DM

Low-cost Apparatus for Real-time Sampling of Water Column based on Internet of Underwater Things

MASTER DISSERTATION

João Pedro Baptista Valente Pereira Freitas
MASTER IN INFORMATICS ENGINEERING

UNIVERSIDADE da MADEIRA
A Nossa Universidade
www.uma.pt

September | 2025

Low-cost Apparatus for Real-time Sampling of Water Column based on Internet of Underwater Things

MASTER DISSERTATION

João Pedro Baptista Valente Pereira Freitas
MASTER IN INFORMATICS ENGINEERING

SUPERVISOR
Marko Radeta

Resumo

Com apenas 5% das profundezas oceânicas exploradas, o desafio de compreender os ecossistemas que residem na biodiversidade marinha e o seu impacto nas alterações climáticas é ainda muito presente. As variáveis oceanográficas essenciais são, tipicamente, avaliadas através de tecnologias de deteção remota e drones. Contudo, estes dispositivos oferecem ou uma baixa resolução com ampla cobertura espacial, ou uma elevada resolução com cobertura espacial limitada. Deste modo, esta dissertação explora a utilização da Internet das Coisas Subaquáticas e o seu papel na facilitação da amostragem da coluna de água, com o intuito de desenvolver uma importante ferramenta a bordo de embarcações de pequeno porte, através de um sistema inovador baseado na amostragem em tempo real da coluna de água, por meio de uma conexão por cabo, aumentando assim a resolução espacial de indicadores essenciais à biodiversidade. O sistema integra vários sensores com capacidade para medir diferentes parâmetros de forma interoperável, garantindo compatibilidade com diversos dispositivos, como portáteis, telemóveis e tablets, permitindo uma maior flexibilidade para especialistas a bordo de embarcações mais pequenas. A validação do sistema demonstra o seu potencial como uma ferramenta para a monitorização marinha.

Keywords: Internet das Coisas Subaquáticas · Amostragem em Tempo Real da Coluna de Água · Tecnologia Marinha de Baixo Custo · Transmissão de Dados Marinhos · Sistemas de Monitorização Ambiental

Abstract

With only 5% of the ocean depths charted, the challenge of understanding marine biodiversity and its role in climate change remains. Essential oceanographic variables are typically assessed using remote sensing technologies and drones. However, these devices offer either low resolution with broad spatial coverage or high resolution with limited spatial coverage. This dissertation explores the use of the Internet of Underwater Things and its role in facilitating real-time sampling of the water column with the intent of developing, onboard small vessels, an innovative system based on tethered real-time sampling, thereby enhancing the spatial resolution of key biodiversity indicators. The system integrates multiple interoperable sensors capable of measuring various parameters and is designed to ensure compatibility with a range of devices such as laptops, smartphones, and tablets. This flexibility supports efficient fieldwork by specialists operating on small boats. The system validation highlights its potential as a tool for environmental monitoring in marine ecosystems.

Keywords: Internet of Underwater Things · Real-Time Water Column Sampling · Low-cost Marine Technology · Oceanic Data Transmission · Environmental Monitoring Systems

Acknowledgments

I would like to express my deepest gratitude to everyone who has accompanied me through this academic journey.

First and foremost, I would like to thank my family for their unconditional love and support through this tough endeavour, and for the many, many challenges they saw me endure while still pushing me forward to reach my academic goals. Without their support, none of this would have been possible.

Secondly, I would like to thank my girlfriend whom, through kindness, support and love helped me push myself harder and to never give up, while always being present to help keep me grounded and hopeful throughout this journey.

To all my friends and colleagues, thank you for all the laughs, jokes and good moments through this academic effort.

I would also like to show my deep gratitude towards my supervisor, Marko Radeta, for all the support, patience and help throughout this thesis.

I also extend my sincere gratitude to ARDITI and MARE for their invaluable support throughout the development of this work. Their provision of essential equipment and access to domain experts was crucial to the quality and depth of this thesis.

Finally, this dissertation is supported by the following Horizon Europe projects: CLIMAREST (101093865), SEAMPHONI (101206245) and TWILIGHTED (101158714). Dissertation also had the support of FCT through the strategic project UIDB/04292/2020 awarded to MARE and through project LA/P/0069/2020 granted to the Associate Laboratory ARNET. Dissertation has been conducted with the support of the Wave Labs[1] - center for deep tech.

[1] `https://wave-labs.org`

Table of Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

ADC Analog-to-Digital Converter

API Application Programming Interface

ARM Architecture Reference Model

ASGI Asynchronous Server Gateway Interface

ASV Autonomous Surface Vehicle

AUV Autonomous Underwater Vehicle

CDOM Coloured Dissolved Organic Matter

CPU Central Processing Unit

CS Chip Select/Slave Select

CTD Conductivity, Temperature, Depth

DHCP Dynamic Host Configuration Protocol

DO Dissolved Oxygen

EBVs Essential Biodiversity Variables

EOVs Essential Oceanic Variables

FTP File Transfer Protocl

GND Ground

GPIO General Purpose Input Output

GPRS General Packet Radio Service

GUI Graphical User Interface

HTTP Hypertext Transfer Protocol

I2C Inter-Integrated Circuit

IPC Inter-Process Commnunication

IoT Internet of Things

IoUT Internet of Underwater Things

JSON JavaScript Object Notation

LAN Local Area Network

LoRa Long Range

MISO Master In Slave Out

MOSI Master Out Slave In

MPSS Multi-Purpose Sensing System

MQTT Message Queuing Telemetry Transport

REST Representational State Transfer

RF Radio Frequency

ROVs Remotely Operated Vehicles

RPi Raspberry Pi

RTC Real-Time Clock

RXD Receive

SCL Serial Clock

SCLK Serial Clock

SDA Serial Data

SMTP Simple Mail Transfer Protocol

SPI Serial Peripheral Interface

SSS Sea Surface Salinity

SST Sea Surface Temperature

TDS Total Dissolved Solids

TRL Technology Readiness Level

TXD Transmit

UART Universal Asynchronous Receiver/Transmitter

VCC Voltage at the Common Collector

1 Introduction

Covering almost two-thirds of the Earth's surface, the oceans remain largely undiscovered, with only 5% of their depths thoroughly explored [1]. Understanding oceans provides insight not only into ecosystems and biodiversity as a whole but also offers a broader understanding of how to mitigate climate change [2, 3].

Despite advancements in sensing technologies, existing methods for assessing Essential Biodiversity Variales (EBVs) and Essential Oceanic Variables (EOVs), including remote sensing and drones, remain constrained in their spatial and temporal resolution [4–6]. These technologies often present a trade-off between high-resolution data and broad spatial coverage, and struggle to capture the complex dynamic of oceanic processes [7].

Moreover, existing oceanic monitoring systems face significant technological and logistical limitations. Issues such as saltwater corrosion, pressure tolerance, waterproofing, and power autonomy impose substantial maintenance demands [6]. Satellite technologies, though valuable for large-scale monitoring, are limited by low pixel resolution and are often hindered by atmospheric interference [4, 8]. Conventional equipment like Conductivity, Temperature and Depth (CTD) Rosette Samplers provide high-precision measurements but require sophisticated deployment mechanisms, large research vessels, and expert operators [5, 9]. Similarly, Remotely Operated Vehicles (ROVs) can collect deep-sea data but are hindered by high energy demands, navigation challenges, and operational complexity [6, 10]. Traditional water quality monitoring systems, still used in certain contexts, remain largely manual, time-intensive, and inadequate for modern, scalable marine research [11]. As such, there is an urgent need for novel underwater observatories capable of continuous, high-resolution monitoring throughout the water column. These observatories should be cost-effective, adaptable to diverse marine conditions, and operable from smaller sea vessels to broaden accessibility.

In response to these limitations, this body of work intends to leverage the current Internet of Things (IoT) and Internet of Underwater Things (IoUT) efforts [12] as a means of facilitating real-time water column sampling. While prior IoUT implementations have focused primarily on stationary surface or seabed settings, this dissertation aims to extend their capabilities to vertical profiling within the water column, a domain that remains largely untapped.

The system was designed for deployment from small-scale vessels, with the goal of increasing spatial and temporal resolution of the following oceanic variables: conductivity, salinity, Total Dissolved Solids (TDS), temperature, Dissolved Oxygen (DO) and depth [13–15]. A key focus was ensuring ease-of-use and real-time data accessibility to support marine biologists during both field collection and post-processing.

To that end, the apparatus integrates real-time tethered data transmission from a microprocessor, while providing live visualization through a cross-device web interface accessible on personal devices. This facilitates researcher work by allowing for data to be monitored and analysed on board, while also enabling more rigorous data examination in a laboratory, a-posteriori. By prioritizing usability, interoperability, and portability, the system is intended to democratize access to high-resolution ocean monitoring tools, particularly for underfunded or small-scale research initiatives.

1.1 Objectives and Research Question

This thesis is driven by the following research question: "*How can the IoUT be leveraged to design a compact, easy-to-use, cost-effective, and real-time monitoring system for profiling the water column from small-scale sea vessels?*". To address this question, the research introduces a low-cost, and portable water sampling system with the intent to fulfil the subsequent objectives:

- **User-friendly interface:** To increase the accessibility to a larger community involved in water column sampling by utilizing an intuitive application for straightforward data retrieval.
- **High-resolution sampling:** To expand geographical coverage by allowing more frequent sampling by e.g. fishing vessels for deep waters ($> 200\ m$).
- **Cost-effective design:** To provide a design with affordable hardware for wider adoption to research initiatives with lower funds. Intially, distributing the system or sharing development files and decisions might help spread the adoption.
- **Simplified deployment:** To reduce logistical limitations by being compact and lightweight, allowing it to be easily transported on smaller vessels and operated without the need for system-specific domain experts.

1.2 Document Structure

This body of work is composed by the following structure:

- **Related Work:** Reviews existing water sampling methods, IoUT efforts in water sampling, employed architectures in the development of IoT systems, and the Raspberry Pi (RPi), a microprocessor employed in the development of said systems.
- **Methodology:** Details the entire implementation process, from user requirements, to hardware and software integration, system architecture, development and the prepared experimental validation.
- **Results:** Describes gathered results and feedback given by domain specific professionals on what was gathered.
- **Discussion:** Debates on the results from the experiment and compares the developed system to market alternatives, while also acknowledging constraints and discussing future system improvements and testing.
- **Conclusion:** Summarizes the conducted work.

2 Related Work

This chapter starts with a review of current technologies and techniques used for sampling water column variables, including Sattelites, CTD Rosettes and ROVs as well as recent advancements within the IoUT. It then examines architectural strategies commonly employed in the design of IoT systems. Given that the IoUT is considered a subset of the broader IoT, these strategies are equally relevant in the development of underwater systems. Finally, the chapter provides an in-depth discussion of the RPi, which serves as a central hardware component in many IoT systems, including the one developed in this work.

2.1 Methods for Sampling Water Variables

Water sampling for oceanography and limnology has been traditionally conducted through various methods, including the remote sensing via Satellites, the deployment of instruments such as the CTD Rosette, the usage of ROVs, and through traditional means [6, 16].

Satellite Technology

Remote sensing provides a comprehensive global perspective of the ocean and atmosphere, facilitating the monitoring of various water properties, including Sea Surface Temperature (SST), Sea Surface Salinity (SSS), DO, chlorophyll-a concentration, turbidity, and the coefficient of Coloured Dissolved Organic Matter (CDOM) [4,16], among others. These measurements are achieved through optical, thermal, and microwave sensors [4], which are able to penetrate the Earth's atmosphere. Satellites offer extensive spatial coverage, which is particularly valuable for studying large-scale phenomena. However, their utility is often constrained by the coarse spatial resolution of many sensors. For example, while Moderate Resolution Imaging Spectroradiometer (MODIS) has a resolution of approximately 1 km, Landsat 8's Operational Land Imager (OLI) has a resolution of 30 m, Thermal Infrared Sensor (TIRS) has a resolution of 100 m resampled to 30 m, and Sentinel-2 Multispectral Imager (MSI) has a resolution of $10 - 30$ m [8, 16]. This limits the ability to capture small-scale features, especially in complex coastal waters. Geostationary satellites, such as Geostationary Ocean Color Imager (GOCI), provide frequent observations of specific regions but face similar spatial resolution challenges and are restricted to fixed areas [4, 8]. Atmospheric conditions, including cloud cover and aerosol interference, further reduce the effectiveness of satellite-based measurements [8].

Satellite data indirectly estimates numerous water quality parameters through proxies or correlations and is limited to surface measurements, failing to account for vertical water-column variability. Sampling errors can also arise due to mismatches in spatial scales between satellite observations and in-situ measurements [17]. Given the surface-only nature of satellite data and the inability to directly measure many essential water quality parameters, research suggests that field measurements with devices such as the CTD Rosette remain crucial [8] and that an integration between these devices and remote sensing is essential to achieve a more comprehensive understanding of water sampling. As such, this technology is better suited as a complementary tool, rather than an independent means of study for marine environments [17].

CTD Rosette

Instruments such as the CTD Rosette are fundamental in oceanographic research, enabling the collection of high-resolution data on seawater properties and water samples from various depths [9]. These devices consist of a CTD sensor package integrated within a frame that holds multiple water sampling bottles. In addition to measuring conductivity, temperature, and depth, some CTD systems often include sensors to assess DO, pH, turbidity, CDOM, and chlorophyll-a [5]. A conducting cable (e.g, Ethernet, umbilical or coaxial) connects the CTD Rosette to a shipboard computer, facilitating real-time data acquisition and precise control over the sampling process [9]. The operational procedure of a CTD Rosette begins with its deployment into the water via a conducting cable, during which sensors continuously transmit real-time data on seawater properties, creating detailed profiles of the water column [9,18]. During upcast, sampling bottles are selectively closed at predetermined depths to collect samples that are subsequently transported to a laboratory for detailed chemical and biological analyses [5]. Some CTD Rosette systems can make readings in extreme depths going up to 4000 m [9].

Despite its indispensable role in oceanographic research, the CTD Rosette has several limitations. For instance, trace element contamination may occur due to metallic components in the frame or other materials [19]. Additionally, sampling while the rosette is in motion can lead to incomplete flushing of the bottles and inadequate equilibration of the CTD sensors, particularly in regions with sharp gradients in water properties [19]. The system is also time-intensive, with a single CTD cast typically requiring between two to five hours to collect a complete set of data, depending on water depth, which can limit the number of samples that can be collected in a

given time frame [9, 19]. This limitation further adds to a limited spatial coverage. Furthermore, mechanical failures, such as improper alignment or obstructions in the sampling module, can occasionally result in the failure of the bottle closure mechanism [18]. Another significant drawback lies in the high cost and logistical complexity associated with their operation. For instance, a single CTD Rosette instrument (with the exclusion of the winch and certain bottle modifications) was reported to cost $133804 in 2003, which would correspond to approximately $235000 in today's terms [19]. In addition to equipment pricing, a significant financial investment is also required for logistical support, as the operation of these systems demands specialized training and experienced personnel [5].

ROVs

The ROVs are versatile and effective tools for collecting data and assessing the conditions of aquatic environments with great adaptability, which allows the collection of diverse types of information in various depths, including visual footage, water quality parameters, and underwater acoustics [10, 20]. The ROVs are commonly equipped with video cameras that facilitate the retrieval of visual data from underwater environments [20] which enables the observation of aquatic life, identification of solid waste and debris, and documentation of the impacts of fishing activities. For water quality assessments, ROVs deploy multi-parameter probes capable of measuring various physical and chemical properties of water, such as conductivity, salinity, temperature, DO, pH, turbidity, depth, chlorophyll-a, and CDOM [7, 10]. These capabilities enable researchers to collect comprehensive data on the health and composition of aquatic systems. In underwater acoustics, ROVs are often equipped with hydrophones to detect and measure underwater noise, capturing sounds generated by machinery, explosions, seismic activity, and natural phenomena [10]. This information is crucial for assessing the effects of noise pollution on marine ecosystems. Additional sensors, including Doppler Velocity Logs (DVL), Inertial Measurement Units (IMU), and GPS systems, enhance navigation and data collection, depending on the research objectives [10, 20]. Data collected by ROV-mounted sensors is transmitted to an onboard computer or a surface control unit via a tethered (e.g Ethernet, umbilical) connection, allowing for real-time monitoring and analysis through various open-source programs [6, 10]. This data can also be logged for post-processing, enabling detailed analyses after field operations.

A widely utilized ROV model, BlueROV2, is regarded as offering a cost-effective yet high-performance drone capable of different oceanic tasks [21]. The base configuration is priced at $4600, but the total cost can exceed $10000 depending on the integration of additional sensors and peripherals [21]. Despite this, it remains significantly more affordable than many commercial ROV alternatives, as these typically range from $\$10000 - \$100000$ for models capable of operating at depths of up to 300 m (with some more expensive models capable of going to depths such as 7000 m [22]). The BlueROV2 employs a RPi as its main computing unit, which is responsible for handling system logic and interfacing with onboard components. Communication between the ROV and the control station is established via a tethered Ethernet connection, ensuring reliable data transfer and command execution to a topside control software, QGroundControl [23].

From a software perspective, the BlueROV2 runs BlueOS, an open-source operating system based on Linux, that is actively maintained and expanded by the marine robotics community [24]. BlueOS is designed to accommodate a wide range of hardware configurations and mission requirements [24]. Its user interface, shown in Figure 1, includes a sidebar on the left for accessing system components, installed extensions, general settings, and a shutdown button. The top bar displays various widgets such as a Central Processing Unit (CPU) and disk usage on the left and operational indicators (e.g., night mode status) on the right. The main content area dynamically displays the selected page, with an overview of all available modules shown in the example.

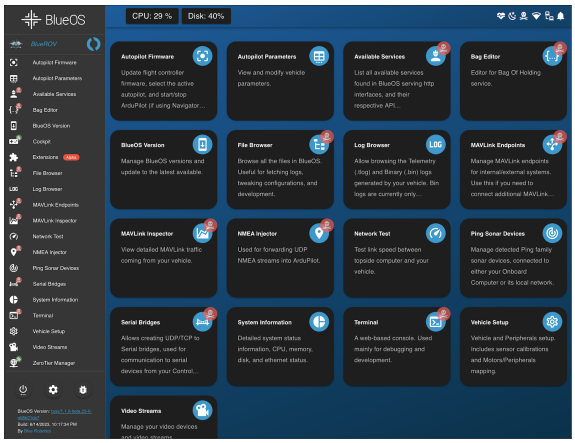


Figure. 1: BlueOS Interface [25]

BlueOS supports integration with a broad array of sensors commonly used in oceanographic research, including those for conductivity, salinity, temperature, DO and pressure [26], however

by default only temperature and pressure sensors are already integrated [27], with extra work needed to add sensors, which also leads to an increase in cost. These integrated sensors can't be natively calibrated within the systems capabilities (with the exception of the pressure sensor) [28]. As illustrated in Figure 2, the interface displays all connected ROV components along with real-time sensor readings, including temperature, freshwater pressure and barometric pressure, from the sensors that are already integrated by default into the drone [26].

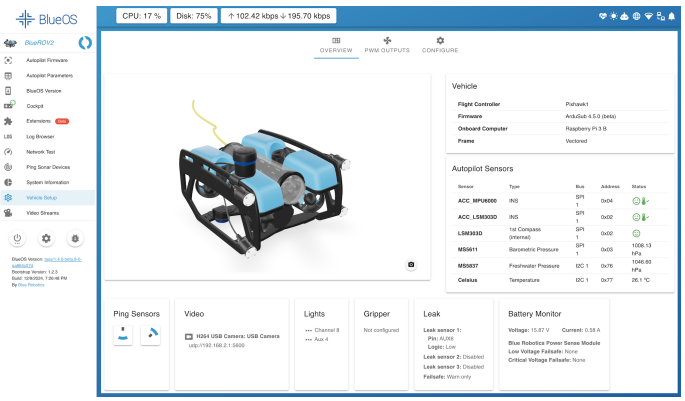


Figure. 2: BlueOS Vehicle Setup [26]

Sensor data acquired through BlueOS can be visualized using graphical tools such as Node-RED, a visual programming platform that facilitates system integration [29]. Node-RED allows users to configure dashboards for real-time visualization of data and supports exporting readings to formats such as CSV or LOG files [29]. Figure 3 provides an example of graphical outputs generated within the BlueOS environment, depicting battery voltage data through a combination of line and circular graphs.

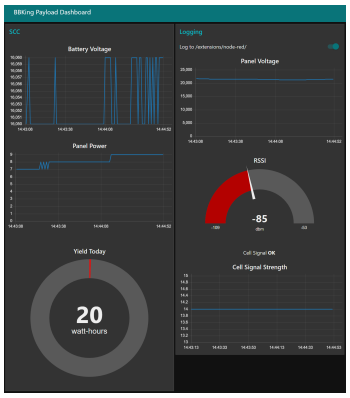


Figure. 3: BlueOS Graphs [29]

Despite their many advantages, ROVs have significant limitations. For instance, their dependence on cables for communication and power supply restricts operational range and depth, as wireless systems are typically unsuitable for transmitting high-frequency data, such as video streams, in underwater environments [6, 10]. Moreover, the cable's weight and size can hinder stability and manoeuvrability [6]. The previously mentioned high cost associated with ROVs, represents a significant barrier to accessibility for many researchers and institutions [6], with BlueROV being one of the most affordable existing alternatives. Operational challenges, including strong currents and high power consumption, further constrain their usability in certain conditions [6, 20].

Traditional Water Sampling Methods

Traditional methods for sampling the water column involved collecting water samples from various locations, which are typically later analysed in a laboratory (in-vitro) to determine water quality [12]. However, this approach often requires repeated visits to the same location, making the process time-consuming and laborious since data needs to be processed and thoroughly examined, contrary to existing real-time solutions. Additionally, its spatial resolution is limited, as only small volumes of water from specific areas can be analysed at a time [11, 13].

2.2 Water Sampling in the IoUT

The IoUT, a ramification of the IoT for underwater environments, has demonstrated significant potential in facilitating water sampling data retrieval by tailoring devices to specific user requirements [1, 11, 12]. As such, this section seeks to analyse different components of the IoUT ecosystem that integrate data retrieval in bodies of water.

Sensors

These systems commonly integrate a wide range of sensors to measure important water parameters that were also retrieved in previously seen sampling methods such as pH, turbidity, conductivity, salinity, TDS, temperature, and DO [13–15]. Additional sensors may be employed to detect other EBVs and EOVs such as humidity, water levels, pressure, nitrates and residual chlorine, depending on the application [13, 30]. In other instances, the observation of marine life through the usage of cameras is also used to fetch visual data [11, 31].

Microprocessors and Microcontrollers

To enable such systems, cost-efficient devices such as single board computers composed of microprocessors (e.g, RPi 4 model B) and microcontrollers (e.g., Arduino Uno or RPi pico) are utilized for different types of computing [32].

Microprocessors are described as a low-cost single-board computer, being capable of running an operating system (such as Linux, Windows 10 IoT), operating headless (without keyboard, mouse, screen) and being remotely controlled and programmed. These components offer more processing power, connectivity, usability and access to data storage compared to microcontrollers [14, 32]. In one study, the RPi functioned as the central control unit in a cost-effective coral monitoring system [31]. It was responsible for remotely initiating image capture by sending commands through an Application Programming Interface (API), automatically capturing images, and temporarily storing them [31]. In another implementation, the RPi was integrated with a display and environmental sensors (pressure and temperature) to facilitate underwater monitoring in a submersible vehicle [33].

Microcontrollers, in contrast to single-board computers, serve to execute a single user-written program and communicate directly with sensors and other electronic components. They are particularly well-suited for performing single, repetitive tasks that do not require further user control, all while consuming significantly less power than single-board computers [32]. In one instance, a network of microcontrollers interfaced to different sensors (pH, turbidity and temperature) was responsible for monitoring river water quality, with the microcontrollers collecting and processing sensor data before transferring it via a Wi-Fi module to other parts of the system [14].

In certain applications, a combination of both microprocessors and microcontrollers can be beneficial. For example, an Arduino may be utilized for analogue sensor data acquisition, while a RPi manages higher-level tasks such as client-side access, automation and online media sharing [32].

In essence, microprocessors offer more processing power and flexibility suitable for control and complex tasks, while microcontrollers are ideal for low-power, repetitive sensing functions.

Communication Mediums

Current communication mediums in the IoUT predominantly rely on wireless technologies such as, ZigBee, Wi-Fi (e.g., ESP8266 modules), Radio Frequency (RF), Acoustic, Magnetically Induced, General Packet Radio Service (GPRS), and LoRa (Long-Range) [13, 34]. These are commonly used

in surface-level applications due to their convenience and performance in air as the line-of-sight allows data offloading. However, their effectiveness diminishes underwater due to significant signal attenuation in due to the salinity, water density and temperature [12, 31].

In underwater environments, communication systems such as magnetically induced and acoustic technologies, while technically being wireless, operate based on different principles and offer robust performance within the water column [34]. Magnetically induced systems use magnetic field coupling, which is ideal for short-range communication, while acoustic technologies rely on mechanical sound waves, enabling long-range communication in challenging underwater conditions [11,34], allowing the transmission of data 5 times faster compared to air. Additionally, tethered systems (e.g., Ethernet, fibre optic, and coaxial cables) provide highly reliable data transmission, especially for stationary underwater installations [1, 11], and in previously discussed ROV instruments. Conversely, while most wireless technologies like ZigBee and Wi-Fi are ideal for surface-level communication between different IoUT components, magnetically induced, acoustic, and cabled systems are indispensable for effective data transmission within the underwater environment [11, 34].

Data Visualization

To visualize real-time data, IoUT systems often employ dashboards or web-based portals, accessible via devices such as smartphones and computers, typically hosted on cloud-based web servers with user-friendly Graphical User Interfaces (GUIs). These platforms present data in tabular formats and visualizations, such as circular and line graphs. While primarily designed for monitoring and control, they also facilitate active sensor management [35,36]. For instance, in Figure 4 a dashboard was developed to visualize data (through a tabular format, a line chart and a circular graph) that was sent remotely through a LoRa, where monitoring and control of sensors was also possible [35].

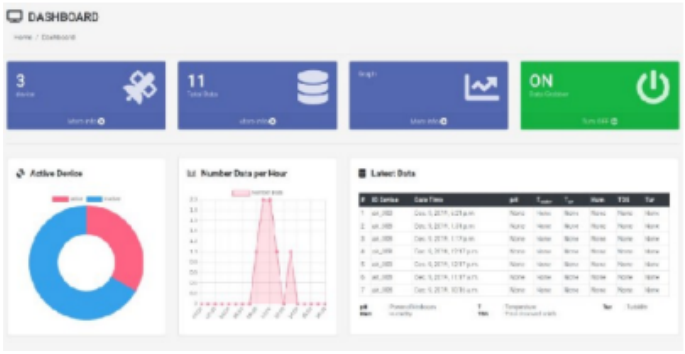


Figure. 4: Dashboard System for data visualization [35]

Layered Models

To better understand IoUT as a means of water sampling, it is helpful to examine how such systems are structured in terms of layered models. Typically the IoUT follows a three-layer architecture as the standard for gathering, processing and displaying data [1, 12]. The three main layers in the commonly IoUT architectures are: Perception Layer, Network Layer and Application Layer, as seen in Figure 5.

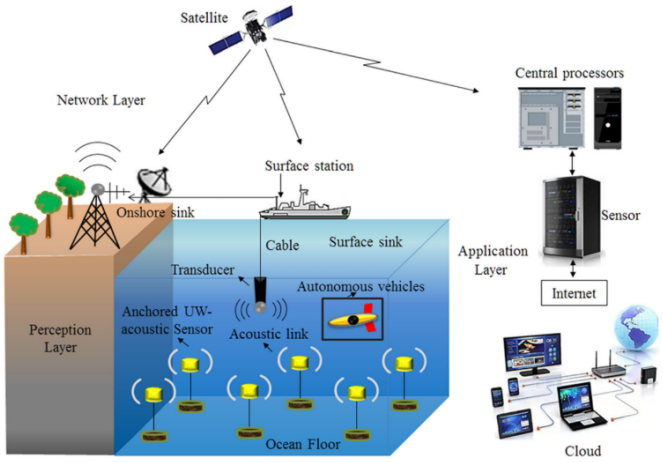


Figure. 5: Three Layer Architecture in the IoUT [11]

– **Perception Layer:** Described as the lowest layer in the IoUT system architecture. Its basic functionalities include identifying objects and gathering information. It is composed by underwater sensors, underwater vehicles such as Autonomous Underwater Vehicles (AUVs) and Autonomous Surface Vehicles (ASVs), monitoring stations, data storage tags, etc [12]. The tasks of the components on this layer are to collect data and initiate actuation. The primary aim of this layer is to gather diverse data regarding water properties, water quality, regular condition monitoring, aquatic life and underwater objects [11].

– **Network Layer:** This layer obtains data from the perception layer and processes it. It is composed of various networks and infrastructure including wired and wireless communication mediums. The network layer is responsible for transmitting the information obtained from the perception layer [12]. It handles bi-directional data packet handling between each endpoint and also translation between adjacent layers. These tasks are performed through internet protocols and data routing [1].

– **Application Layer:** This layer represents a set of solutions that apply IoUT technology to satisfy user needs [12]. It comprises all GUI based front-end services for the analysis of sensed information [11]. In IoUT the application layer is responsible for identifying each object (such as sensor type), and then collecting, processing, and delivering necessary commands. The application can integrate with the Web, through a Representational State Transfer (REST) style architecture, data can be accessed from IoUT devices through Hypertext Transfer Protocol (HTTP) methods like GET and POST [12].

Domain Applications

These systems find applications across various domains including mainly real-time water quality assessment in rivers, lakes and coastal zones; wastewater quality monitoring; marine pollution analysis; aquaculture environmental monitoring; and natural disaster prevention, such as flood detection [1,11,30]. Moreover, these systems are primarily designed for deployment in shallow water depths, with most system components positioned at the surface level of the water column [30, 34]. In some instances, these systems are also deployed in the oceans seabed [12,37]. Additionally, these are not typically optimized for use on smaller vessels by domain experts but are instead intended as stationary systems, that facilitate continuous data transmission [1,11]. However, in some instances, the usage of AUVs and acoustic communication allow for retrieval of data in deeper depths when needed [34, 37].

With this, we conclude that the IoUT remains applied to continuous stationary data retrieval in surface or seabed settings while lacking applicability in sampling the water column. As such, this thesis aims to deeply explore and understand this specific domain by developing an apparatus for water-column sampling.

Limitations

Despite the current IoUT advancements, several challenges persist. Developing cost-effective, energy-efficient, and high-accuracy systems remains a significant hurdle [1, 11]. Reliable communication in underwater environments is particularly challenging due to unique water-based constraints that are more prominent in wireless communication systems, such as high signal attenuation and the Doppler effect [34]. Water-based medium constraints impacts the capability of such systems to withstand greater depths. Furthermore, the integration of diverse technologies, including sensors,

microcontrollers, communication protocols, and IoT platforms requires substantial effort and meticulous system design [11, 37].

2.3 System Architectures in the IoT

The IoT, as a broader domain encompassing the IoUT, is composed by a wide range of systems and architectures designed for specific use cases. Therefore, understanding the architectural decisions commonly employed in the broader IoT, for the development of applications is essential to developing a system of this nature.

2.3.1 Reference Architectures

IoT environments are characterized by a high degree of heterogeneity, encompassing devices with diverse capabilities, functionalities, and network protocols [38]. While various IoT platforms have been proposed to abstract device specifics and promote interoperability, the lack of standardization in the IoT context means these platforms often do not fully address important requirements like scalability, privacy and security, and can adopt incompatible programming models [39].

In order to mitigate these limitations and the challenges that arise from the wide heterogeneity in the IoT, reference architectures arise as a means to specify and guide the development of such systems [40]. As identified through different sources in this research, multiple technology companies have proposed their own reference architectures for IoT systems [41]. However, these architectures are typically designed for large-scale implementations and rely heavily on cloud-based infrastructures [42, 43]. For example, Microsoft Azure IoT [42] is oriented toward sectors such as healthcare, retail, manufacturing, and energy; AWS IoT [43] supports industrial and smart home applications by leveraging Amazon's cloud services; and Intel's reference architecture [41] emphasizes edge analytics by enabling computation at the device level. While these architectures serve their intended large-scale applications effectively, they are not optimally suited for smaller, resource-constrained, local systems [44].

Consequently, this thesis undertakes a more focused examination of two reference architectures better aligned with smaller-scale and more versatile IoT deployments: the IoT Architecture Reference Model (ARM), a European initiative designed to provide a baseline framework applicable to IoT systems regardless of their complexity [39]; and the WSO2 reference architecture, which,

despite being developed by a commercial entity, offers a flexible model intended to support diverse IoT application domains, whether it be with cloud or server solutions [44].

IoT ARM

The IoT ARM is a conceptual framework developed as a part of the European research initiative known as IoT-A (Internet of Things - Architecture). Its primary objective was to establish a standardized reference architecture to support the development of interoperable and scalable IoT ecosystems [39]. This model provides a foundational structure that outlines essential building blocks, design principles, and architectural guidelines for IoT systems. It was established upon a reference model intended to be a baseline for IoT architectures, and takes a top-down approach [38].

IoT ARM provides high-level architectural views and perspectives (as illustrated in Figure 6). These architectural views offer distinct representations of the system that are particularly useful during different stages of design and implementation. Each view emphasizes specific concerns relevant to stakeholders such as developers, system architects, or business analysts. The primary views defined by IoT ARM are as follows:

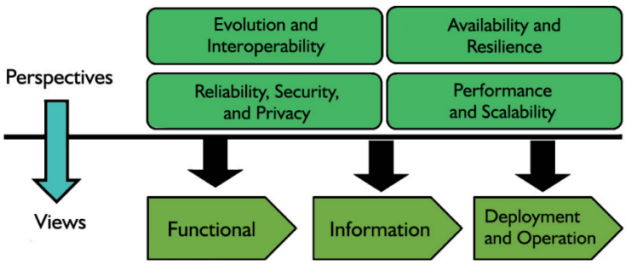


Figure. 6: IoT ARM [38]

- **Functional View:** This view defines the functional building blocks of an IoT system. It categorizes functionality into nine distinct groups, each composed of one or more functional components. These components represent the main capabilities of an IoT system, although their specific interactions are intentionally left open, so that the system can tailor to specific use cases when being designed [38]. An overview of these functional groups and components is shown in Figure 7.
- **Information View:** This view focuses on the representation, flow, and lifecycle of information within an IoT system. It addresses both static and dynamic aspects of data management,

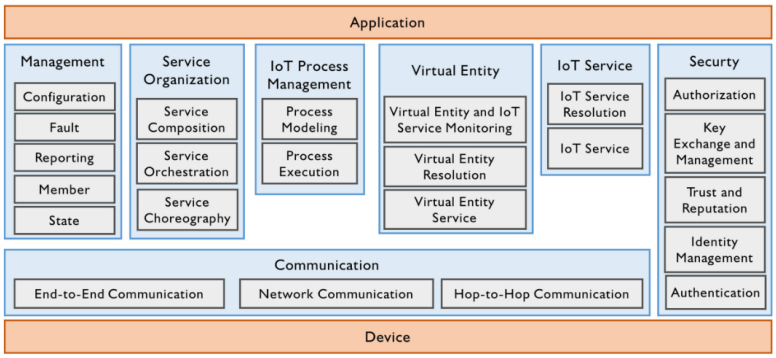


Figure. 7: Functional groups and components on the Functional View of the IoT ARM [38]

including how information is structured, processed and exchanged. Central to this view is the concept of the Virtual Entity (represented on Figure 7), which represents a digital counterpart to a physical object in the system. This layer enables interaction with physical components through abstractions such as digital twins [39].

– **Deployment and Operation View:** This view deals with the practical realization of an IoT system, more specifically with offering guidance on selecting technologies and ensuring interoperability among components. It emphasizes three primary elements: devices, resources and services. This view is also responsible for providing insights into how functional components operate and interact in real-world deployments [39].

In addition to these views, IoT ARM also introduces four architectural perspectives, that focus on cross-cutting concerns, offering tactics, directives, and design decisions aimed at ensuring the system meets specific quality attributes [38, 39]:

– **Evolution and Interoperability:** Deals with how the system can change overtime and communicate with other systems [38, 39].

– **Availability and Resilience:** Focuses on the system's ability to remain operational and handle failures, with a main focus on robustness [38, 39].

– **Reliability, Security and Privacy:** Ensures the system's dependability in protecting data and user privacy. It provides a set of tactics for achieving desired security attributes [38, 39].

– **Performance and Scalability:** Concerned with the system's ability to handle increasing workloads and data volume efficiently [38, 39].

Despite attempts at standardization, the IoT ARM still has its limitations [38]. One major drawback is its relative immaturity. Many proposed architectures based on IoT ARM remain at a conceptual level, and require further research and refinement to fully address the system's practical needs. The high level of abstraction, while useful for general guidance, can hinder the detailed specification of component interactions or accurate modelling of highly diverse physical entities. As such, applying IoT ARM to real-world systems often requires substantial architecture interpretation and adaption, which includes careful consideration of aspects such as external interfaces, security protocols and system adaptability, many of which may be simplified or under-represented in the reference model [39]. Moreover, important IoT requirements such as dynamic adaptation and real-time responsiveness are sometimes overlooked or inadequately addressed [38].

To summarize, while IoT ARM provides a relevant foundation for conceptualizing and design IoT systems, its successful application requires complementary domain-specific insights and careful architectural decisions to bridge the gap between abstract models and practical implementations [38, 39].

WSO2

The WSO2 reference architecture, developed by the technology company WSO2, is a bottom-up design with the primary objective to offer system architects and developers a comprehensive and flexible starting point of addressing the diverse requirements of IoT applications, with the intent to also reduce system heterogeneity [38, 44].

As illustrated in Figure 8, the architecture is structured into multiple layers, each of which can be implemented using a variety of technologies. It also incorporates transversal layers that span across the vertical stack to address cross-cutting concerns [44, 45]. The main architectural layers, arranged from bottom to top, are as follows:

- **Device Layer:** This is the bottom layer of the architecture. In this layer, each device should have a unique identifier and the capability for direct or indirect communication with a network [44].
- **Communication Layer:** Responsible for enabling connectivity between devices and the system, supporting various communication protocols such as Message Queuing Telemetry Transport (MQTT) and HTTP [44].

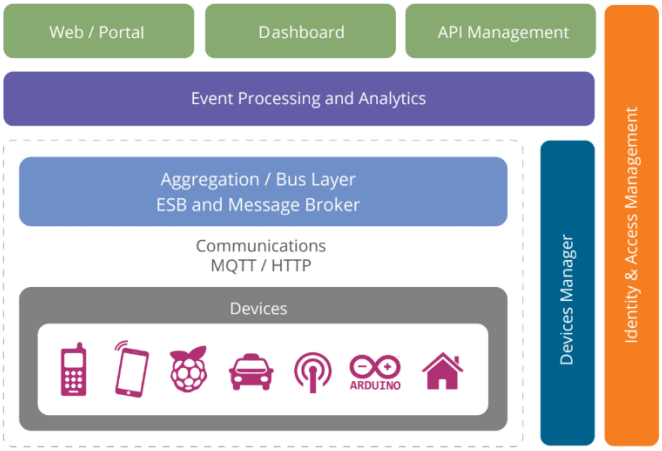


Figure. 8: WSO2's reference architecture [44]

– **Aggregation/Bus Layer:** Facilitates the integration of device communication through interfaces such as HTTP servers and MQTT brokers. It's responsible for supporting, aggregating and combining communication from several devices, while also serving as a bridge to transform data along different protocols, e.g., mediating HTTP-based APIs into MQTT messages for devices [44].

– **Event Processing and Analytics Layer:** This layer is responsible for processing and reacting upon events coming from the Aggregation/Bus Layer. It can also perform data storage, while also providing the core functionality or logic of the platform. Although the layers do not specifically define functional components, this layer could contain components for processing contextual information and managing large data volumes [45].

– **Client/External Communications Layer:** Through this layer, users can interact with devices and access data available in the system. It allows communication outside the device-oriented system using three main approaches: web-based front-ends/portals, dashboards for analytics and event processing views, and APIs for machine-to-machine communication with external systems. An API management system is used to manage and control these APIs, providing a developer portal, a gateway for access control, throttling, routing, load-balancing, and publishing usage data to analytics [44].

In addition to these vertical layers, there are also two transversal layers present in this architecture:

- **Device Management Layer:** This layer offers remote management functionalities for IoT devices using various communication protocols, allowing active configuration and monitoring [44].

- **Identity and Access Management:** This layer is responsible for access control and security directives. It provides services such as OAuth2 token issuing and validation, support for other identity services (like SAML2 SSO and OpenID Connect), an XACML Policy Decision Point, a user directory, and policy management for access control [44].

The WSO2s architecture was made to not be bound to a specific set of technologies, with each layer being capable of being instantiated using specific technologies that best suit the IoT system under construction [38]. It is also designed to be modular and scalable, supporting the addition or removal of capabilities and addressing requirements across a wide variety of use cases. It also encompasses devices and both server-side and cloud architectures needed to interact with and manage these devices [44, 45].

According to [38] while the WSO2 architecture is considered promising and covers most requirements for IoT platforms in a complete or partial way, it has been noted that it lacks elements for dynamic adaptation. Unlike the IoT ARM, the WSO2 architecture does not include a comprehensive reference model for the IoT context, which defines common concepts and relationships for the domain. However, when compared to IoT ARM, the same source highlights WSO2 as being more promising, due to supporting more system requirements.

2.3.2 Architectural Patterns

After understanding a variety of different reference architectures, it is important to understand which architectural patterns are employed to further specify and define system structure. A systematic literature review aimed at guiding developers in the use of architectures and software design within the IoT identified a total of 143 distinct design patterns in 32 different academic papers [46]. Among these, 82 patterns (approximately 57%) were classified as non-IoT-specific. These are general-purpose design patterns that can be applied across various software systems, provided the contextual requirements and problem domains are comparable.

This prevalence of non-IoT patterns indicates that IoT software and system development often relies on conventional architectural and design principles [46]. From the set of non-IoT-specific

patterns the Publisher-Subscriber and Client-Server were selected for more in-depth analysis due to their frequent occurrence and widespread applicability in IoT-related implementations [46].

Publisher-Subscriber

Publisher-Subscriber is a widely adopted architectural design pattern consisting of three core components: publishers, subscribers and a broker [47, 48]. Publishers send messages to specific topics, and subscribers receive messages by registering their interest in particular topics [49]. Communication is typically handled by a message broker, which acts as an intermediary, routing messages from publishers to interested subscribers (as shown in Figure 9). While a centralized broker is a common approach, it can become a performance bottleneck, which can be mitigated by utilizing a network of brokers that cooperate [48, 49].

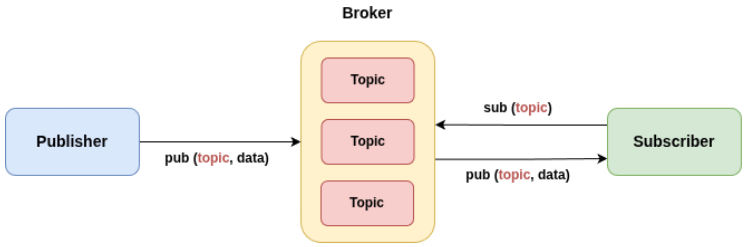


Figure. 9: Publisher-Subscriber Architectural Pattern, based on [47]

One of the main features of this model is the loose coupling between subscribers and publishers [50]. This loose coupling exists across three dimensions:

- **Space decoupling**: Publishers and subscribers do not need to know each other's identity or location [48].
- **Time decoupling**: Allows parties to operate independently. A publisher can send messages while a subscriber is offline and a subscriber can receive notifications about events published while it was disconnected [48].
- **Synchronization decoupling**: Ensures publishers are not blocked when producing events, and subscribers receive notifications asynchronously. This complete decoupling increases scalability [48].

This architectural pattern can be divided into three different variations, with each differing in how subscribers define their interest in events or messages [47]:

- **Topic-based:** In this model, subscribers express their interest by registering for specific, predefined topics. Publishers send messages categorized under a particular topic and the system routes messages based on a match between the message's topic and the subscriber's subscribed topic. The set of available topics is usually known in advance, often defined during the design phase. Subscribers can choose to follow a single topic or multiple ones. While this approach is relatively static and less expressive than content-based alternatives, it is widely used in IoT systems due to its simplicity, efficiency and suitability in constrained environments and hardware [47, 48].

- **Content-based:** In this model, subscribers specify filters that describe the characteristics or content of the messages they wish to receive, rather than just a topic name. This allows for more control over which messages are received, making it the most flexible of the three. However, it's flexibility comes at the cost of increased complexity and greater runtime overhead, as the system must evaluate each message against the subscriber-defined conditions [47, 48].

- **Type-Based:** This less common approach filters messages based on their data type. Subscribers express interest in receiving messages of a specific type, often defined in the programming language or middleware being used. This model supports type safety at compile time and can enable features like sub-typing, making it more integrated with the system [47, 48].

A common lightweight publish-subscribe protocol for IoT is, the previously mentioned, MQTT. MQTT was designed for constrained devices and bandwidth-limited communications, operating effectively over unreliable networks with small bandwidth and high latency. Its asynchronous nature is well-adapted to such distributed environments and edge computing. The pattern hides the complexity of the underlying network from application developers, allowing data delivery based on interests (topics) rather than explicit network addresses [47, 49].

This architectural pattern has some advantages, particularly in IoT environments. It's one-to-many distribution and decoupling of data producers and consumers make it highly effective in dynamic and heterogeneous systems. By supporting asynchronous messaging, it allows for communication across constrained networks, making it especially suitable for remote systems, a characteristic in line with the needs of IoUT deployments, where sensor data is often retrieved from distant or inaccessible locations [49, 50].

Despite these strengths, implementation-specific challenges exist, especially in constrained environments. For instance, ensuring reliable triggering of messages on node failure or managing reliable message delivery from the broker to clients can be challenging with current API implementations. Handling the sleep cycles and duty modes of client devices also presents difficulties. The effort required to add metadata for general content-based systems can also incur too high an overhead for highly constrained platforms [47].

Client-Server

Client-Server is a broadly implemented architectural pattern for designing network-based applications [51]. This model outlines a system into two primary logical components: the server, which offers services or resources, and the client, which requests and consumes those services (as evident in Figure 10) [51,52]. This division establishes a clear separation of concerns, facilitating a modular and scalable system design. In this architecture, the client initiates requests for services or data, acting as the consumer, while the server, as the provider, processes these requests and responds accordingly [51,53]. The server is typically responsible for executing the majority of computational logic, data storage, and service processes, thereby relieving the client of heavy processing tasks and improving overall performance [54].

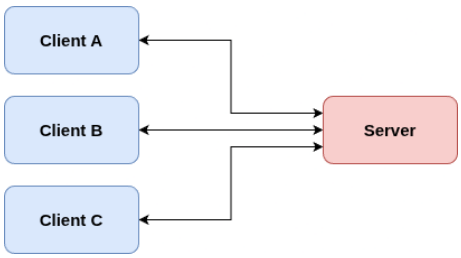


Figure. 10: Client-Server Architectural Pattern, based on [53]

Communication between the client and server is achieved thorough inter-process communication (IPC) mechanisms, which facilitate the exchange of data over a network [53]. This communication typically relies on standardized protocols such as the File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP), and the more commonly used HTTP [51, 54]. These interactions are often mediated by middleware, a distributed software layer that abstracts underlying network complexities and provides transparent access to remote services and resources [50]. Middleware

components may include APIs, and other mechanisms for physical and data-layer transmission, operating on both client and server ends [53].

Client-server systems can be categorized into three primary architectural tiers [54] when being defined:

– **Two-Tier System:** In a two-tier system, the client (e.g. computer) communicates directly with a server (commonly a database server). Application logic can reside on either side or be shared between them [51, 54].

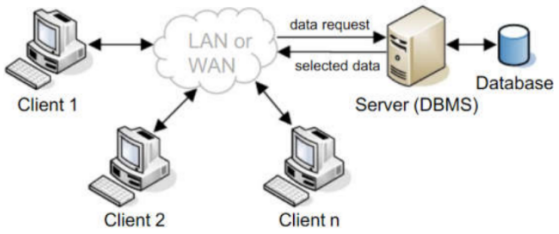


Figure. 11: Two-Tier Client-Server Architecture [51]

– **Three-Tier System:** This architecture introduces an intermediate application server, separating the presentation logic (client), business logic (application server), and data management (database server). The three-tier model promotes modularity, simplifies client-side development and enhances robustness, due to the wider separation of concerns [51, 53, 54].

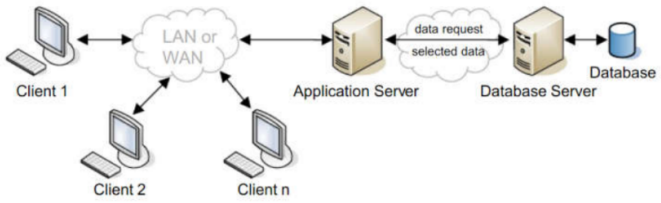


Figure. 12: Three-Tier Client-Server Architecture [51]

– **N-Tier System:** This architecture is an extension of the three-tier model. An N-tier application distributes the application logic across multiple tiers. It is similar to the three-tier architecture, but the number of application servers or layers is increased and represented in individual tiers to further distribute business logic. This architecture can benefit many advanced applications by allowing the integration of multiple data sources separately. For instance, data

warehousing applications can have four possible tiers: individual data repositories, a server that unifies the view of this data, an application server that performs queries based on the unified view and the front-end [51–53].

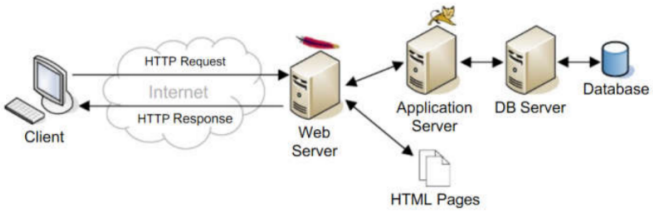


Figure. 13: N-Tier Client-Server Architecture [51]

Functionality in client-server architectures can also vary based on the distribution of application logic, giving rise to classifications such as fat clients and fat servers, where a part of the system has a disproportionate amount of functionality [53].

- **Fat server systems:** In this configuration, the server manages most of the application logic. This approach simplifies client-side maintenance and deployment, as updates are centralized on the server. It is particularly common in mission-critical or web applications [53].
- **Fat client systems:** In this configuration, the client handles a greater portion of the application logic, which is often seen in database applications. While this can make development more straightforward, it also increases the client's dependency on the server's data structure and organization [53].

The client-server model is prevalent across different domains, including ATM networks, web services, and database management systems [53, 54]. It is also highly relevant in IoT applications, where devices often assume the role of either client or server depending on their function within the network [50].

The client-server pattern offers several significant advantages. By distributing application processing across multiple machines, it enables more efficient resource utilization and scalability [53]. Servers centralize data management, reducing redundancy and the risk of inconsistencies across the system [54]. This centralization also simplifies the maintenance and updating of data and application logic, especially in systems with numerous clients. The separation of responsibilities between clients and servers allows for clearer modular design, where each component has a well-

defined role (clients typically handle the presentation layer, while servers manage business logic and data) [53]. This modularity supports more manageable and flexible application development, testing and deployment [54]. Furthermore, because clients and servers communicate over standardized protocols, the system facilitates location-transparent access to services. Components can be replaced, upgraded, or scaled independently, which contributes to easier customization and server-side enhancements. Additionally, client-server systems can scale horizontally by adding more clients with minimal impact on performance, or vertically by upgrading server capabilities to handle more processing or storage [53].

Despite these benefits, the client-server model has several drawbacks. One of the primary challenges is the complexity and cost associated with designing and implementing a system of this nature, particularly as the scale of deployment increases. Security is also a major concern, as servers often have access to sensitive client data and system operations [52, 54]. The communication between clients and servers is vulnerable to various threats, including unauthorized access, data interception, and malware attacks, especially in systems with inadequate encryption or authentication measures [53,54]. Moreover, the reliance on a central server introduces potential single points of failure, which can impact system reliability and availability [54]. Maintenance can be cumbersome in distributed environments where components need to be synchronized, particularly when updates must be rolled out across multiple clients [53]. Achieving high performance is essential in client-server systems, yet it poses significant challenges, particularly in managing network traffic efficiently. Additionally, interoperability is also highlighted as a key issue, as it is also particularly challenging to exchange information in heterogeneous environments [52].

2.4 The RPi

The RPi is a versatile, low-cost single-board computer widely recognized for its diverse applications in research and prototyping due to its capabilities of being used in-situ deployments [32, 55]. As this thesis explores sensor integration and the utilization of microprocessors for IoUT systems, there will be a brief overview of communication protocols, a simple description of the General Purpose Input/Output (GPIO) header for peripheral integration, and advantages and limitations for in-situ deployments. The characteristics that will be further discussed apply broadly to all RPi single-board models rather than any specific version.

Communication Protocols

The RPi employs the Inter-Integrated Circuit (I2C), Serial Peripheral Interface (SPI) and Universal Asynchronous Receiver/Transmitter (UART) protocols to handle efficient and reliable communication with a variety of peripheral devices and sensors [32].

The I2C protocol is a serial bus interface designed for low-to-medium speed data transmission between integrated peripherals [56]. It operates using only two transmission lines: Serial Data (SDA) for bidirectional data transfer and Serial Clock (SCL) for the clock signal, which is generated by the master device [57]. In typical configuration, the RPi functions as the master, while connected sensors act as slave devices [57], each accessible via a unique address on the bus, although in some cases a multimaster configuration is utilized instead [58]. I2C is mostly utilized for interfacing with low-speed components, such as environmental sensors [57, 59].

The SPI protocol, on the other hand, is a synchronous serial communication interface designed for high-speed data exchange [58]. It generally uses a four-wire configuration: Master In Slave Out (MISO) for data from the slave to the master, Master Out Slave In (MOSI) for data from the master to the slave, Serial Clock (SCLK) for the clock signal generated by the master, and Chip Select/Slave Select (CS) to enable communication with specific slave devices [60]. Alternatively, SPI can operate in a three-wire mode by using either MISO or MOSI, depending on the direction of communication [60]. The RPi typically acts as the master in SPI setups, with devices such as FPGAs or sensors serving as slaves [61]. SPI supports data transfer rates of up to 50 MHz, making it ideal for high-speed, short-distance communication applications, such as on-chip interfacing and monitoring systems [58, 60].

The UART protocol enables simple, point-to-point serial communication [62]. Unlike I2C and SPI, UART is asynchronous and does not require a shared clock line, instead relying on two dedicated signal lines: Transmit (TXD) and Receive (RXD) for full-duplex communication [63]. Its internal architecture typically comprises of a baud rate generator responsible for setting the communication speed through clock division, a transmitter module for converting parallel data to serial format, and a receiver module for decoding incoming serial data [63]. The RPi includes a single UART interface, which is commonly used for integrating various peripherals such as sensors or Bluetooth communication modules [62]. While UART is limited in speed compared to SPI and

lacks the multi-device support of I2C, it remains an important protocol in embedded and IoT systems [63].

GPIO Pins

The GPIO pins are a fundamental feature that distinguishes the RPi from conventional computers [56]. These pins provide direct access to the onboard circuitry, enabling developers to interface the RPi with a variety of external electronic components and devices [59]. By leveraging the GPIO interface, the RPi can function not only as a stand-alone computer but also as a controller within large embedded systems. The RPi's GPIO header consists of 40 physical pins [60], each designed for specific functions, as illustrated in Figure 14. These include digital input/output capabilities, power supply lines, and communication interfaces for protocols such as I2C, SPI and UART [62].

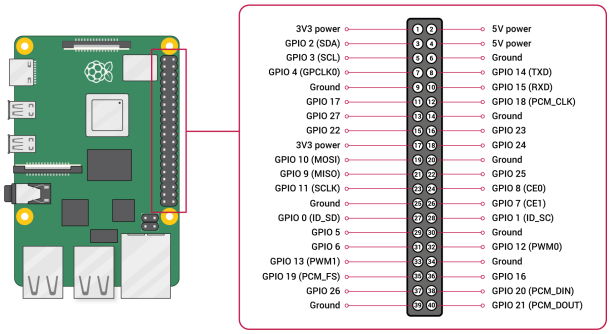


Figure. 14: RPi GPIO Header [64]

GPIO pins can be configured through software to act either as inputs or outputs. As evident in official RPi documentation [64], common usage includes general digital I/O (e.g. pin 17 or pin 27), as well as dedicated lines for communication protocols such as I2C (pin 2 for SDA and pin 3 for SCL), SPI (pin 9 for MISO, pin 10 for MOSI, pin 11 for SCLK, and pin 8 or 7 for CS) and UART (pin 14 for TXD and pin 15 for RXD). The header also includes 3.3V and 5V power output pins along with multiple ground connections. Unlike microcontrollers such as Arduino, the RPi lacks in analogue input integration [32]. Therefore, reading analogue signals from sensors or modules requires an external Analog-to-Digital Converter (ADC), such as MCP3008 chip [59].

Through its GPIO interface, the RPi can be connected to a broad range of peripherals, including output devices like LEDs, motors, and relays, as well as input devices such as buttons, switches,

joysticks, and various sensors. This flexibility makes it highly suitable for a wide range of IoT and embedded applications [59].

Advantages for in-situ deployments

The RPi is low-cost, providing an affordable computing platform, making it accessible for experimentation and development [56]. Its low price has contributed to its widespread adoption in different scientific projects [65,66]. Moreover, it is compact in size, being highly portable and easily integrable into various projects [56,67], its size is particularly advantageous for applications where space is a constraint [59]. Regarding the processing power, it is equipped with a relatively fast processor compared to microcontroller-based platforms, the RPi can effectively perform computationally demanding tasks, making it suitable for many data-intensive applications [65]. Additionally, with its multiple GPIO pins for external device connections and compatibility with various ADCs, the platform allows for diverse sensor integration essential for real-time monitoring [59, 65]. The RPi also supports multiple communication protocols, which enable interaction with peripheral components via its GPIO pins [56]. When it comes to the software compatibility, it operates with numerous programming languages (e.g. Python, C++, Java) and is able to run various operating systems, like other microprocessors, making it suitable for a wide range of software development needs [56]. Most RPi models include Ethernet, Wi-Fi and Bluetooth capabilities, simplifying integration into networks and connection with other devices [59,65]. The RPi is capable of significant customization and flexibility, enabling users to tailor it to specific project requirements [56,65].

Limitations for in-situ deployments

Although the RPi has many benefits, several limitations exist. For instance, the RPi lacks built-in storage and relies on an SD card for data storage, which is typically slower and less reliable than other storage options such as eMMC, which is used in other single-board computers [65]. Additionally, the absence of a Real Time Clock (RTC) necessitates external modules for accurate timekeeping in offline scenarios [56, 59]. Another limitation is its computation power. Although robust for its size, the RPi does not match the processing capabilities of a traditional desktop computer [32], restricting its use in some high-performance applications. Another disadvantage is that the RPi lacks a built-in ADC which further hampers the integration with analogue sensors [56,59]. The RPi's bare circuit board, requires additional protective casing for harsh environments,

increasing deployment costs and complexity [32,56,65]. Another constraint may be seen in its heat dissipation. Intensive computational tasks can cause the RPi to overheat, necessitating the use of heat sinks or cooling fans to prevent damage [32]. Nevertheless, in an underwater setting, the actual depth and water acts as a passive cooler [68]. The number of GPIO pins is also limited, potentially requiring external expansion modules for projects with higher I/O demands, although this can be expanded [56]. Regarding the energy consumption, and as evident in previous sections, the RPi consumes more power than microcontrollers, which may affect its suitability for battery-operated systems [32, 56].

3 Methodology

This chapter outlines the strategies, design choices, and procedures implemented in the development of the apparatus, application and its integrated system components.

3.1 System Requirements

To begin the development process, a domain expert specialized in plankton distribution (which is studied by analysing parameters that are retrieved when sampling water) was consulted through a series of 3 semi-structured interviews. These interviews were conducted to elicit essential requirements by exploring the expert's research needs, operational expectations and environmental constraints. While some requirements were taken from these interviews, others were defined to address system operation, maintenance, and performance which were not explicitly requested, but were later approved upon review. The latter ones are highlighted with an asterisk **(*)** and include a brief justification for their inclusion. These requirements were separated into 5 different categories, depending on their role within the system:

- **System Recording Requirements:** Defines how the system captures and records sensor data, including sampling methods, precision and real-time visualization.
- **System Accessibility and Monitoring Requirements:** Based on accessibility, monitoring and device management.
- **Sensor Configuration Requirements:** Related to the ability to configure and calibrate sensors to ensure accurate measurements.
- **Data Related Requirements:** Addresses how recorded data is stored, managed and accessed through CSV and Log files.
- **Error Handling and Data Integrity Requirements:** Based on reliability of data collection and system response in the event of errors or hardware issues.

These requirements are classified as functional or non-functional, with functional requirements describing specific system behaviours or functions, while non-functional requirements define performance constraints, usability, reliability, and other quality attributes that influence how the system operates rather than its specific functionality. Each requirement is named by its category, with an **F** or **NF** indicating a functional or non-functional requirement, respectively.

System Recording Requirements

This category covers all requirements related to how the system captures and records environmental data from sensors. It includes specifications on which parameters are recorded, the precision of data, configurable recording intervals or triggers (such as time or depth), and options for real-time visualization and user control of the recording process.

Table 1: System Recording Requirements

ID	Type	Requirement Description
SR1	F	The system shall measure and record key parameters: conductivity, DO, temperature, depth.
SR2	NF	The system shall have a sensor data precision of 0.1.
SR3	F	The system shall enable the user to configure the method and interval of sensor recordings.
SR3.1	F	The system shall allow the user to select time as a method of recording.
SR3.1.1	F	The system shall have a minimum sensor reading interval of one reading every 5 seconds when time is the method of recording.
SR3.1.2	F	The system shall have a maximum sensor reading interval of one reading every 60 seconds when time is the method of recording.
SR3.2	F	The system shall allow the user to select depth as a method of recording.
SR3.2.1	F	The system shall have a minimum sensor reading interval of one reading every 5 m when depth is the method of recording.
SR3.2.2	F	The system shall have a maximum sensor reading interval of one reading every 25 m when depth is the method of recording.
SR4	F	The system shall provide the user with configurable options for terminating the recording process, allowing for both automated and manual control.
SR4.1	F	The system shall enable the user to stop recordings automatically depending on the method of recording.
SR4.1.1	F	The system shall automatically stop the recording when a specified time duration has elapsed (for time-based recordings).
SR4.1.2	F	The system shall automatically stop the recording upon reaching a predetermined depth threshold (for depth-based recordings).
SR4.2	F	The system shall allow the user to manually terminate the recording at any point through the user interface.
SR5	F	The system shall display the most recent readings in real time.
SR6	F	The system shall allow for visualization of recorded data on a real-time updated graph for analysis.

System Accessibility and Monitoring Requirements

This category groups requirements concerning user access, device control, and ongoing system monitoring. It includes functionality for safe shutdown and reboot, user guidance through manuals, real-time display of critical safety parameters (battery, depth, CPU), user limits on simultaneous connections, and compatibility with various device types for flexible access.

Table 2: System Accessibility and Monitoring Requirements

ID	Type	Requirement Description
SMR1	F	The system shall allow the user to reboot the device if needed.
SMR2	F	The system shall provide an integrated manual for user guidance and troubleshooting.
SMR3	F	The system shall display safety measure parameters at all times.
SMR3.1	F	The system shall display the device's remaining battery level to monitor power status.
SMR3.1.1	F	The system shall notify the user if the remaining battery is critically low.
SMR3.2	F	The system shall continuously display the current depth as a safety measure.
SMR3.2.1	F	The system shall notify the user if the current depth is over 300 m. (*)
SMR3.3	F	The system shall indicate the CPU usage on the device. (*)
SMR3.3.1	F	The system shall notify the user if the CPU usage is critically high. (*)
SMR4	NF	The system shall allow for maximum of 3 users to be connected at the same time. (*)
SMR4.1	F	The system shall allow for only one user to be responsible for sensor readings and sensor calibration. (*)
SMR5	NF	The system shall be accessible on mobile devices (e.g laptops, mobile phones).
SMR6	F	The system shall allow for customization of visual aspects to improve visibility.
SMR6.1	F	The system shall allow the user to customize the interface colour scheme for accessibility. (*)
SMR6.2	F	The system shall allow the user to switch between light and night theme modes. (*)

– **SMR3.2.1:** As discussed later in this chapter, the sensors integrated into this system are limited to a maximum operational depth of approximately 350 m. To ensure system safety, it's important to set a lower threshold for where the system should operate (300 m) to prevent potential damage to the sensors.

- **SMR3.3:** While the domain expert initially suggested monitoring available storage space, the use of a lightweight SQLite database for storing sensor readings and temporary log files, which are either automatically deleted or can be removed by the user, makes for significant changes in storage space unlikely. Therefore, monitoring available CPU is more relevant for this system, as it better reflects real-time system performance and potential operational constraints. CPU usage was also an monitored parameter in the observed BlueOS system utilized for BlueROV2 [24].

- **SMR3.3.1:** Excessive CPU consumption can lead to system instability, including unexpected crashing or failures. Alerting the user in such cases is essential to ensure they are informed of the underlying cause and can take appropriate action.

- **SMR4:** Given that the system is hosted on RPi, simultaneous access by multiple users may overwork its limited computational resources and impact overall system performance. However, as the deployment typically occurs on small research vessels with a limited number of users, supporting a large number of concurrent connections is not crucial or a user need.

- **SMR4.1:** If more than one user is capable of stopping sensor recordings or doing calibrations, errors may occur. By limiting these operations to one user the system is less error-prone.

- **SMR6.1** and **SMR7.2:** The researcher highlighted that in certain cases, recordings are done in harsher weather and locations where visibility is more constrained. As such, allowing the user to change the colours of the application and themes to facilitate visibility is important.

Sensor Configuration Requirements

These requirements define how users interact with the system's sensors to ensure accurate and reliable data collection. They focus on configuring sensor parameters, performing calibrations through guided procedures, displaying sensor statuses, and validating inputs to prevent incorrect sensor settings. All of these requirements are defined to ensure that sensors can be configured and calibrated without the need to remove them from the system.

Table 3: Sensor Configuration Requirements

ID	Type	Requirement Description
SC1	F	The system shall allow for sensor configuration.
SC1.1	F	The system shall allow for the modification of sensor parameters.
SC1.2	F	The system shall allow for the calibration of individual sensors through the user interface.
SC1.2.1	F	The system shall guide the user through calibration procedures, where applicable.
SC2	F	The system shall display the current status and configuration of each sensor.
SC3	F	The system shall validate configuration input to prevent invalid or unsupported sensor settings.

Data Related Requirements

This set of requirements addresses the management of recorded data, including how sensor data and system logs are stored, accessed, visualized, downloaded, and deleted. It also highlights the importance of maintaining logs for monitoring system activity and supporting maintenance and troubleshooting.

Table 4: Data Related Requirements

ID	Type	Requirement Description
DR1	F	The system shall save different types of data.
DR1.1	F	The system shall save all sensor recording data in CSV files.
DR1.1.1	F	The system shall provide an option to delete recorded CSV files when they are no longer needed.
DR1.1.2	F	The system shall provide an option to download recorded CSV files.
DR1.1.3	F	The system shall provide an option to display recorded CSV file data on a graph.
DR1.2	F	The system shall save all server and recording activity in Log files. (*)
DR1.2.1	F	The system shall allow for Log data to be downloaded. (*)
DR1.2.2	F	The system shall automatically delete Log files older than 3 days. (*)

– **DR1.2:** Logging system activity in files is essential for monitoring user interactions, server activity and for facilitating debugging in case of system failures.

– **DR1.2.1:** Enabling Log files to be downloaded facilitates further analysis and long-term archiving of system activity, which may be necessary for troubleshooting.

– **DR1.2.2:** Automatically deleting Log files older than 3 days helps conserve storage space by removing data that is unlikely to be needed.

Error Handling and Data Integrity Requirements

Requirements in this category ensure that the system reliably handles sensor connectivity and data validity issues. They focus on detecting sensor failures, alerting the user appropriately, attempting automatic recovery, preserving data integrity during interruptions, and logging errors for future analysis.

Table 5: Error Handling and Data Integrity Requirements

ID	Type	Requirement Description
EDR1	F	The system shall verify that sensor connection has been successfully established. (*)
EDR1.1	F	The system shall prevent back-end server operations from proceeding until all required sensors are properly connected. (*)
EDR1.2	F	The system shall alert the user if a sensor fails to respond. (*)
EDR1.3	F	The system shall alert the user if a sensor provides invalid data. (*)
EDR1.4	F	The system shall automatically attempt to reconnect to any temporarily unresponsive sensor devices. (*)
EDR2	F	The system shall log all detected errors in the previously mentioned Log file for further analysis. (*)
EDR3	F	The system shall ensure that interrupted recordings (e.g., due to system reboot or power loss) are safely handled and previous data is preserved. (*)

– **EDR1:** As system operations rely heavily on sensor input, ensuring that all sensors are properly connected is critical for normal operation.

– **EDR1.1:** Allowing the system to operate without proper sensor connections may result in incorrect or misleading outputs. Therefore, halting operations until the sensors are correctly connected helps maintain data integrity and prevents system malfunctions.

– **EDR1.2:** User awareness through system feedback is essential when components fail to operate as intended.

- **EDR1.3:** Even if the sensor is connected, it may produce incorrect readings due to malfunction or interference. Notifying the user of such occurrences is also relevant to allow for better decision-making to fix the issue.
- **EDR1.4:** Implementing an automatic reconnection mechanism ensures more robustness and minimizes manual intervention by making the system correct itself.
- **EDR2:** Maintaining logs of system errors facilitates troubleshooting and supports the identification and resolution of (possible) recurring issues.
- **EDR3:** Ensuring data integrity is critical in systems of this nature, as the completeness and accuracy of the data are essential for reliable analysis of environmental variables.

3.2 System Architecture

After defining the system requirements, initial work focused on designing a robust architecture. As such, initially a simple conceptual architecture was outlined based on the researched architectural references, and a architectural diagram, based on system requirements and architectural patterns.

3.2.1 Conceptual Architecture

The WSO2 reference architecture was chosen to serve as the foundation for the proposed system's conceptual architecture. Its comprehensive support for the entire IoT stack (including device communication, data aggregation, processing and API management) makes it particularly well-suited for small to medium scale systems such as the one presented in this work [44]. As shown in Figure 15, the conceptual architecture is separated into 4 main layers.

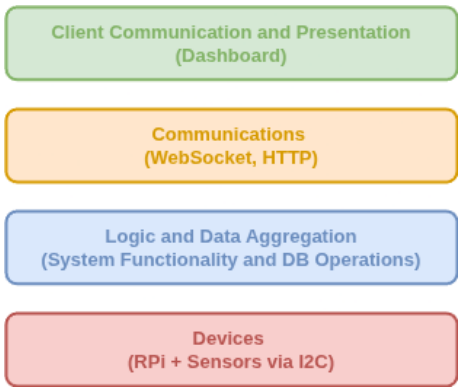


Figure. 15: Conceptual Architecture.

- **Devices:** This foundational layer includes all hardware components integrated into the system through the I2C communication protocol, such as environmental sensors, the microprocessor (e.g., RPi), and any peripheral modules. These devices perform the actual data acquisition and are interfaced with the back-end logic to facilitate real-time processing and control.

- **Logic and Data Aggregation:** This layer handles all back-end operations and system logic. It is responsible for sensor management, event logging, database interactions, and the generation of structured data files such as CSVs. As such, this layer ensures data consistency, integrity, and proper integration between hardware and software components.

- **Communications:** This layer is responsible for all network communication protocols facilitating data exchange between the front-end and back-end. HTTP endpoints are employed for standard data retrieval and control operations, such as sending commands to the sensors. For real-time data transmission, WebSocket connections are utilized due to their lower latency and capability of ensuring immediate delivery of sensor readings.

- **Client Communication and Presentation:** The topmost layer encompasses all client-side interactions and system visualization. It enables the user to initiate and control recording sessions, retrieve stored data, and manage the overall system operation. Through this layer, users can monitor sensor data via tabular and graphical representations. Functionally, this layer is implemented as a front-end dashboard that provides control and real-time feedback, serving as the primary interface between the user and the system.

Compared to WSO2's reference architecture [44], which includes a broad set of transversal components such as Identity and Access Management, Device Management, and Analytics, the implemented system adopts a simplified structure tailored to its specific use case. Transversal components like Identity and Access Management were excluded, as the system is intended for single-user or small-group, isolated local deployments, making authentication and role-based access control unnecessary and cumbersome. Similarly, Device Management was not implemented, as the system relies on a single board operated locally, and does not require remote operation, firmware updates, or large-scale device control. The Analytics layer was also condensed into the Logic and Data Aggregation layer, as all processing occurs centrally within the same module. Nonetheless, the core layers of communication, logic, and device interaction remain aligned with WSO2's structure, maintaining a clear separation of concerns while supporting maintainability and future scalability.

3.2.2 Architectural Diagram

As illustrated in Figure 16, the system adopts a two-tier client-server architectural pattern [51]. This architecture was deemed most appropriate for the system's intended use, which involves deployment in a resource-constrained environment and is primarily accessed by a small number of researchers. More complex architectures, such as three-tier or N-tier models, would introduce unnecessary logistical and technical overhead, making deployment less efficient for this use-case. Similarly, incorporating additional components such as intermediary message brokers commonly found in publisher-subscriber architectures, would unnecessarily complicate deployment and maintenance. Given the limited user base and the fact that data is collected locally by the users rather than remotely, the use of a publish-subscribe model (with its reliance on concepts such as topics, publishers, and subscribers) is neither necessary nor practical for this specific system. By consolidating the business logic and data access into a single back-end server, the system reduces network overhead, minimizes latency and simplifies deployment and maintenance.

As described in the conceptual architecture, real-time sensor data is acquired via a WebSocket connection. Other forms of data interaction between the client and server are handled through a RESTful API, which uses standard HTTP methods to facilitate the retrieval and management of relevant information.

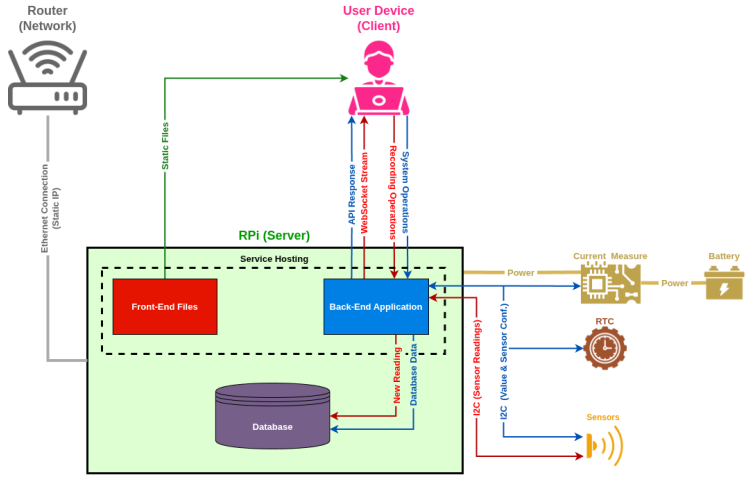


Figure. 16: Architectural Diagram.

The architectural diagram depicts the integration of hardware and software components to enable the acquisition, processing, storage, and visualization of environmental sensor data. As evident with typical client-server architectural patterns, this diagram is divided into 3 parts:

1. **RPi (Server):** composed by the RPi and its components, and peripherals (sensors, RTC, current measurer and battery). Within the three-layer IoUT model [12], the RPi also assumes the role of the perception layer, as its integrated sensors are responsible for acquiring environmental data. For the purposes of this dissertation, the RPi was selected as the computational unit, as it is capable of supporting all required functionalities within a centralized platform. Employing a microcontroller alongside the RPi was deemed unnecessary for this use case since the combined power draws would make power consumption significantly higher, which is undesirable in energy-constrained underwater environments relying on power sources (such as batteries) of limited capacity.

2. **Router (Network):** a LAN established by a router for communication between the server and the client. In the context of the three-layer IoUT model [12], the router and its associated networking components compose the network layer, ensuring data transmission across system elements.

3. **User Device (Client):** user interaction with the system through a GUI. According to the three-layer IoT model [12], this corresponds to the application layer, where data is presented to the user and system functionalities are accessed.

Although not illustrated in the diagram, all components, with the exception for the router and the user device, are to be submerged during operation. Among the submerged components, only the sensors are in direct contact with the surrounding water to collect environmental data. The remaining hardware should be housed within a durable, waterproof casing to ensure protection against the underwater environment, with the apparatus being deployed manually inside a bucket that is slowly descended by an operator on boat.

RPi (Server)

As the system's server, the *RPi 4 Model B* [69] is responsible for managing hardware interfaces and serving software components to the user:

– **Front-End Files (Red Square):** Built using Vue.js [70], a JavaScript framework based on development through reusable components. Its responsiveness and reactivity enable real-time feedback and a dynamic interface that reflects system status and user interactions. To enhance code quality and maintainability, TypeScript [71] is employed. As a statically typed superset

of JavaScript, TypeScript offers robust type checking and better tooling support, which results in cleaner and more reliable code. For state management Pinia [72] is utilized, which simplifies the process of sharing and maintaining states across components without the need for constant database calls. To comply with graphical visualization requirements, Chart.js [73] is responsible for rendering interactive and real-time graphs of sensor data. All of these technologies compose the front-end files with a simple GUI for system operation.

– **Back-End Application (Blue Square):** Built using Python [74], selected for its simplicity, versatility, and extensive library ecosystem. Python supports both configuration and communication with the chosen sensors, as well as development of low-level server-side functionalities required by the system, and database interactions. To build the back-end API, FastAPI [75] was chosen. This technology was employed due to its high performance in developing RESTful endpoints and establishing WebSocket communication, which are both important functionalities in this architecture. To serve the FastAPI application, Uvicorn [76] was chosen due to being lightweight and asynchronous. As an Asynchronous Server Gateway Interface (ASGI) server, it provides an efficient interface between the application and the underlying system, enabling fast response times and effective execution of the FastAPI application. All of these technologies help develop the logic and server logic responsible for communicating with the client.

– **Server Hosting (Black Dotted Square):** Responsible for handling server hosting, NGINX [77] was chosen for its ability to efficiently manage and handle communication between both front-end and back-end services by configuring a single file. Its ease of configuration, high performance and low resource consumption make it well-suited for running both server roles on a device like the RPi. For the front-end, NGINX is used to serve static files when accessing a specific static IP address, allowing access to the GUI. In this case, the RPi hosts the application files, while their execution occurs on the client's device. The back-end, which runs directly on the RPi, requires a different handling approach. As such, NGINX's role is to act as a reverse proxy between the front-end and the back-end server. This choice enhances security by hiding server details (albeit not a crucial concern for this system), improves performance through caching and potential load balancing, centralizing access to the server. As a result, all API endpoints and WebSocket communications are managed safely and efficiently.

– **Database (Purple Cylinder):** Utilized for storing sensor data through SQLite [78], a serverless and lightweight option to store data. Its simplicity eliminates the need for a separate database server, since all data is stored in a single file, minimizing overhead while maintaining data.

– **Sensors (Orange):** Used for gathering environmental data, three sensors were employed: the *Atlas Scientific Electrical Conductivity 1.0K Probe* [79] which is capable of measuring conductivity, salinity, TDS and specific gravity; the *Atlas Scientific Zero Dissolved Oxygen Probe* [80] which captures DO levels; and the *Keller 4LD* [81] used for determining both temperature and pressure. While the primary parameters to be recorded are conductivity, DO, temperature and depth, salinity and TDS were also accommodated. Although these two are not classified as essential parameters (as evident by system requirements) prior studies have highlighted their significance in underwater sensing applications [11, 35], and after approval from the domain expert, these parameters were included in the system's capabilities. Since the *Keller 4LD* sensor provides pressure rather than depth, the depth can be calculated by the following formula:

$$\text{Depth (m)} = \frac{P - P_0}{\rho \cdot g}$$

Where P is the measured pressure in Pascals (Pa), P_0 is the reference surface pressure (typically $101325\,\text{Pa}$), ρ is the density of the fluid, which takes a value ranging between $1020 - 1029\,\text{kg/m}^3$ for seawater, and g is the gravitational acceleration, approximately $9.81\,\text{m/s}^2$. The selection of sensors was guided by several criteria, such as: durability (with each sensor being operational to depths of up to 300 m) and compatibility (with the RPi's integrated GPIO pins through the I2C protocol). These parameters align with the system's objective of offering a reliable yet affordable alternative to existing market solutions without compromising on functionality or reliability. An operational depth of a maximum of 300 m was selected based on the domain expert's advice, as depth variations between $0 - 150$ m in certain environments offer little to no variation of environmental variables.

– **RTC (Brown):** Accurate data retrieval is essential, and precise timekeeping is crucial both for maintaining Log files and for calculating the time intervals between sensor readings. However,

as previously noted, RPi devices do not include an integrated RTC, which can compromise timestamp accuracy, especially after power cycles or when operating offline. Since this system is intended for continuous use in aquatic environments, maintaining an internet connection for time synchronization is not possible. To address this limitation, a *DS3231 RTC* [82] module was integrated into the system via the I2C communication interface, replacing the built-in fake clock present in the RPi, providing reliable and accurate timekeeping.

- **Battery and Current Measurement (Yellow):** The battery will serve as the primary power source for the entire system. It should be connected to an *INA219* [83] current measurement module, which monitors the electrical current and sends it to the RPi via the I2C communication protocol. Current values are used to estimate the remaining battery percentage, fulfilling system requirements for monitoring power availability. For the current version of this system, a battery has not yet been implemented as testing and development is done in-vitro, and the utilization of a wall outlet power source is more suited until the system is ready to be deployed in its intended environment.

For all the integrated hardware peripherals, the I2C communication protocol is employed due to its multi-device integration, and the fact that all selected peripherals natively support it. Standardizing communication through a single protocol reduces software complexity, and minimizes wiring, while maintaining functionality. Although SPI offers higher data transferring, this advantage is not critical in the present context, as most sensors require internal processing time after data retrieval (e.g documentation for one of the sensors in [84]). Consequently, I2C is well-suited for the application, offering an effective balance between simplicity, integration and performance.

Router (LAN)

To enable wireless access to the system, a TP-Link *TL-MR3420* router [85] was employed. The router is connected to the RPi via an Ethernet cable, with the RPi configured to operate with a static IP address on its Ethernet interface. This setup allows the router to establish a Local Area Network (LAN) and broadcast the connection over Wi-Fi. As a result, wireless devices onboard the vessel can connect to the RPi through the assigned static IP address and access the GUI.

User Device (Client)

The user device, operated onboard the vessel, connects to the LAN provided by the router. Through a web browser, users are able to load the front-end application files through the static IP, and run the GUI through their chosen device.

System Cost

The total estimated cost of the system components (including the microprocessor, sensors, cabling, router, protective casing and battery) is approximately $1200 at the time of writing. It is important to note that this figure represents a high-end estimate, as the final casing design has not yet been developed and the battery component has not been implemented in the current version. Therefore, the overall cost is expected to be lower in the version of the system ready for field use.

3.2.2.1 Main System Operations

In the architectural diagram (Figure 16), the system encompasses three distinct operations, each represented by a different arrow colour: green, red, and blue.

Access to Front-End GUI (Green Arrow)

The first operation, which is the most straightforward, refers to how the user accesses the front-end interface. As previously described, the front-end files are served by the NGINX server at a static IP address corresponding to the RPi's Ethernet interface. Consequently, when a user connects to this IP address via a web browser, the front-end files are loaded onto their device, enabling interaction with the system and access to its full range of functionalities.

Real-Time Sensor Data Retrieval (Red Arrows)

This operation is initiated when the user wants to start sensor recording operations. This triggers the transmission of the pre-defined parameters to the back-end. Subsequently, the connection with the sensors through I2C begins the retrieval of real-time environmental data that is then stored in the database, leading to a simultaneous transmission of recorded data via the WebSocket connection as soon as it reaches the database.

The flowchart illustrated in Figure 17 expands upon this process. It details how sensor readings are acquired and delivered to the user. The steps below assume the user is connected to the back-end and the WebSocket connection is established. From this point onward, the term "trip" is defined on the developed system as the process of recording environmental data.

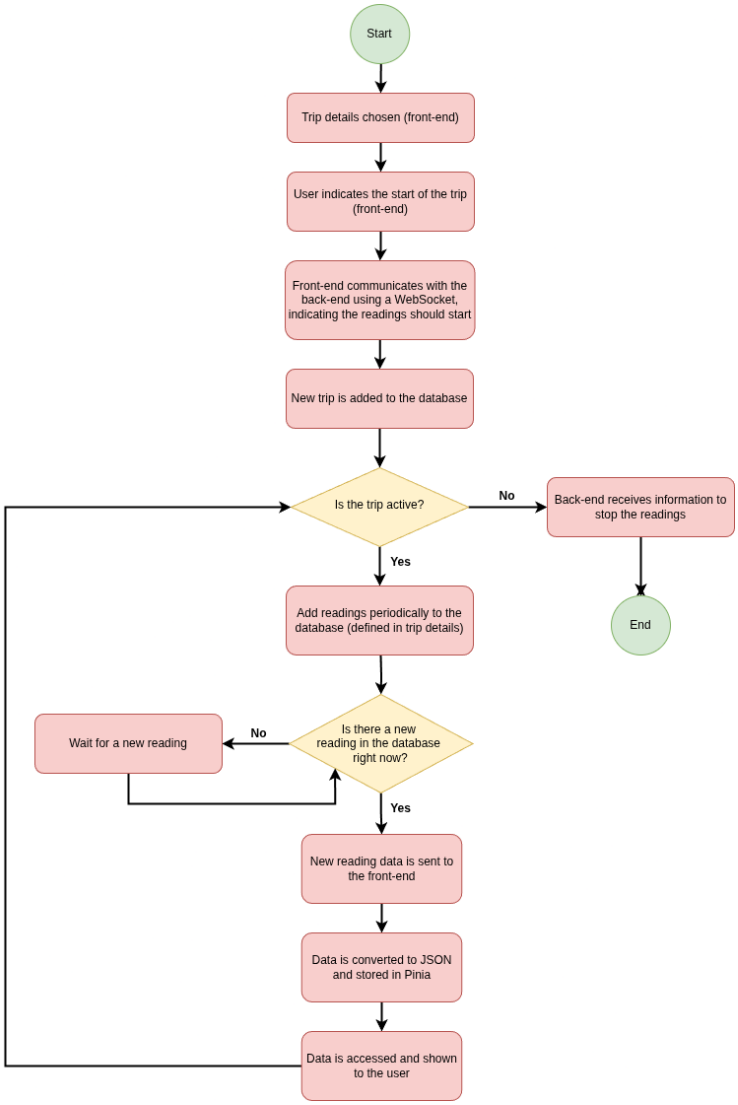


Figure. 17: Flowchart of Real-Time Sensor Data Retrieval.

1. **Trip details chosen (front-end):** The process begins with the user configuring the parameters of the trip through a form in the front-end. These parameters include conditions such as the total duration of the trip (e.g., 5 minutes) and the interval between readings (e.g., every 10 seconds).

2. **User indicates the start of the trip (front-end):** Once the form is completed to custom preference, the user initiates the trip process through the front end interface.

3. **Front-end communicates with the back-end using a WebSocket, indicating the readings should start:** Upon initiation, a message is sent from the front-end to the back-end via WebSocket, signalling the system to begin collecting sensor readings.

4. **New trip is added to the database:** A new trip entry is created in the SQLite database, initializing the data recording session.

5. **Is the trip active?:** From this point forward, the system continuously checks whether the trip is active. The flowchart diverges based on the trip's status:

   – **(If the trip is no longer active) Back-End receives information to stop the readings:** The system stops collecting sensor readings. This can occur when one of the following conditions is met: the pre-defined time duration is exceeded, a maximum depth threshold is reached, or the user manually terminates the trip. The back-end finalizes the data recording session and closes the trip, concluding the real-time data acquisition process.

   – **(If the trip is active) Add readings periodically to the database (defined in trip details):** Sensor readings are periodically collected and stored in the database based on the user's initial configuration (e.g., every 5 seconds, every 5 m). The real-time data handling proceeds as follows:

      (a) **Is there a new reading in the database right now?:** The system checks for a new sensor reading in the database:

         - **(If a new reading is available) New reading data is sent to the front-end:** The data is immediately sent to the front-end through the WebSocket connection.

         - **(If no new reading is available) Wait for a new reading:** The WebSocket remains open and waits until a new reading is recorded, repeating the check.

      (b) **Data is converted to JSON and stored in Pinia Store:** Once received, the data is parsed into JavaScript Object Notation (JSON) format and stored in the Pinia Store for state management.

      (c) **Data is accessed and shown to the user in the front-end:** The stored sensor data in Pinia is then accessed by various components of the front-end and displayed to the user in real time (e.g. graphs, tables). The system then returns to step 5 to continue the loop.

Concisely, the entire process relies on a back-end WebSocket connection to deliver continuous, real-time sensor data to the front-end client as soon as it is stored in the database.

RESTful API (Blue Arrows)

The RESTful API facilitates communication between the front-end and back-end components of the system. This operation starts when a certain type of system operation is done by the user or independently by the application, where a corresponding request is sent to the back-end. The back-end then interacts with the appropriate data source (whether it be sensors, the current measurement module or the database). Once the requested data is retrieved, it's returned to the front-end through a structured API response, enabling the user interface to reflect the latest system status or data.

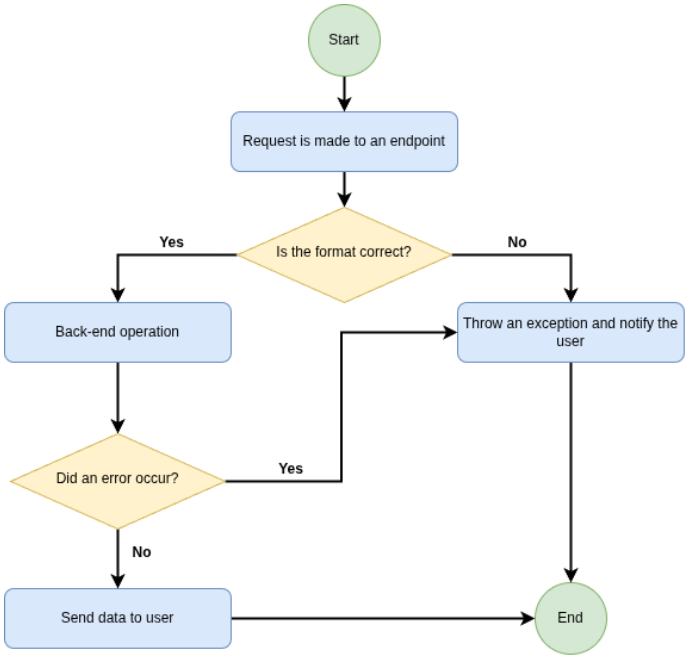


Figure. 18: Flowchart of the RESTful API.

The flowchart illustrated in Figure 18 expands on this process. It details how certain requests are made to the back-end by the user, or automatically by the system. The steps below assume the user is connected to the back-end.

1. **"Request is made to an endpoint":** The first step in the flowchart involves the initiation of a request, which may be triggered either by the user or automatically by the system to a specific endpoint. User-initiated requests typically involve actions such as retrieving all trips and their associated data from the database, modifying sensor parameters, or executing sensor calibration. In contrast, system-initiated requests are used to periodically update real-time values displayed in the interface (such as battery level, current depth, and CPU usage). Two

types of HTTP requests are employed in this process: GET requests, used for retrieving data without requiring additional parameters, and POST requests, used when the front-end must transmit data to the back-end in order to receive a response.

2. **"Is the format correct?":** This conditional step verifies whether the incoming request adheres to the expected format. Although the system is designed to minimize the occurrence of improperly formatted requests, this check serves as a precautionary measure to ensure robustness. From here onward, the flowchart diverges based on the format of the request:

   – **(If the format is incorrect) "Throw an exception and notify the user":** If the request contains formatting errors, the server raises an HTTP exception. To notify the user, an error pop-up is shown indicating that the operation has failed and the request process has finalized unsuccessfully.

   – **(If the format is correct) "Back-end operation":** The system processes the incoming request and performs the corresponding operation, which may include sensor configuration, sensor calibration, retrieval of database records, access to a specific sensor reading, or querying the current battery percentage.

      (a) **(If no errors occur) "Send data to the user":** The server returns an HTTP 200 status code, along with the requested data. The user is also notified that the requested process was completed correctly.

      (b) **(If an error occurs) "Throw an exception and notify the user":** Similarly to what occurs in an incorrect format, an exception is raised, and the user is notified.

In summary, the flowchart depicts a conventional RESTful API process: structured HTTP requests are sent to predefined endpoints, validated, processed by the back-end, and responded to with appropriate data or exceptions.

3.3 Database ER Diagram

Following the design of the system architecture, an entity-relationship (ER) diagram, shown in Figure 19, was developed to define the structure of the SQLite database responsible for storing sensor data and associated readings. Unlike more detailed relational database management systems, SQLite supports only four primitive data types: binary large object (blob), integer, real, and

text. The absence of common types such as boolean and timestamp introduces certain challenges, particularly in representing boolean values and temporal data, which are addressed in the design described below.

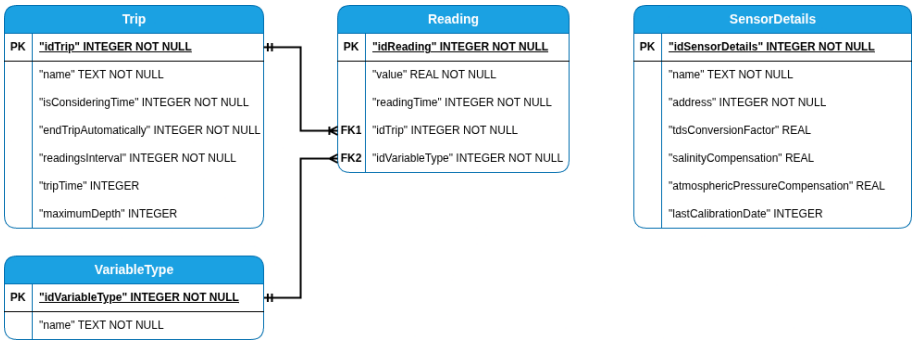


Figure. 19: Entity-relationship model of the database.

Trip Table

This table stores data related to each recording session, including user-defined parameters that determine how data acquisition was conducted. The table is structured around a primary key, **idTrip**, which uniquely identifies each trip. In addition to this key, the table contains the following fields:

- **name**: A non-null text field that stores the name of the trip. This name serves as an identifier for easier retrieval and review of past recordings. The file name follows the format TRIP-[TIME]-[DATE].txt, to ensure name uniqueness and guarantee easier traceability.

- **isConsideringTime**: A non-null integer field that indicates whether the trip uses time-based or depth-based sampling. Due to the lack of a native boolean type in SQLite, this field uses 0 to represent false (depth-based recording) and 1 to represent true (time-based recording), in accordance with system requirements.

- **endTripAutomatically**: A non-null integer field that determines whether the trip should terminate automatically upon reaching a predefined threshold. A value of 0 indicates manual termination of recordings, while 1 enables automatic termination instead. This behaviour applies to both sampling methods: for time-based trips, termination occurs when the **tripTime** limit is reached; for depth-based trips, it occurs when the specified **maximumDepth** is surpassed.

- **readingsInterval**: A non-null integer field that defines the interval between sensor readings. The unit of measurement depends on the selected sampling method (in **isConsideredTime**): for time-based sampling, the value is expressed in seconds; for depth-based sampling, in meters.
- **tripTime**: An optional integer field that specifies the total duration of the trip in minutes. This field is relevant only when the trip employs time-based sampling and is set to null otherwise.
- **maximumDepth**: An optional integer field that indicates what should be the longest possible depth. This field is only applicable for depth-based sampling and is null otherwise.

VariableType Table

This table is the smallest in the database and serves to define the different types of environmental variables that the system can record. It consists of two fields: a primary key, **idVariableType**, and a non-null text field, **name**, which stores the name of each variable type. In the current implementation, the table includes a predefined set of variable types: conductivity, salinity, TDS, DO, depth, and temperature. This table was designed to facilitate data organization and expandability, enabling clear association between sensor readings and their corresponding environmental variables in a structured and maintainable manner in the **Reading** table.

Reading Table

This table records all information belonging to a single reading made from a specific environmental variable. The table is composed by a primary key, **idReading**, which uniquely identifies each reading. In addition to this key, the table is composed by the following fields:

- **value**: A non-null real that stores the value of a reading taken from a sensor.
- **readingTime**: A non-null integer that stores the precise moment a sensor reading was taken. Due to the absecnce of a native timestamp data type in SQLite, the system stores time utilizing the Unix epoch format (i.e., the number of seconds elapsed since January 1st, 1970, UTC). This approach not only allows a workaround the storage of date formats, but also allows for a more precise time tracking.
- **idTrip** (FK1): This foreign key references the **idTrip** field in the **Trip** table. The relationship is one-to-many, where a single trip can have multiple sensor readings, but each reading is associated with only one trip.

- **idVariableType** (FK2): This foreign key references the **idVariableType** field within the **VariableType** table. The relationship is also one-to-many: a single variable type can appear in multiple different readings, but each reading corresponds to exactly one variable type.

SensorDetails Table

This table is the only table in the database schema that does not maintain a relationship with any other table. It's primary purpose is to store metadata and configuration parameters related to the sensors used in the system. Initially, the database only included the **Trip**, **Reading**, and **VariableType** tables, as the pressure sensor has no possible configuration or mutable parameters as evident by the sensor documentation [86]. However, after reviewing the documentations for the conductivity [87] and DO [84] sensors, it became evident that these probes were highly configurable, with the majority of configurable parameters (with the exception of temperature compensation) persisting in the sensor's embedded circuit even after power cycles. Given that parameters such as TDS conversion factor, salinity compensation and atmospheric pressure compensation remain stored on sensor hardware, the system does not need to continuously query the sensors for these values. Instead, when a user modifies these configuration parameters on the sensor via the GUI, the new values are stored in the database for reference and display purposes.

- **name**: A non-null text field that stores the name of the sensor.
- **address**: A non-null integer field responsible for storing the I2C address assigned to the sensor on the communication bus.
- **tdsConversionFactor**: An optional real field that stores the TDS conversion factor for the conductivity sensor. For the DO sensor, this field remains null.
- **salinityCompensation**: An optional real field that stores the salinity compensation value for the DO sensor. This field is null for the conductivity sensor.
- **atmosphericPressureCompensation**: An optional real field that stores the atmospheric pressure compensation value for the DO sensor. For the conductivity sensor, this value is null.
- **lastCalibrationDate**: A non-null integer field that records the timestamp of the sensor's most recent calibration. As with **readingTime** in the **Reading** table, this value is stored in Unix epoch format.

3.4 Development

After defining the system architecture and database, the system could be developed through the integration of the previously mentioned peripherals and development of the user application.

3.4.1 Initial Setup

To start development and integration of sensors to the apparatus, work began on configuring the RPi to the system's needs.

Ethernet

To enable reliable communication between the RPi and external systems via a wired Ethernet connection, it was necessary to configure the eth0 network interface with a static IP address. This configuration ensures consistency in network access, which is essential for stable communication and system integration.

Upon initial system boot and execution of the **ifconfig** command across multiple sessions, it was observed that the IP address assigned to eth0 varied dynamically. This variability, resulting from the default behaviour of dynamic IP allocation via the Dynamic Host Configuration Protocol (DHCP), posed a challenge to maintaining consistent network connectivity. To address this, the **dhcpcd.conf** configuration file was edited using the command **sudo nano /etc/dhcpcd.conf**.

DHCPCD is a client daemon that automates the assignment of IP addresses, subnet masks, default gateways, and DNS servers. By modifying the configuration file, a static IP address was assigned to the eth0 interface, as shown below:

```
interface eth0
static ip_address=192.168.137.181/24
static routers=192.168.137.1
static domain_name_servers=8.8.8.8
```

Listing 1: Static IP Configuration for eth0

These directives ensure that the eth0 interface, responsible for wired Ethernet connectivity, is consistently assigned the IP address **192.168.137.181** with a subnet mask of /24 (equivalent to **255.255.255.0**). The **static routers** directive sets the default gateway to **192.168.137.1**, which corresponds to the local network router and facilitates routing of packets destined for external networks, with this setting being crucial to maintain a consistent address for router integration.

Finally, the **static domain_name_servers** line specifies Google's public DNS server (**8.8.8.8**) to resolve domain names into IP addresses.

I2C

To enable communication between the RPi and the sensors, it was necessary to activate and configure the I2C interface. The following steps were tackled to achieve this:

1. Access the system configuration utility by executing the command: **sudo raspi-config**.
2. Navigate to **Interface Options** → **I2C**, and enable the I2C interface.
3. Install the I2C tools package using the command: **sudo apt-get install i2c-tools**.
4. Modify the RPi hardware configuration file in the path: **/boot/firmware/config.txt**, and append the following line: "dtoverlay=i2c0". This configuration enables the use of an additional I2C bus (bus 0), complementing the default bus 1. The decision to use two I2C buses instead of one was made due to conflicts arising from having the *Atlas Scientific* sensors and the *Keller 4LD* sensor in the same bus.
5. Reboot the RPi through **sudo reboot now** to apply the changes.

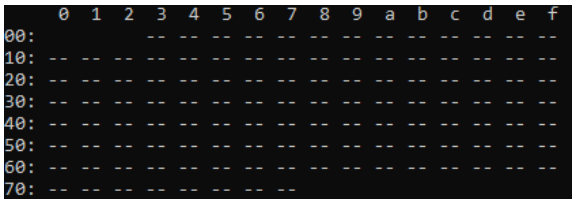


Figure. 20: Empty I2C Bus.

To verify that the I2C interfaces have been correctly enabled, the commands **sudo i2cdetect -y 1** (for bus 1) and **sudo i2cdetect -y 0** (for bus 0) can be executed. If both return output similar to that shown in Figure 20, it indicates that the I2C buses are successfully active and ready for use.

Peripheral Integration

Once I2C communication was enabled on the RPi, the subsequent step involved the physical integration of all peripheral sensors necessary for data acquisition. The wiring diagram presented in Figure 21 illustrates the electrical connections established between the sensors and the RPi's GPIO header (with a more detailed pinout of the GPIO header previously provided in Figure 14).

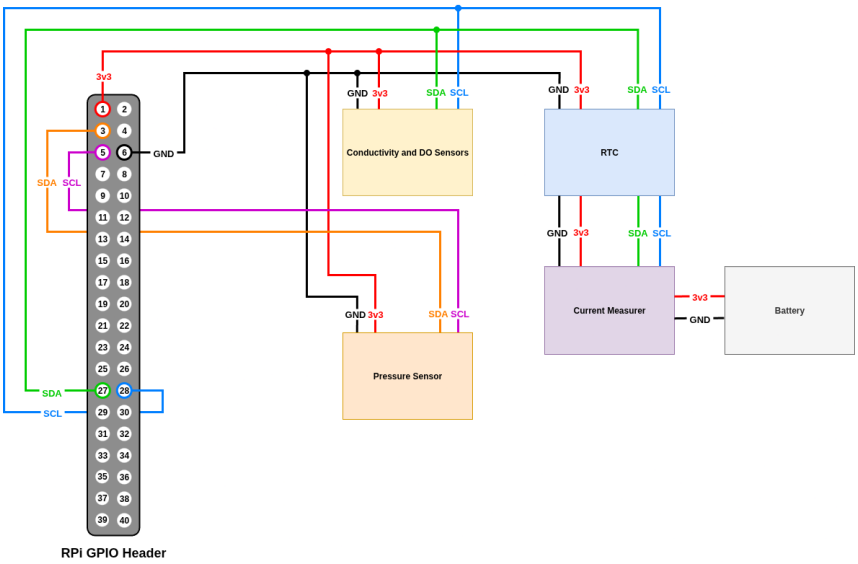


Figure. 21: Peripheral Integration Diagram

As shown in the diagram, the conductivity and DO sensors, the RTC, and the current measurement module are all connected to I2C Bus 0. These peripherals share four jumper wires: red wire suppling 3.3 Voltage at the Common Collector (VCC) from pin 1; the black wire for ground (GND) connected to pin 6; the green wire for the data line (SDA) connected to pin 27; and blue wire for the clock line (SCL) connected to pin 24.

The pressure sensor, although powered through the same 3.3V and GND lines, is connected to a separate I2C bus (Bus 1) to avoid address conflicts and ensure proper communication. In this setup, the orange wire corresponds to the SDA line connected to pin 3 and the purple wire to the SCL line connected to pin 5.

In the graph, the battery is connected to the current sensor via only the 3.3V and GND lines. This minimal connection is sufficient, as the purpose is solely to monitor the current being drawn by the system rather than communicate data via I2C. Power is supplied to the RPi by connecting its USB-C power input to the USB output of the current measurement module. This configuration enables real-time monitoring of the system's power consumption while simultaneously powering the device. As previously explained, in the current implementation, the current measurer is not yet connected to the system, and the battery is also not currently integrated, as testing is still being done in-vitro and a wall outlet power source is more reliable for testing. When integrated into the system, the current sensor will be assigned to I2C bus 0, as illustrated in the corresponding figure. This configuration is necessary because it shares the same default address as the pressure

sensor. By allocating these devices to separate buses, potential address conflicts and communication interference are avoided.

Once all sensors were physically connected, the I2C communication buses were verified using the **sudo i2cdetect -y 0** and **sudo i2cdetect -y 1** commands, corresponding to bus 0 and bus 1, respectively. The resulting device address maps, confirming successful detection of the connected peripherals, are presented in Figure 22.

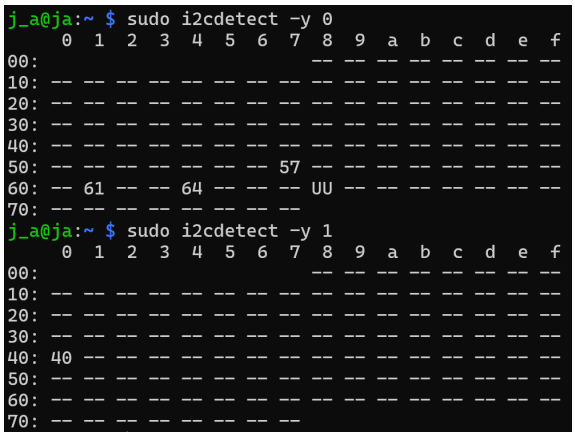


Figure. 22: Both I2C buses after connection

On I2C Bus 0, the address 0x61 (97 in decimal) indicates the presence of the DO sensor, while the address 0x64 (100 in decimal) corresponds to the conductivity probe. The addresses 0x57 and 0x68 are associated with the RTC module. Specifically, 0x57 refers to the module's built-in EEPROM (which is not utilized in the current implementation) whereas 0x68 (104 in decimal) is the address of the RTC itself. In the I2C scan output, this address appears as UU, indicating that it is currently in use by a driver in the system kernel. This status change occurred after the RTC module was configured as the device's default system clock, replacing the RPi's pre-installed fake hardware clock.

On I2C Bus 1, only the Keller 4LD sensor is connected, identified by the address 0x40 (64 in decimal). Following this configuration, all sensors are ready for operation. The developed system scripts are capable of initiating communication with each sensor by referencing their respective I2C addresses.

3.4.2 User Application

After most hardware and RPi configuration had been concluded, work began on developing the user application to operate the system. As such, a prototype was developed first, and then a final version was implemented.

3.4.2.1 Prototype

Initially, a prototype was developed[2] and co-designed with the domain expert to ensure accessibility for researchers with varying levels of technical expertise. The layout was divided into three distinct sections: **Sidebar**, **Top Bar**, and **Main Content**. Both **computer** and **smartphone** versions were outlined, with the computer version addressed first.

Sidebar

(a) Collapsed (b) Expanded

Figure. 23: Sidebar (Prototype)

The sidebar operates in two states: collapsed and expanded, as shown in Figure 23. The collapsed state, shown in (a), is composed of an arrow button responsible for the expansion of the sidebar, and icons that allow navigation to specific pages. The expanded state, shown in (b), displays the system name, the name of the pages in front of each icon, and a clock at the bottom that provides current date and time for user convenience. At this stage, the system was designated as Multi-Purpose Sensing System (MPSS).

Top Bar

The top bar (Figure 24) provides important information and controls. The left section displays the name of the current page, keeping users oriented within the application. The right section showcases

[2] `https://figma.fun/FqRXFM`

essential system metrics in colour-coded labels: current depth (orange), battery level (green) and usage CPU (yellow). Additionally two icons are positioned on the right: a circular arrow button for system reboot and a settings button for accessing system preferences. Displaying these metrics coincides with previously established requirements that certain variables should always be visible, as well as functionality for rebooting the system.

Figure. 24: Top Bar (Prototype)

Main Content

The main content area dynamically adapts to the active page. Five primary sections were developed in the prototype: **New Trip Pages**, **Configure Sensors Pages**, **Saved Data Page**, **Manual Page**, and **Settings Page**.

- **New Trip Pages:** Represents the sensor recording process during trips. Users utilize the New Trip page to configure trip settings as defined in system requirements, and proceed to the Real-Time Trip page, where live sensor data, trip settings, and system logs are displayed.
- **Configure Sensors Pages:** Allows users to select which sensor they wish to adjust and calibrate in the Configure Sensors page. After selecting a sensor users are redirected to the Configure Sensor page.
- **Saved Data Page:** Enables users to access downloadable CSV data and Log files. It also includes functionality to delete undesired data, to ensure storage management by the user.
- **Manual Page:** Serves as a comprehensive guide, providing detailed information about each system section. Users can select specific topics or pages to view relevant instructions.
- **Settings Page:** The Settings page includes various system customization options such as colour scheme and a theme (dark and light mode). This section was created to give users further liberty with their customization and also allow for better visibility in certain conditions.

Smartphone Version

According to user requirements, mobile compatibility with smartphones (and tablets) is important, and as such a simple layout was developed. The prototype layout is in landscape, with different page

components (in the image shown, "Current Values") being aligned vertically, instead of horizontally and vertically like in laptop oriented pages.

3.4.2.2 Final Version

Following the definition of the prototype, development of the system commenced using previously discussed technologies. All developed code and configuration files can be found through a Git repository in Bitbucket[3]. Throughout development, the domain expert was consulted as a means of improving certain aspects, with the addition or removal of certain functionalities not accounted for in the system requirement phase. The **Sidebar** retained its original structure from the prototype, and as such there will not be an in depth explanation to it, as doing so would be redundant. Therefore, the focus of this section will be on the various main pages comprising the **Top Bar** and the **Main Content** areas, as outlined in the prototype, along with their respective functionalities. Finally, a brief overview of the **smartphone version** of the interface is provided.

Top Bar

The **Top Bar** component maintained a design consistent with the version presented in the prototype (Figure 25).

Figure. 25: Top Bar

On the left-hand side, the current page name is still displayed to orient the user within the application. Initially, the right-hand side presents the same system parameters as before, depth (orange), battery level (green), and CPU usage (yellow), which are retrieved through periodic HTTP GET requests to the back-end. With the exception of the battery values, which is still a placeholder, the other two values are retrieved from their respective sources. The difference that was introduced in the final version was the ability to both reboot and power off the device directly from the interface, as illustrated in Figure 26. While the prototype only supported system reboot, including a shutdown option was also of importance by, enabling researchers to power down the device after field operations to save power. The settings button remained unchanged.

New Trip Pages

[3] `https://bitbucket.org/wave-mps/iot/src/main/`

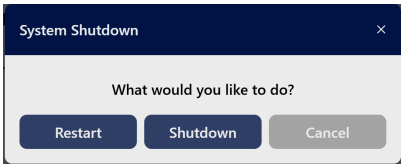
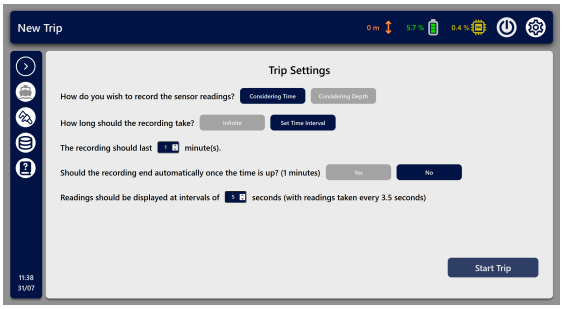
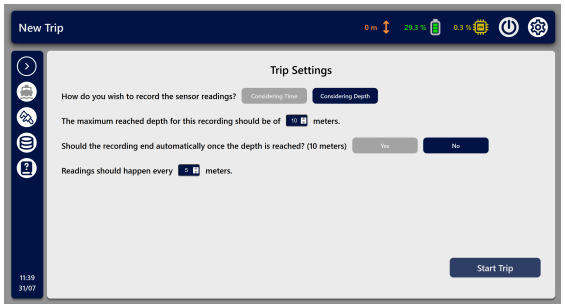


Figure. 26: Reboot and Shutdown Modal

Similarly to the prototype, **New Trip Pages** is composed by a page of the same name, and by **Real-Time Trip** page, where data is displayed and seen by the user. The **New Trip** page is composed of a **Trip Settings** component with configurable options for initiating data collection.



(a) Time-based Options



(b) Depth-Based Options

Figure. 27: New Trip Page

As illustrated in Figure 27, users can choose between time-based and depth-based recording modes. In Figure 27a time-based readings include an indefinite recording mode and a mode with a predefined duration, in which case the system automatically stops the recording once the specified time has elapsed. Users can also define the display interval for incoming data, in seconds. Notably, this parameter was modified from the prototype to the final version to account for delays in sensor readings, introducing an approximate 1.5 second delay between data acquisition and visualization. On the other hand, Figure 27b presents the depth-based configuration options. Here, users can specify a maximum depth threshold for data collection. If desired, the system can be configured to automatically terminate the recording once this depth is reached. Similar to time-based configuration, users can define the interval between depth-based readings, but using meters as a metric instead. After the user chooses the parameters that will define the recording process, the data is then sent to the back-end WebSocket, where the server will handle the recording operations and save trip details to the database, as previously shown in the flowchart in Figure 17. The defined parameters are also stored temporarily in the Pinia store as a means of allowing the front-end application to be aware of trip status and details.

The **Real-Time Trip** page is composed of two components: **Trip Information** and **Sensor Graph**. The Device Log component from the prototype was removed due to not being necessary, according to the domain expert, and the **Trip Settings** was combined with **Current Values** (also from the prototype) to form the final **Trip Information** component.

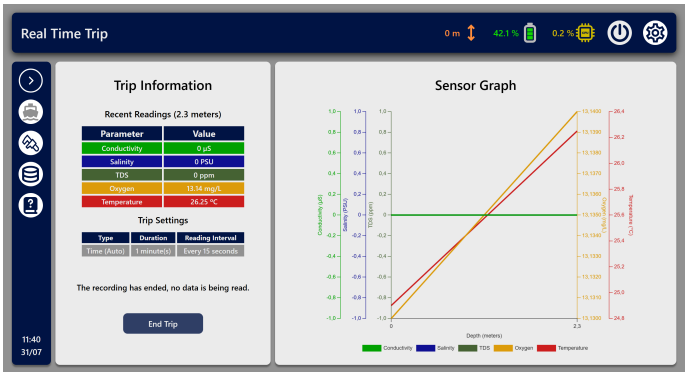


Figure. 28: Real-Time Trip Page

The **Trip Information** component is designed to present important details about an ongoing trip in a structured manner. At the top, a table displays the most recent sensor readings. Directly beneath it, another table outlines the trip parameters as defined by the user. Below these, a status message provides real-time feedback on the trip's progress, and a button is available to allow the user to manually terminate the trip at any time. As illustrated in the example, the displayed trip is time-based, configured to run for one minute with sensor readings captured at five-second intervals. The status message includes a countdown timer indicating the remaining duration of the trip before it ends automatically. Another feature present in this component, is that when a user clicks on a value from the recent readings table, a modal appears showing the margin of error associated with each reading (example is shown in Figure 29).

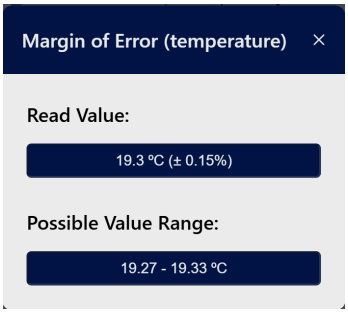


Figure. 29: Margin of Error Modal

The **Sensor Graph** component visualizes the collected data using a multi-axis chart implemented with Chart.js. It features five y-axes, each corresponding to a specific recorded variable, and a shared x-axis representing depth. Users can toggle the visibility of each variable by interacting with the labels positioned beneath the graph. The graph allows for value display on hover and real-time updates, which occur whenever a new sensor reading is received. Figure 30 shows back-end sensor data being stored in Pinia where it is then fetched and utilized to be either displayed graphically, or tabularly. The server process of retrieving sensor data and storing it in Pinia occurs continuously until readings either stop manually or automatically where the WebSocket stops sending data.

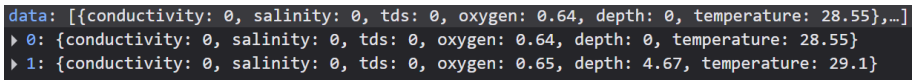


Figure. 30: Pinia Store recording

To prevent the initiation of concurrent trips or the possibility of leaving an ongoing trip unstopped, access to certain pages is restricted based on the trip status. Specifically, while a trip is active, the user is unable to access the **New Trip**. Conversely, when no trip is active, the **Real-Time Trip** page is inaccessible. To also ensure that no trips start concurrently, only the user that first connects to the system can start and stop trips through his device, with other users being prohibited of starting trips until the first one disconnects. This design ensures that only one trip can be managed at a time, avoiding data problems, operational inconsistency and sensor communication problems.

Configure Sensors Page

In contrast to the prototype, the **Configure Sensors** was consolidated into a single page, retaining the same name, as shown in Figure 31. This design decision aimed to simplify the user interface by reducing the number of navigable pages, thereby enhancing usability. To maintain functionality while minimizing complexity, modal dialogues were adopted for specific user interactions.

The Configure Sensors page consists of two primary components: **Choose Sensor** and **Sensor Configuration**.

The **Choose Sensor** component provides three buttons, each corresponding to a specific sensor that the user may wish to configure or visualize sensor information.

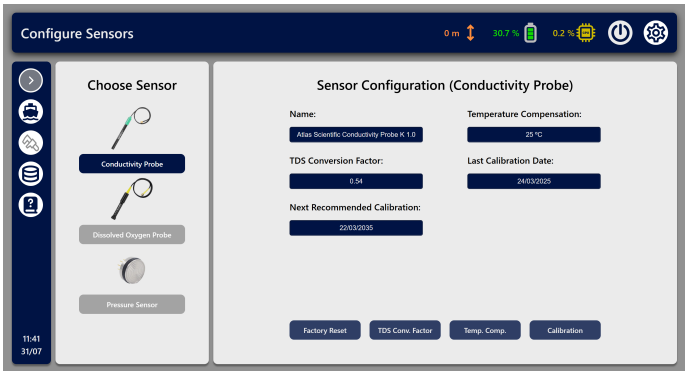
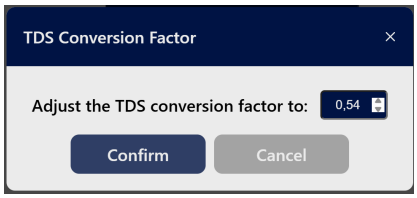


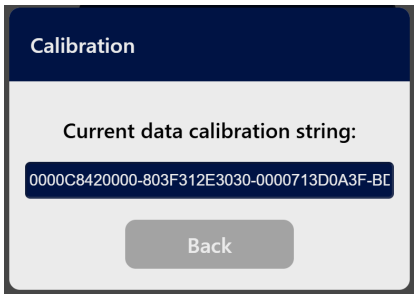
Figure. 31: Configure Sensors

The core functionality, however, is located within the **Sensor Configuration** section. Here, users can view detailed information about the selected sensor, including its name, configuration parameters, and the date of its last calibration. This section also allows users to perform various configuration (e.g. TDS conversion factor, temperature compensation) and calibration tasks, as illustrated in Figure 32. These operations are supported only for the conductivity and DO sensors, as the pressure sensor lacks configurability or calibration capabilities as previously mentioned.

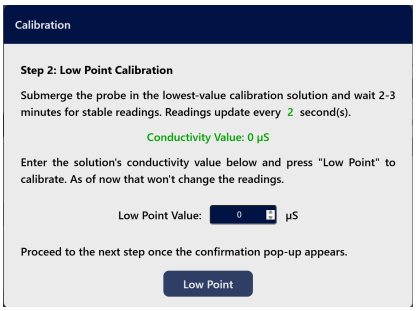


(a) Configure Specific Parameters

(b) Calibration Operations



(c) Export Calibration String



(d) Manual Calibration Example

Figure. 32: Calibration Options

For example, Figure 32a displays a modal box that enables users to modify the TDS conversion factor of the conductivity probe. Additionally, Figure 32b presents the available calibration operations, including importing or exporting a calibration string and executing manual calibration

procedures in a step-by-step manner. As seen in Figure 32c, calibration strings are predefined by the probe's internal firmware and represent sensor-specific calibration data. These strings consist of a sequence of hexadecimal values arranged in a specific order, which the sensor's integrated circuit interprets to apply or restore calibration settings. When a calibration string is imported, it is transmitted directly to the sensor via the communication interface, allowing for rapid restoration of previous calibration states without manual intervention. However, in certain cases manual calibration might be desirable, and as seen in Figure 32d the application guides the user in a step-by-step tutorial in how to calibrate sensor as instructed per sensor documentation [84,87]. All sensor operations are executed through HTTP requests sent to the RESTful API endpoints, using either the GET or POST method depending on the nature of the operation. Upon successful execution, the API responds with a success message; otherwise, it returns an appropriate exception. This request-response behaviour is outlined in the RESTful API flowchart presented in Figure 18.

Sensor configuration and calibration are only permitted when no trip is active and can only be done by the user that first connected to the system. This constraint was introduced to prevent potential interference with sensor operation during active data collection and to also ensure that no two users were changing parameters of the sensor at the same time. By restricting these actions, the system ensures that sensor settings remain consistent and stable throughout the recording process, thereby preserving data integrity and minimizing the risk of configuration-related errors.

Saved Data Page

The **Saved Data** page had several modifications when compared to the initial prototype. Notably, the **Log Files** section was removed following feedback from the domain expert, who indicated that log-related information in this section was not relevant for marine researchers. Its presence was considered unnecessary and potentially confusing, adding complexity without contributing meaningful value. Consequently, the final design (illustrated in Figure 33) was changed to include only two main sections: **CSV Files** and **Sensor Graph**, both recommended by the same expert.

The **CSV Files** component includes a search bar for filtering the list of recorded trips displayed in the table below. This table consists of two columns: "Name", which follows the naming convention defined in the creation of the database diagram, and is clickable to reveal trip details through an HTTP request to the back-end where SQLite data is fetched through a query (as shown in Figure 34); and "Action", which includes three buttons, each with a distinct purpose.

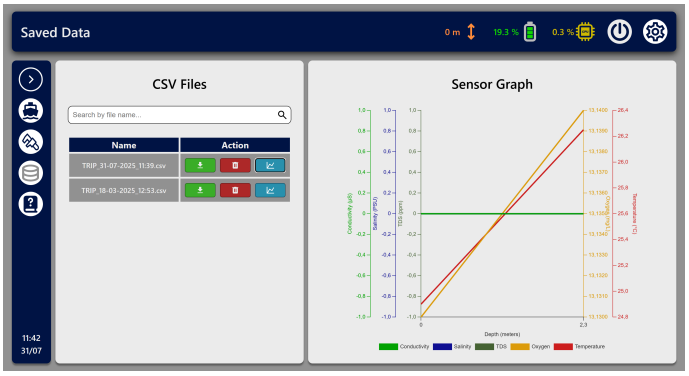


Figure. 33: Saved Data Page

Figure. 34: Trip Details

The green download button triggers an HTTP request to a back-end REST endpoint, dynamically generating a CSV file from the corresponding trip data stored in the SQLite database through the RPis memory. The red delete button initiates a confirmation modal, warns the user that deletion of a specific trip is permanent. If confirmed, it sends a deletion request to the back-end, removing all associated trip data from the database. Lastly, the blue graph display button, added through domain expert suggestion, displays the selected trip data in a visual format, similar to that used in the **Real-Time Trip** page, enabling users to analyse the recorded values more intuitively. All of these operations also follow the RESTful API flowchart in Figure 18.

Data from the most recent trip cannot be downloaded while the trip is still active. This constraint was implemented to ensure data consistency and to prevent users from accessing incomplete or potentially unreliable information before the recording session has concluded.

Manual Page

The **Manual** page (Figure 35) follows the structure of the prototype, containing a single component titled **System Information**, which provides information to guide users utilizing the application.

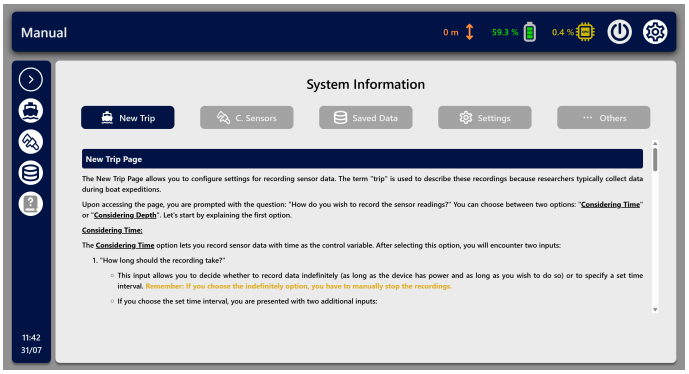
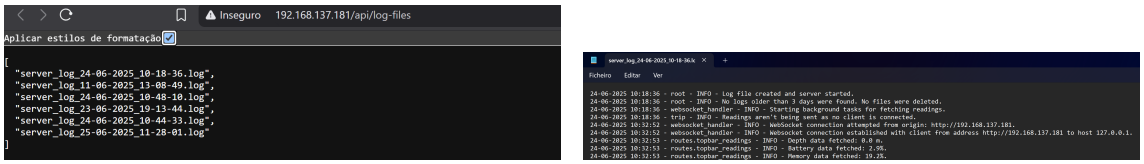


Figure. 35: Manual Page

The content is organized into five categories corresponding to the main pages of the system, "Settings" and an additional "Others" category. This latter section includes explanations of different interface elements such as the top bar and sidebar.



(a) Log list (b) Example of log file

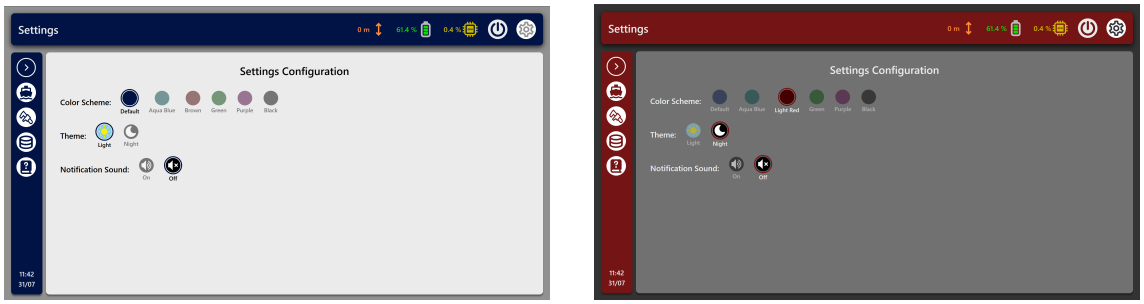
Figure. 36: Log operations

Furthermore, a simple explanation in how to access system logs (in a case of necessity) has been added to this section, where users are instructed on how to access them through accessing a RESTful endpoint through their browser (as shown in Figure 36a, with a Log file example in Figure 36b). This design decision was made to ensure that system logs remain accessible and downloadable without the need for direct access to the RPi via SSH or the use of external peripherals such as a keyboard and monitor, even if they are not directly present on the front-end application as requested.

Settings Page

The **Settings** page remained largely consistent with the version presented during the prototyping phase and is composed solely of the **Settings Configuration** component.

The system allows users to customize the interface through twelve colour schemes (six designed for light mode and six for night mode) as illustrated in Figure 37. Since some colours vary significantly in contrast and visibility depending on the active theme, the system dynamically adjusts the



(a) Default settings page

(b) Settings page with night theme and colour scheme

Figure. 37: Comparison between two visual configurations of the Settings page

available colour options to preserve usability and visual clarity. An example of this adaptability is shown on Figure 37b, where the colour scheme has been set to light red, and the theme has been switched to night mode.

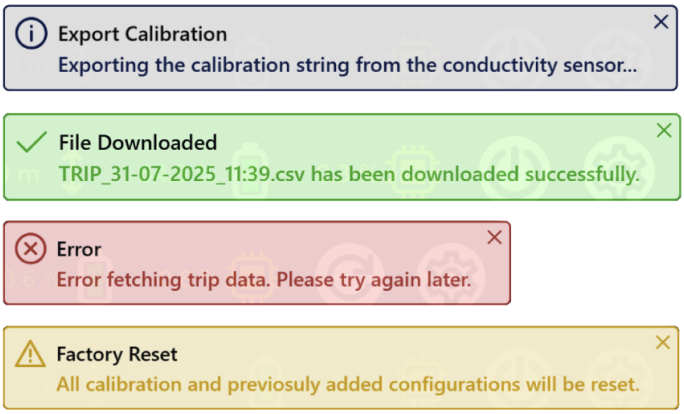


Figure. 38: System Notifications

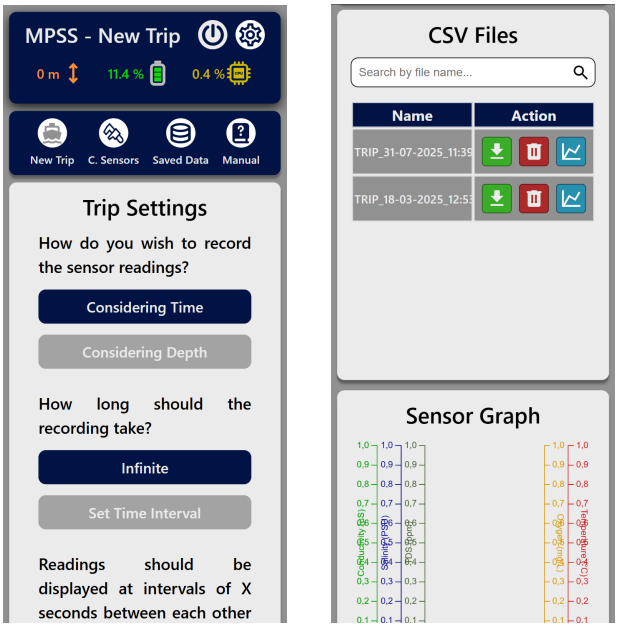
The most significant addition to the **Settings** page was the option to enable or disable notification sounds. These notifications refer to the toast messages displayed within the interface to inform the user and provide immediate feedback on system operations. The various types of system notifications are illustrated in Figure 38, presented from top to bottom.

The first toast, highlighted in the currently selected application colour (as defined in the colour scheme settings), serves an informational purpose. In the example shown, it notifies the user that the calibration string of the conductivity sensor is being exported, thereby preventing the misperception that the application has become unresponsive or encountered an error. The second toast, displayed in green, signifies a successful operation. As illustrated, it informs the user that a CSV file has been successfully downloaded. The third toast is red, indicating an error. In the provided example,

it alerts the user that an issue occurred while retrieving trip data from the database. Lastly, the fourth toast appears in yellow and serves as a warning. It cautions the user that certain actions may have consequences or lead to unexpected outcomes. In the figure, the system warns that performing a factory reset on the sensor will erase all calibration and configuration data associated with that device. All notification toasts are accompanied by an alert sound to more effectively draw the user's attention. However, this most recent setting allows users to disable the notification sound should they find it unnecessary or intrusive.

Smartphone Version

The smartphone version of the application varies from the initial prototype. The biggest change is that it defaults to portrait orientation, rather than landscape, aligning more appropriately with standard smartphone usage patterns. Furthermore, the layout was redesigned to offer a more concise and space-efficient interface, as illustrated in Figure 39a.



(a) New Trip (b) Saved Data

Figure. 39: Smartphone Version

To optimise the limited screen dimensions, the **Top Bar** was restructured to efficiently accommodate its various elements. Additionally, the **Sidebar**, which in the desktop version was positioned to the left, was relocated beneath the **Top Bar**. In the mobile version, only the es-

sential navigation buttons were maintained, while secondary information (such as date, time, and system name) was omitted to reduce visual clutter and facilitate user interaction.

Moreover, the overall layout transitioned from a predominantly horizontal arrangement to a vertical distribution of components (seen in Figure 39b when compared to desktop version in Figure 33). This change was specifically intended to enhance usability on devices with narrower screens, ensuring that the interface remains accessible and user-friendly across different smartphone dimensions.

3.4.3 NGINX Configuration

Following the development of both the front-end and back-end components, it was necessary to integrate these services using NGINX. The steps involved in configuring NGINX for this purpose are as follows:

1. Install NGINX by executing the following command: **sudo apt install nginx**.
2. Open the main configuration file located at **/etc/nginx/nginx.conf**.
3. Modify the configuration file to match the setup shown below:

```
user www-data;
worker_processes auto;
pid /run/nginx.pid;
error_log /var/log/nginx/error.log;
include /etc/nginx/modules-enabled/*.conf;

events {
     worker_connections 768;
}

http {
    sendfile on;
    tcp_nopush on;
    types_hash_max_size 2048;

    include /etc/nginx/mime.types;
    default_type application/octet-stream;

    ssl_protocols TLSv1 TLSv1.1 TLSv1.2 TLSv1.3;
    ssl_prefer_server_ciphers on;

    access_log /var/log/nginx/access.log;
    gzip on;

```

```
    include /etc/nginx/conf.d/*.conf;

    server {

        # HTTP port (80) and eth0 interface address (192.168.137.181)
        listen 80;
        server_name 192.168.137.181;

        # Location of compiled front-end files
        root /var/www/vuejs-app/dist;
        index index.html;

        location / {
            try_files $uri /index.html;
        }

        # RESTful API address and configuration
        location /api/ {
            proxy_pass http://127.0.0.1:8000/;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
        }

        # WebSocket address and configuration
        location /ws/ {
            proxy_pass http://127.0.0.1:8000;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
        }

        gzip_types text/plain application/javascript;
        gzip_min_length 256;

        add_header X-Frame-Options "SAMEORIGIN";
        add_header X-XSS-Protection "1; mode=block";
        add_header X-Content-Type-Options "nosniff";
    }
}
```

Listing 2: NGINX Configuration for Frontend and Backend Routing

As shown in the configuration above, some adjustments were required. The server was configured to listen on port 80 (HTTP) and assigned the IP address of the eth0 interface to allow access via Ethernet. HTTPS was not implemented, as the system operates within a pri-

vate, controlled network and does not have strict security requirements. The root path was set to point to the directory containing the compiled front-end files, ensuring NGINX could serve them to clients. Additionally, reverse proxy configurations were defined for the RESTful API (under the /**api**/ path) and for WebSocket connections (under the /**ws**/ path), enabling communication between the client and the back-end services.

4. To apply the configuration and ensure the NGINX service is running and initiates at boot time, the following command was executed: **sudo systemctl start nginx**.

After this configuration, the RPi is now set up to automatically serve the front-end and reverse-proxy back-end components of the system upon boot, ensuring the application is ready for use immediately after startup at **http://192.168.137.181:80**/.

3.4.4 Back-End Server Service

Following the configuration of NGINX, the front-end interface becomes accessible automatically upon system boot. However, the back-end server, responsible for handling API requests, WebSocket communication, and database interactions still requires manual activation. To ensure full system availability at startup, a dedicated systemd service was implemented to automate the back-end server's initialization process. The setup procedure involves the creation of a systemd service unit file as follows:

1. Create the service configuration **back-end-server** file through the command: **sudo nano /etc/systemd/system/back-end-server.service**.

2. Edit the file to the following configuration:

```
[Unit]
# Simple description to the system
Description=Back-End Uvicorn Server for FastAPI

# Start this service only after the network is available
After=network.target

[Service]
# Run as the j_a user (changes depending on the user of the RPi).
User=j_a

# Path of the application.
WorkingDirectory=/home/j_a/mpss-back-end-app

```

```
# Command to execute the server.
ExecStart=/home/j_a/mpss-back-end-app/.venv/bin/uvicorn app:app --host
    ↪ 0.0.0.0 --port 8000

# The service will always restart even if the app exits cleanly.
Restart=always

# Makes the application stop cleanly when the RPi shuts down or reboots.
KillSignal=SIGINT

[Install]
# Make this service start automatically during the boot sequence.
WantedBy=multi-user.target
```

Listing 3: Back-End Server Service

This service file is structured as follows:

- The **[Unit]** section provides a brief description and specifies that the service should only begin once the system's network is available, ensuring the server can be accessed by remote clients.
- The **[Service]** section defines the user context (j_a) under which the server will run, the working directory containing the application code, and the ExecStart command responsible for launching the server using Uvicorn. This command points to the virtual environment's Uvicorn binary and specifies the application module, binding it to 0.0.0.0 (to make it accessible over the network) on port 8000.
- The Restart=always directive ensures that the service is automatically restarted in the event of failure or unexpected shutdown.
- The KillSignal=SIGINT instruction allows the application to shut down cleanly when the system powers off or reboots.
- The **[Install]** section includes WantedBy=multi-user.target, which ensures that the service is enabled to start during the normal boot sequence.

After both the NGINX configuration file is finished, and this service is created, the system is finally starting automatically and ready for full use to users that connect to its GUI address.

3.4.5 Router Integration

With the system fully implemented and capable of starting automatically, the final step involved enabling wireless access to the system by integrating a Wi-Fi router instead of only directly by Ethernet cable. To ensure that devices connected via Wi-Fi would reside within the same local network as the RPi and be able to access the system interface hosted by the NGINX web server, the following configuration steps were carried out:

1. A personal computer was first connected to the router via either Ethernet or Wi-Fi, and the router's web-based configuration interface was accessed through its default address at **192.168.0.1** or **192.168.1.1**, as indicated in the official documentation [88].

2. The router was set to operate in Wireless Router mode, with DHCP enabled to automatically assign IP addresses to client devices joining the network.

3. Within the configuration interface, the router's LAN IP address was changed to **192.168.137.1**, as had been defined in the static routers for the eth0 interface in the RPi. The subnet mask was set to **255.255.255.0**.

4. The DHCP server settings were then modified to define the dynamic IP address pool, with a starting address of **192.168.137.100** and an ending address of **192.168.137.150**. The default gateway was set to **192.168.137.1** (i.e., the router itself), and the primary DNS server was configured as **8.8.8.8**.

Following these configurations, the LAN created by the router enables any Wi-Fi-capable device (e.g., laptop, smartphone, or tablet) to connect and interact with the system. As previously described, access to the user interface hosted on the RPi is achieved through a web browser by navigating to the system's static IP address via the NGINX server.

3.5 System Validation

After development, in order to assess whether the developed system can be meaningfully compared to existing commercial alternatives, a controlled laboratory test was conducted. Throughout this thesis, the proposed system will be referred by its given name, MPSS, while the reference device will be denoted as YSI, which refers specifically to the *YSI Professional Plus* [89] model provided by MARE [90] researchers. A CTD Rosette or a ROV could not be employed for such a test due to its unavailability and unsuitability for controlled laboratory testing, as these instruments are

typically designed for field deployments. Instead, the YSI instrument which measures the same environmental variables as the MPSS but is designed for shallower depths, was selected as the reference device to compare these EOVs. This testing is intended to ensure that the system reaches a Technology Readiness Level (TRL) of 4 [91].

3.5.1 Experimental Setup

The validation focuses on evaluating the measurement consistency of the MPSS relative to the YSI device under identical and controlled conditions. Both systems were placed side by side in an aquarium with a total volume of 23.5 L at the same depth. Various substances were introduced into the water to induce measurable variations in the parameters of interest: conductivity, salinity, TDS, temperature, and DO. Although the two instruments serve distinct purposes, with MPSS being designed for water column profiling at greater depths, while the YSI is primarily intended for shallower water sampling, it was still possible to compare the developed system's ability to retrieve environmental variables with the reference instrument. Before testing, MPSS sensors were calibrated using manufacturer-provided calibration solutions and procedures [84,87], while the YSI instrument was utilized with its existing field calibration.

This laboratory-based approach was selected for two main reasons: the MPSS is not yet equipped with its final enclosure or integrated battery, which prevents immediate field deployment; and simulating realistic depth variations in a laboratory environment without specialized equipment is challenging and beyond the scope of this thesis.

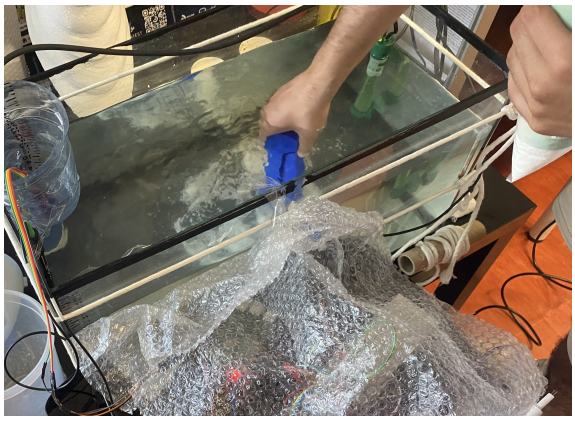


Figure. 40: Experimental Setup

To verify the comparison between instruments in controlled environments, four distinct water conditions were prepared to evaluate both systems: plain freshwater; freshwater with added salt; freshwater with added salt and sodium bicarbonate; and finally, freshwater with added salt, sodium bicarbonate, and ice. Seawater could have been used instead of freshwater, but the intent was to test the devices capabilities in salted and non-salted enviornments. For each test, the prepared environment was left to stabilize for 15 minutes to ensure proper mixing and homogenization. Figure 40 presents a photograph of the experimental setup during the testing phase. Salt was actively mixed into the water to ensure proper homogenization of the solution. As the protective casing is yet to be developed, the system was temporarily shielded using plastic wrapping to prevent water ingress and protect the electronic components from potential damage.

Both devices will then record simultaneously at 15 second intervals over a one minute period, for each condition. This duration was chosen based on perliminary results indicating a smaller variation between systems within this timeframe. The collected data is compared in the later chapter through graphical representations (dispersion plots) to asses measurement consistency and identify potential systematic offsets between the two systems.

4 Results

This chapter demonstrates the results of the previously described experiment. Across all four test scenarios, MPSS mostly tracked the temporal stability of the YSI, showing that its sensors respond predictably to changes in water composition.

4.1 Freshwater

The first test involved plain freshwater. In this test, the MPSS system exhibited stable and consistent trends comparable to the YSI reference instrument, albeit with a variation in values between both systems.

Table 6: Comparison of Conductivity (Freshwater)

Time	**MPSS (µS/cm)**	**YSI (µS/cm)**
00:00	215.7	225.6
00:15	215.7	224.8
00:30	215.9	226.7
00:45	215.7	223.7
01:00	215.8	224.9

Table 7: Comparison of Salinity (Freshwater)

Time	**MPSS (PSU)**	**YSI (PSU)**
00:00	0.1	0.11
00:15	0.1	0.11
00:30	0.1	0.11
00:45	0.1	0.11
01:00	0.1	0.11

Table 8: Comparison of TDS (Freshwater)

Time	**MPSS**	**YSI**
00:00	116	150
00:15	116	149
00:30	116	150
00:45	117	149
01:00	117	149

The conductivity values (Table 6) for both systems remained nearly constant over time, with YSI averaging 225.1 μS/cm and MPSS measuring 215.8 μS/cm, corresponding to a 4.2% difference.

Table 9: Comparison of Temperature (Freshwater)

Time	MPSS (ºC)	YSI (ºC)
00:00	22.55	23.9
00:15	22.55	23.9
00:30	22.5	23.9
00:45	22.5	23.8
01:00	22.5	23.9

Table 10: Comparison of DO (Freshwater)

Time	MPSS (mg/L)	YSI (mg/L)
00:00	2.29	4.31
00:15	2.40	4.17
00:30	4.08	4.07
00:45	4.50	3.97
01:00	4.80	3.89

As salinity (Table 7) is derived directly from conductivity, the MPSS reported 0.10 PSU compared to 0.11 PSU from the YSI, which presents an extremely small 0.01 PSU variation, which can be considered negligible in practical terms. TDS (Table 8) which also derives from conductivity values, also showed a slightly larger offset, with MPSS averaging 116 while YSI measured 149, resulting in a 22% variation between both systems. Temperature measurements (Table 9) were also highly stable for both instruments, with MPSS readings averaging 22.5 °C compared to 23.9 °C from the YSI, corresponding to a 5.8% variation. DO (Table 10) was the only parameter showing more dynamic behaviour. At the start of the measurement, MPSS reported lower DO concentrations (2.29 mg/L vs. 4.31 mg/L for YSI, 46.9% lower), but over time the values gradually converged and even slightly surpassed the YSI readings by the final measurement (23% higher, 4.80 mg/L vs. 3.89 mg/L). This pattern suggests a longer stabilization period for the MPSS DO sensor rather than a systematic error.

4.2 Freshwater with Salt

For the second test, 1 kg of salt was added to the freshwater and mixed until the the solution was fully dissolved, which would expectedly lead to an increase in conductivity, TDS and salinity.

Conductivity (Table 11) remained highly stable throughout the test, with YSI averaging 39388 μS/cm and MPSS 38872 μS/cm, a minor 1.3% difference. Salinity (Table 12) reflected an offset of 4.7%, with MPSS reporting 24.7 PSU compared to 25.91 PSU from the YSI. TDS values (Table 13) were essentially identical, with MPSS readings averaging 26434 compared to 26368 from YSI, a

Table 11: Comparison of Conductivity (Freshwater with Salt)

Time	MPSS (µS/cm)	YSI (µS/cm)
00:00	38870	39388
00:15	38870	39388
00:30	38880	39387
00:45	38880	39388
01:00	38860	39387

Table 12: Comparison of Salinity (Freshwater with Salt)

Time	MPSS (PSU)	YSI (PSU)
00:00	24.71	25.91
00:15	24.7	25.91
00:30	24.71	25.91
00:45	24.7	25.91
01:00	24.7	25.91

Table 13: Comparison of TDS (Freshwater with Salt)

Time	MPSS	YSI
00:00	26437	26368
00:15	26433	26368
00:30	26439	26368
00:45	26427	26368
01:00	26435	26368

Table 14: Comparison of Temperature (Freshwater with Salt)

Time	MPSS (ºC)	YSI (ºC)
00:00	22.7	23.5
00:15	22.7	23.5
00:30	22.7	23.5
00:45	22.75	23.5
01:00	22.75	23.5

Table 15: Comparison of DO (Freshwater with Salt)

Time	MPSS (mg/L)	YSI (mg/L)
00:00	4.47	4.02
00:15	4.26	3.98
00:30	3.97	3.96
00:45	3.80	3.93
01:00	3.65	3.88

0.25% difference, demonstrating that both instruments produce consistent outputs in this experiment. Temperature measurements (Table 14) were equally stable, with MPSS averaging 22.7 °C versus 23.5 °C from the YSI, which corresponds to a 3.3% difference. DO (Table 15) showed a slight divergence over time, starting higher in MPSS (4.47 mg/L vs. 4.02 mg/L from YSI, 11% higher) before gradually converging and ending slightly lower (3.65 mg/L vs. 3.88 mg/L, 6% lower). This

shift may be once again attributed to sensor stabilization dynamics or differences in response time between the two instruments instead of systematic error.

4.3 Freshwater with Salt and Sodium Bicarbonate

For the third test, 100 g of sodium bicarbonate was added to the freshwater with salt and mixed until the the solution was fully dissolved. This addition, as the last one, led to another increase in conductivity-based values such as salinity and TDS.

Table 16: Comparison of Conductivity (Freshwater with Salt and Sodium Bicarbonate)

Time	**MPSS (µS/cm)**	**YSI (µS/cm)**
00:00	40630	41343
00:15	40640	41347
00:30	40650	41386
00:45	40660	41405
01:00	40650	41413

Table 17: Comparison of Salinity (Freshwater with Salt and Sodium Bicarbonate)

Time	**MPSS (PSU)**	**YSI (PSU)**
00:00	25.94	27.1
00:15	25.94	27.1
00:30	25.95	27.1
00:45	25.97	27.1
01:00	25.96	27.1

Table 18: Comparison of TDS (Freshwater with Salt and Sodium Bicarbonate)

Time	**MPSS**	**YSI**
00:00	28442	27461
00:15	28449	27461
00:30	28456	27480
00:45	28475	27493
01:00	28466	27500

Conductivity (Table 16) remained stable throughout the test, with the YSI averaging a value of 41379 μS/cm and the MPSS measuring 40646 μS/cm, corresponding to a 1.8% lower reading. As expected due to conductivity values, salinity values (Table 17) reflected a similar offset, with MPSS reporting approximately 25.95 PSU compared to 27.10 PSU from the YSI, which was 4.3% lower. TDS values (Table 18) showed a slight positive deviation, with MPSS averaging 28458 compared

Table 19: Comparison of Temperature (Freshwater with Salt and Sodium Bicarbonate)

Time	MPSS (ºC)	YSI (ºC)
00:00	22.7	23.9
00:15	22.65	23.9
00:30	22.7	23.9
00:45	22.65	23.9
01:00	22.65	23.9

Table 20: Comparison of DO (Freshwater with Salt and Sodium Bicarbonate)

Time	MPSS (mg/L)	YSI (mg/L)
00:00	4.12	4.26
00:15	4.08	4.13
00:30	4.18	4.00
00:45	4.25	4.06
01:00	4.80	4.13

to 27479 from YSI (3.6% higher). Temperature (Table 19) remained highly stable across both instruments, with MPSS readings averaging 22.7 °C, about 5.1% lower than the YSI's 23.9 °C. DO (Table 20) displayed a gradual divergence over time: initially, MPSS recorded slightly lower values (4.12 mg/L vs. 4.26 mg/L, 3% lower), but as the test progressed, the readings increased relative to YSI, ultimately reporting a higher value at the final measurement (4.80 mg/L vs. 4.13 mg/L, 16% higher). This pattern suggests that the MPSS DO sensor may have a slightly different stabilization behaviour in chemically enriched solutions.

4.4 Freshwater with Salt, Sodium Bicarbonate and Ice

In the final test, ice was added to the existing solution, slightly reducing the overall temperature and altering the solution's properties, increasing DO and leading to the decrease of conductivity-based variables.

Table 21: Comparison of Conductivity (Freshwater with Salt, Sodium Bicarbonate and Ice)

Time	MPSS (µS/cm)	YSI (µS/cm)
00:00	37860	38039
00:15	37850	38066
00:30	37840	38094
00:45	37820	38110
01:00	37810	38123

Conductivity values (Table 21) remained very similar between both instruments, with YSI averaging 38086 μS/cm and MPSS 37876 μS/cm, a negligible 0.55% lower reading. Salinity (Table 22)

Table 22: Comparison of Salinity (Freshwater with Salt, Sodium Bicarbonate and Ice)

Time	MPSS (PSU)	YSI (PSU)
00:00	24.5	26.15
00:15	24.6	26.17
00:30	24.5	26.19
00:45	24.6	26.21
01:00	24.5	26.21

Table 23: Comparison of TDS (Freshwater with Salt, Sodium Bicarbonate and Ice)

Time	MPSS	YSI
00:00	26503	26550
00:15	26496	26570
00:30	26490	26595
00:45	26479	26615
01:00	26469	26620

Table 24: Comparison of Temperature (Freshwater with Salt, Sodium Bicarbonate and Ice)

Time	MPSS (ºC)	YSI (ºC)
00:00	20.6	21.7
00:15	20.6	21.7
00:30	20.6	21.7
00:45	20.6	21.7
01:00	20.6	21.7

Table 25: Comparison of DO (Freshwater with Salt, Sodium Bicarbonate and Ice)

Time	MPSS (mg/L)	YSI (mg/L)
00:00	5.97	5.20
00:15	5.84	5.35
00:30	5.53	5.38
00:45	5.80	5.42
01:00	5.60	5.55

followed a similar trend, with MPSS reporting approximately 24.5 PSU compared to 26.2 PSU from the YSI (6.3% lower). TDS values (Table 23) were nearly identical, differing by only 0.4% between the two systems (YSI: 26590 vs. MPSS: 26487). Temperature readings (Table 24) reflected the cooling effect of the added ice, with MPSS consistently measuring 5.1% lower (20.6 °C vs. 21.7 °C) from YSI). DO (Table 25) initially showed a higher value in the MPSS sensor (5.97 mg/L vs. 5.20 mg/L, 14.8% higher), but the readings gradually converged over the measurement period, ending with a very small 0.9% difference (5.60 mg/L vs. 5.55 mg/L).

4.5 Expert feedback

After gathering and presenting the results of this experiment, the domain expert that closely followed the development of the system gave insights into the gathered values. Although, the developed system demonstrates consistent measurement behaviour across different conditions, the comparison revealed systematic offsets between the instruments. Parameters derived from conductivity, such as salinity and TDS, generally differed by 4–6% for salinity and 0.5–4% for conductivity and TDS under saline conditions, although TDS in freshwater showed a larger deviation (22%). Temperature exhibited a stable offset of 0.7–1.3 °C (approximately 3–6%), while dissolved oxygen displayed more dynamic differences, sometimes starting higher or lower before gradually converging with the YSI measurements. These offsets are non-negligible in oceanographic contexts, where even small variations can have meaningful implications. According to the domain expert, distinguishing between water masses often relies on subtle differences of 1 *PSU* in salinity, meaning that a 4–9% deviation could obscure key hydrological boundaries. Likewise, temperature gradients as small as 0.5–1 °C can delineate stratification layers or ecological niches, making a 0.7–1.3 °C offset significant for marine studies. The domain expert further attributed the observed discrepancies to the differences in calibration. The YSI instrument is professionally calibrated for field research in the oceans, whereas the MPSS sensors were calibrated using manufacturer-provided standard solution pouches.

A marine researcher who reviewed the data also noted that the variation between both systems could also be attributed to the different calibrations, and that utilizing a water homogenizer would also guarantee better results as the water would constantly flow and most likely decrease the discrepancy between values in both instruments. The same researcher outlined that the sampling density (five readings per condition) was relatively low for robust statistical analysis. Although a longer period of time was not set to measure these results due to bigger diversions in values between devices in longer periods, increasing the number of measurements per test would have enhanced the statistical power of the comparison and allowed for a more rigorous assessment of variability and sensor stabilization behaviour.

Although the results had some discrepancies, the biologists highlighted that the MPSS application proved to be a simple, useful and functional, with similar functionalities to the YSI instrument

(such as data retrieval, acquisition and sensor calibration). However, to prove itself as a capable alternative to existing sampling instruments better calibration and testing is needed.

5 Discussion

This chapter seeks to discuss experiment results, system comparison with other shown alternatives, constraints, and the planned future work.

5.1 Experiment Results

After receiving researcher feedback, it is essential to clarify that the scope of this thesis, grounded in informatics engineering, was the design and development of a functional system capable of retrieving, processing and calibrating sensor data reliably as a cost-effective alternative to existing instruments. The precise calibration process of sensors (especially to research-grade standards) is a specialized task that generally requires domain-specific expertise and resources beyond the scope of this work. Calibration in this study relied on manufacturer guidelines and standard solutions for functional validation of the MPSS; however, full scientific calibration for both sensors and optimization remain areas for future collaboration with marine instrumentation experts that were not present for this specific experiment.

As stated by the domain expert and the marine researcher that were consulted, the developed system proves to be simple, useful and resourceful to aid marine researchers with their field work apart from the discrepancies that resulted from the calibration that was utilized for these tests. Moreover, it represents a significantly more cost-effective solution: with the total cost of the MPSS instrument being of approximately $1200 with intended water column profiling capabilities, whereas the YSI system costs around $3000 (according to the consulted researcher) while being limited to shallower depths.

No formal testing was conducted to evaluate the GUI's ease of use and operability due to time constraints, which represents a limitation of this work. Although marine biologists were consulted for feedback, a more rigorous evaluation involving a broader range of users with varying levels of expertise would provide a stronger validation of the system's usability.

To summarize the deductions taken from the gathered results, the MPSS can be meaningfully compared with the YSI as a water sampling tool and demonstrates strong potential as a cost-effective water-column monitoring solution. However, without additional calibration and thorough testing (including in-vitro, in-situ and usability evaluations) the system cannot yet replace a fully calibrated scientific-grade sensor in precision-dependent studies.

5.2 System Comparison with Alternatives

This subsection aims to compare the developed apparatus with other alternatives through its physical aspect and application with other researched options.

5.2.1 Physical Aspect

When addressing the physical configuration of the system, namely, the apparatus and its integrated hardware, the MPSS incorporates many of the same sensor technologies utilized in commercial alternatives and existing IoUT platforms for marine environmental monitoring. As demonstrated throughout this thesis, the MPSS includes important features that enable the collection of essential EOVs, aligning with the capabilities typically employed by marine researching tools.

Nevertheless, while the developed system is engineered to operate at greater depths than those supported by the researched IoUT systems, it remains limited when compared to the studied market alternatives such as CTD rosettes and ROVs. Some of these systems, as documented in prior research, are capable of deployment at depths reaching 7000 m, significantly beyond the 300 m operational limit of the MPSS (which, although not formally tested, must be restricted to around this limit to prevent potential sensor damage). This limitation is primarily due to the structural and material constraints inherent to the sensors themselves. Despite the use of a reinforced protective casing, the manufacturer-recommended maximum depth rating for the MPSS sensors remains of around 350 m, so operating at depths over this threshold will always lead to unwanted results or sensor malfunction. However, although not as capable in terms of depth profiling, the MPSS is more portable and easier to deploy as the only equipment necessary to transport is a router, the sensing equipment, the RPi and a battery (with the latter two intended to be enclosed in a casing in the future), while CTD Rosettes and ROVs need different types of equipment (such as cranes for the CTD Rosettes) to operate. This difference allows for more geographical coverage and frequent sampling, as no other factors are necessary.

However, this discrepancy in supported depth and sensor precision is also reflected on cost. When compared to the previously mentioned CTD Rosettes and ROVs, this system drastically reduces financial costs due to its cost of around $1200. The fact that the system is easier to transport and operate also reduces logistical demands for transport and the need of specialized personnel to utilize.

5.2.2 User Application

The user interface developed for the MPSS system can be meaningfully compared to existing platforms commonly employed in environmental research, particularly within the context of IoUT systems and ROV-based software.

In IoUT systems, user-facing applications typically adopt a dashboard-based architecture, presenting data through tabular formats and graphical visualizations. Some platforms also integrate functionalities for sensor management. The MPSS application follows a similar paradigm, providing a centralized interface that consolidates real-time data display, sensor control, and logging capabilities.

When compared to ROV-based solutions, more specifically the BlueROV2 system running BlueOS, it is relevant to draw comparisons at both the interface and functionality levels. While BlueOS is technically an operating system, its user interface shares several structural similarities with MPSS. Both systems employ a sidebar on the left for page navigation and a persistent top bar for displaying system metrics. In BlueOS, the top bar displays performance metrics such as CPU usage, disk activity, and Ethernet packet transmission rates as seen in Figure 2. MPSS similarly displays real-time readings of CPU load, battery voltage, and depth as seen for example, in Figure 27. In both cases, access to system settings and power management controls (reboot/shutdown) is consistently available (albeit in different positions) ensuring operational control at all times. Furthermore, the core content area in both interfaces remains dedicated to task-specific functionality, framed by the sidebar and top bar to maintain navigational and contextual consistency.

Functionally, however, the two systems diverge in purpose and complexity. BlueOS, being a modular open-source operating system designed for wide-ranging marine operations, supports a broad ecosystem of extensions contributed by the research community. These extensions enable BlueOS to adapt to the specific needs of a given deployment with the BlueROV2.

MPSS, in contrast, is purpose-built for a more focused set of functionalities, more specifically the acquisition, logging, and visualization of environmental data, along with sensor calibration capabilities. While BlueOS can perform water sampling and visualize sampled data through graphs or export it in formats such as CSV or Log files, documentation suggests that sensor calibration is typically limited to the devices natively supported by BlueOS, with normally extra APIs or scripts being needed to fully accommodate full sensor calibration and configuration. MPSS offers a clear

distinction in this regard by including integrated support for calibration of externally connected sensors, enhancing its applicability in scientific data collection where sensor precision and accuracy are critical.

However, BlueOS offers greater expandability and user configurability compared to MPSS. While MPSS is a purpose-built application designed specifically for water sampling, its architecture is more constrained and requires additional development to support new functionalities. In contrast, BlueOS functions as a modular open-source operating system that supports the integration of diverse capabilities through community-developed extensions or lightweight scripts, making it inherently more flexible for a wide range of underwater research and operational tasks.

5.3 Constraints

While the developed system demonstrates significant potential as a low-cost solution for underwater data retrieval, it also presents some limitations that affect its performance and applicability in certain environments:

- **Depth and pressure resistance:** One of the primary limitations of the system lies in its restricted operational depth. Consequently, this depth ceiling restricts the system's applicability in deep-sea research or operations requiring high-pressure tolerance.
- **Power consumption and autonomy:** Due to the RPi's battery consumption compared to other more energy efficient microprocessors. As such, batteries like the one that will be later implemented for this system are drained faster, which can limit the system's operational duration, making it better suited for short-to-medium duration missions unless larger and more expensive power sources are integrated.
- **Mechanical stability and connectivity:** Similarly to the ROVs, the system uses a tethered connection for data transmission and remote control. While this provides reliable communication, it introduces mechanical challenges in dynamic aquatic environments. In particular, strong currents or wave activity can lead to tension or instability in the tether, which may hinder manoeuvrability, affect sensor accuracy, or disrupt consistent data collection. These factors make the system less ideal for use in turbulent or unpredictable marine conditions.
- **Less customizability:** When compared to systems like the BlueOS, the incapability of MPSS of expanding the system in a more streamlined manner can be a disadvantage, as the archi-

tecture of the system might need to be tweaked to accomodate to new functionalities and sensors.

5.4 Future Work

To better improve the apparatus and prepare it for field ready operations, the following developments are proposed:

- **Comprehensive calibration and validation:** Future work will focus on conducting more rigorous experiments, ideally in collaboration with domain experts, to improve the system's calibration accuracy and usability. These calibration efforts aim to validate the MPSS's sensor performance under controlled and real-world conditions through extended data collection and comparative testing with reference systems (such as YSI) and verify if the discrepancy in readings is a constraint (which will lead to replacement of sensors) or a parameter to be perfected.In terms of usability, methods such as think-aloud protocols and user surveys will be employed to gather feedback, helping identify which aspects of the system require improvement and which are functioning effectively.
- **Casing and power integration for field deployment:** A robust waterproof casing must be designed and fabricated to ensure sensor protection and pressure resistance in marine environments. In parallel, the integration of a dedicated battery system is essential to enable autonomous deployments, replacing the current reliance on a power from a wall outlet and making the system suitable for depth-profile testing in the field.

6 Conclusion

This thesis has presented a contribution to marine water sampling technologies through the development of the MPSS, a low-cost, real-time water column sampling apparatus designed to support marine research.

The work began with a comprehensive review of existing technologies and their role in deep-sea sampling, alongside a discussion of relevant IoT architectural approaches, and with attention given to the RPi, a suitable core for system implementation due to its adaptability and widespread use in similar projects.

Subsequently, the system development process was outlined in detail with the elicitation of requirements from a domain expert, the design of the system architecture, the integration of hardware and software components, and the creation of a GUI to ensure ease of use.

An experimental deployment of the system followed, during which feedback indicated that the MPSS is intuitive and functionally comparable to existing market solutions, although not validated formally. However, the experiment also highlighted the need for further calibration and validation to ensure reliability and scientific accuracy of retrieved EOVs as well as system usability.

A comparative analysis was then conducted between the developed prototype and other marine IoUT systems, including ROV platforms like those using BlueOS, reinforcing the novelty and potential of the proposed system within the current technological landscape both in terms of physical and software parts of the system.

While additional implementation and rigorous field testing are required before the system can be considered ready for operational deployment, the core objectives: developing a cost-effective, user-friendly, and modular platform for high-resolution marine sampling, have been successfully achieved, with further in-situ testing needed to verify its capabilities as a water column operating tool.

Regarding the initial research objectives, the system demonstrated varying degrees of success. Feedback from marine biologists indicated that the user interface is intuitive and accessible; however, formal usability testing is required to quantitatively validate this assessment. In terms of high-resolution sampling, while the system successfully integrated sensors rated for depts exceeding 200 meters, the current version is restricted to laboratory enviornments and thus, in situ functionality

at such pressures has not yet been verified. In terms of cost, the design proved more cost-effective than existing market alternatives, despite its functional trade-offs. Finally, the objective of simplified deployment was achieved through a compact, lighweight architecture that eliminates the need for specialized personnel and ease of transportation to operate, although the transition to a fully field-ready iteration remains a future hurdle to surpass. Consequently, while this study met several set objectives associated with the proposed research question, a comprehensive resolution requires further development and testing before a definitive answer can be reached.

In conclusion, this thesis has demonstrated the feasibility of the MPSS as a promising tool for marine researchers. With continued refinement and validation, the system will seek to become a valuable asset in the field of marine environmental monitoring.

References

[1] S. A. H. Mohsan, Y. Li, M. Sadiq, J. Liang, and M. A. Khan, "Recent advances, future trends, applications and challenges of internet of underwater things (iout): A comprehensive review," *Journal of Marine Science and Engineering*, vol. 11, no. 1, p. 124, 2023.

[2] A. R. Thurber, A. K. Sweetman, B. E. Narayanaswamy, D. O. Jones, J. Ingels, and R. Hansman, "Ecosystem function and services provided by the deep sea," *Biogeosciences*, vol. 11, no. 14, pp. 3941–3963, 2014.

[3] L. A. Levin, C.-L. Wei, D. C. Dunn, D. J. Amon, O. S. Ashford, W. W. Cheung, A. Colaço, C. Dominguez-Carrió, E. G. Escobar, H. R. Harden-Davies *et al.*, "Climate change considerations are fundamental to management of deep-sea resource extraction," *Global Change Biology*, vol. 26, no. 9, pp. 4664–4678, 2020.

[4] Y. J. Kim, D. Han, E. Jang, J. Im, and T. Sung, "Remote sensing of sea surface salinity: challenges and research directions," *GIScience & Remote Sensing*, vol. 60, no. 1, p. 2166377, 2023.

[5] D. L. Moreira, A. G. Dalto, A. G. Figueiredo Jr, A. M. Valerio, A. M. Detoni, A. C. Bonecker, C. N. Signori, C. Namiki, D. K. Sasaki, D. V. Pupo *et al.*, "Multidisciplinary scientific cruises for environmental characterization in the santos basin–methods and sampling design," *Ocean and Coastal Research*, vol. 71, no. suppl 3, p. e23022, 2023.

[6] F. Azis, M. Aras, M. Rashid, M. Othman, and S. Abdullah, "Problem identification for underwater remotely operated vehicle (rov): A case study," *Procedia Engineering*, vol. 41, pp. 554–560, 2012.

[7] A. F. Casper, B. Dixon, E. T. Steimle, M. L. Hall, and R. N. Conmy, "Scales of heterogeneity of water quality in rivers: Insights from high resolution maps based on integrated geospatial, sensor and rov technologies," *Applied Geography*, vol. 32, no. 2, pp. 455–464, 2012.

[8] G. Zheng and P. M. DiGiacomo, "Uncertainties and applications of satellite-derived coastal water quality products," *Progress in oceanography*, vol. 159, pp. 45–72, 2017.

[9] J. Jijesh, M. Susmitha, M. Bhanu, P. Sindhanakeri *et al.*, "Development of a ctd sensor subsystem for oceanographic application," in *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*. IEEE, 2017, pp. 1487–1492.

[10] F. Hidalgo, J. Mendoza, and F. Cuéllar, "Rov-based acquisition system for water quality measuring," in *OCEANS 2015-MTS/IEEE Washington*. IEEE, 2015, pp. 1–5.

[11] S. A. H. Mohsan, A. Mazinani, N. Q. H. Othman, and H. Amjad, "Towards the internet of underwater things: A comprehensive survey," *Earth Science Informatics*, vol. 15, no. 2, pp. 735–764, 2022.

[12] M. C. Domingo, "An overview of the internet of underwater things," *Journal of Network and Computer Applications*, vol. 35, no. 6, pp. 1879–1890, 2012.

[13] V. Lakshmikantha, A. Hiriyannagowda, A. Manjunath, A. Patted, J. Basavaiah, and A. A. Anthony, "Iot based smart water quality monitoring system," *Global Transitions Proceedings*, vol. 2, no. 2, pp. 181–186, 2021.

[14] M. S. U. Chowdury, T. B. Emran, S. Ghosh, A. Pathak, M. M. Alam, N. Absar, K. Andersson, and M. S. Hossain, "Iot based real-time river water quality monitoring system," *Procedia computer science*, vol. 155, pp. 161–168, 2019.

[15] C. S. Ranganathan, R. Raman, S. Parikh, S. Rajesh, R. Meenakshi, and M. Muthulekshmi, "Iot applications in marine monitoring: Protecting ocean health and biodiversity," in *2023 International Conference on Sustainable Communication Networks and Application (ICSCNA)*. IEEE, 2023, pp. 305–310.

[16] R. C. Trinh, C. G. Fichot, M. M. Gierach, B. Holt, N. K. Malakar, G. Hulley, and J. Smith, "Application of landsat 8 for monitoring impacts of wastewater discharge on coastal water quality," *Frontiers in Marine Science*, vol. 4, p. 329, 2017.

[17] N. Vinogradova, T. Lee, J. Boutin, K. Drushka, S. Fournier, R. Sabia, D. Stammer, E. Bayler, N. Reul, A. Gordon *et al.*, "Satellite salinity observing system: Recent discoveries and the way forward," *Frontiers in Marine Science*, vol. 6, p. 243, 2019.

[18] W. M. Smethie Jr, D. Chayes, R. Perry, and P. Schlosser, "A lightweight vertical rosette for deployment in ice-covered waters," *Deep Sea Research Part I: Oceanographic Research Papers*, vol. 58, no. 4, pp. 460–467, 2011.

[19] C. I. Measures, W. M. Landing, M. T. Brown, and C. S. Buck, "A commercially available rosette system for trace metal—clean sampling," *Limnology and Oceanography: Methods*, vol. 6, no. 9, pp. 384–394, 2008.

[20] R. M. Gomes, A. Martins, A. Sousa, J. Sousa, S. Fraga, and F. L. Pereira, "A new rov design: issues on low drag and mechanical symmetry," in *Europe Oceans 2005*, vol. 2. IEEE, 2005, pp. 957–962.

[21] Blue Robotics, "BlueROV2 – Product Store Page," https://bluerobotics.com/store/rov/bluerov2/, 2025, (accessed 28-07-2025).

[22] H. Yoshida, T. Aoki, H. Osawa, S. Ishibashi, Y. Watanabe, J. Tahara, T. Miyazaki, and K. Itoh, "A deepest depth rov for sediment sampling and its sea trial result," in *2007 Symposium on Underwater Technology and Workshop on Scientific Use of Submarine Cables and Related Technologies*. IEEE, 2007, pp. 28–33.

[23] Blue Robotics, "BlueROV2 Buyers Guide by Options," https://bluerobotics.com/learn/bluerov2-buyers-guide-by-options/, 2025, (accessed 28-07-2025).

[24] BlueOS, "Blue OS Documentation – Usage Overview," https://blueos.cloud/docs/stable/usage/overview/, 2025, (accessed 28-07-2025).

[25] Blue OS, "Blue OS Documentation – Advanced Usage," https://blueos.cloud/docs/stable/usage/advanced/, 2025, (accessed 28-07-2025).

[26] BlueOS, "Blue OS Documentation – Additional Sensors Integration," https://blueos.cloud/docs/stable/integrations/hardware/additional/other-sensors/, 2025, (accessed 28-07-2025).

[27] Blue Robotics, "BlueROV2 Datasheet," https://bluerobotics.com/wp-content/uploads/2025/04/BROV2-DATASHEET.pdf, 2025, (accessed 28-07-2025).

[28] ——, "BlueROV2 Software Setup – Sensor Calibration," https://bluerobotics.com/learn/bluerov2-software-setup/#sensor-calibration, 2025, (accessed 28-07-2025).

[29] ——, "BlueOS and Node-RED Guide," https://bluerobotics.com/learn/blueos-and-node-red-guide/, 2025, (accessed 28-07-2025).

[30] C.-C. Kao, Y.-S. Lin, G.-D. Wu, and C.-J. Huang, "A comprehensive study on the internet of underwater things: applications, challenges, and channel models," *Sensors*, vol. 17, no. 7, p. 1477, 2017.

[31] A. F. Abdillah, M. H. Berlian, Y. Y. F. Panduman, M. A. W. Akbar, M. A. Afifah, A. Tjahjono, S. Sukaridhoto, and S. Sasaki, "Design and development of low cost coral monitoring system for shallow water based on internet of underwater things," *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, vol. 9, no. 2-5, pp. 97–101, 2017.

[32] J. W. Jolles, "Broad-scale applications of the raspberry pi: A review and guide for biologists," *Methods in Ecology and Evolution*, vol. 12, no. 9, pp. 1562–1579, 2021.

[33] G. D. Priya and I. Harish, "Raspberry pi based underwater vehicle for monitoring aquatic ecosystem," *International Journal of Engineering Trends and Applications*, vol. 2, no. 2, pp. 65–71, 2015.

[34] M. Jouhari, K. Ibrahimi, H. Tembine, and J. Ben-Othman, "Underwater wireless sensor networks: A survey on enabling technologies, localization protocols, and internet of underwater things," *IEEE Access*, vol. 7, pp. 96 879–96 899, 2019.

[35] T. Adiono, A. M. Toha, S. Pamungkas, N. Sutisna, and E. Sumiarsih, "Internet of things for marine aquaculture," in *2021 International Symposium on Electronics and Smart Devices (ISESD)*. IEEE, 2021, pp. 1–5.

[36] N. Vijayakumar and R. Ramya, "The real time monitoring of water quality in iot environment," in *2015 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS)*. IEEE, 2015, pp. 1–5.

[37] H. Lan, Y. Lv, J. Jin, J. Li, D. Sun, and Z. Yang, "Acoustical observation with multiple wave gliders for internet of underwater things," *IEEE Internet of Things Journal*, vol. 8, no. 4, pp.

2814–2825, 2020.

[38] E. Cavalcante, M. P. Alves, T. Batista, F. C. Delicato, and P. F. Pires, "An analysis of reference architectures for the internet of things," in *Proceedings of the 1st International Workshop on Exploring Component-based Techniques for Constructing Reference Architectures*, 2015, pp. 13–16.

[39] M. Bauer, N. Bui, C. Jardak, and A. Nettsträter, "The iot arm reference manual," *Enabling Things to Talk*, vol. 213, 2013.

[40] M. Weyrich and C. Ebert, "Reference architectures for the internet of things," *IEEE software*, vol. 33, no. 1, pp. 112–116, 2015.

[41] L. Boyanov, V. Kisimov, and Y. Christov, "Evaluating iot reference architecture," in *2020 International Conference automatics and informatics (ICAI)*. IEEE, 2020, pp. 1–5.

[42] B. Di Martino, M. Rak, M. Ficco, A. Esposito, S. A. Maisto, and S. Nacchia, "Internet of things reference architectures, security and interoperability: A survey," *Internet of Things*, vol. 1, pp. 99–112, 2018.

[43] G. Fortino, A. Guerrieri, P. Pace, C. Savaglio, and G. Spezzano, "Iot platforms and security: An analysis of the leading industrial/commercial solutions," *Sensors*, vol. 22, no. 6, p. 2196, 2022.

[44] P. Fremantle *et al.*, "A reference architecture for the internet of things," *WSO2 White paper*, pp. 02–04, 2015.

[45] J. Guth, U. Breitenbücher, M. Falkenthal, F. Leymann, and L. Reinfurt, "Comparison of iot platform architectures: A field study based on a reference architecture," in *2016 Cloudification of the Internet of Things (CIoT)*. IEEE, 2016, pp. 1–6.

[46] H. Washizaki, S. Ogata, A. Hazeyama, T. Okubo, E. B. Fernandez, and N. Yoshioka, "Landscape of architecture and design patterns for iot systems," *IEEE Internet of Things Journal*, vol. 7, no. 10, pp. 10 091–10 101, 2020.

[47] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "Mqtt-s—a publish/subscribe protocol for wireless sensor networks," in *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08)*. IEEE, 2008, pp. 791–798.

[48] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM computing surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[49] A. Rizzardi, S. Sicari, D. Miorandi, and A. Coen-Porisini, "Aups: An open source authenticated publish/subscribe system for the internet of things," *Information Systems*, vol. 62, pp. 29–41, 2016.

[50] B. Tekinerdogan and Ö. Köksal, "Pattern based integration of internet of things systems," in *International Conference on Internet of Things*. Springer, 2018, pp. 19–33.

[51] S. Kumar, "A review on client-server based applications and research opportunity," *International Journal of Recent Scientific Research*, vol. 10, no. 7, pp. 33 857–3386, 2019.

[52] S. Ali, R. Alauldeen, and A. Ruaa, "What is client-server system: architecture, issues and challenge of client-server system," *HBRP Publication*, vol. 2, no. 1, pp. 1–6, 2020.

[53] S. M. Lewandowski, "Frameworks for component-based client/server computing," *ACM Computing Surveys (CSUR)*, vol. 30, no. 1, pp. 3–27, 1998.

[54] H. S. Oluwatosin, "Client-server model," *IOSR Journal of Computer Engineering*, vol. 16, no. 1, pp. 67–71, 2014.

[55] S. Kurkovsky and C. Williams, "Raspberry pi as a platform for the internet of things projects: Experiences and lessons," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, 2017, pp. 64–69.

[56] M. Maksimović, V. Vujović, N. Davidović, V. Milošević, and B. Perišić, "Raspberry pi as internet of things hardware: performances and constraints," *design issues*, vol. 3, no. 8, pp. 1–6, 2014.

[57] G. A. M. do Nascimento, M. M. L. Neto, and W. B. da Silva, "Uma aplicação didática do protocolo i2c em sistemas de comunicação a didactic application of the i2c protocol in communication systems," *Brazilian Journal of Development*, vol. 7, no. 10, pp. 94 837–94 853, 2021.

[58] S. B. Jamkhandi, N. V. Kenchanagouda, A. G. Patil, N. Divakar, P. Kallimani, and J. Mallidu, "Raspberry pi-based real-time health monitoring and alert system for enhanced patient care," in *2024 5th International Conference on Smart Electronics and Communication (ICOSEC)*. IEEE, 2024, pp. 1164–1170.

[59] S. Karthikeyan, R. A. Raj, M. V. Cruz, L. Chen, J. A. Vishal, and V. Rohith, "A systematic analysis on raspberry pi prototyping: Uses, challenges, benefits, and drawbacks," *IEEE Internet of Things Journal*, vol. 10, no. 16, pp. 14 397–14 417, 2023.

[60] M. Shah, J. Patel, and V. Patel, "Development of interactive data storage unit using raspberry pi," in *2018 International Conference on Inventive Research in Computing Applications (ICIRCA)*. IEEE, 2018, pp. 825–830.

[61] H. Hajjar and H. Mourad, "Implementation of an fpga-raspberry pi spi connection," in *The Twelfth International Conference on Advances in Circuits, Electronics and Micro-electronics*, 2019, pp. 7–12.

[62] N. Agarwal, S. N. Reddy *et al.*, "Design & development of daughter board for raspberry pi to support bluetooth communication using uart," in *International Conference on Computing, Communication & Automation*. IEEE, 2015, pp. 949–954.

[63] R. R. Pahlevi, M. Abdurohman *et al.*, "Fast uart and spi protocol for scalable iot platform," in *2018 6th International Conference on Information and Communication Technology (ICoICT)*. IEEE, 2018, pp. 239–244.

[64] Raspberry Pi Foundation, "Raspberry Pi Documentation: Computers," https://www.raspberrypi.com/documentation/computers/raspberry-pi.html, 2025, (accessed on 22-05-2025).

[65] H. D. Ghael, L. Solanki, and G. Sahu, "A review paper on raspberry pi and its applications," *International Journal of Advances in Engineering and Management (IJAEM)*, vol. 2, no. 12, p. 4, 2020.

[66] W. Hajji and F. P. Tso, "Understanding the performance of low power raspberry pi cloud for big data," *Electronics*, vol. 5, no. 2, p. 29, 2016.

[67] K. M. Hosny, A. Magdi, A. Salah, O. El-Komy, and N. A. Lashin, "Internet of things applications using raspberry-pi: a survey." *International Journal of Electrical & Computer Engineering (2088-8708)*, vol. 13, no. 1, 2023.

[68] M. Radeta, A. Zuniga, N. H. Motlagh, M. Liyanage, R. Freitas, M. Youssef, S. Tarkoma, H. Flores, and P. Nurmi, "Deep learning and the oceans," *Computer*, vol. 55, no. 5, pp. 39–50,

2022.

[69] Raspberry Pi, “Raspberry Pi 4 Model B – Product Page,” https://www.raspberrypi.com/products/raspberry-pi-4-model-b/, 2025, (accessed on 19-05-2025).

[70] VueJS, “Vue.js – The Progressive JavaScript Framework,” https://vuejs.org/, 2025, (accessed on 20-05-2025).

[71] TypeScriptLang, “TypeScript – JavaScript With Syntax for Types,” https://www.typescriptlang.org/, 2025, (accessed on 20-05-2025).

[72] Pinia, “Pinia – The Intuitive Store for Vue.js,” https://pinia.vuejs.org/, 2025, (accessed on 20-05-2025).

[73] ChartJS, “Chart.js – Simple Yet Flexible JavaScript Charting,” https://www.chartjs.org/, 2025, (accessed on 20-05-2025).

[74] Python, “Python – Official Website,” https://www.python.org/, 2025, (accessed on 20-05-2025).

[75] FastAPI, “FastAPI – Modern, Fast (High-Performance) Web Framework for Python,” https://fastapi.tiangolo.com/, 2025, (accessed on 20-05-2025).

[76] Uvicorn, “Uvicorn – The Lightning-fast ASGI Server,” https://www.uvicorn.org/, 2025, (accessed on 20-05-2025).

[77] NGINX, “NGINX – Official Website,” https://nginx.org/en/, 2025, (accessed on 20-05-2025).

[78] SQLite, “SQLite – Official Website,” https://www.sqlite.org/, 2025, (accessed on 20-05-2025).

[79] Atlas Scientific, “Conductivity K 1.0 Kit – Product Page,” https://atlas-scientific.com/kits/conductivity-k-1-0-kit/, 2025, (accessed on 20-05-2025).

[80] ——, “EZO Complete DO Kit – Product Page,” https://atlas-scientific.com/kits/ezo-complete-do-kit/, 2025, (accessed on 20-05-2025).

[81] Keller Druck, “Series 4LD – OEM Pressure Transmitters,” https://keller-pressure.com/en/products/pressure-transmitters/oem-pressure-transmitters/series-4ld, 2025, (accessed on 20-05-2025).

[82] BotnRoll, "Módulo RTC DS3231 – Product Page (PT)," https://www.botnroll.com/pt/outros/1614-modulo-rtc-ds3231.html, 2025, (accessed on 20-05-2025).

[83] ——, "Sensor de Corrente INA219 26V DC 3.2A máx – Product Page (PT)," https://www.botnroll.com/pt/corrente-/3051-m-dulo-sensor-corrente-ina219-26v-dc-3-2a-max.html, 2025, (accessed on 20-05-2025).

[84] Atlas Scientific, "DO EZO Datasheet," https://files.atlas-scientific.com/DO_EZO_Datasheet.pdf, 2025, (accessed on 22-05-2025).

[85] TP-Link Technologies Co., Ltd., "TL-MR3420 – 3G/4G Wireless N Router (Portuguese Product Page)," https://www.tp-link.com/pt/home-networking/5g-4g-router/tl-mr3420/, 2025, (accessed on 20-05-2025).

[86] Keller Druck, "Series 4LD – Product Documentation," https://download.keller-pressure.com/api/download/2LfcGMzMbeHdjFbyUd5DWA/en/2021-07.pdf, 2021, (accessed on 20-05-2025).

[87] Atlas Scientific, "EC EZO Datasheet," https://files.atlas-scientific.com/EC_EZO_Datasheet.pdf, 2025, (accessed on 22-05-2025).

[88] TP-Link Technologies Co., Ltd., "TL-MR3420(EU) V5.0 User Guide," https://static.tp-link.com/2018/201803/20180327/1910012177_TL-MR3420(EU)%205.0_UG.pdf, 2018, (accessed on 04-07-2025).

[89] YSI Inc., "ProPlus Handheld Multiparameter Instrument," https://www.ysi.com/proplus, 2025, (accessed on 21-07-2025).

[90] Mare Madeira, "Mare madeira – homepage," https://mare-madeira.pt/, 2025, (acessed on 18-05-2025).

[91] TWI Global, "Technology Readiness Levels – Technical Knowledge," https://www.twi-global.com/technical-knowledge/faqs/technology-readiness-levels, 2025, (acessed on 18-05-2025).