_str}'
25
26    geo_json = dlx.dicts_to_geojson(devices_location, lat="latitude", lon="longitude")
27
28    return geo_json, no_update, no_update
```

Listing 14: Dashboard device location callback

As depicted in Figure 46, when the user interacts with the location pin, he can view the ID of the device and the timestamp of when the location was captured.

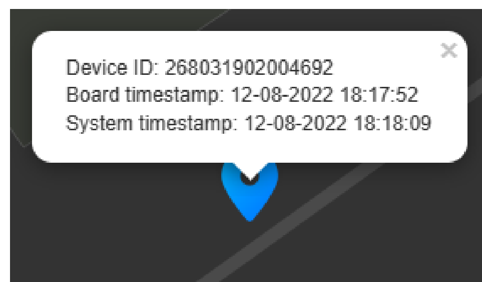


Figure 46: Overview page device location pin

In the form of a list, the notifications section displays the device's critical notifications. Each entry in the list displays the device's identifier, the notification's timestamp, and its type. When a user clicks on an alert, a window containing additional information appears, similar to the example in Figure 47. Through the click of a button, the user can delete a single notification or all of them.

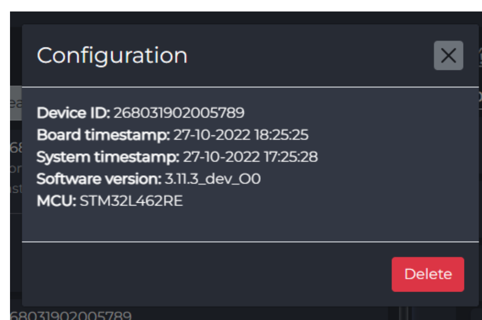


Figure 47: Overview page notification

The notifications are stored in the browser's local storage, which means they persist across browser sessions and disappear only when the user deletes them. A strict limit of one hundred notifications is imposed to prevent the page from experiencing performance issues due to loading a large number of

notifications. The notifications are computed in the callback in Listing 15, which also handles all user actions, such as deleting a single or all notifications. Requests are sent to three distinct endpoints for configurations, emergencies, and command results in order to acquire the required data. Each notification is also assigned a unique identifier, which is required for keeping track of them within the list.

```
@app.callback(
    Output("notification-data", "data"),
    Input("changestreams-ws", "message"),
    Input({"type": "delete-btn", "index": ALL}, "n_clicks"),
    Input("clear-notifications", "n_clicks"),
    State("notification-data", "data"),
    State("notification-data", "modified_timestamp"),
)
def update_notification_data(ws_message, delete_btn, clear_btn, json_not_data, modified_timestamp):
    if json_not_data is None:
        not_data = list()
    else:
        not_data = json.loads(json_not_data)

    new_ts = datetime.now().astimezone().timestamp()

    if ctx.triggered_id == "changestreams-ws":
        ...

    if ctx.triggered_id == "clear-notifications":
        not_data = []

    elif (isinstance(ctx.triggered_id, dict) and ctx.triggered_id["type"] == "delete-btn"):
        not_data = list(filter(lambda x: x["idx"] != ctx.triggered_id["index"], not_data))

    else:
        if not modified_timestamp:
            lower_ts = new_ts
        else:
            lower_ts = modified_timestamp / 1000

        new_configurations = requests.get(f"{utils.DEVICES_API_URL}/devices/configurations", params={"lower-ts": int(
            ↪ lower_ts)}).json()
        ...

        configurations = [dict(cfg, **{"type": "configuration"}) for cfg in new_configurations]
        ...

        not_data = [*not_data, *configurations, *emergencies, *commands]

    not_data = [dict(obj, **{"idx": str(uuid.uuid4())}) for obj in not_data]

    sorted_not_data = sorted(not_data, key=lambda x: x["system_timestamp"])

    if len(sorted_not_data) >= MAX_NOTIFICATION_NUM:
        sorted_not_data = sorted_not_data[len(sorted_not_data) - MAX_NOTIFICATION_NUM :]

    return json.dumps(sorted_not_data)
```

Listing 15: Dashboard notifications callback

4.3.2.3 Metrics

The metrics page displays graphs containing sensor data for each configured sensor. In Figure 48, an example of the implemented layout is displayed using two distinct sensors with varying layout options. The data in the plot is presented over a user-selected time scale, and two gauges or displays on both sides indicate the most recent value and the average or mean value. In addition, there is an *Export to Excel*

button that, when clicked, generates and automatically downloads an Excel spreadsheet containing the sensors divided by pages, with two columns per plot line (time and value) and an image of the plot for each page.

Figure 48: Metrics page implementation

A settings menu is also provided, which contains the options shown in Figure 49, to allow for a degree of customization. This allows the user to select the sensors he wishes to view, as well as modify certain plot parameters. These settings are stored locally within the browser and are therefore retained between sessions.

Although the settings menu allows the user to customize the page, the majority of configuration is performed in a YAML configuration file outside of the dashboard. This file is also where new sensors and their corresponding IDs can be added, along with the sensor variables. Unless a sensor is configured in this file, it will not be displayed on the dashboard. This configuration is illustrated in Figure 50.

The configuration file permits the definition of various types of components, such as gauges and displays, for displaying statistics, and the statistics themselves can be either average or mean values. The sensor variables are contained within the properties, where the ID cannot be arbitrary and must correspond to the one defined in the cloud for that variable. In this section, the color of the plot, as well as the labels, units, decimal places, and minimum/maximum values for the gauges, can be specified.

This page's code contains multiple callbacks, but the most important one is in Listing 16, which retrieves sensor data from the devices API and generates the primary plot. This callback utilizes pattern-matching callbacks [77] to retrieve data and generate graphs for multiple sensors simultaneously. It is

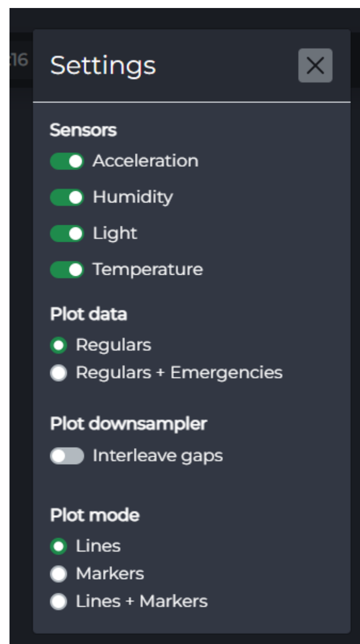


Figure 49: Metrics page settings menu

```yaml
sensors:
  humidity:
    name: Humidity
    properties:
      rel_humidity:
        id: sHUM
        color: rgb(26, 188, 156)
        label: Relative Humidity (%RH)
        min: 0
        max: 100
        unit: "%RH"
        decimal_places: 1
    component_type: gauge
    plot_statistic: average
```

Figure 50: Metrics page configuration

activated when the user presses the apply button, and it utilizes the current state of multiple components, such as the user-selected device ID, date/time period, and configurations chosen from the settings menu, and outputs the results to a plot and the statistic column.

The *fetch* method of the *SamplesFetcher* class is invoked in lieu of a direct call to the device's API to retrieve the standard sensor data. The *SamplesFetcher* class is utilized as it is the class responsible for not only the implementation of the time series cache, but also the acquisition and processing of sensor data from the device's API, as described in greater detail below. It returns the data as a *DataFrame* structure from the Pandas library, which the Plotly graphing libraries can utilize directly.

Moreover, since the sensor data can consist of millions of samples depending on the selected time period, it would be extremely challenging to interact with the plot without experiencing severe performance issues. To prevent this, the Plotly-Resampler [78] library was used to implement a downsampling technique

that can reduce millions of samples to thousands without losing valuable information.

```python
@app.callback(
   Output({"type": "sensor-plot", "index": MATCH}, "figure"),
   Output({"type": "plot-stat-comp", "index": MATCH}, "children"),
   Output({"type": "plot-stat-col", "index": MATCH}, "style"),
   Input("apply-btn", "n_clicks"),
   State({"type": "sensor-switch", "index": MATCH}, "value"),
   State("devices-dropdown", "value"),
   State("date-time-picker", "startDate"),
   State("date-time-picker", "endDate"),
   State("data-mode-input", "value"),
   State("plot-mode-input", "value"),
   State("interleave-switch", "value"),
   State("session-token-id", "data"),
   prevent_initial_call=True,
)
def plot_callback(apply_btn, sensor_switch, device_id, start_date, end_date, data_mode_input, plot_mode_input,
    ↪ interleave_switch, session_token):
   if device_id is None:
      raise PreventUpdate

   utils.set_selected_device(session_token, device_id)

   if not sensor_switch:
      raise PreventUpdate

   ...

   for property_data in sensors[ctx_sensor]["properties"].values():
      sample_data = SamplesFetcher(device_id, property_data["id"]).fetch(dt_start, dt_end)

      if sample_data.empty:
         continue

      sample_data["values"] = round(sample_data["values"], property_data["decimal_places"])

      plot.add_trace(
         go.Scattergl(
            x=sample_data["timestamps"],
            y=sample_data["values"],
            mode=plot_mode_input,
            name=f'{property_data["label"]}<br>(regulars)',
            line=dict(color=property_data["color"]),
         )
      )

      if "emergency" in data_mode_input:
         ...

      if sensors[ctx_sensor]["plot_statistic"] == "average":
         stat_value = round(sample_data["values"].mean(skipna=True), property_data["decimal_places"])
      elif sensors[ctx_sensor]["plot_statistic"] == "median":
         stat_value = round(sample_data["values"].median(skipna=True), property_data["decimal_places"])
      else:
         continue

      if sensors[ctx_sensor]["component_type"] == "gauge":
         ...
      elif sensors[ctx_sensor]["component_type"] == "digital":
         ...

   ...

   return plot, plot_stat_layout, plot_stat_style
```

Listing 16: Dashboard metrics plot generation

4.3.2.4 Sensors

As depicted in Figure 51, the sensors page allows for the configuration of the device sensor variables. The page's presentation is dynamic, which allows the page to load as many containers as required for

all variables. As the default ID values are not particularly explicit, the labels of the containers can also be configured within the YAML file used for the metrics page.

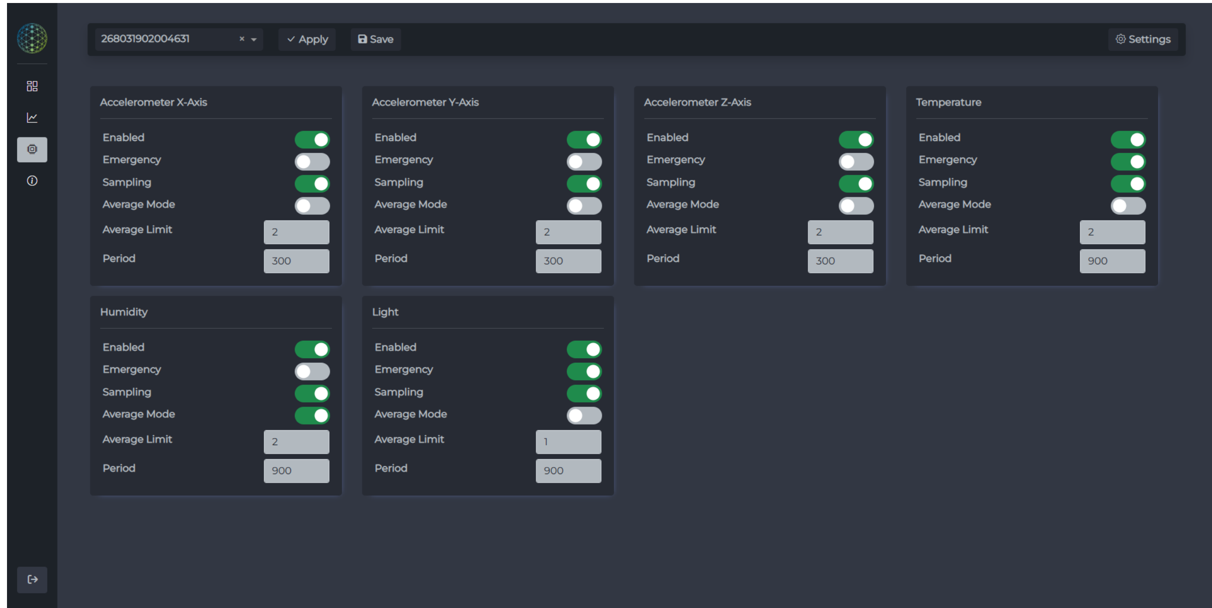


Figure 51: Sensors page implementation

In contrast to the metrics page, the settings window does not alter the page layout, but rather modifies the parameters of the command sent to the device's API when the changes are saved. These parameters include an MD5 hash for validating data integrity, the choice of parsing microservice to consume the command, and the persistence of the command when consumed.

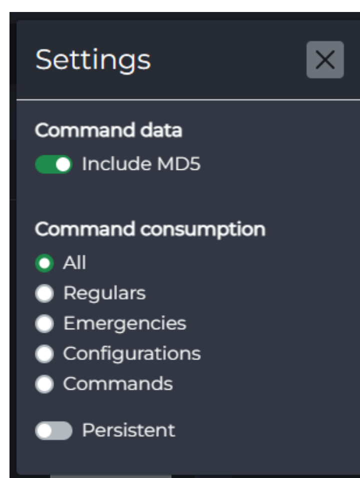


Figure 52: Sensors page configuration

The *sensor_variables_callback* shown in Listing 17 is responsible for generating the containers, storing the current state of the sensor variables, and sending this information to the devices API. This callback

is triggered when the user presses either the apply or save button. The sensors data is retrieved from the devices API for the apply function, and containers with parameters are generated for the available variables. In the save function, a comparison is performed between the new and old parameter values to determine which ones have changed. The modified values are then sent as a command to the device's API. If no changes are detected, no data will be transmitted.

```python
@app.callback(
    Output("sensors-container", "children"),
    Output("device-data-store", "data"),
    Output("sensors-toast", "is_open"),
    Output("sensors-toast-txt", "children"),
    Input("btn-apply", "n_clicks"),
    Input("btn-save", "n_clicks"),
    State("devices-dropdown", "value"),
    State("device-data-store", "data"),
    State("command-input", "value"),
    State("md5-switch", "value"),
    State("persistent-switch", "value"),
    State({"type": "sensor-prop-bool", "index": ALL}, "id"),
    State({"type": "sensor-prop-bool", "index": ALL}, "value"),
    State({"type": "sensor-prop-val", "index": ALL}, "id"),
    State({"type": "sensor-prop-val", "index": ALL}, "value"),
    State("session-token-id", "data"),
)
def sensor_variables_callback(apply_clicks, save_clicks, device_id, device_data, command_consumer, md5, persistent,
    ↪ props_bool_ids, props_bool, props_val_ids, props_val, session_token):
    if device_id is None:
        raise PreventUpdate

    if ctx.triggered_id == "btn-save":
        return save_btn_click(save_clicks, device_id, device_data, command_consumer, md5, persistent, props_bool,
            ↪ props_bool_ids, props_val, props_val_ids)
    else:
        return apply_btn_click(apply_clicks, device_id, session_token)

def apply_btn_click(n_clicks, device_id, session_token):
    ...

    configs = requests.get(f"{DEVICES_API_URL}/devices/{device_id}/configurations/latest").json()

    ...

    return container, json.dumps(configs), no_update, no_update

def save_btn_click(n_clicks, device_id, device_data, command_consumer, md5, persistent, props_bool, props_bool_ids,
    ↪ props_val, props_val_ids):
    json_data = json.loads(device_data)
    old_data = copy.deepcopy(json_data)

    ...

    changed_props = get_changed_props(old_data, json_data)

    if not changed_props:
        return no_update, json.dumps(old_data), True, "Failed to send command. No changes detected."

    ...

    response = requests.post(f"{DEVICES_API_URL}/devices/{device_id}/commands/sensors", data=json.dumps(post_data),
        ↪ headers=headers)

    if response.status_code == 200:
        return no_update, json.dumps(json_data), True, f"Sent command with data: {response.json()}."
    else:
        return no_update, json.dumps(old_data), True, "Failed to send command."
```

Listing 17: Dashboard sensor variables callback

4.3.2.5 Specifications

The specifications page in Figure 53 displays the main specifications of the device that are not sensor specifications, as this information is available on the sensors page. In addition, two containers for frequently used commands, the alarm command and the FOTA update command, are also present. The first allows the user to set the frequency of periodic messages sent from the device to the cloud, while the second allows the user to update the device's firmware. A settings window is also available, for the same reason as on the sensors page, to allow the user to configure the command parameters.

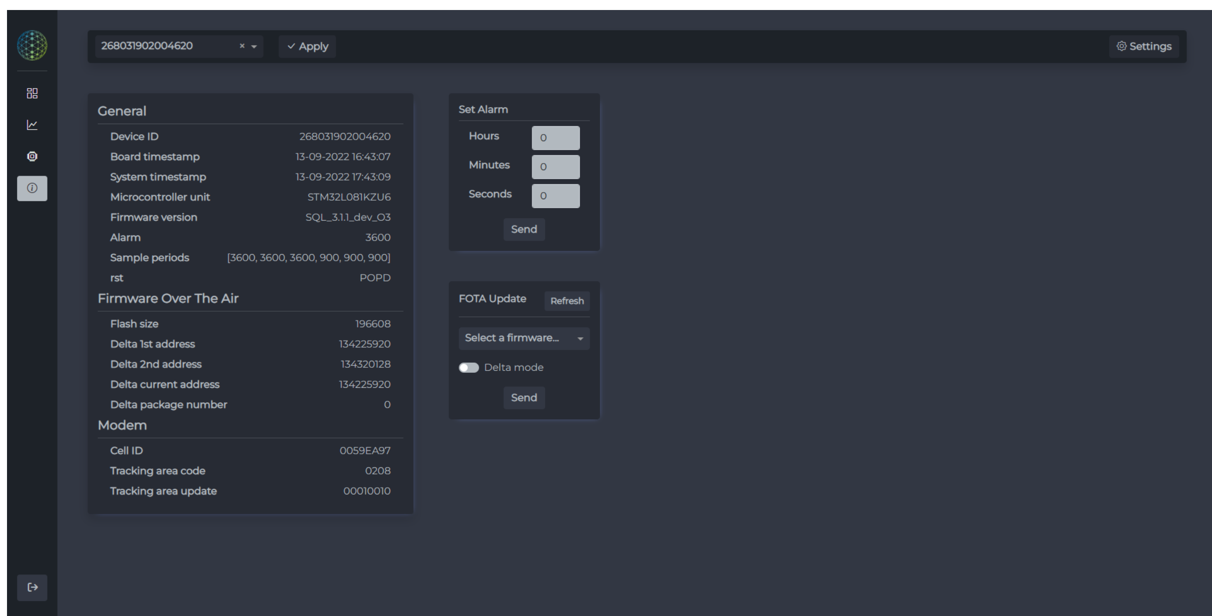


Figure 53: Specifications page implementation

The callback displayed in Listing 18 is invoked when the apply button is clicked and returns the layout containing the multiple device specifications. It begins by retrieving the data from the configurations endpoint of the devices API, which is then processed by the function *prop_layout* to generate the container containing the various device specifications.

```
1  @app.callback(
2     Output("info-container", "children"),
3     Output("config-layout", "class_name"),
4     Input("spec-apply-btn", "n_clicks"),
5     State("spec-devices-dropdown", "value"),
6     State("session-token-id", "data"),
7  )
8  def apply_btn_callback(n_clicks, device_id, session_token):
9     if device_id is None:
10       raise PreventUpdate
11
12    set_selected_device(session_token, device_id)
13
14    config = requests.get(f"{DEVICES_API_URL}/devices/{device_id}/configurations/latest").json()
15
16    layout = dbc.Container([], style={"background-color": "#272b34"}, class_name="p-3 shadow rounded-3")
17
```

```
18    for prop in config:
19        layout.children = prop_layout(prop)
20
21    return layout, SHOW_CONTAINER_CLASS
```

Listing 18: Dashboard specifications apply callback

The callbacks for the alarm and FOTA update commands are not displayed because they are nearly identical to those previously displayed in Listing 17, specifically within the *save_btn_click* function. In each instance, the current state of the user-defined settings for the command is retrieved, and the command is created with the settings in addition to the main command content and sent to the devices API. Due to the impossibility of receiving a prompt command response from the device, the status code of the response indicates only whether the other section of the cloud successfully received the command.

4.3.3 Real-time updates

In order to provide real-time updates to the dashboard, a separate service based on MongoDB's change streams feature was developed. Motor, an asynchronous Python driver for MongoDB, was paired with Tornado, a Python web framework that enables the creation of a WebSockets server, to implement this feature. This is particularly important because the dashboard and this service communicate using WebSockets, which provides a persistent connection with low latency and overhead.

The WebSockets server implementation is displayed in Listing 19. The *ChangesHandler* class overrides the methods on the Tornado *WebSocketHandler* class and provides methods to initiate and terminate client connections, as well as forward data to clients. The *on_change* method formats the data, which is then transmitted to connected clients via the *send_updates* method.

```
1  class ChangesHandler(tornado.websocket.WebSocketHandler):
2      connected_clients = set()
3
4      def check_origin(self, origin):
5          return True
6
7      def open(self):
8          ChangesHandler.connected_clients.add(self)
9
10     def on_message(self, message):
11         logger.debug(message)
12
13     def on_close(self):
14         ChangesHandler.connected_clients.remove(self)
15
16     @classmethod
17     def send_updates(cls, message):
18         logger.debug(message)
19         for connected_client in cls.connected_clients:
20             connected_client.write_message(message)
21
22     @classmethod
23     def on_change(cls, change):
24         message = dict(tags=change["tags"], document=change["fullDocument"])
25
26         ChangesHandler.send_updates(message)
```

Listing 19: Dashboard change streams handler

The primary service implementation is displayed in Listing 20. As seen in the code, the asynchronous function *watch* is created and used as the callback function for database collection collections. Once the execution loop has begun, it is only interrupted by this function, which then invokes the *ChangesHandler* object to process the database data. The web application that employs WebSockets is assigned a URI (*changestreams*), a handler, and a port for communicating with clients (5500).

```
async def watch(collection, pipeline):
    global change_stream

    async with collection.watch(pipeline, "updateLookup") as change_stream:
        async for change in change_stream:
            ChangesHandler.on_change(change)

app = tornado.web.Application([(r"/changestreams", ChangesHandler)])
app.listen(5500)

client = MotorClient(DB_URL, DB_PORT)

loop = tornado.ioloop.IOLoop.current()
loop.add_callback(watch, client[DB_NAME][REGULARS_COLLECTION], REG_PIPELINE)
loop.add_callback(watch, client[DB_NAME][EMERGENCIES_COLLECTION], EMERG_PIPELINE)
loop.add_callback(watch, client[DB_NAME][CONFIGURATIONS_COLLECTION], CFG_PIPELINE)
loop.add_callback(watch, client[DB_NAME][CONFIGURATION_HIST_COLLECTION], CFG_HIST_PIPELINE)
loop.add_callback(watch, client[DB_NAME][COMMAND_RESULTS_COLLECTION], CMD_RES_PIPELINE)
```

Listing 20: Dashboard change streams main

4.3.4 Time series cache

The implementation of the time series cache consists of two classes with multiple methods. One of these classes, *WeeklyBuckets*, functions as a helper class to enable the computation of weekly buckets. It implements a pydantic [79] model with an *align_start* validator to ensure weekly buckets can be stacked by converting the date to the closest past Thursday midnight UTC, since Thursday is the epoch reference date.

```
class WeeklyBucket(BaseModel):
    start: datetime

    @property
    def end(self) -> datetime:
        return self.start + week

    @validator("start")
    def align_start(cls, v: datetime) -> datetime:
        seconds_in_week = week.total_seconds()

        return datetime.fromtimestamp(
            (v.timestamp() // seconds_in_week * seconds_in_week), timezone.utc
        )

    def next(self) -> "WeeklyBucket":
        return WeeklyBucket(start=self.end)

    def cache_key(self) -> str:
```

```
20        return f"{int(self.start.timestamp())}"
```

Listing 21: Dashboard WeeklyBucket class

The SamplesFetcher class in Listing 22 is the main class as it provides the *fetch* method, which abstracts the remaining code. This method receives the user-supplied start and end date/times and uses them to create buckets resorting to the *WeeklyBuckets* class, which are then used in the cache keys. The results for the provided cache keys are obtained in a single call, then iterated over to determine which cache keys have been assigned data. The *get_samples_from_upstream* method is used to make a data request to the devices API when a key does not contain any data. The data is subsequently filtered to ensure that it remains within the specified date and time range and in the correct format for the graphing libraries.

```
1  class SamplesFetcher:
2     ...
3
4     def get_cache_key(self, bucket: WeeklyBucket) -> str:
5        return f"samples:{self.device_id}:{self.sample_id}:{bucket.cache_key()}"
6
7     def get_samples_from_upstream(self, bucket: WeeklyBucket) -> pd.DataFrame:
8        start_date, end_date = bucket.start, bucket.end
9
10       ...
11
12       sample_objects = requests.get(f"{DEVICES_API_URL}/devices/{self.device_id}/samples", params=request_params).json
             ↪ ()
13
14       self.compute_dataframe(sample_objects)
15
16    def fetch(self, start_date: datetime, end_date: datetime) -> pd.DataFrame:
17       ...
18
19       buckets = get_buckets(start_date, end_date)
20
21       records_df = pd.DataFrame(dict(values=[], timestamps=[]))
22
23       key_list = [self.get_cache_key(bucket) for bucket in buckets]
24
25       cache_results = cache.mget(key_list)
26
27       for idx, cached_raw_samples in enumerate(cache_results):
28          # Check if there's anything in cache
29          if cached_raw_samples is not None:
30             raw_df = pickle.loads(cached_raw_samples)
31
32             records_df = pd.concat([records_df, raw_df])
33
34             continue
35
36          # Fetch the value from the upstream
37          self.get_samples_from_upstream(buckets[idx])
38          records_df = pd.concat([records_df, self.samples_df])
39
40          key_ttl_dict[key_list[idx]] = get_cache_ttl(buckets[idx])
41          cache_data_dict[key_list[idx]] = pickle.dumps(self.samples_df, protocol=pickle.HIGHEST_PROTOCOL)
42
43       ...
44
45       return fil_rec_df
46
47    ...
```

Listing 22: Dashboard SamplesFetcher class

Chapter 5

Results

Multiple tests were conducted on the production server to validate the cloud's architecture and ensure that all components function as expected. Using log files, database collections, and the dashboard itself, the results are displayed in a variety of ways. The test setup included two devices deployed on the Digital Transformation CoLab (DTx) facilities and a remote computer connected to the cloud's VPN with complete access to the cloud's systems.

This test deployment setup resembles a real deployment in that the devices are deployed in difficult to access locations and interaction with them is conducted remotely. The tests were conducted from the perspective of a user interacting with the dashboard as he manages devices and views data transmitted by them. Following the user's interaction with the dashboard, the data paths are followed to ensure that the data arrives where and when it should.

5.1 Login

Logging into the dashboard with the correct credentials is the initial step. In this instance, the administrator login was utilized, as displayed on Figure 54. After clicking the *Login* button, a POST request is sent to the Users microservice API at the *users/authenticate* endpoint shown in Figure 55. After successfully logging in, the user is redirected to the homepage.

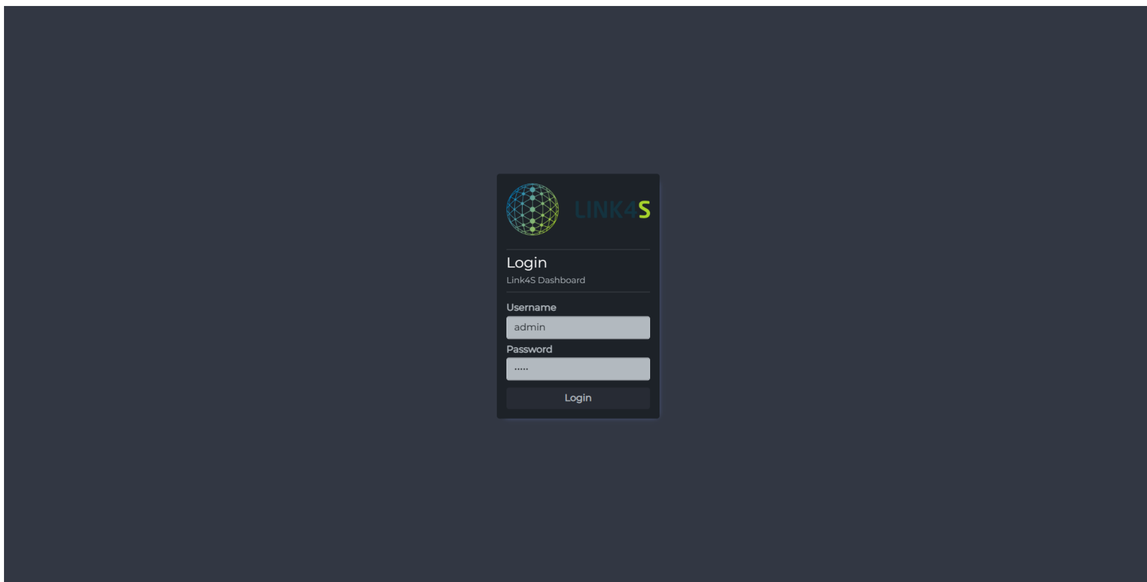


Figure 54: Dashboard login page test

```
1    2023-01-19 08:25:21,627 root INFO <Request: POST 'http://10.8.0.102:5600/v1/users/authenticate'>
2    2023-01-19 22:49:50,947 root INFO <Request: POST 'http://10.8.0.102:5600/v1/users/authenticate'>
3    2023-01-19 23:26:33,307 root INFO <Request: POST 'http://10.8.0.102:5600/v1/users/authenticate'>
```

Figure 55: Users microservices authentication request logs

5.2 Overview

As the devices were connected for some time prior to the tests, some activity is already available on the overview page. As seen in Figure 56, the two test devices with the IDs *268031902005750* and *268031902005790* are online and transmitting messages every hour.

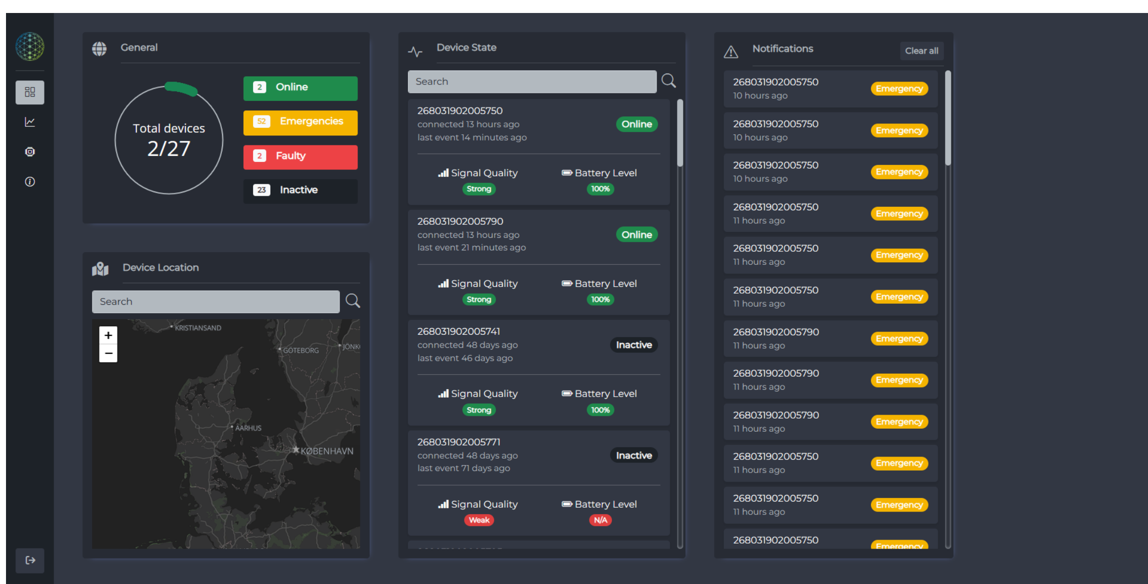


Figure 56: Dashboard overview page

In addition, some emergencies were also received as a result of the default configuration of these devices, as some sensor variables reached abnormal values. Clicking on the most recent emergency notification opens the window in Figure 57 that displays additional information, including the timestamp of the occurrence on the same day as well as the flag it belongs to, which is the accelerometer Z-axis in this case (ACz). These emergencies can also be seen on the metrics page's plots.

Unfortunately, it is not possible to view results for the device location section because none of the devices in this test are equipped to transmit location data. However, the device with the ID *268031902004692* is capable of doing so on the development server, as shown in Figure 58. A pin is placed on the device's most recent known location, and upon clicking it, the device ID and timestamps of the last known location are displayed.

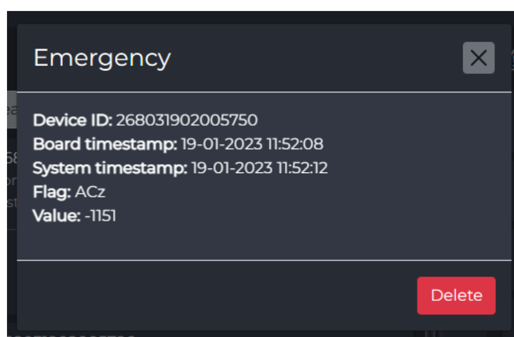


Figure 57: Dashboard emergency notification details

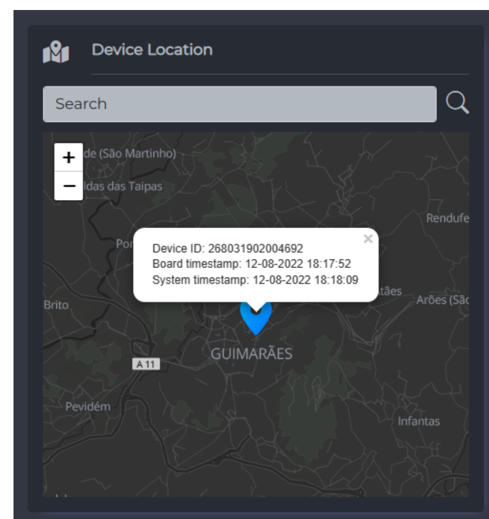


Figure 58: Dashboard device location section

5.3 Metrics

On the metrics page, a time range for the previous twenty-four hours was selected. As seen in the logs in Figure 59, after clicking the *Apply* button, a GET request for the sensor data was sent to the Devices microservice because the data was not readily available in the cache. In this log, the requests for the sensor variable *ACx* range from day 12 to day 19, and from day 19 to day 26 respectively. As the sensor data is divided into weekly buckets, this is how requests for caching purposes are made. In Figure 60, the logs of GET requests on the Devices microservice display the same time ranges converted to the Unix time format.

```
546   2023-01-19 23:56:15,969 app INFO Fetch self.sample_id='sACx' for self.device_id=268031902005790
547   (start_date=datetime.datetime(2023, 1, 12, 0, 0, tzinfo=datetime.timezone.utc), end_date=datetime.datetime(2023, 1, 19, 0, 0, tzinfo=datetime.timezone.utc)) from the upstream
548   2023-01-19 23:56:16,085 app INFO Fetch self.sample_id='sHUM' for self.device_id=268031902005790
549   (start_date=datetime.datetime(2023, 1, 12, 0, 0, tzinfo=datetime.timezone.utc), end_date=datetime.datetime(2023, 1, 19, 0, 0, tzinfo=datetime.timezone.utc)) from the upstream
550   2023-01-19 23:56:16,091 app INFO Fetch self.sample_id='sACx' for self.device_id=268031902005790
551   (start_date=datetime.datetime(2023, 1, 19, 0, 0, tzinfo=datetime.timezone.utc), end_date=datetime.datetime(2023, 1, 26, 0, 0, tzinfo=datetime.timezone.utc)) from the upstream
```

Figure 59: Dashboard fetch sensor data logs

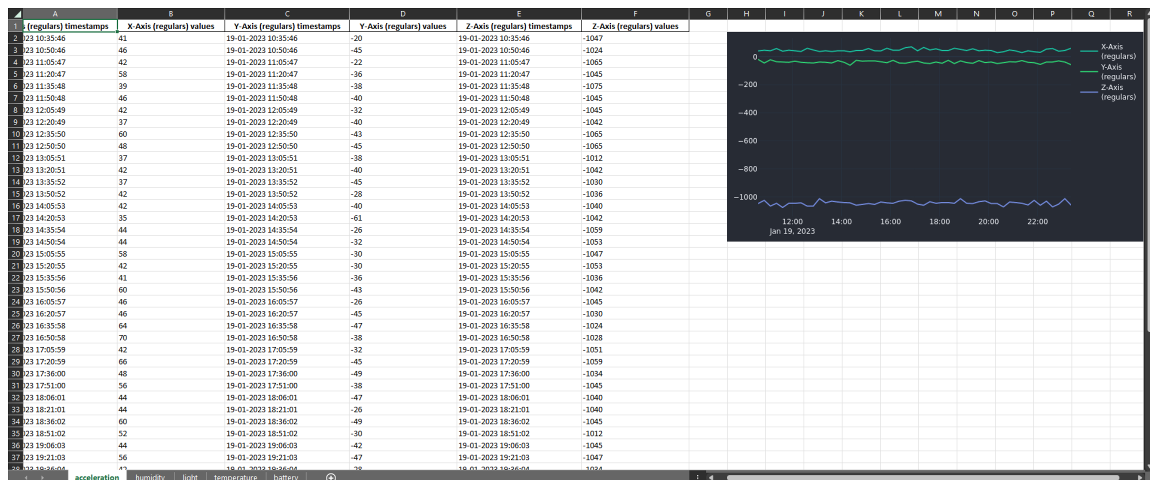
```
1453   2023-01-19 23:56:16,043 root INFO <Request: GET 'http://10.8.0.102:5601/v1/devices/268031902005790/samples?sample-id=sACx&lower-ts=1673481600&higher-ts=1674086400'>
1454   2023-01-19 23:56:16,090 root INFO <Request: GET 'http://10.8.0.102:5601/v1/devices/268031902005790/samples?sample-id=sHUM&lower-ts=1673481600&higher-ts=1674086400'>
1455   2023-01-19 23:56:16,137 root INFO <Request: GET 'http://10.8.0.102:5601/v1/devices/268031902005790/samples?sample-id=sACx&lower-ts=1674086400&higher-ts=1674691200'>
```

Figure 60: Devices microservice metrics request logs

Once the sensor data for all available sensors has been retrieved, the plots are generated as shown in Figure 61. Despite the fact that the data acquired from the Devices microservice API corresponds to a much longer period due to the caching strategy, the displayed data is entirely contained within the user-selected time range. In this instance, all sensors are enabled and the average is chosen as the optional graph.

Figure 61: Dashboard metrics page

Using the *Export to Excel* function, an Excel spreadsheet containing all the data previously presented to the user is generated and automatically downloaded. In Figure 62, the spreadsheet is separated into multiple pages based on the sensors. Within each page, the data for each variable is presented in separate columns alongside the corresponding timestamps, and an image of the plot is also created for each page.



	A	B	C	D	E	F
1	(regulars) timestamps	X-Axis (regulars) values	Y-Axis (regulars) timestamps	Y-Axis (regulars) values	Z-Axis (regulars) timestamps	Z-Axis (regulars) values
2	023 10:35:46	41	19-01-2023 10:35:46	-20	19-01-2023 10:35:46	-1047
3	023 10:50:46	46	19-01-2023 10:50:46	-45	19-01-2023 10:50:46	-1024
4	023 11:05:47	42	19-01-2023 11:05:47	-22	19-01-2023 11:05:47	-1065
5	023 11:20:47	58	19-01-2023 11:20:47	-36	19-01-2023 11:20:47	-1045
6	023 11:35:48	39	19-01-2023 11:35:48	-38	19-01-2023 11:35:48	-1075
7	023 11:50:48	46	19-01-2023 11:50:48	-40	19-01-2023 11:50:48	-1045
8	023 12:05:49	42	19-01-2023 12:05:49	-32	19-01-2023 12:05:49	-1045
9	023 12:20:49	37	19-01-2023 12:20:49	-40	19-01-2023 12:20:49	-1042
10	023 12:35:50	60	19-01-2023 12:35:50	-43	19-01-2023 12:35:50	-1065
11	023 12:50:50	48	19-01-2023 12:50:50	-45	19-01-2023 12:50:50	-1065
12	023 13:05:51	37	19-01-2023 13:05:51	-38	19-01-2023 13:05:51	-1012
13	023 13:20:51	42	19-01-2023 13:20:51	-40	19-01-2023 13:20:51	-1042
14	023 13:35:52	37	19-01-2023 13:35:52	-45	19-01-2023 13:35:52	-1030
15	023 13:50:52	42	19-01-2023 13:50:52	-28	19-01-2023 13:50:52	-1036
16	023 14:05:53	42	19-01-2023 14:05:53	-40	19-01-2023 14:05:53	-1040
17	023 14:20:53	35	19-01-2023 14:20:53	-61	19-01-2023 14:20:53	-1042
18	023 14:35:54	44	19-01-2023 14:35:54	-26	19-01-2023 14:35:54	-1059
19	023 14:50:54	44	19-01-2023 14:50:54	-32	19-01-2023 14:50:54	-1053
20	023 15:05:55	58	19-01-2023 15:05:55	-30	19-01-2023 15:05:55	-1047
21	023 15:20:55	42	19-01-2023 15:20:55	-30	19-01-2023 15:20:55	-1053
22	023 15:35:56	41	19-01-2023 15:35:56	-36	19-01-2023 15:35:56	-1036
23	023 15:50:56	60	19-01-2023 15:50:56	-43	19-01-2023 15:50:56	-1042
24	023 16:05:57	46	19-01-2023 16:05:57	-26	19-01-2023 16:05:57	-1045
25	023 16:20:57	46	19-01-2023 16:20:57	-45	19-01-2023 16:20:57	-1030
26	023 16:35:58	64	19-01-2023 16:35:58	-47	19-01-2023 16:35:58	-1024
27	023 16:50:58	70	19-01-2023 16:50:58	-38	19-01-2023 16:50:58	-1028
28	023 17:05:59	42	19-01-2023 17:05:59	-32	19-01-2023 17:05:59	-1051
29	023 17:20:59	66	19-01-2023 17:20:59	-45	19-01-2023 17:20:59	-1059
30	023 17:36:00	48	19-01-2023 17:36:00	-49	19-01-2023 17:36:00	-1034
31	023 17:51:00	56	19-01-2023 17:51:00	-38	19-01-2023 17:51:00	-1045
32	023 18:06:01	44	19-01-2023 18:06:01	-47	19-01-2023 18:06:01	-1040
33	023 18:21:01	44	19-01-2023 18:21:01	-26	19-01-2023 18:21:01	-1040
34	023 18:36:02	60	19-01-2023 18:36:02	-49	19-01-2023 18:36:02	-1045
35	023 18:51:02	52	19-01-2023 18:51:02	-30	19-01-2023 18:51:02	-1012
36	023 19:06:03	44	19-01-2023 19:06:03	-42	19-01-2023 19:06:03	-1045
37	023 19:21:03	56	19-01-2023 19:21:03	-47	19-01-2023 19:21:03	-1047

Figure 62: Dashboard metrics exported Excel spreadsheet

5.4 Specifications

On the specifications page, the majority of a device's specifications can be viewed, and alarm and FOTA update commands can be sent. As mentioned previously, the devices involved in these tests communicate with the cloud once per hour. To test the commands, this interval will be reduced to 5 minutes. In Figure 63, a 5-minute alarm is set for one of the devices, and the same is done for the other.

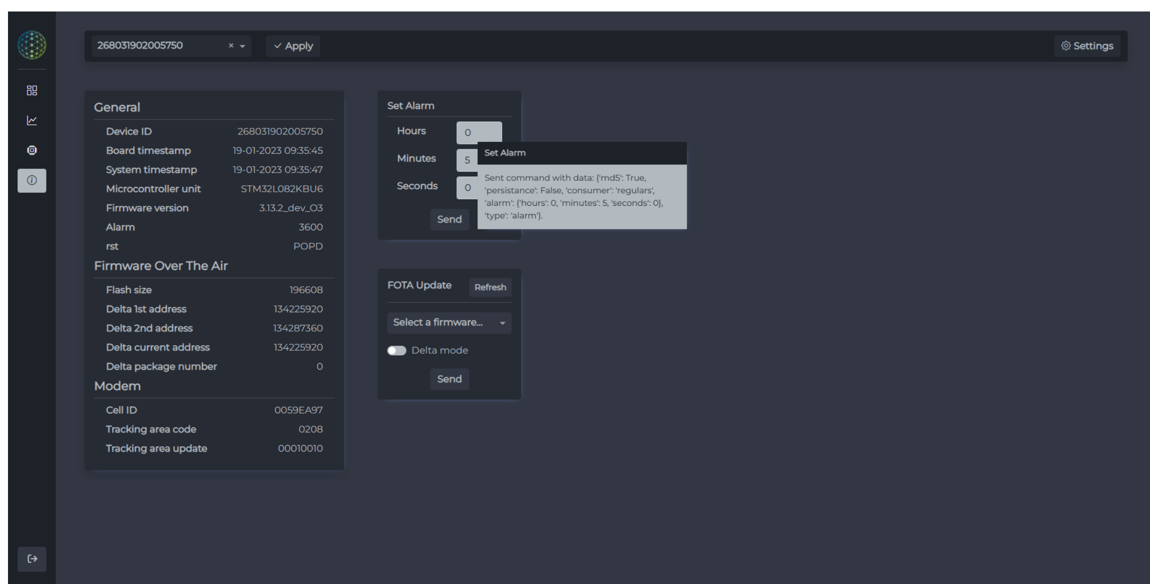


Figure 63: Dashboard specifications set alarm

The commands' consumer is set to *regulars*, which means they will be consumed by the devices the next time they send a regular message to the cloud. In the Process Data microservice logs displayed in Figure 64, it is possible to verify that the commands successfully passed through the Mediator, also known

as the Orchestrator, and onto the Device Ecosystem. As shown in Figure 65, they were then successfully saved in the database. Since their persistence is set to false, they will be deleted from the database when consumed.

```
95   2023-01-19 22:53:15,062 root INFO Command received from orchestrator: device_id='268031902005750',
96   command={'md5': True, 'persistance': False, 'consumer': 'regulars', 'alarm': {'hours': 0, 'minutes': 5, 'seconds': 0}, 'type': 'alarm'}
97   2023-01-19 22:54:11,122 root INFO Command received from orchestrator: device_id='268031902005790',
98   command={'md5': True, 'persistance': False, 'consumer': 'regulars', 'alarm': {'hours': 0, 'minutes': 5, 'seconds': 0}, 'type': 'alarm'}
```

Figure 64: Process Data microservice alarm command logs

```
  _id: ObjectId('63c9c9dbc88bd519a1f1497c')
  board_id: 268031902005750
> options: Object
  opcode: 0
> datagram: Array
  microservice: "regulars"
  description: "Set Interval Mode: True, Hours: [0], Min: [5], Sec [0]"

  _id: ObjectId('63c9ca13c88bd519a1f1497d')
  board_id: 268031902005790
> options: Object
  opcode: 0
> datagram: Array
  microservice: "regulars"
  description: "Set Interval Mode: True, Hours: [0], Min: [5], Sec [0]"
```

Figure 65: Database alarm commands

After a few minutes, the devices received the commands and, after processing them, transmitted the command results along with the new configuration containing the updated alarm value. In the overview page's notification section, there are four new notifications, two for each device, as shown in Figure 66.

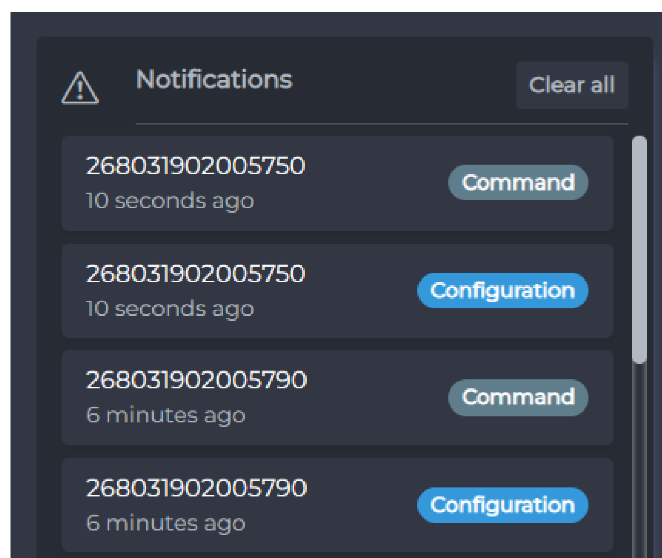


Figure 66: Dashboard notifications section

Clicking on the command notification in Figure 67 reveals that the opcode and description correspond to the command that was sent to the device, and that it was executed successfully as both the set and

get parameters are true. The configuration notification in Figure 68 displays some relevant information regarding the device, which was not changed in this instance.

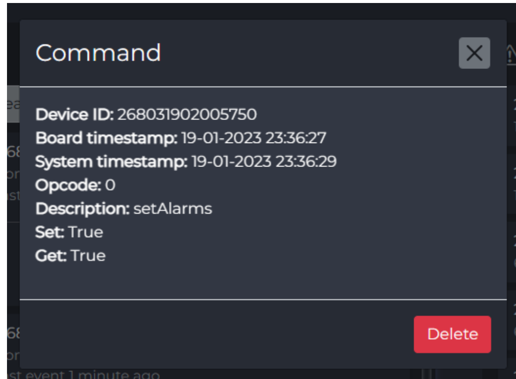


Figure 67: Dashboard command notification

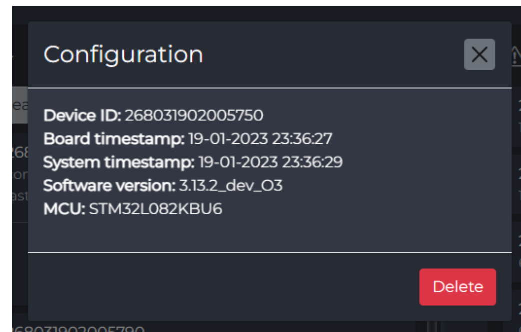


Figure 68: Dashboard configuration notification

It is also important to note that both of these notifications were received in real-time using the change streams feature, as seen in the logs in Figure 69, which shows the detected changes to the CommandResult, Configurations, and ConfigurationHistory collections of the database.

```
53  [D 230119 23:30:11 changeshandler:22] {'tags': ['state', 'command_result'], 'document': {'device_id': 268031902005790, 'board_timestamp': '2023-01-19T23:30:10.000Z', 'system_timestamp':
54  [D 230119 23:30:11 changeshandler:22] {'tags': ['state', 'configuration'], 'document': {'device_id': 268031902005790, 'board_timestamp': '2023-01-19T23:30:10.000Z', 'system_timestamp': '
55  [D 230119 23:30:11 changeshandler:22] {'tags': ['state', 'configuration_history'], 'document': {'device_id': 268031902005790, 'board_timestamp': '2023-01-19T23:30:10.000Z', 'system_times
```

Figure 69: Dashboard change streams logs

5.5 Sensors

On the dashboard's sensors page, the current parameters for each sensor variable are displayed and can be altered. In this case, the period of the accelerometer variables was changed to 600 seconds, and the light and humidity sensors were deactivated, as shown in Figure 70.

The logs in Figure 71 indicate that the Process Data microservice received the command successfully, and the modified variable data is as expected. The command's consumer is set to *regulars*, and it will be consumed when the next regular message is sent within 5 minutes. After a few seconds, the *regular* endpoint receives a POST request, indicating that the device has sent a regular message. After the POST request, a GET request is sent to the *command* endpoint to retrieve a command, if one is available in the database. The sensors variable command is returned to the device in byte array format.

The device sends a command result message after processing the command, which is displayed in Figure 72. Both set and get parameters are true, and the opcode and command description correspond

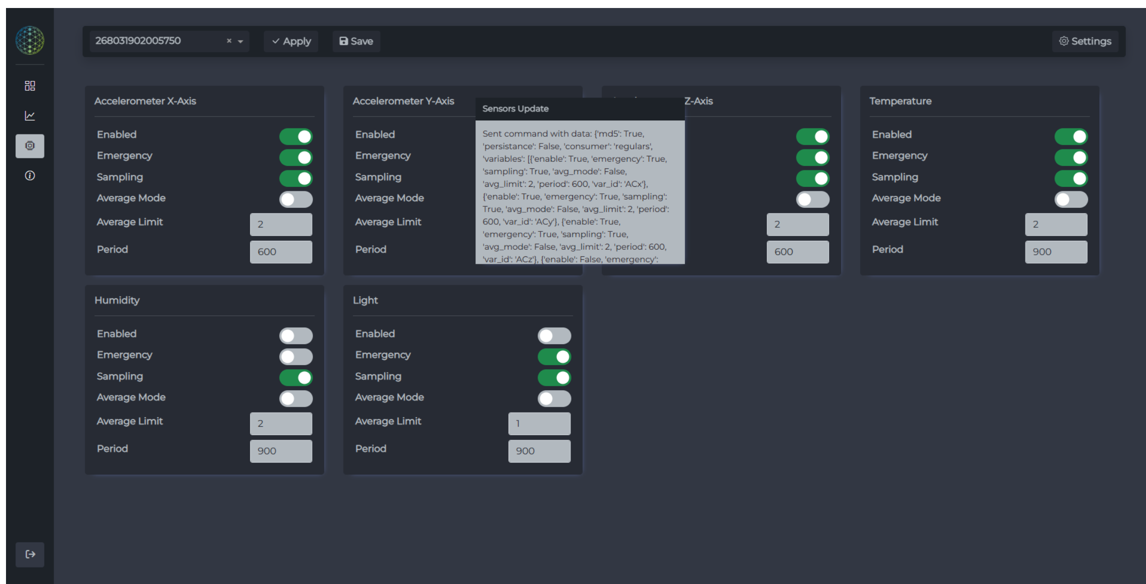


Figure 70: Dashboard sensors page parameters update

```
118  2023-01-19 23:41:17,812 root INFO Command received from orchestrator: device_id='268031902005750',
119  command={'md5': True, 'persistance': False, 'consumer': 'regulars',
120  'variables': [{'enable': True, 'emergency': True, 'sampling': True, 'avg_mode': False, 'avg_limit': 2, 'period': 600, 'var_id': 'ACx'},
121  {'enable': True, 'emergency': True, 'sampling': True, 'avg_mode': False, 'avg_limit': 2, 'period': 600, 'var_id': 'ACy'},
122  {'enable': True, 'emergency': True, 'sampling': True, 'avg_mode': False, 'avg_limit': 2, 'period': 600, 'var_id': 'ACz'},
123  {'enable': False, 'emergency': False, 'sampling': True, 'avg_mode': False, 'avg_limit': 2, 'period': 900, 'var_id': 'HUM'},
124  {'enable': False, 'emergency': True, 'sampling': True, 'avg_mode': False, 'avg_limit': 1, 'period': 900, 'var_id': 'LGH'}],
125  'type': 'variables'}
126  2023-01-19 23:41:32,627 root INFO <Request: POST 'http://10.8.0.102:5602/v1/regular'>
127  2023-01-19 23:41:32,687 root INFO <Request: GET 'http://10.8.0.102:5602/v1/command?board-id=268031902005750&microservice=regulars'>
128  2023-01-19 23:41:32,692 root INFO Returning Set Sensors Variables Config
129      S: 0, Var: 0, Status: 39, Period: 600, Threshold to Set: H, Type: ['GE'], Values: [200], Hyst to set: H, Type: ['GE'], Values: [5];
130      S: 0, Var: 1, Status: 39, Period: 600, Threshold to Set: H, Type: ['GE'], Values: [200], Hyst to set: H, Type: ['GE'], Values: [5];
131      S: 0, Var: 2, Status: 39, Period: 600, Threshold to Set: H, Type: ['GE'], Values: [200], Hyst to set: H, Type: ['GE'], Values: [5];
132      S: 1, Var: 1, Status: 36, Period: 900, Threshold to Set: H, Type: ['GE'], Values: [200], Hyst to set: H, Type: ['GE'], Values: [5];
133      S: 2, Var: 0, Status: 22, Period: 900, Threshold to Set: H, Type: ['GE'], Values: [200], Hyst to set: H, Type: ['GE'], Values: [5];
134
135  Payload bytearray(b"\x8d\x0b\xadh\xa7\x9a%\xe6\x92\x05\x0b={\xa6\x9e\x9b\x96\x03\xc4\x12\x00\x00\'X\x02\x00\x00H\x05\xc8\x00\x00\x00\x05\x05\x00\x00\x00\xc4\x12\x00\x01\'X\x02\x00\x01H\x
136  Length 118
```

Figure 71: Process Data microservice sensor variable command logs

to the command that was sent. After initiating communication with the cloud, the time required for the device to receive the command, execute it, and send the results back to the cloud was approximately 5 seconds from the user's perspective.

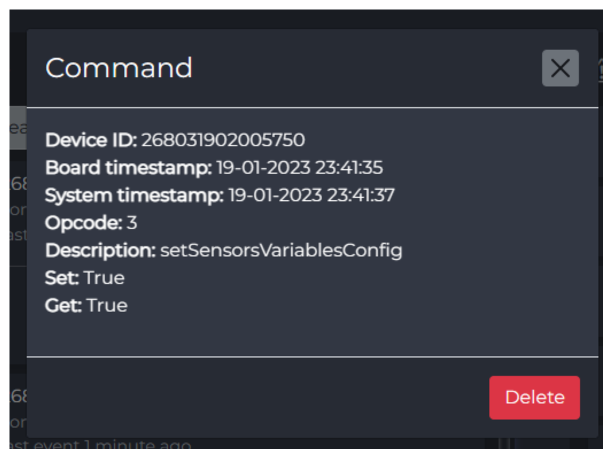


Figure 72: Dashboard change streams logs

Finally, in it is possible to view the changes made to the configuration of the device in the database collection. The updated configuration is received alongside the command result and the stored values match what was initially selected on the dashboard.

In Figure 73, it is possible to view the modifications made to the database Configurations collection. The updated configuration is delivered alongside the command result, and the stored values correspond to the values that were initially selected on the dashboard.


```
1     board_id: 268031902005750              Int64
2     alarms: 300                            Int32
3     board_timestamp: 1674171695            Int32
4     mcu: "STM32L082KBU6"                   String
5   > modem_config: Object                   Object
6   > ota: Object                            Object
7   > p: Array                               Array
8     sw_version: "3.13.2_dev_O3"            String
9     system_timestamp: 1674171697.811944    Double
10  v variables: Object                      Object
11    v ACx: Object                          Object
12        enable: true                       Boolean
13        emergency: true                    Boolean
14        sampling: true                     Boolean
15        avg_mode: false                    Boolean
16        avg_limit: 2                       Int32
17        period: 600                        Int32
18        emergency_value: 4                 Int32
19    > ACy: Object                          Object
20    > ACz: Object                          Object
21    > TMP: Object                          Object
22    v HUM: Object                          Object
23        enable: false                      Boolean
24        emergency: false                   Boolean
25        sampling: true                     Boolean
26        avg_mode: false                    Boolean
27        avg_limit: 2                       Int32
28        period: 900                        Int32
29        emergency_value: 0                 Int32
30    v LGH: Object                          Object
31        enable: false                      Boolean
32        emergency: true                    Boolean
33        sampling: true                     Boolean
34        avg_mode: false                    Boolean
35        avg_limit: 1                       Int32
36        period: 900                        Int32
37        emergency_value: 2                 Int32
38    timestamp: 1634912447                  Int32
39  > variables_interrupts: Object           Object
```


Figure 73: Dashboard change streams logs

Chapter 6

Conclusion

The Internet of Things (IoT) is a rapidly expanding field that promises to transform how we live and work by connecting physical objects to the internet and enabling them to communicate and share data. This has led to an increasing number of IoT devices being used for a wide range of applications, including environmental monitoring, data acquisition, and machine learning. As the number of IoT devices increases, the importance of cloud-based architectures that are efficient and dependable for managing and analyzing the data they collect grows.

This dissertation was developed as part of the Link4S project, which seeks to generate new scientific knowledge regarding the design, development, and testing of a new generation of connectivity devices and their corresponding platforms. A segment of the cloud infrastructure had already been developed alongside end-devices prior to this dissertation. The architecture of these ultra-low-power monitoring systems is typical of the Internet of Things. They spend most of their lives in a dormant state, awakening only to measure environmental variables using multiple sensors or to signal an anomaly, using an NB-IoT transceiver to communicate with the cloud and a microcontroller as the system's brain.

The original cloud architecture consisted of a microservices-based architecture that utilized the Blackwing framework and only implemented microservices for parsing and storing device-sent data in a MongoDB database. This approach made it difficult to analyze device data because there were no available user interfaces, and all device interaction, such as sending commands, had to be performed manually using Python scripts.

In the context of this dissertation, the emphasis was on improving the original cloud architecture to an

CMP compliant microservices-based architecture. The CMP architecture enables interoperability between IoT ecosystems by providing the ability to "link any data source(s) to any data sink(s), independently of the data formats and/or protocols that are used", and can be implemented modularly using microservices. Additionally, a dashboard application was required to facilitate the management and analysis of device data. To achieve the proposed goals, a literature review was made on the subject of microservices architectures, to better understand how they work and their benefits over monolithic architectures, and the best practices on the design of microservices. On the topic of dashboards, the best practices for dashboard design and the frameworks best suited to the task at hand were evaluated.

The requirements for this project, which influenced the development of this dissertation, were related to the cloud and data visualization. As new microservices were created to implement the components of the CMP architecture, and a data visualization and device management platform was created in the form of a Dashboard, the project successfully met all requirements.

To enhance the functionality of the Dashboard application, additional components such as real-time updates and a time series cache were added. In addition, the creation of REST APIs to access and manipulate device and user data enables the development of additional applications given the data's ease of access. Testing and validating the final cloud architecture, including validation with project partners, and deploying the cloud architecture on a production-ready server were also accomplished.

In the context of the Link4S project, the implementation of this microservices-based architecture has resulted in a more efficient and effective cloud architecture that is better suited to manage and analyze data from a large number of IoT devices. This work contributes to the ongoing efforts to improve the reliability and scalability of IoT systems and has the potential to have an impact on the IoT and cloud computing fields.

The end-result of this project in an CMP-based cloud architecture that is more autonomous, secure, and reconfigurable at runtime via CLI commands. The cloud architecture is based on microservices and enables scalability to a large number of devices; end-devices are fully integrated with the cloud and have an estimated field lifetime of 23 years. It supports secure end-to-end communications using the RSA and AES encryption algorithms, as well as FOTA updates and other configurations. Overall, this dissertation emphasizes the significance of microservices architecture in the IoT space and how the CMP architecture can serve as a base for future IoT architectures.

6.1 Future work

Containerization technology such as Docker could be used to move cloud components into containers, and these containers could be managed using a container orchestration service such as Kubernetes. Containers are a perfect fit for microservices due to their decentralized nature, as demonstrated in the State of the Art chapter, and this deployment method can greatly improve their scalability and reliability. This approach could also be combined with a CI/CD pipeline to streamline the development process, which includes automatic builds on a dedicated server and integration tests to ensure that new code changes do not break existing code.

Using a database for time series purposes, such as MongoDB's own time series implementation or TimescaleDB, can also improve the storage of sensor data and make the retrieval of data more efficient.

Adding more features to the dashboard could overwhelm the current framework, as its capabilities for large and complex web applications are somewhat limited. In this situation, it may be preferable to migrate the current implementation to JavaScript using the React framework and HTML/CSS. As the current Dash framework is based on React, the majority of the existing codebase can be leveraged to facilitate this transition.

Bibliography

[1] S. B. Atitallah, M. Driss, and H. B. Ghzela, “Microservices for data analytics in iot applications: Current solutions, open challenges, and future research directions,” *Procedia Computer Science*, vol. 207, pp. 3938–3947, 2022, Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 26th International Conference KES2022, issn: 1877-0509. doi: https://doi.org/10.1016/j.procs.2022.09.456. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050922013503.

[2] K. Dineva and T. Atanasova, “Architectural ml framework for iot services delivery based on microservices,” in *Distributed Computer and Communication Networks*, V. M. Vishnevskiy, K. E. Samouylov, and D. V. Kozyrev, Eds., Cham: Springer International Publishing, 2020, pp. 698–711, isbn: 978-3-030-66471-8.

[3] S. P. R. Asaithambi, R. Venkatraman, and S. Venkatraman, “Mobda: Microservice-oriented big data architecture for smart city transport systems,” *Big Data and Cognitive Computing*, vol. 4, no. 3, 2020, issn: 2504-2289. doi: 10.3390/bdcc4030017. [Online]. Available: https://www.mdpi.com/2504-2289/4/3/17.

[4] Z. Li, D. Seco, and A. E. Sánchez Rodríguez, “Microservice-oriented platform for internet of big data analytics: A proof of concept,” *Sensors*, vol. 19, no. 5, 2019, issn: 1424-8220. doi: 10.3390/s19051134. [Online]. Available: https://www.mdpi.com/1424-8220/19/5/1134.

[5] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, “The pains and gains of microservices: A systematic grey literature review,” *Journal of Systems and Software*, vol. 146, pp. 215–232, 2018, issn: 0164-1212. doi: https://doi.org/10.1016/j.jss.2018.09.082. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121218302139.

[6] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018. doi: 10.1109/MS.2018.2141039.

[7] COMPETE 2020. (2022). (link4s)ustainability: Um sistema de conectividade de nova geração, [Online]. Available: https://www.compete2020.gov.pt/noticias/detalhe/Newsletter_46122_(Link4S)ustainability (visited on 06/01/2023).

[8] J. Borges, "Robust software services for iot embedded systems," Master's thesis, University of Minho, 2021.

[9] NOS Comunicações. (2022). (link4s)ustainability, [Online]. Available: https://www.nos.pt/institucional/PT/a-nos/inovacao/Paginas/(Link4S)ustainability.aspx (visited on 06/01/2023).

[10] P. Mestre, E. Dogruluk, C. Ferreira, R. Cordeiro, J. Valente, S. Branco, B. Gaspar, and J. Cabral, "A Platform Architecture for m-Health Internet of Things Applications," in *Wireless Mobile Communication and Healthcare. MobiHealth 2022. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, Springer, Cham, 2022.

[11] N. Dragoni, S. Giallorenzo, A. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds., Springer, Sep. 2017. [Online]. Available: https://hal.inria.fr/hal-01631455.

[12] H. J. La, J. S. Bae, S. H. Chang, and S. D. Kim, "Practical methods for adapting services using enterprise service bus," in *Proceedings of the 7th International Conference on Web Engineering*, ser. ICWE'07, Springer-Verlag, 2007, pp. 53–58, isbn: 9783540735960.

[13] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *2015 10th Computing Colombian Conference (10CCC)*, 2015, pp. 583–590. doi: 10.1109/ColumbianCC.2015.7333476.

[14] J. Lewis and M. Fowler. (2014). Microservices, [Online]. Available: https://martinfowler.com/articles/microservices.html (visited on 12/08/2022).

[15] F. Auer, V. Lenarduzzi, M. Felderer, and D. Taibi, "From monolithic systems to microservices: An assessment framework," *Information and Software Technology*, vol. 137, Sep. 2021, issn: 09505849. doi: `10.1016/j.infsof.2021.106600`.

[16] P. Krivic, P. Skocir, M. Kusek, and G. Jezic, "Microservices as agents in iot systems," in *Agent and Multi-Agent Systems: Technology and Applications*, Jan. 2018, pp. 22–31, isbn: 978-3-319-59393-7. doi: `10.1007/978-3-319-59394-4_3`.

[17] R. M. Munaf, J. Ahmed, F. Khakwani, and T. Rana, "Microservices architecture: Challenges and proposed conceptual design," in *2019 International Conference on Communication Technologies (ComTech)*, 2019, pp. 82–87. doi: `10.1109/COMTECH.2019.8737831`.

[18] B. Butzin, F. Golatowski, and D. Timmermann, "Microservices approach for the internet of things," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016, pp. 1–6. doi: `10.1109/ETFA.2016.7733707`.

[19] S. Newman, *Building Microservices*, 2nd. O'Reilly Media, Inc., 2021, isbn: 9781492034025.

[20] M. Waseem, P. Liang, M. Shahin, A. D. Salle, and G. Márquez, "Design, monitoring, and testing of microservices systems: The practitioners' perspective," *Journal of Systems and Software*, vol. 182, Dec. 2021, issn: 01641212. doi: `10.1016/j.jss.2021.111061`.

[21] Microsoft. (2021). Using domain analysis to model microservices, [Online]. Available: `https://docs.microsoft.com/en-us/azure/architecture/microservices/model/domain-analysis` (visited on 12/08/2022).

[22] E. Norelus. (2019). Implementing domain-driven design for microservice architecture, [Online]. Available: `https://medium.com/design-and-tech-co/implementing-domain-driven-design-for-microservice-architecture-26eb0333d72e` (visited on 12/08/2022).

[23] C. Ramalingam. (2020). Building domain driven microservices, [Online]. Available: `https://medium.com/walmartglobaltech/building-domain-driven-microservices-af688aa1b1b8` (visited on 12/08/2022).

[24] Microsoft. (2021). Using tactical ddd to design microservices, [Online]. Available: `https://docs.microsoft.com/en-us/azure/architecture/microservices/model/tactical-ddd` (visited on 12/08/2022).

[25] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[26] Microsoft. (2021). Identifying microservice boundaries, [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/microservices/model/microservice-boundaries (visited on 12/08/2022).

[27] IBM. (2021). Remote procedure call, [Online]. Available: https://www.ibm.com/docs/en/aix/7.3?topic=concepts-remote-procedure-call (visited on 15/08/2022).

[28] gRPC. (2021). Introduction to grpc, [Online]. Available: https://grpc.io/docs/what-is-grpc/introduction (visited on 15/08/2022).

[29] Red Hat. (2020). What is a rest api? [Online]. Available: https://www.redhat.com/en/topics/api/what-is-a-rest-api (visited on 15/08/2022).

[30] IBM. (2020). Message brokers, [Online]. Available: https://www.ibm.com/cloud/learn/message-brokers (visited on 15/08/2022).

[31] Apache Kafka. (2021). Kafka 3.0 documentation, [Online]. Available: https://kafka.apache.org/documentation (visited on 15/08/2022).

[32] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014. doi: 10.1109/MCC.2014.51.

[34] F. Paraiso, S. Challita, Y. Al-Dhuraibi, and P. Merle, "Model-Driven Management of Docker Containers," in *9th IEEE International Conference on Cloud Computing (CLOUD)*, IEEE, Ed., San Francisco, United States, Jun. 2016, pp. 718 –725. doi: 10.1109/CLOUD.2016.0100. [Online]. Available: https://hal.inria.fr/hal-01314827.

[33] Docker. (2021). Docker overview, [Online]. Available: https://docs.docker.com/get-started/overview (visited on 14/08/2022).

[35] IBM. (2021). Container orchestration, [Online]. Available: https://www.ibm.com/cloud/learn/container-orchestration (visited on 14/08/2022).

[36] Stratoscale. (2019). Everything kubernetes: A practical guide, [Online]. Available: https://www.stratoscale.com/resources/ebooks/everything-kubernetes-5/ (visited on 15/08/2022).

[37] Kubernetes. (2021). Kubernetes components, [Online]. Available: https://kubernetes.io/docs/concepts/overview/components (visited on 14/08/2022).

[38] S. Few, *Information dashboard design: Effective visual communication of data*, en. Sebastopol, CA: O'Reilly Media, 2006, isbn: 9780596100162.

[39] O. M. Yigitbasioglu and O. Velcu, "A review of dashboards in performance management: Implications for design and research," *International Journal of Accounting Information Systems*, vol. 13, no. 1, pp. 41–59, 2012, issn: 1467-0895. doi: https://doi.org/10.1016/j.accinf.2011.08.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1467089511000443.

[40] A. Sarikaya, M. Correll, L. Bartram, M. Tory, and D. Fisher, "What do we talk about when we talk about dashboards?" *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 1, pp. 682–692, 2019. doi: 10.1109/TVCG.2018.2864903.

[41] K. Pauwels, T. Ambler, B. Clark, P. LaPointe, D. Reibstein, B. Skiera, B. Wierenga, and T. Wiesel, "Dashboards as a service : Why, what, how, and what research is needed?" *Journal of Service Research*, vol. 12, pp. 175–189, Oct. 2009. doi: 10.1177/1094670509344213.

[42] E. B. Goldstein, *Sensation and Perception*. Belmont, CA: Wadsworth Publishing, 2006.

[43] P. Moore and C. Fitz, "Gestalt theory and instructional design," *Journal of Technical Writing and Communication*, vol. 23, no. 2, pp. 137–157, 1993. doi: 10.2190/G748-BY68-L83T-X02J. eprint: https://doi.org/10.2190/G748-BY68-L83T-X02J. [Online]. Available: https://doi.org/10.2190/G748-BY68-L83T-X02J.

[44] W. Dilla and P. Steinbart, "The effects of alternative supplementary display formats on balanced scorecard judgments," *International Journal of Accounting Information Systems*, vol. 6, pp. 159–176, Sep. 2005. doi: 10.1016/j.accinf.2004.12.002.

[45] T. Amer, "An experimental investigation of multi-cue financial information display and decision making.," *Journal of Information Systems*, vol. 5, no. 2, pp. 18–34, 1991.

[46] I. Vessey and D. Galletta, "Cognitive fit: An empirical study of information acquisition," *Information Systems Research*, vol. 2, no. 1, pp. 63–84, 1991, issn: 10477047, 15265536. [Online]. Available: http://www.jstor.org/stable/23010613.

[47] I Vessey and D Galletta, "Cognitive fit: An empirical study of information acquisition," *Information Systems Research*, vol. 2, no. 1, pp. 63 –84, Jan. 1991. [Online]. Available: http://d-scholarship.pitt.edu/13949/.

[48] N. S. Umanath and I. Vessey, “Multiattribute data presentation and human judgment: A cognitive fit perspective*,” *Decision Sciences*, vol. 25, no. 5-6, pp. 795–824, 1994. doi: https://doi.org/10.1111/j.1540-5915.1994.tb01870.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1540-5915.1994.tb01870.x. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1540-5915.1994.tb01870.x.

[49] L. Diamond and F. Lerch, “Fading frames: Data presentation and framing effects*,” *Decision Sciences*, vol. 23, pp. 1050 –1071, Jun. 2007. doi: 10.1111/j.1540-5915.1992.tb00435.x.

[50] Thingsboard. (2022). Thingsboard documentation, [Online]. Available: https://thingsboard.io/docs (visited on 22/08/2022).

[51] Ubidots. (2022). Ubidots documentation, [Online]. Available: https://ubidots.com/docs (visited on 22/08/2022).

[52] OpenJS Foundation, Node-RED contributors. (2022). Node-red documentation, [Online]. Available: https://nodered.org/docs (visited on 22/08/2022).

[53] Grafana Labs. (2022). Grafana documentation, [Online]. Available: https://grafana.com/docs (visited on 22/08/2022).

[54] Plotly. (2022). Dash documentation & user guide, [Online]. Available: https://dash.plotly.com (visited on 22/08/2022).

[55] B. Mataloto, J. C. Ferreira, and N. Cruz, “Lobems—iot for building and energy management systems,” *Electronics*, vol. 8, no. 7, 2019, issn: 2079-9292. doi: 10.3390/electronics8070763. [Online]. Available: https://www.mdpi.com/2079-9292/8/7/763.

[56] R. Chetty, S. M.S., S. D., and R. R. Marathe, “A lorawan based open source iot solution for monitoring rural electrification policy,” Institute of Electrical and Electronics Engineers Inc., 2020, pp. 888–890, isbn: 9781728131870. doi: 10.1109/COMSNETS48256.2020.9027490.

[57] A. Ali, C. Coté, M. Heidarinejad, and B. Stephens, “Elemental: An open-source wireless hardware and software platform for building energy and indoor environmental monitoring and control,” *Sensors*, vol. 19, p. 4017, Sep. 2019. doi: 10.3390/s19184017.

[58] L. O. Aghenta and M. T. Iqbal, "Design and implementation of a low-cost, open source iot-based scada system using esp32 with oled, thingsboard and mqtt protocol," *AIMS Electronics and Electrical Engineering*, vol. 4, no. 1, pp. 57–86, 2020, issn: 2578-1588. doi: 10.3934/ElectrEng.2020.1.57. [Online]. Available: https://www.aimspress.com/article/doi/10.3934/ElectrEng.2020.1.57.

[59] P. Diaz, K. Potter, G. Johnson, and A. Lopez, "Uncertainty visualization for renewable energy potential," in *Proceedings of the Twelfth ACM International Conference on Future Energy Systems*, Association for Computing Machinery, Jun. 2021, pp. 335–340, isbn: 9781450383332. doi: 10.1145/3447555.3466593. [Online]. Available: https://doi.org/10.1145/3447555.3466593.

[60] S. Branco. (2018). Blackwing specifications, [Online]. Available: https://blackwing.readthedocs.io/en/latest (visited on 21/12/2022).

[61] NordPass. (2022). Learning password security jargon: Password peppering, [Online]. Available: https://nordpass.com/blog/pepper-password (visited on 10/10/2022).

[62] A. Biryukov, D. Dinu, and D. Khovratovich, "Argon2: New generation of memory-hard functions for password hashing and other applications," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016, pp. 292–302. doi: 10.1109/EuroSP.2016.31.

[63] MongoDB. (2022). Mongodb, [Online]. Available: https://www.mongodb.com (visited on 10/10/2022).

[64] Flask. (2022). Welcome to flask — flask documentation (2.2.x), [Online]. Available: https://flask.palletsprojects.com/en/2.2.x (visited on 12/10/2022).

[65] React. (2022). A javascript library for building user interfaces, [Online]. Available: https://reactjs.org/ (visited on 12/10/2022).

[66] Plotly. (2022). Plotly open source graphing library for python, [Online]. Available: https://plotly.com/python (visited on 12/10/2022).

[67] MDN Web Docs. (2022). Mvc - mdn web docs glossary: Definitions of web-related terms, [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/MVC (visited on 12/10/2022).

[68] Plotly. (2022). Basic dash callbacks, [Online]. Available: https://dash.plotly.com/basic-callbacks (visited on 20/10/2022).

[69] UXPin. (2022). Ui design and prototyping tool, [Online]. Available: https://www.uxpin.com (visited on 24/10/2022).

[70] Plotly. (2022). Live updating components, [Online]. Available: https://dash.plotly.com/live-updates (visited on 25/10/2022).

[71] MongoDB. (2022). An introduction to change streams, [Online]. Available: https://www.mongodb.com/blog/post/an-introduction-to-change-streams (visited on 25/10/2022).

[72] Flask Cache. (2022). Flask-caching documentation, [Online]. Available: https://flask-caching.readthedocs.io/en/latest (visited on 28/10/2022).

[73] G. Duarte. (2009). Page cache, the affair between memory and files, [Online]. Available: https://manybutfinite.com/post/page-cache-the-affair-between-memory-and-files (visited on 29/10/2022).

[74] R. Imankulov. (2022). Time series caching with python and redis, [Online]. Available: https://roman.pt/posts/time-series-caching (visited on 29/10/2022).

[75] AWS. (2022). Caching strategies - amazon elasticache, [Online]. Available: https://docs.aws.amazon.com/AmazonElastiCache/latest/mem-ug/Strategies.html (visited on 29/10/2022).

[76] Redis. (2022). Introduction to redis, [Online]. Available: https://redis.io/docs/about (visited on 29/10/2022).

[77] Plotly. (2022). Pattern-matching callbacks, [Online]. Available: https://dash.plotly.com/pattern-matching-callbacks (visited on 20/11/2022).

[78] J. Van Der Donckt, J. Van Der Donckt, E. Deprost, and S. Van Hoecke, *Plotly-resampler: Effective visual analytics for large time series*, 2022. doi: 10.48550/ARXIV.2206.08703. [Online]. Available: https://arxiv.org/abs/2206.08703.

[79] pydantic. (2022). Models, [Online]. Available: https://docs.pydantic.dev/usage/models (visited on 22/12/2022).