Department of Electrical and Computer Engineering

ECE 4600 GROUP DESIGN PROJECT

FINAL REPORT

Design and Implementation of IoT Lake/River Real-time Depth Monitoring System

Prepared by Group # 9: Gunjan Modi, Sukhmeet Singh Hora, Sudipta Dip, Matt Fryatt, Het Patel, and Justine Ugalde

Faculty Supervisor:

DR. KEN FERENS

March 14, 2025

This page intentionally left blank

Abstract

This project presents an innovative sonar-based mapping system designed to provide accurate, real-time water depth data that is both accessible and publicly available. The system is compact, user-friendly, and easily attachable to any boat. As the boat moves, it continuously records water depth and geographic location, enabling users to monitor depth and positioning in real-time on a map while simultaneously generating a detailed depth profile of the area.

The collected data is automatically uploaded to a cloud-based platform, ensuring open access for users who require up-to-date depth information. Advanced mapping techniques and comparative analyses are then applied to detect anomalies, track depth changes, and identify objects resulting from natural events such as sediment shifts, erosion, or seasonal variations, as well as human activities like dredging or construction. This information also serves as a valuable resource for updating and improving the accuracy of existing navigational maps.

By leveraging cloud technology and real-time data collection, this system enhances maritime navigation, supports environmental monitoring, and facilitates research. It provides a practical and powerful tool for boaters, researchers, and conservationists seeking to better understand and document water depth variations over time.

Acknowledgements

We would like to extend our sincere gratitude to the University Swimming Pool for their approval and unwavering support, which allowed us to conduct multiple rounds of system integration testing. Their cooperation played a crucial role in validating our system's accuracy.

A special thank you goes to Courtney Dietrich for her invaluable assistance when the sensor's receiving wire broke. Her expertise and quick intervention helped us repair it efficiently, allowing us to proceed with our testing without unnecessary delays.

We are also deeply grateful to ECE department technical staff—Gordon Glatz, Oleg Shevchenko, Zoran Trajkoski, and Cory Smit—for their continuous support throughout the project. Their technical knowledge and readiness to assist whenever needed greatly contributed to the successful execution of our work. Their insightful suggestions regarding hardware integration proved invaluable at every stage of our project.

A special acknowledgment goes to Alexander Wall, the Civil Lab Technician, for granting us access to the civil lab water flume for testing. This facility provided an ideal environment for our experiments, enabling us to evaluate our system comprehensively. His enthusiasm, curiosity, and willingness to engage with our project helped us refine several critical elements that might have otherwise been overlooked.

We extend our heartfelt gratitude to our advisor, Dr. Ken Ferens, for his continuous guidance and mentorship throughout the project. His expertise was instrumental in shaping our approach and ensuring the success of our work.

Lastly, we sincerely appreciate the contributions and support of Aidan Topping and Dr. Derek Oliver for their assistance in the successful completion of our project.

Division of Work

Contributions from each member to the technical work of the project are listed in Table 1.

Table 1 Project Contributions

	Sukhmeet	Sudip	Gunjan	Matt	Het	Justine
Project Planning	○	○	○	○	○	○
Embedded Systems (including sensors, MCU and power supply)			●	○		
Prototype Boat Designing			●	○		
MCU Programming	●	○				
Cloud Storage and Communication Setup	●				○	
Application (iOS, Android)	●	○			○	
Path Mapping					●	
Hazard Differentiation		●				●
Integration and Testing	○	○	○	○		

Legend:

● - Leading Member

○ - Contributing Member

Table of Contents

List of Figures

List of Tables

Nomenclature

AWS	Amazon Web Services
BLE	Bluetooth Low Energy
CW	Clockwise
CCW	Counterclockwise
GATT	Generic Attribute Profile
GPS	Global Positioning System
GUI	Graphical User Interface
HMI	Human Machine Interface
IoT	Internet of Things
IP	Internet Protocol
KNN	K-Nearest Neighbors
MCU	Microcontroller Unit
ML	Machine Learning
MQTT	Message Queuing Telemetry Transport
RBF	Radial Basis Function
RC	Remote Control
RFC	Random Forest Classifier
Rx	Receiver
SSID	Service Set Identifier
SVM	Support Vector Machine
Tx	Transmitter
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus
UUID	Universally Unique Identifier
Wi-Fi	Wireless Fidelity

This page intentionally left blank

1 Introduction

1.1 Project Overview

Bathymetry is the study of underwater topography of bodies of water. The data that it provides has several uses. First, it is used for safe navigation by providing information about water depth and potential hazards. Next, it is used for studying habitats of marine life by providing a topographical map of a body of water. Additionally, it is used to design and construct submerged foundations for building structures on water. The bathymetric data is collected using echosounders. The echosounder sends sound pulses downwards and measures the time it takes for the pulse to reflect to the sensor. This measurement is used to calculate the depth. Using this sensor, we have created a depth monitoring system for use by boaters. The system consists of three different components. The first component is a device capable of taking bathymetric surveys of lakes and rivers. The second component is cloud storage. Any data collected by a boater is saved here for later use. The third component is an app. The app allows the boater to control the device and view the data collected. Each component will be discussed in greater detail later in the report. Our system will make it easy for boaters to monitor the depth of lakes and rivers using our depth monitoring device and app.

1.2 Problem Statement

Our project, "Design and Implementation of IoT Lake/River Real-time Depth Monitoring System," aims to enable boaters to conduct bathymetric surveys of lakes or rivers and leverage the survey data to enhance their boating experience and safety. Boaters can integrate our device onto their boats to monitor real-time depth and location data via our phone app and upload this data to our database.

1.3 Scope

The main purpose of our project is to create a depth monitoring system for lakes and rivers. There were significant challenges when developing and testing this system because of our limited access to bodies of water and boats during the winter season. Therefore, we chose to focus on creating a system that can function in our controlled testing environment while remaining adaptable for real-world deployment with minor modifications.

Additionally, we developed features for our system that can help boaters with navigation like the hazard differentiation and path mapping features. However, the purpose of our project does not include offering

navigational services. Improvements to our system, like offering navigational services, are discussed in future recommendations. The core purpose of this project is to develop a real-time depth monitoring system that functions both locally and remotely while providing access to pre-mapped routes with data analyzes for hazard and safe route prediction. Additionally, the system enables device control through a dedicated app, facilitating in-depth analysis of lakes and rivers.

1.4 Project Specifications

Table 2 Performance Metrics

Component / Feature	Specific Field	Proposed Target	Outcome
Sensor	Minimum depth (Civil Lab validation)	0.3 m	0.379 m
	Maximum depth (Pool validation)	100 m	3.69 m[1]
Real-time depth monitoring from Raspberry Pi 4 to application	Locally BLE communication when offline with time delay	< 2 s	0.492 s
	Remote MQTT over Wi-Fi communication with time delay	< 2 s	0.469 s
Frequency of readings	Time between subsequent readings	< 5 s	3.14 s (Simulating or GPS device)
			7.32 s (using Wi-Fi Geolocation)
File upload from Raspberry Pi 4 to cloud storage	Speed of uploading large files (JSON messages) to the cloud (when Wi-Fi is available)	> 20 kB/s	27.13 kB/s
Application	View real-time UI updates	< 2 s	1 s
	Retrieve and view pre-mapped routes from Amazon S3 storage with time delay	< 5 s	4.419 s
	Hazard Alerts for pre-mapped routes when user uses that map “x” m before	15m	20m
Hazard differentiation	Accuracy of model during training data	> 85%	81.61%[2]
	Accuracy of model during testing data	> 80%	91.43%[2]

[1]Outcome value due to limitation of testing facility.
[2]The model's accuracy is solely based on the data collected and tested within the same testing facilities.

2 Prototype Boat and Hardware Interfaces

To ensure that all connections are properly established, we aimed for a compact design to accommodate essential hardware components such as a power bank, a microcontroller, and a USB-UART converter in a waterproof box. Additionally, we wanted to create a structure that housed these components efficiently and maintained stability and functionality in a freshwater body environment. Furthermore, our design considerations extended beyond just the internal components—we also needed to ensure that the structure could float on water and give us an accurate result. This requirement led us to explore the design and development of a prototype boat, carefully considering factors such as buoyancy, weight distribution, and material selection to achieve optimal performance.

2.1 Sensor Selection

Our project focused on developing an underwater application, which required a sensor specifically designed for underwater use. Most of the sensors we initially considered were intended for air or environments with minimal humidity. Using a sensor not rated for underwater conditions posed significant risks, including potential damage and inaccurate measurements. In such cases, investing in a high-quality, specialized sensor is often the most efficient and cost-effective approach, preventing wasted time and effort.

We reviewed various sensors, including ultrasonic, pressure, laser, and radar, to identify the best option. After a thorough evaluation, we shortlisted two ultrasonic sensors from different manufacturers. The first sensor we considered was the *XL-MaxSonar* from *Maxbotix*. It offered all the essential features needed for our hardware interface and aligned well with our project budget. However, Maxbotix had not tested this sensor for underwater use, which raised concerns about its reliability in our application. As a result, we had to rule it out.

Ultimately, we turned to *Blue Robotics*, a company specializing in underwater and surface drone technology. Their business model focuses on innovation and accessibility, providing well-documented products widely used in educational institutions. They were also highly responsive and helpful in discussing product options. Even though it would consume our entire budget, choosing the Blue Robotics Ping 2 sensor was the most logical and efficient decision to ensure the success of our project.

2.1.1 Working Principle of the Ping2 Sensor

The Blue Robotics Ping 2 sensor is designed for underwater distance measurement, offering a 100-meter range and a 25-degree beam width. It operates by emitting a brief 115 kHz acoustic pulse from its transducer and then listening for returning echoes. As sound waves travel through water, they reflect off solid objects and return to the sensor. The distance to an object is calculated using the formula [1]:

$$Distance = \frac{Speed\ of\ Sound\ in\ water * Echo\ return\ time}{2}$$

Speed of Sound in water = 1500 m/s

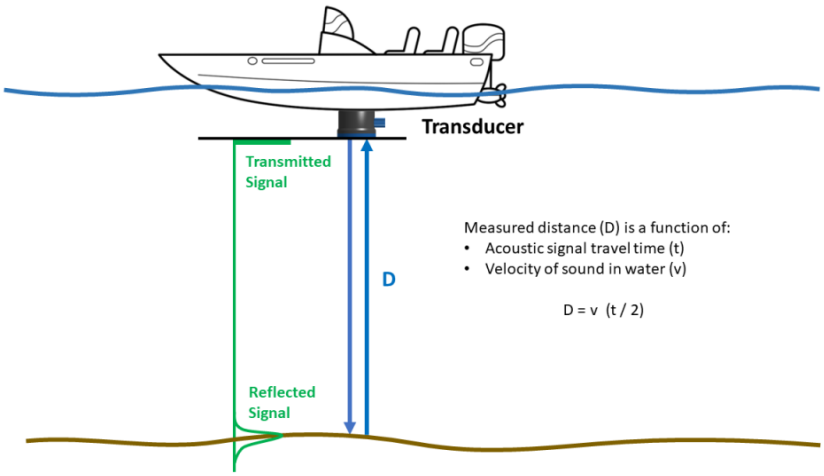


Figure 1 Working Principle of the Ping2 Sensor

Additionally, the sensor provides a confidence measurement based on the strength of the returned signal. A strong signal results in 100% confidence, while noisy environments with a low signal-to-noise ratio reduce the confidence level [1]. The Ping 2 sonar communicates via a serial UART interface using the Ping Protocol, a binary communication standard. More details on its hardware interface are provided in section 2.1.2 below.

2.1.2 Hardware Interface

As part of the final hardware integration, Raspberry Pi 4 (RPi4) is directly connected to a 5V power bank, ensuring a stable voltage supply for consistent operation. The USB-UART converter is then plugged into the USB port of the RPi4. The specific connections between the USB-UART converter and the Ping2 sensor are detailed in the table below.

Table 3 USB-UART Wiring Connections

Pin on USB-UART converter	Ping2 Sensor wire color
Vc	Red
Gnd	Black
Tx	White
Rx	Green

Initially, to minimize costs, we used an Arduino Uno 3, which we obtained for free from the tech shop. It functioned perfectly; however, since it introduced an additional microcontroller and occupied more space within the container, we opted to replace it with a USB-UART converter. This change allowed us to achieve a more compact and efficient system design.

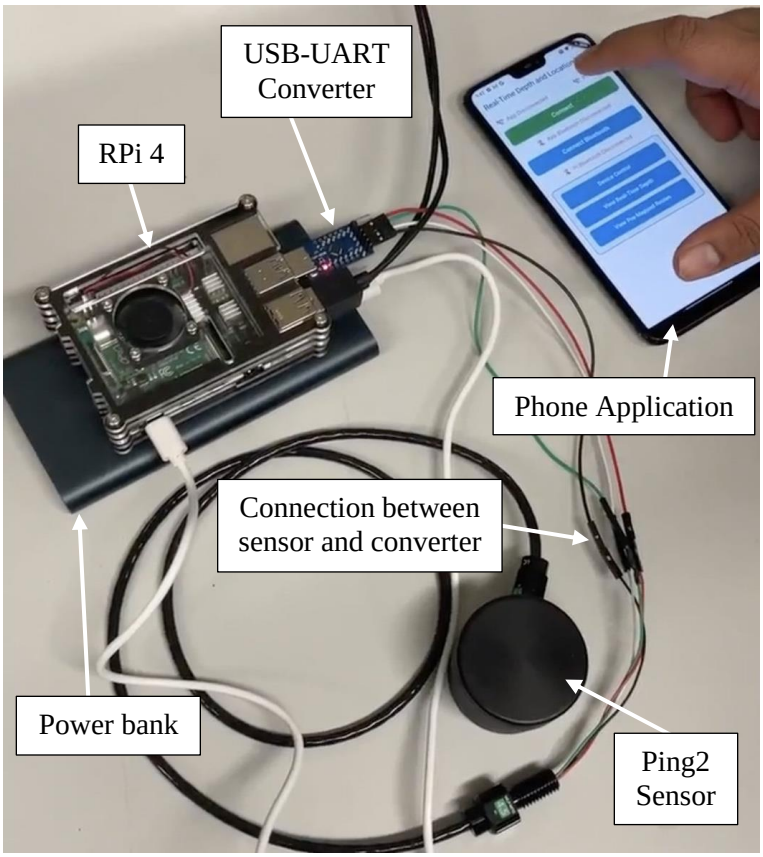


Figure 2 Connecting the USB-UART Converter to the Raspberry Pi

2.2 Design of a prototype boat

To minimize costs, we prioritized using materials readily available at either a tech shop or within the university, eliminating the need for additional expenses. We repurposed a pair of swimming kickboards from the university swimming pool area for the boat's floatation. We used a large circular coffee container to

Figure 3 Top & Side view of the Prototype boat

house our hardware components, which provided ample space for secure placement. The top of the container was intentionally left open to allow easy access to the internal hardware connections. Given the sufficient height of the design above the water surface, we were not concerned about water entering the container. Additionally, we securely attached the Ping2 sensor to the underside of the kickboards using double-sided tape.

2.3 System Integration on Boat

Initially, our plan was to integrate all the hardware components directly into the remote-controlled boat. However, when we tested the setup with a similar weight to the actual hardware, we noticed that the front end of the boat began to sink slightly as shown in Figure 4. This imbalance indicated that the boat would struggle to stay afloat under the additional load. As a result, we had to abandon the idea of placing any significant weight on the boat. Otherwise, our carefully designed components would risk submersion, potentially compromising the functionality and stability of the entire system.

Figure 4 RC-Boat when similar weight is applied

However, in a real-world environment with an actual boat, this issue would not arise. Unlike the smaller remote-controlled version, a full-sized boat has greater buoyancy and stability, allowing us to integrate our entire system seamlessly. By creating a designated hole in the hull, we can securely attach the sensor to the bottom of the boat, ensuring it remains submerged for accurate data collection. Meanwhile, other essential components, such as the Raspberry Pi 4, power bank, and USB-UART converter, can be safely housed inside the boat, positioned in a dry and secure area to protect them from water exposure. This setup would allow for a fully functional and efficient integration of our system without compromising the boat's stability or performance.

3 MCU Programming

The Raspberry Pi 4 Model B was chosen for its superior processing power, built-in Wi-Fi, and extensive data storage capabilities, which streamlined cloud connectivity and data management [2]. Its ability to handle real-time data processing and run complex algorithms significantly enhanced the project's efficiency.

The Raspberry Pi 4 was set up by installing the Raspberry Pi OS on an SD card and configuring system settings through updates and interface adjustments [3]. A virtual environment was created, and essential libraries, including AWS IoT SDK, Boto3, Blue Robotics Ping, PySide6, and Requests, were installed to enable seamless script execution at startup. This setup ensured proper communication, data handling, and cloud connectivity for the project.

3.1 Wi-Fi Connection on Raspberry Pi 4

In the implementation of the IoT Lake/River Depth Measurement and Monitoring System, reliable internet connectivity was essential for real-time depth monitoring when Wi-Fi and network access were available, as well as for storing the locally saved mapped route to cloud storage. While a dedicated modem would have been ideal for seamless internet connectivity in a real-world deployment, the project relied on connecting the Raspberry Pi 4 (RPi4) to a mobile hotspot via Wi-Fi. This setup ensured that collected data could be uploaded to the cloud once a network connection became available, allowing remote access, visualization, and real-time monitoring when possible.

To automate Wi-Fi connectivity, a script was developed to enable Wi-Fi and connect to a known network on boot. The script first checked if Wi-Fi was blocked using the *rfkill* list command and unblocked it, if

necessary, with sudo rfkill unblock wifi. The Wi-Fi interface was then turned on using sudo nmcli radio wifi on, followed by a 4-second delay to allow the hardware to initialize. The system then attempted to connect to the predefined Wi-Fi network using sudo nmcli dev wifi connect <SSID> password <Password>. If the connection failed, the script retried every 5 seconds until successful, ensuring continuous network access. To verify connectivity and diagnose any issues, several commands were executed manually during testing. The *rfkill list* command was used to check if the Wi-Fi interface was blocked by software or hardware, ensuring potential connectivity issues could be identified. If the interface was blocked, sudo rfkill unblock wifi was executed to restore functionality [4]. The nmcli dev wifi list command listed available Wi-Fi networks, allowing verification of detected networks before attempting a connection [5]. The IP link show command displayed details of network interfaces, helping confirm the status and availability of the Wi-Fi adapter. The sudo nmcli radio wifi on command ensured that the Wi-Fi radio interface was enabled for scanning and connecting to networks. Finally, sudo nmcli dev wifi connect <SSID> password <Password> was used to establish a connection to the specified Wi-Fi network, enabling internet access for data transmission and real-time monitoring.

By implementing this automated Wi-Fi connectivity process, the project maintained reliable data transmission for cloud synchronization and real-time depth monitoring when Wi-Fi and network access were available. This approach enabled the system to function effectively within the constraints of the project while highlighting the need for a dedicated modem for practical real-world deployment.

3.2 Interfacing Ping2 Sensor on Raspberry Pi 4

The Ping2 sensor was connected to the Arduino UNO using Software Serial on pins 9 (RX) and 10 (TX), with power and ground wired accordingly. The *ping-arduino* library was installed, and the *ping1d-simple* code was used to read depth and confidence values via Serial Monitor. The Arduino acted as a master in the UART communication with Ping2, retrieving sensor data and transmitting it to the Raspberry Pi via USB. A Python script was implemented on the Raspberry Pi to read data from the serial port, writing incoming values to a file for real-time logging and analysis. The Raspberry Pi handled data processing while leveraging the Arduino as a signal translator.

To enhance efficiency and simplify our setup, we revised our design by replacing the Arduino UNO with a USB-UART converter for interfacing the Ping2 sensor with the Raspberry Pi 4. This change was primarily driven by the need to implement a *request-response protocol* while reducing power consumption and improving compactness for easier integration on a boat. Initially, the Arduino was used to provide library support for sensor communication, but we addressed this by utilizing the *Blue Robotics Python library*

directly on the Raspberry Pi. While a vendor-specific serial adapter was considered, budget constraints led us to choose a *cost-effective USB-UART converter*, which effectively translates UART serial signals from the Ping2 sensor into USB-compatible communication with the Raspberry Pi.

3.2.1 Implementing the Request-Response Protocol

To facilitate communication between the Raspberry Pi 4 and the *Ping2 sensor*, we installed the *Blue Robotics ping-python* library, which provides a Python-based implementation of the *Ping messaging protocol* and a device API for the *Ping1D sonar* [5]. We created a virtual environment on the Raspberry Pi 4 to manage dependencies and installed the necessary libraries within it. This ensures an isolated environment, preventing conflicts with system-wide packages and allowing greater flexibility in package management.
By installing the ping-python library within the virtual environment, we ensured that the package dependencies remained contained, avoiding system-wide modifications. Using this library, we implemented a request-response protocol, allowing the Raspberry Pi to actively request data from the Ping2 sensor and receive real-time measurements in return. This method is more efficient than continuous streaming as it reduces unnecessary data transmission, lowers power consumption, and ensures that only the required data is processed at any given time.

To establish direct communication between the Ping2 sensor and the Raspberry Pi 4, we developed a Python script that interacts with the USB-UART converter, allowing the system to send commands and retrieve distance and confidence readings. As illustrated in the sequence diagram, the implementation follows a request-response protocol using the Blue Robotics Ping1D library, optimizing data retrieval while minimizing power consumption.

The process begins with the initialization phase, where the Raspberry Pi sets up the environment and establishes a serial connection with the Ping2 sensor (myPing.connect_serial(device, baudrate)). This step ensures stable communication over the USB-UART interface (/dev/ttyUSB0, 115200 baud). As shown in the diagram, the script then calls myPing.initialize() to verify sensor detection before proceeding. If the initialization fails, the script exits to prevent execution errors.

Following successful initialization, the Raspberry Pi sends a request to retrieve sonar readings (myPing.get_distance()). The Ping2 sensor responds with distance (mm) and confidence (%) values, completing one cycle of the request-response protocol. This approach, reflected in the diagram's loop

structure, ensures that data is only retrieved when needed, reducing unnecessary power consumption. To prevent excessive requests, a 100ms delay (time.sleep(0.1)) is introduced between consecutive readings.

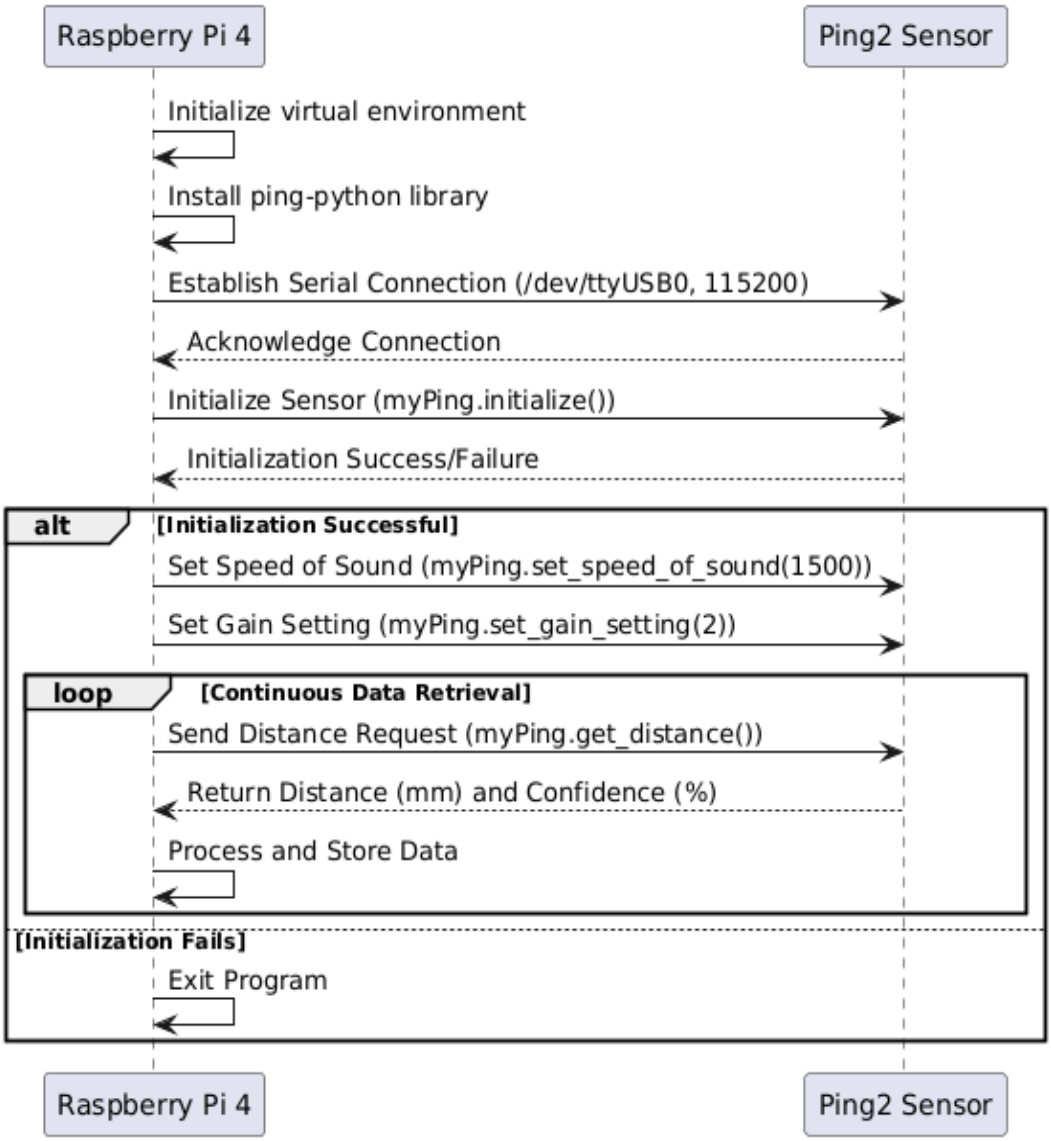


Figure 5 Request Response protocol to interface Ping2 sensor on RPi4

To enhance measurement accuracy, the diagram highlights additional configurations, such as setting the speed of sound (myPing.set_speed_of_sound(1500)) to account for environmental factors like temperature and salinity. Additionally, gain sensitivity adjustments (myPing.set_gain_setting(2)) help filter weak reflections, ensuring clearer sonar readings.

By structuring communication as a request-response model, we significantly improved real-time sonar readings, minimized power usage, and enhanced system efficiency. As depicted in the final steps of the diagram, integrating the Ping1D library directly on the Raspberry Pi 4, without an Arduino intermediary, streamlined the design while maintaining robust and reliable sonar measurements.

3.3 Interfacing Global Position System on Raspberry Pi 4

3.3.1 Interfacing the GPS Device on RPi4

The GU-504GGB GPS module was connected to the Raspberry Pi 4 via USB. To verify the connection, the *lsusb* command was executed, confirming the module's recognition. The assigned serial port was identified using dmesg | grep tty, which displayed the port as /dev/ttyUSB0. To check the raw GPS data transmission, the command cat /dev/ttyUSB0 was used, and for a structured view of NMEA sentences, screen /dev/ttyUSB0 115200 was run, ensuring the baud rate was set to 115200 bps as specified in the module's datasheet.

The GNGLL (Geographic Position – Latitude/Longitude) NMEA sentence was utilized to obtain location data. It followed the structure:

$GNGLL, ddmm.mmmm,a,dddmm.mmmm,a,hhmmss.sss,A,a*hh

where ddmm.mmmmm and dddmm.mmmmm represented latitude and longitude in degrees and minutes. The hemispheres were indicated by 'N' for North, 'S' for South, 'E' for East, and 'W' for West. The time was recorded in UTC, while the status field ('A' for valid data, 'V' for invalid) determined data reliability [6]. We implemented a method to read and parse *GNGLL sentences* from the serial port to extract latitude, longitude, and accuracy. The system continuously monitored incoming serial data, identifying lines containing the "GNGLL" identifier. When a valid sentence was detected, the latitude, longitude, and status were extracted and stored in a structured format.

To ensure proper interpretation, the GNGLL sentence was broken down into its individual components, including raw latitude, latitude hemisphere (N/S), raw longitude, longitude hemisphere (E/W), and the status indicator. The latitude and longitude values were then converted from the NMEA ddmm.mmmmm format to decimal degrees by separating degrees and minutes, calculating the decimal value, and adjusting the sign based on the hemisphere (negative for South/West, positive for North/East).Accuracy was determined based on the status flag (valid (A) or invalid (V)).

Overall, the script effectively read real-time GPS data from the serial port, parsed and converted it into a more usable format, and stored it for further use in navigation or tracking applications.

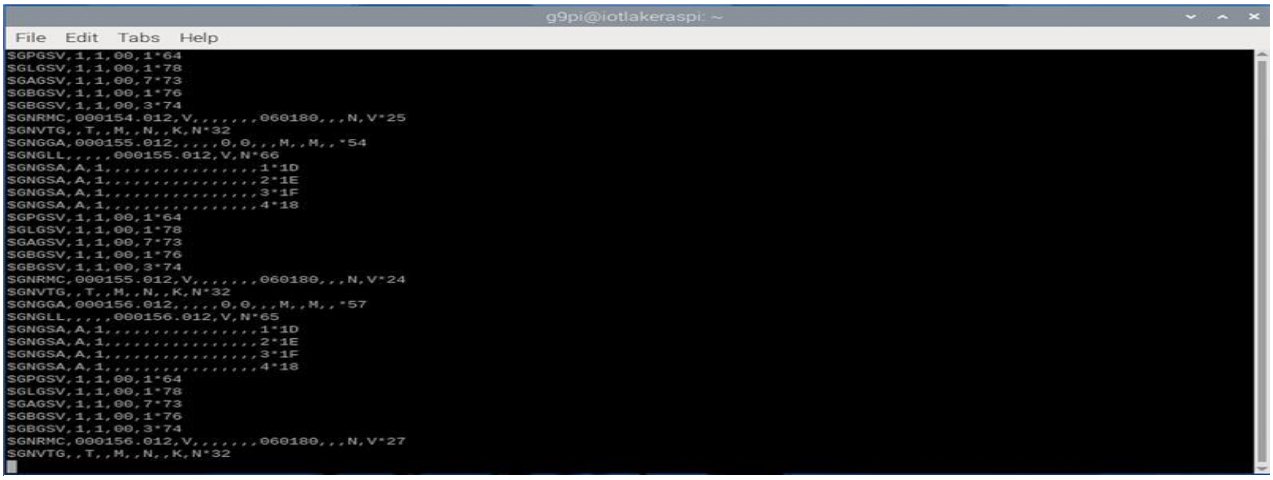


Figure 6 Output showing the NMEA messages from GPS module

However, as shown in Figure 6 acquiring a GPS signal indoors was not possible due to building obstacles and weak satellite signals. GPS devices typically do not function indoors, as their receivers require an unobstructed view of satellites to obtain valid data unless repeaters are used. In contrast, Wi-Fi geolocation can determine latitude and longitude by referencing the MAC addresses of nearby Wi-Fi access points. However, this method is only viable when a Wi-Fi network is available. In cases where neither GPS nor Wi-Fi geolocation is accessible, GPS simulation is utilized, generating random latitude and longitude values with consistent variation to mimic real movement. For outdoor environments such as lakes or rivers, the GPS device provides accurate latitude, longitude, direction, and time data.

3.3.2 Wi-Fi Geolocation

Wi-Fi geolocation was implemented by scanning nearby Wi-Fi access points and using their MAC addresses and signal strengths to estimate location. A Google Maps Geolocation API key was generated and included in API requests as a query parameter [7]. The command sudo iwlist wlan0 scan was used to retrieve a list of available Wi-Fi access points, extracting MAC addresses and signal strengths. A POST request was then sent to the Google Geolocation API with this data in JSON format, allowing the API to return estimated latitude, longitude, and accuracy in meters [8]. If geolocation failed, a GPS simulation was used as a fallback when Wi-Fi network is not available. This method is only used for testing indoors and validation the actual location on map.

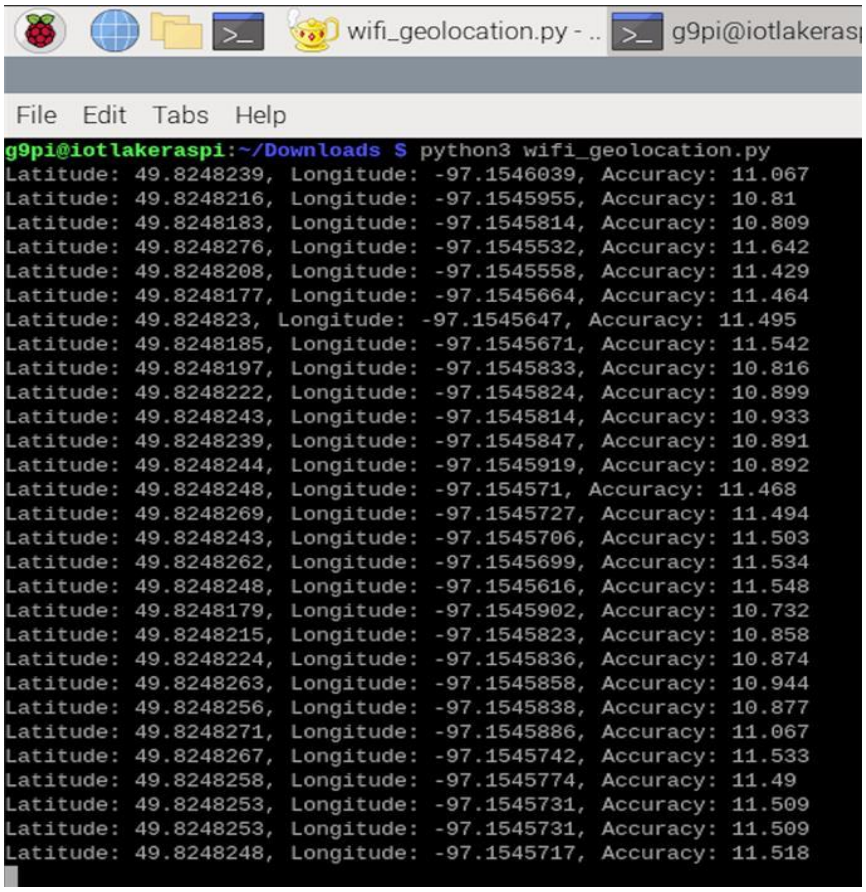


Figure 7 Wi-Fi Geolocation returning Latitude, Longitude & Accuracy using Wi-Fi access points

3.3.3 Reverse Geo Coding

Reverse geocoding is implemented to convert latitude and longitude coordinates into a human-readable address, making location data more meaningful and accessible. This process involves sending a request to the Google Maps Geocoding API with the latitude, longitude, and a valid API key [9]. If the request is successful, the API responds with a JSON object containing location details, including a formatted address. The script extracts this formatted address and uses it to generate a meaningful filename before uploading the file to an S3 bucket. If the API request fails or no address is found, appropriate error handling ensures smooth execution.

3.3.4 GPS Simulation

GPS values are simulated by initializing latitude and longitude with predefined coordinates. Small random variations within a set speed range are added to these values to mimic movement. The updated coordinates, along with an accuracy value, are then stored in a shared data structure. This process generates approximate

location data for testing and visualization. This method is only used when testing indoors and Wi-Fi is not available during testing scenarios.

3.3.5 GPS Methods and Usage

GPS on the Raspberry Pi 4 was implemented using three methods based on availability. When no network was available indoors, GPS data was simulated using random values. If Wi-Fi access points were detected, Wi-Fi geolocation was used to obtain GPS coordinates via the Google API. In real-world outdoor environments, the external GPS device would provide valid latitude and longitude data in presence of satellite signals and receivers, making it the preferred method for accurate positioning. Our software is designed to seamlessly switch between GPS data, Wi-Fi geolocation, and GPS simulation, ensuring reliable positioning based on the availability of GPS signals and Wi-Fi networks.

3.4 Local Storage

Mapped routes are stored locally on the SD card of the Raspberry Pi 4 to ensure data persistence before being uploaded to the cloud. The system collects depth and location data in real-time, storing it in a structured list. When a mapping session is completed, the data is saved as a JSON file with a unique filename, generated using a combination of a UUID and the current date and time. This ensures that each recorded session is uniquely identified and easily retrievable. The locally stored files serve as a backup and allow the system to retain route data even if there is a delay or issue with cloud synchronization. Once saved, these files can be later accessed for further processing, analysis, or uploading to AWS S3 for remote access and visualization within the app.

4 Cloud Storage and Communication Setup

4.1 Wi-Fi Communication

Using Wi-Fi communication and MQTT with AWS IoT Core as the broker, the system was implemented to enable real-time depth monitoring by transmitting data from an RPi4-based depth measurement device to a mobile app. The RPi4 is programmed to collect sonar-based depth data and publish the message to a topic via MQTT, allowing remote users to access and visualize water depth and hazard information whenever network connectivity was available.

4.1.1 MQTT Protocol

MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol designed for IoT communication, enabling efficient data transmission over limited-bandwidth networks. It follows a publish-subscribe (pub-sub) model, where clients (devices or applications) either publish messages or subscribe to receive them. An MQTT broker acts as the central hub, managing message distribution between publishers and subscribers based on predefined topics. Clients establish a connection with the broker, subscribe to topics of interest, and receive real-time data updates whenever a new message is published under that topic. Topics are hierarchical strings used to categorize and filter messages exchanged between clients. They act like channels where publishers send messages and subscribers receive relevant data. Topics are structured using a / separator, allowing for efficient message routing. Clients subscribing to a topic receive all messages published under it, ensuring real-time updates based on their selected topics of interest.[10]

AWS IoT Core acted as the MQTT broker, enabling IoT devices like the RPi4 to connect, publish, and subscribe to messages securely. It facilitated real-time data exchange between devices and cloud services, ensuring seamless communication. Using the AWS IoT Device SDK for Python, we integrated MQTT on the RPi4 to send and receive depth monitoring data via AWS IoT and a secure communication [11].

4.1.2 RPi4 Device Connection with AWS

The Raspberry Pi 4 was configured as an AWS IoT Thing by installing the AWS IoT Device SDK for Python, along with necessary dependencies like CMake, libssl-dev, Git, and Python3 with pip3. IoT resources, including a thing object, policy, and X.509 certificates, were created to enable secure authentication and communication with AWS IoT Core. The device certificates and keys were installed on the Raspberry Pi, allowing it to establish an MQTT connection using mutual TLS authentication [12].

A custom Python script was implemented to handle MQTT communication. The *connect_and_subscribe ()* function established a secure connection to AWS IoT Core using the *mqtt_connection_builder.mtls_from_ path ()* method, specifying the endpoint, device certificate, private key, and root CA certificate. The connection was set up with a persistent session and a 30-second keep-alive interval to maintain continuous communication. Upon a successful connection, the script allowed the device to subscribe to relevant topics. For publishing data, whenever the Raspberry Pi 4 measured depth, GPS coordinates, and other sensor data, it structured the data in JSON format and used the *publish_message()* function to send it to AWS IoT Core under the topic "g9capstone/readValues". If Wi-Fi is available, used the *mqtt_connection.publish()* method to ensure reliable delivery with QoS level 1 (AT_LEAST_ONCE). This setup enabled real-time

synchronization of depth and locations readings with AWS IoT Core. Additionally, once connected to AWS IoT, the RPi4 starts publishing a heartbeat message every minute to the topic "g9capstone/piHeartbeat", indicating its connection status and ensuring continuous monitoring of the device's availability. This setup enables real-time synchronization of depth, location readings, and device health with AWS IoT Core.

4.1.3 App Connection with AWS

On the App side, Fleet Provisioning with Claim Certificates enables the seamless and scalable onboarding of devices by dynamically assigning each user a unique device certificate upon their first connection to AWS IoT Core. This approach eliminates the need for manually preloading device certificates, making it ideal for large-scale IoT deployments.

Fleet Provisioning is a mechanism in AWS IoT that automates the registration and authentication of devices. It allows devices to use a provisioning claim certificate—a temporary certificate embedded in the app—to connect to AWS IoT Core and request a unique device certificate. The app first connects using claim certificates, subscribes to the AWS provisioning topics, and initiates a certificate creation request. Once the new certificate is issued, the app registers it, attaches necessary IoT policies, and then switches to using the unique certificate for all future communications [12].

To facilitate real-time communication, the app subscribes to *g9capstone/#,* a wildcard topic that captures updates from all subtopics. Using MQTT methods, it establishes a subscription with QoS level 1 (client.subscribe(topic, MqttQos.atLeastOnce)) and listens for incoming messages asynchronously (client.updates!.listen()). The received messages are processed based on their topic, such as "g9capstone/readValues" for real-time sensor data from RPi4, "g9capstone/piHeartbeat" for monitoring device health, and responseTopic for AWS IoT provisioning updates. This ensures a secure, efficient, and seamless IoT communication flow between the RPi4 and AWS IoT Core. The RPi4 continuously measures depth, GPS location, and other telemetry data, publishing it to AWS IoT Core whenever Wi-Fi is available. By leveraging Fleet Provisioning, the app ensures a secure and automated way to onboard devices, maintain authenticated communication, and enable real-time IoT data streaming between the RPi4 and the application.

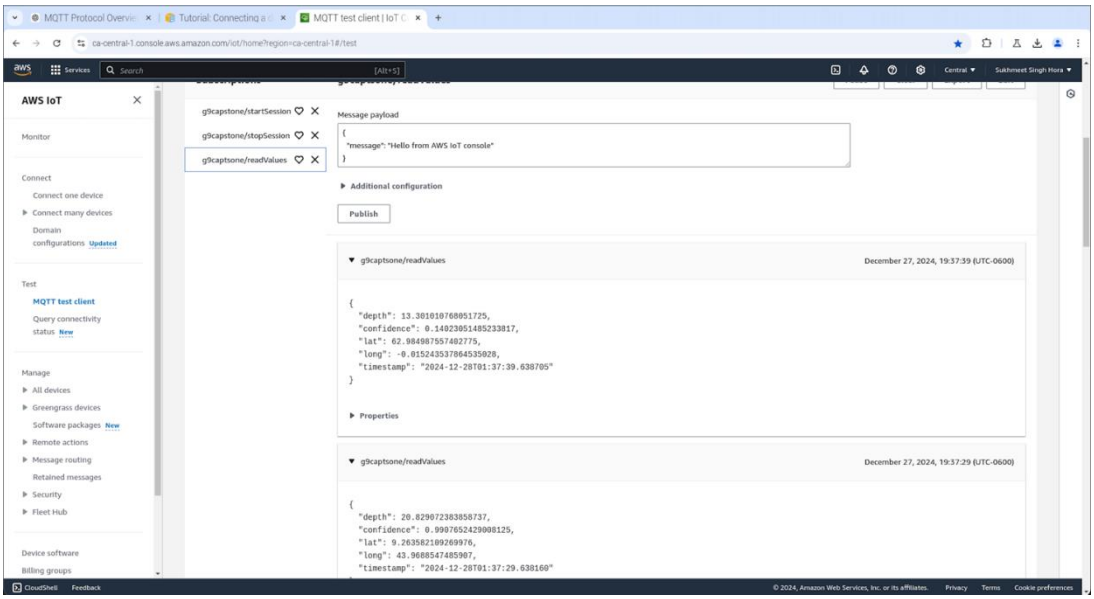


Figure 8 Example of publishing and receiving value using RPi4 and MQTT Test Client [13]

4.1.4 Wi-Fi Communication Flow from RPi4 to App via AWS IoT Core

1. RPi4 connects to Wi-Fi and then connect to AWS IoT Core using certificates and keys.
2. Every 1 minute, RPi4 publishes a heartbeat to g9capstone/piHeartbeat.
3. The App connects to AWS IoT Core using Fleet Provisioning by claim certificates and subscribes to g9capstone/#.
4. RPi4 reads sensor data (Depth, GPS) and publishes to g9capstone/readValues.
5. AWS IoT Core forwards both messages (heartbeat & sensor data) to the App.
6. The App processes heartbeat & sensor data separately.

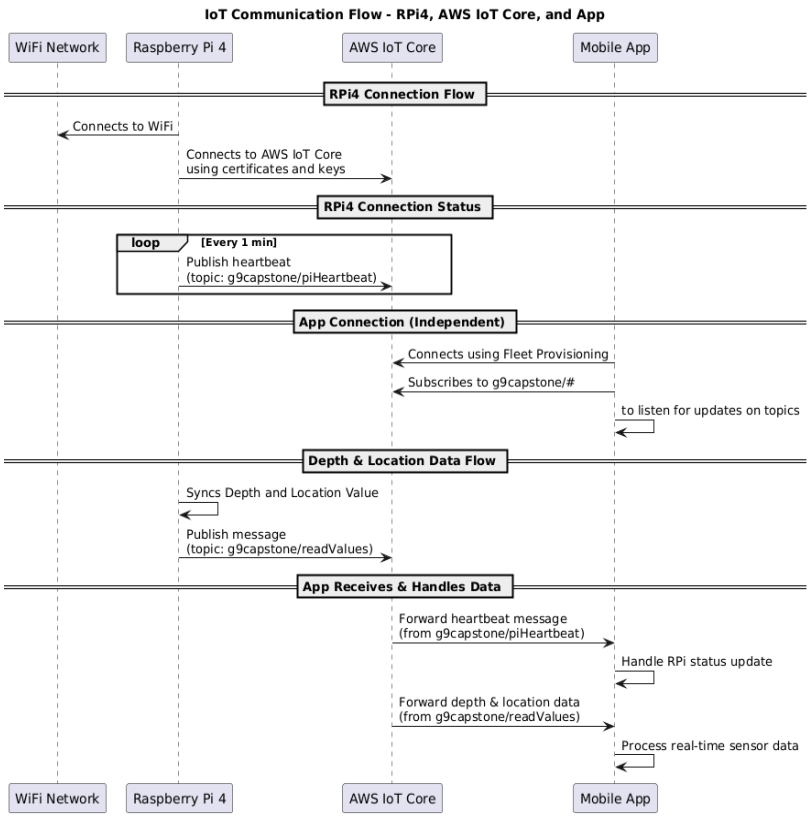


Figure 9 Wi-Fi Communication Flow from RPi4 to App via AWS IoT Core

4.2 Bluetooth Communication

Our depth monitoring device was able to use Wi-Fi to transfer data from the device to cloud storage and the app. However, we decided to add Bluetooth communication to our device for one main reason. We cannot rely on our users having access to Wi-Fi on lakes and rivers. Therefore, we used a short-range communication protocol for data transfer when Wi-Fi is unavailable.

We used this Bluetooth communication protocol to provide two functionalities. The first functionality is to provide a Human Machine Interface (HMI) functionality on our Graphical User Interface (GUI) app to control the depth monitoring device. We chose to implement a software HMI instead of a physical HMI for the following reasons.

- Using a physical HMI requires buttons for controls and LEDs or a small LCD screen for feedback. This will add additional expenses to our budget.
- Using a physical HMI limits device control to the number of buttons. Using software HMI allows us to exchange any data/commands we wish.
- Using a software application HMI on GUI is the more modern approach.
- Additionally, our users should have our application with them anyways on their phone. So, it is convenient to integrate an HMI into our application

The second functionality is to share real-time data collected with the device with our users through the app. The device will package the real-time data collected by the device into a JSON. This JSON will then be shared directly with the app through Bluetooth communication.

4.2.1 Bluetooth Low Energy Protocol

BLE works in a similar way to client-server connections. The depth monitoring device will function as a server AKA a peripheral device. The app will function as a client AKA a central device. BLE works by creating a generic attribute profile (GATT) which contains three different types of attributes: service attribute, characteristic attribute, and descriptor attribute. To create a BLE server, one must create a combination of these attributes which results in the creation of a GATT table. The BLE server advertises its services. The BLE client looks at all nearby devices and what services they offer and initiates a connection with one of the devices. Upon connection the client can view the GATT table and start exchanging data. See appendix B for more in-depth knowledge of the Bluetooth protocol or check [14].

4.2.2 Device Server

Table 4 GATT table for device server

Attribute Name	UUID	Attribute Permissions	Attribute Value
Device Control Service (Service Declaration)	0x2800	Read	UUID: 37dd28eb-b0e5-4714-b874-0fa1f50f88bf
Device State Characteristic (Characteristic Declaration)	0x2803	Read	Properties: Read & Write
			UUID: 9232ed36-2122-4773-b1c8-31c2d5114e96
Device State Characteristic (Value Declaration)	9232ed36-2122-4773-b1c8-31c2d5114e96	Read & Write	Device State (START, STOP, UPLOAD)
Depth Monitoring Service (Service Declaration)	0x2800	Read	UUID: cbc3bb98-e29b-4b8d-8a1b-3e90aa65a790
Depth Value Characteristic (Characteristic Declaration)	0x2803	Read	Properties: Notify
			UUID: 6943ec7e-cb2e-4b44-9adc-7f5d12837bd1
Depth Value Characteristic (Value Declaration)	6943ec7e-cb2e-4b44-9adc-7f5d12837bd1	Notify	Depth Value (Depth Value JSON)
Client Characteristic Configuration Descriptor (Descriptor Declaration)	0x2902	Read & Write	Unsubscribe: 0x0 Subscribe: 0x1

For our project, we created a BLE server and added our custom attributes to the GATT table, which shown above. To provide the first functionality discussed above, we created a Device Control Service attribute along with a Device State Characteristic attribute. This service is used to control the state of the device. It has a characteristic value that the client can use to send commands to the device. The characteristic property is set so that a central device can read and write values. The client can write "START", "STOP", or "UPLOAD" as utf-8 bytes to the characteristic value to send commands to the device.

To provide the second functionality above, we created a Depth Monitoring Service attribute and a Depth Value Characteristic attribute. This service is used by the server to send the most recent depth value read to the client. It has a characteristic value which contains a JSON with the most recent reading. The

characteristic property is set so that the server notifies connected clients with new readings. The client can subscribe and unsubscribe to receive depth data notifications from the server.

4.2.3 App Client

The app uses the flutter_reactive_ble library to interact with Bluetooth Low Energy (BLE) devices [15]. It follows a structured process to scan for devices, establish a connection, subscribe to real-time data, and send commands. The reactive BLE library provides comprehensive support for interacting with Bluetooth Low Energy (BLE) devices. It enables seamless device discovery, allowing the app to scan for nearby BLE devices that advertise specific services. Additionally, it includes functionality to monitor the BLE status of the host device, ensuring that the app can detect changes such as Bluetooth being turned off or permissions being revoked. Once a suitable device is found, the library facilitates establishing and maintaining connections, even across multiple devices. It ensures that connection status updates are continuously tracked, making it easier to manage simultaneous connections. While service discovery happens implicitly during this process, the library provides the necessary mechanisms to access characteristics for reading and writing data. To support real-time data exchange, the library allows subscribing to characteristics, enabling automatic notifications whenever new data is available. It also includes advanced features such as clearing the GATT cache to resolve stale data issues and negotiating the Maximum Transmission Unit (MTU) size, optimizing data transfer efficiency. These capabilities ensure reliable and efficient communication between the app and BLE devices.

```
D/BluetoothAdapter(11770): isLeEnabled(): ON
D/BluetoothLeScanner(11770): onScannerRegistered() - status=0 scannerId=8 mScannerId=0
W/BluetoothAdapter(11770): getBluetoothService(), client: android.bluetooth.BluetoothDevice$1@9551640
D/BluetoothAdapter(11770): isLeEnabled(): ON
I/flutter (11770): Discovered Device IDs: [DC:A6:32:1A:7A:72]
I/flutter (11770): Connecting to Device ID: DC:A6:32:1A:7A:72
D/BluetoothGatt(11770): connect() - device: DC:A6:32:1A:7A:72, auto: false, eattSupport: false
D/BluetoothGatt(11770): registerApp()
D/BluetoothGatt(11770): registerApp() - UUID=c3707d3d-8585-4f7b-ab1e-180707079526
D/BluetoothGatt(11770): onClientRegistered() - status=0 clientIf=8
I/flutter (11770): Connection state: DeviceConnectionState.connecting
D/BluetoothGatt(11770): onClientConnectionState() - status=0 clientIf=8 device=DC:A6:32:1A:7A:72
I/flutter (11770): Connection state: DeviceConnectionState.connected
D/BluetoothGatt(11770): discoverServices() - device: DC:A6:32:1A:7A:72
D/BluetoothGatt(11770): onConnectionUpdated() - Device=DC:A6:32:1A:7A:72 interval=6 latency=0 timeout=500 status=0
D/BluetoothGatt(11770): onSearchComplete() = Device=DC:A6:32:1A:7A:72 Status=0
D/BluetoothGatt(11770): onConnectionUpdated() - Device=DC:A6:32:1A:7A:72 interval=36 latency=0 timeout=500 status=0
D/BluetoothGatt(11770): setCharacteristicNotification() - uuid: 6943ec7e-cb2e-4b44-9adc-7f5d12837bd1 enable: true
I/flutter (11770): Successfully connected and subscribed to BLE device.
```

Figure 10 Example output of App discovering, connecting and subscribing to BLE characteristics

Scanning for Devices

The app initiates a BLE scan to discover nearby devices advertising a specific service. It filters results based on the targeted service UUID and collects unique device identifiers. The scan runs for a limited duration, after which the discovered device IDs are processed. If no devices are found, the process can be retried after a short delay.

Connecting to a Device

Once a device advertising the required service is identified, a connection request is made. The app ensures that the device supports the expected service and attempts to establish communication. Throughout the connection process, state updates are monitored to detect any changes, such as connection failures or unexpected disconnections.

Subscribing and Listening for Data

After a successful connection, the app subscribes to a characteristic responsible for transmitting depth values. The characteristic supports BLE notifications, enabling real-time updates whenever new data is available. Incoming values are decoded and processed, allowing the app to display or utilize the received data.

Sending Commands

To control the BLE device, the app writes specific commands to a designated characteristic. Commands like "START," "STOP," or "UPLOAD" are encoded into a format the device can interpret and then sent over BLE. The write operation ensures that the device receives and processes the commands as expected.

4.2.4 BLE Communication Flow

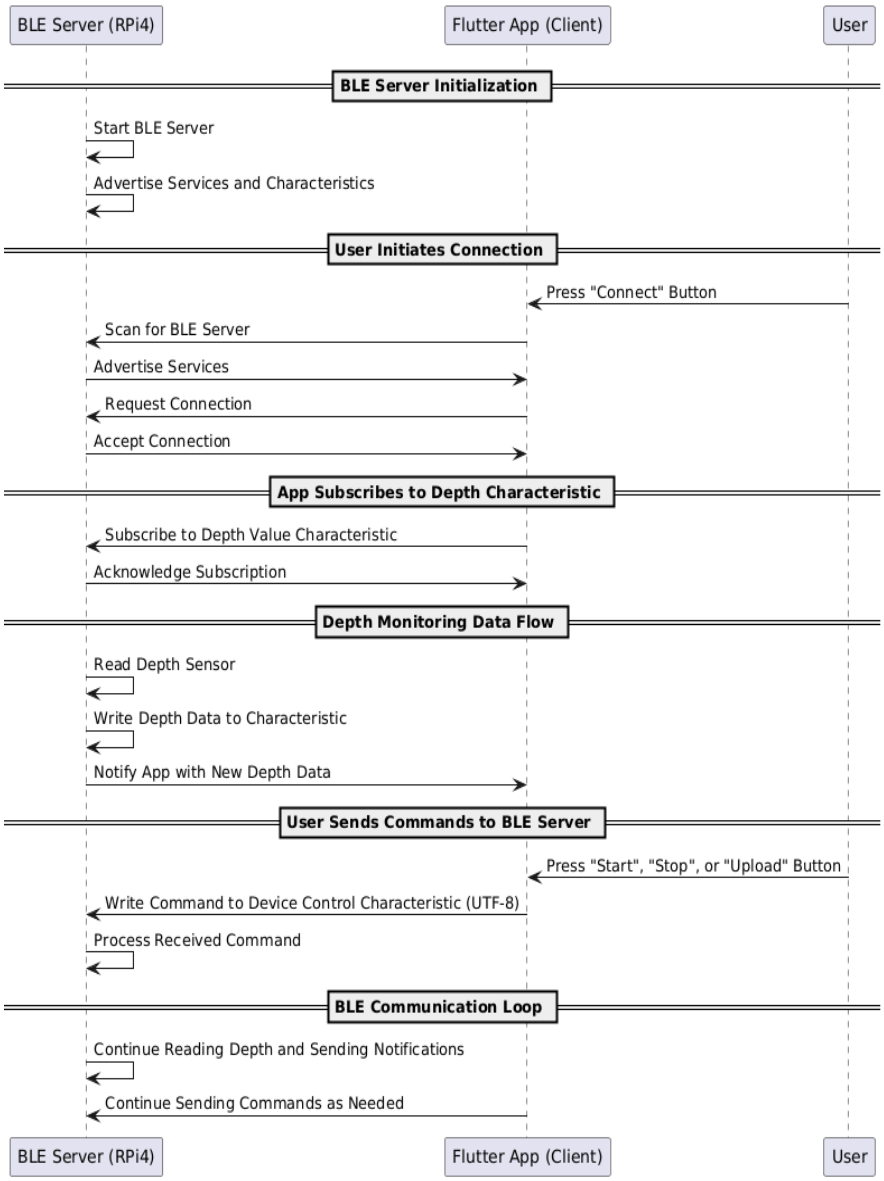


Figure 11 BLE Communication Flow: Connect, Real Time Depth Monitoring and Device Control

4.2.5 BLE Communication Overview

As a result, the depth monitoring device can communicate with the app through Bluetooth. The user can control the device using the app. They can start and stop a data recording session. When a data recording session is complete, they upload the data to our cloud storage given a Wi-Fi connection. When Wi-Fi is unavailable, the device will share real-time data that is being recorded with the user via BLE. The data will also be stored in the SD card of the device, so that it can be uploaded to our cloud storage later when Wi-Fi is available.

4.3 Real Time Communication Message Passing

RPi4 reads the depth, and location values and communicates a JSON message to App via Wi-Fi using MQTT protocol over AWS IoT Core and via BLE when Wi-Fi is not available.

Table 5 Real Time JSON Message Contains

distance	integer Num
confidence	integer Num
latitude	decimal (Float)
longitude	decimal (Float)
accuracy	decimal (Float)
timestamp	yyyy-mm-dd hh:mm:ss

Example of a JSON message: {

"distance": 12,

"confidence": 0,

"latitude": 49.8083267,

"longitude": -97.1345926,

"accuracy": 15.314,

"timestamp": "2025-01-29 15:20:40"

}

4.4 Cloud Storage

Amazon Simple Storage Service (Amazon S3) is a cloud-based object storage service that provides high scalability, security, and reliability. It allows users to store and retrieve any amount of data for various use cases, including backups, data lakes, mobile applications, and IoT storage. With built-in management features, S3 helps optimize data access, security, and compliance needs for businesses of all sizes.

4.4.1 Usage and Setup of AWS S3

AWS S3 is used in this project to store user-specific IoT certificates, pre-mapped routes from Raspberry Pi 4, and machine learning model files. The setup involved creating an S3 storage bucket via the AWS Amplify

backend studio. For integration within the Flutter app, the amplify_storage_s3 package was used to enable seamless file uploads and retrieval.

To access stored pre-mapped route files, the app retrieves a list of all files stored in the *public/MappedRoutes* directory. This is done using the Amplify Storage list function, which allows the app to scan and fetch details of all files within the specified S3 path. The function ensures that all available pre-mapped route files are listed and can be accessed by the user. The storage listing process is configured to retrieve all objects within the directory by utilizing plugin-specific options, ensuring that no files are missed.

On Raspberry Pi 4, the boto3 library was used for direct file uploads to S3. The upload process was implemented using the boto3.client('s3') function [16], where AWS credentials (access_key_id and secret_access_key) are used for authentication. The script uploads files by calling an S3 upload function, ensuring that collected data, including pre-mapped routes and analyzed results, is securely stored and accessible from the cloud for users to retrieve and visualize within the app.

Pre-Mapped Routes to S3

In this project, pre-mapped routes are generated and saved locally on the Raspberry Pi 4 (RPi4) after collecting depth and location data. The RPi4 system continuously logs depth readings and GPS coordinates, storing them as a timestamped JSON message list. These JSON files also contain analyzed data from a hazard differentiation algorithm, which processes the collected information to identify potential hazards and navigation insights.

To make this analyzed data accessible to users, the JSON files are uploaded to AWS S3. This ensures that boaters using the mobile app can retrieve pre-mapped routes, review hazard assessments, and make informed navigation decisions for improved safety. The upload process involves reading the locally stored JSON files, extracting latitude and longitude coordinates, and performing reverse geocoding to generate a human-readable filename.

For the actual upload, the script uses the *boto3* library in Python, specifically the boto3.client('s3'). upload_file() function. This function takes the local file path, the target S3 bucket name, and the destination path (public/MappedRoutes/) as parameters. The app reads pre-mapped route files stored in AWS S3 by first listing all files in the *public/MappedRoutes/* directory. This ensures that all available mapped route data is retrieved for processing. Once the files are listed, the app logs the retrieved items and iterates through each file to download its content. After downloading, the file data is decoded into a UTF-8 string and parsed as JSON. Each JSON object in the list is then processed to extract relevant depth and location information,

enabling users to visualize and analyze the mapped routes. Throughout this process, the app implements error handling mechanisms to manage issues such as missing files, incorrect formatting, or connectivity problems, ensuring a seamless user experience.

Hazard Differentiation Model Storage

The hazard differentiation model is an SVM-based machine learning model stored as a *.pkl* file on AWS S3. The Raspberry Pi system ensures it always has the latest version by automatically downloading the most updated model from S3. This allows the device to use the most recent training data for making accurate predictions. To retrieve the model, the script initializes an S3 client using boto3, authenticated with AWS credentials. It first checks if the designated local folder exists and creates it if necessary. The model file is then downloaded from the specified S3 bucket to this folder using the s3.download file () function. If any issues arise, such as missing credentials, incorrect paths, or network errors, appropriate error handling mechanisms are in place to ensure a smooth download process. A New models can be trained remotely using pre-mapped route data and upload updated *.pkl* files to S3. This enables seamless deployment, allowing Raspberry Pi to fetch and apply the latest model without requiring physical access to the device.

5 Hazard Differentiation

5.1 Overview

Hazard differentiation is a feature we added to improve the safety of users during boating operations. We wanted to utilize machine learning to identify hazards in previously measured depths, as well as evaluating the level of danger in the spotted hazards. From there, we integrated it with the mobile application to send alerts to the user when approaching any of these hazards.

5.2 Machine Learning

We employed supervised learning to train a model using labeled data, enabling it to predict hazard severity based on sensor inputs. Supervised learning is an approach that involves having a dataset that has been labelled to train a model. The model learns the relationship between the inputs and outputs through a selected number of features from the dataset, then uses that to make predictions on unseen data. [17]

5.2.1 Dataset Development

We looked at some bathymetry datasets available online to use for training. Most of the datasets we found were measuring ocean depths, which were not suitable since they are much deeper than lakes or rivers. We also found lake datasets, but we did not have information for underwater hazards, so we could not use them for supervised learning.

We created a custom data set using our IoT system composed of the Raspberry Pi, GPS, and sensor. The features we obtained from the sensor were distance and confidence value; and from the GPS, we obtained longitude, latitude, accuracy, and timestamp. We chose four features for training, which were depth, confidence value, latitude, and longitude. The target labels were numbered 0-5 to classify the danger level of the hazard, with no hazard being detected and 5 being the highest level. Our proprietary dataset used several field tests conducted at a river-model apparatus available in a Hydraulics Lab on campus.

5.2.2 Model Selection and Training

We evaluated seven ML algorithms for compatibility with the Raspberry Pi 4's constraints (limited CPU/RAM, 4GB model). Out of those considerations, Artificial Neural Networks (ANN), AdaBoost, Decision Trees and GPIO Zero were excluded due to hardware limitations or overfitting risks. We moved forward with further investigation into the model accuracy for Random Forest Classifier (RFC), Support Vector Machine (SVM), and K-Nearest Neighbors (KNN).

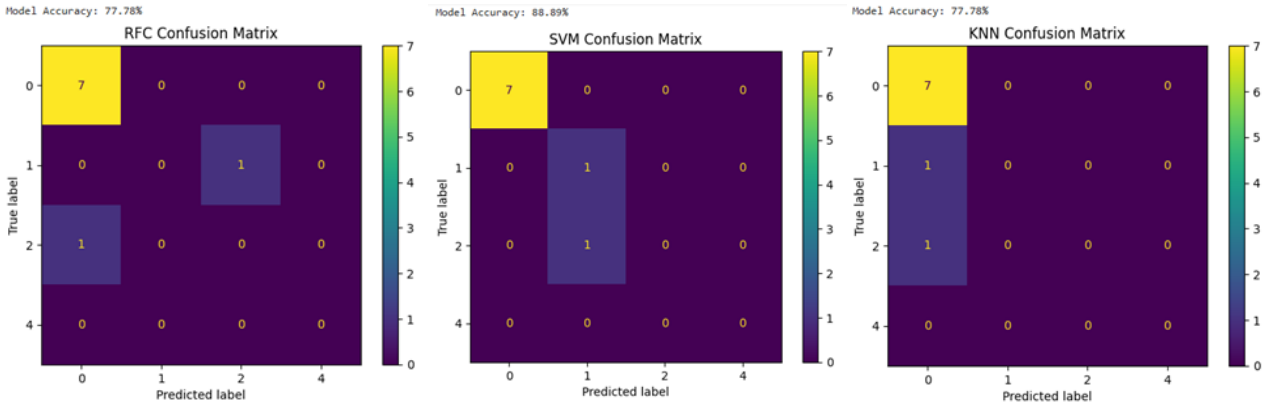


Figure 12 Confusion Matrices of Model Accuracy for RFC, SVM, and KNN

We decided to create a Support Vector Machine classifier using the Scikit Learn library to perform our classification task. As shown in Figure D1, we achieved 88.89% accuracy on multi-class classification (6 labels) whereas it was 77.78% for Random Forest and KNN. Moreover, SVM provides us with a compact model size that is ideal for Raspberry Pi deployment and its linear kernel avoids the computational overhead of non-linear alternatives. SVM finds the optimal hyperplane to separate the dataset into different classes. It

maximizes the margin between the closest data points that are in different classes from each other. These closest data points are the support vectors, and the margin is the distance from them to the hyperplane. [18] We used K-fold cross validation to evaluate the model's performance five times. The dataset was divided into five groups (folds), each making up 20% of data. One of the folds was treated as the test set, and the remaining folds were used as the training set. The test set was changed to a different fold after performing classification. The dataset we used had a majority class of 0, which also made up most of the dataset. The other classes were in much lower numbers, so our dataset had an imbalanced distribution of classes. To aid this, we used Stratified K-fold cross validation which divides the data to have an even distribution of classes for each fold. After performing classification on all 5 folds, the results were an average accuracy of 81.61%.

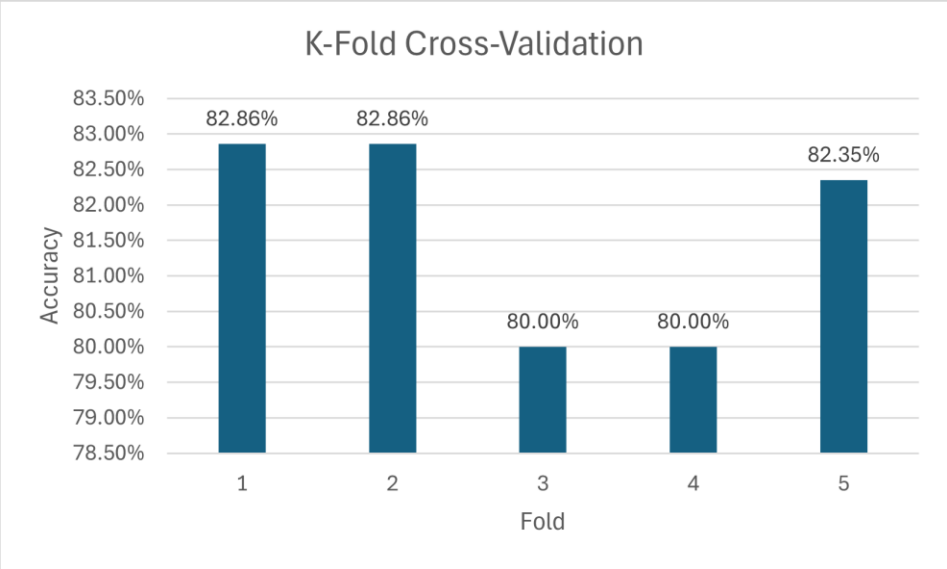


Figure 13 Accuracy for K-fold cross validation (K = 5)

5.3 Model Update and Generalization for Real-World Deployment

To ensure robust performance in real-world conditions, we propose a comprehensive, cloud-assisted model update pipeline that continuously refines our hazard differentiation model. Presently, our model is trained on data gathered from a controlled river-model apparatus in the Hydraulics Lab, which provides a stable and predictable environment. Recognizing that this controlled environment does not capture the full variability of lakes and rivers, we propose a plan to integrate real-world data into our training process through a three-phase deployment strategy.

In the first phase, we can deploy our current model on the Raspberry Pi at strategically selected locations near shorelines, where conditions are moderately variable. These test sites will be equipped with our sensor

package and, optionally, supplementary underwater cameras to capture additional contextual information such as water turbidity and object visibility. The onboard Raspberry Pi will record sensor outputs along with metadata like weather conditions and will periodically upload this data to our secure cloud storage. This real-world dataset will then serve as the basis for an iterative retraining process in the cloud.

Secondly, the cloud-based retraining pipeline, secured via our IoT core services, will re-evaluate the model's parameters using advanced validation techniques, including extended cross-validation and anomaly detection mechanisms. These mechanisms will be included in our future recommendations for this system to help the system to generalize into the real-world scenario. The additional context like camera footage would come as a great help for the system to provide accurate feedback to the system which can be done automatically or manually or in both ways. When the updated model achieves our target performance (for example, maintaining or exceeding a 91% testing accuracy under varied environmental conditions), it is automatically pushed to our AWS cloud. As our deployed edge devices (Raspberry Pi units) fetch the model file from AWS cloud after every recording session, this process would eventually ensure that our hazard differentiation system always operates with the most recent and robust version of the model on the Raspberry Pi units.

To further guarantee system reliability, we incorporate a real-time monitoring feedback loop on the Raspberry Pi in the final phase. The system continuously monitors the performance of the deployed model against new incoming data. If this performance falls below a predefined threshold, indicating a potential misfit with the current environmental conditions, the device will revert to a previously stable model version until a successful update is confirmed. This fallback mechanism is crucial for ensuring continuous, reliable hazard detection even in the face of fluctuating environmental variables.

This comprehensive approach, which combines systematic data collection under real-world conditions with continuous cloud-assisted retraining and secure deployment, ensures that our hazard differentiation model remains accurate and reliable. By incorporating additional sensors for context and robust feedback mechanisms, we address potential uncertainties and guarantee that the system can adapt and scale to the diverse challenges of actual lake and river environments.

6 Path Mapping

Path mapping is a feature we added to our system to find safe routes between points in a depth map. The feature works on an open body of water. Boaters can use our device to collect bathymetry data of an area. The collected samples are fed to some algorithms to provide a safe route between two points on a map. The collected samples provide the depth values and hazard levels, from our hazard differentiation feature, for each reading. We considered a safe route to be any route that avoids shallow and hazardous areas.

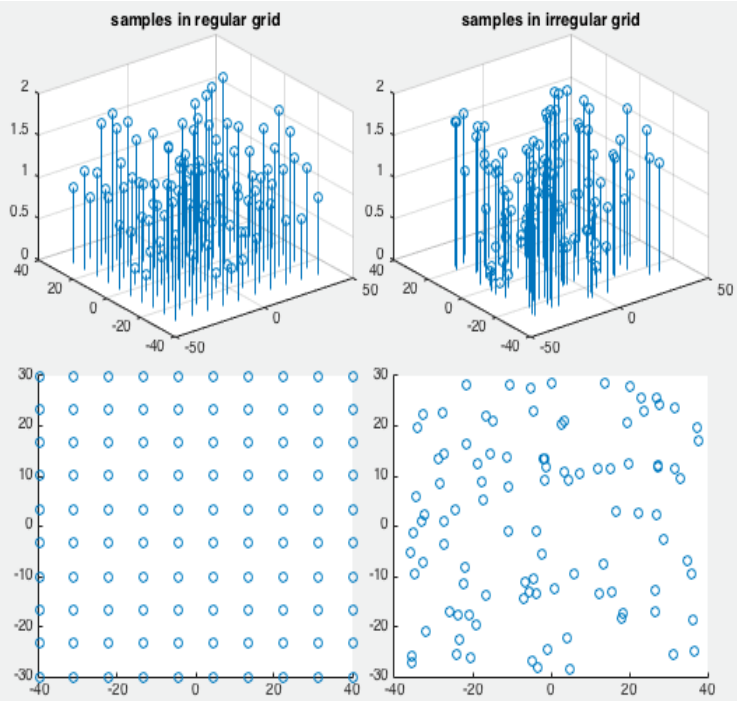


Figure 14 Difference between structured and unstructured data [19]

To develop this feature, we must create an algorithm that takes a list of samples with scattered data and produces a list of coordinates that connects the two points. Since the collected samples are unstructured (scattered data), the first method we considered was to use graph data structures and algorithms. This would mean finding the set of edges in a graph that connects nodes together to form a safe route. This method was not the best approach for several reasons.

1. Each sample is a node in the graph. However, it is not straightforward how the nodes in the graph should be connected.
2. The algorithm to determine a safe route could be expensive.
3. Most importantly, the route formed from a graph data structure will be jagged, since samples are scattered, which is not ideal for creating routes.

The second method for developing this feature converts unstructured data to structured data using interpolation. The structured data will form a matrix of depth values for evenly spaced latitudes and longitudes. The matrix will then be fed to a routing algorithm which produces a safe route between two points.

Although either method can work, we chose the second method because it is the better and safer choice for the reasons stated above. The details of how these algorithms work is described below.

6.1 Interpolation Algorithm

Radial Basis Function (RBF) interpolation is used to convert scattered data to gridded data. To interpolate new points, the method considers the distance between the new point (interpolant) and each sample point. The sample points that are closer to the interpolant weigh more, while the sample points that are further away weigh less. RBF interpolation uses a kernel function to determine the weight of each sample based on distance. The kernel function we chose was the gaussian kernel shown in the equation below. As distances increase, the gaussian kernel will decay to zero resulting in that point adding nothing to the sum.

$$K(\overrightarrow{x_1}, \overrightarrow{x_2}) = e^{-\frac{|\overrightarrow{x_1} - \overrightarrow{x_2}|^2}{\mu}}$$

$$f(\overrightarrow{x_j}) = \sum_{i=1}^{N} w_i \, K(\overrightarrow{x_i}, \overrightarrow{x_j}) = \sum_{i=1}^{N} w_i \, e^{-\frac{|\overrightarrow{x_i} - \overrightarrow{x_j}|^2}{\mu}}$$

The interpolation algorithm had one tunable parameter, μ, which controls the quality of the interpolation. When this parameter is set to a low value, it results in the interpolated surface passing closely through the sample points. This is because the gaussian function decays faster so points further away are not considered. When this parameter is set to a high value, it results in a smoother and flatter interpolated surface. This is because the gaussian function decays slower, so more points weigh in on the interpolant. Additionally, the domain of the coordinates needs to be scaled with this parameter. To scale the domain, we used min-max feature scaling on the list of samples. After that, the optimal value of the parameter was found to be μ=0.03 using gradient descent, as shown in figure 15. The tunable parameter was set to that constant. We also added another constant that trims the sample domain. This was done because interpolating near the edges is not reliable. More details on the optimization of this parameter are shown in appendix D.

The algorithm takes the list of scattered data collected during a recording session. The data is normalized using min-max feature scaling. Next, the weights are found for each sample point using matrix

multiplication. Finally, using the weights, the interpolants are found once again using matrix multiplication. The algorithm outputs interpolated depth values that are evenly spaced along latitudes and longitudes. More details of how values are interpolated are shown in appendix C.

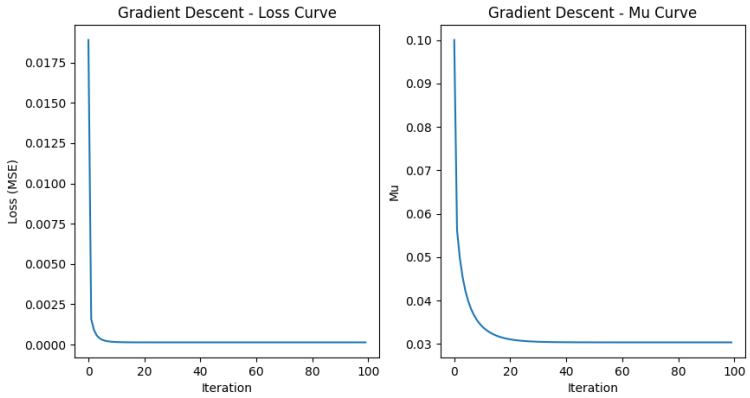


Figure 15 Gradient descent curve of interpolated surface

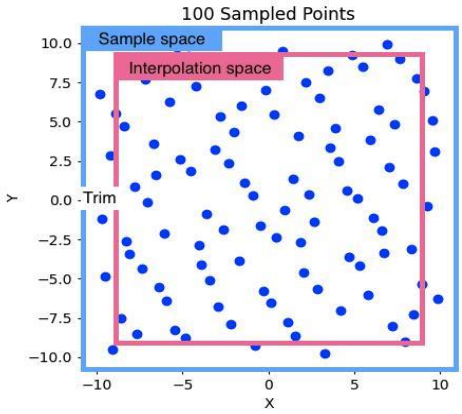


Figure 16 Trim applied to sample space

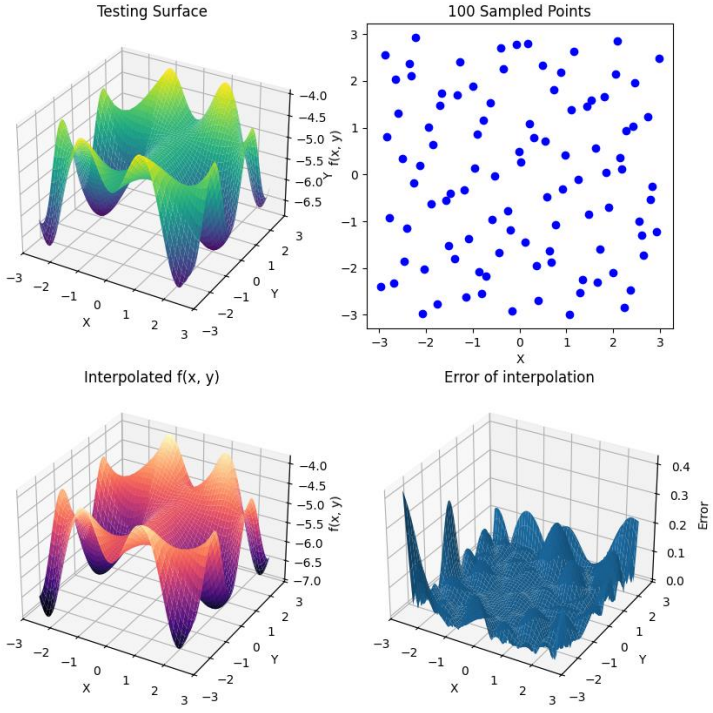


Figure 17 Interpolation of a testing surface with 100 random samples

6.1.1 Integrating Hazards

The interpolation algorithm converts an unstructured list of depth values into a structured matrix of depth values. However, a safe route is found by evaluating depth and hazard level, determined from the hazard differentiation feature. The hazard level needs to be incorporated into the structured matrix of depth values. To do this, we went through the list of scattered data and for each hazard, a block of depths around the hazardous sample was turned to zero in the matrix. For a low hazard level, a small block of depths is

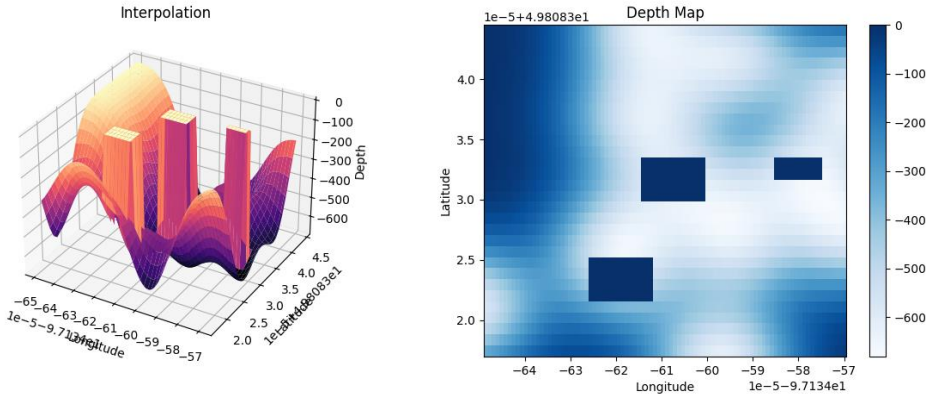


Figure 18 Hazards integrated into interpolated surface

changed. For a high hazard level, a large block of depths is changed. This effectively incorporates hazards into the converted data, since a safe route should avoid shallow areas.

6.2 Routing Algorithm

The structured matrix is fed into the routing algorithm. Along with that, the start and end coordinates and the minimum depth are required. Since the data is structured, developing the routing algorithm became easier. The idea behind the algorithm is to make incremental progress towards the end. The route starts at the start coordinate in the matrix. One step, in the matrix, is taken at a time and added to the route. Upon reaching a shallow area, the route traces around the unsafe area until it can once again move towards the end. However, when tracing around an unsafe area, the algorithm does not know what the best option for tracing is. It can be traced clockwise (CW) or counterclockwise (CCW). To address this issue, the algorithm uses recursion. One recursive branch will trace CW, while the other branch will trace CCW. This means each time an unsafe area is encountered, the algorithm splits into two branches. When the algorithm reaches the end coordinates, the branch returns to its route list. Each pair of branches in the recursion stack are compared, and the shorter route list is returned.

One thing to add is that, to make it simpler for the algorithm to choose the direction to move in, we created a compass data structure. It was created using a circular doubly linked list with CW or CCW rotation. The compass and a copy of the most recent route list is passed to each branch.

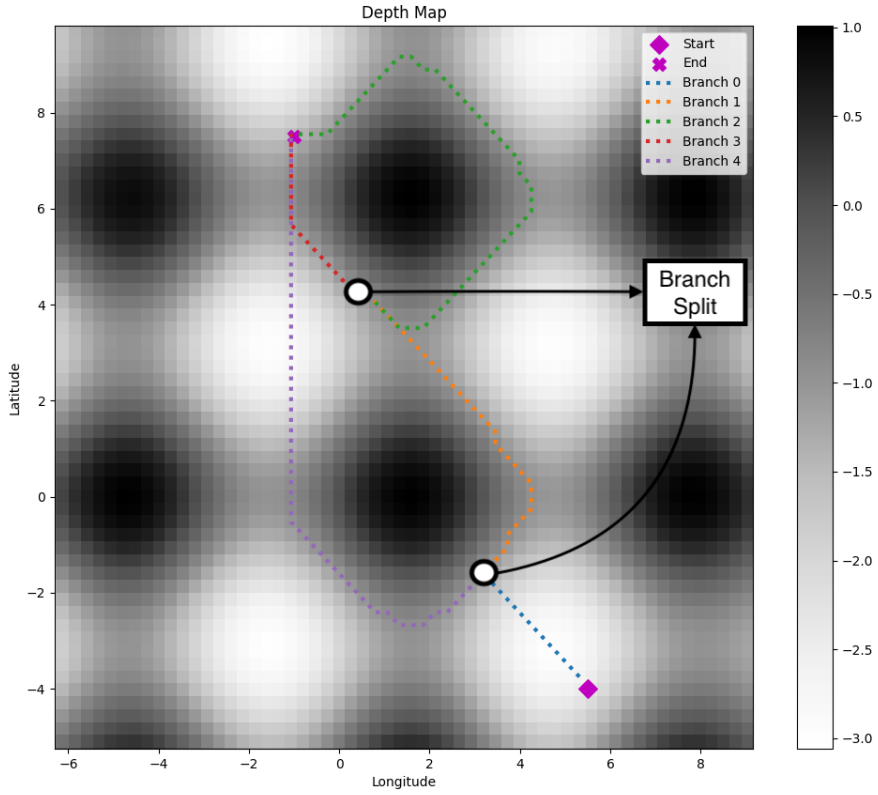


Figure 19 Routing algorithm area tracing and branching

An example of the working algorithm is shown in figure 19. The algorithm starts routing as shown with the blue route. It collides with an unsafe area and splits into two branches. The first branch, shown with the orange route, starts tracing with CCW rotation and collides with another unsafe area. It splits in two branches again. The two branches reach the end coordinate and return their route. The algorithm returns to the first splitting point and starts tracing with CW rotation, shown with the purple route. The purple route reaches the end and returns its route. The shortest route will be chosen which will be the combination of the blue, orange and red routes.

To summarize how the algorithm works, it makes progress towards the end in a while-loop. There are three sections in this loop. The first section moves towards the end location one step at a time. The second section makes two recursive calls upon encountering an unsafe area. The branch that made the call stops making progress and waits for a route list to be returned. The last section traces around the unsafe area based on the

rotation of the compass. If it can move towards the end safely, it escapes, and the algorithm moves back to section one. The algorithm ends once all branches return, or the maximum number of iterations is reached. The number of iterations is capped to the size of the matrix. The shortest route list is returned as the safe route. If no safe route is found or the maximum number of iterations is reached, an empty list is returned.

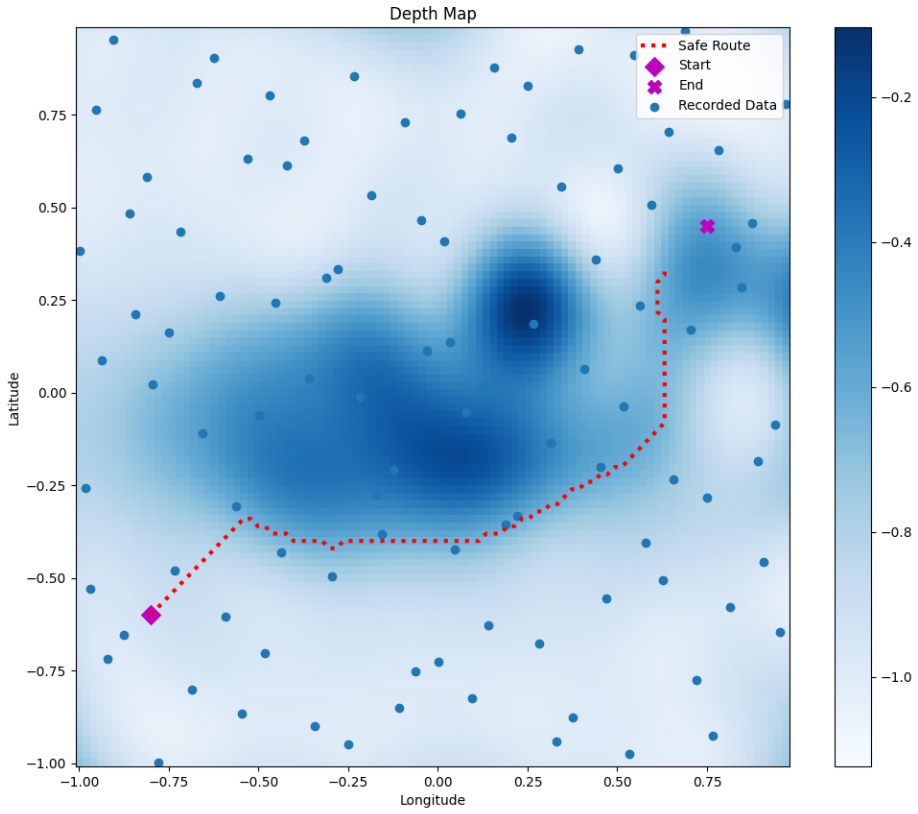


Figure 20 Shortest route provided from start to end

7 Application development

AWS Amplify [20] simplifies backend integration for Flutter apps, streamlining authentication and cloud services setup. Using the Amplify CLI, the project was initialized, and AWS services like Cognito were configured for authentication. The Amplify Authenticator UI was leveraged to implement user authentication seamlessly. The app setup included installing Flutter, configuring iOS and Android environments, and verifying with flutter doctor. Amplify's significance lies in providing a scalable and efficient way to integrate

AWS services without extensive backend coding. Amplify Studio was enabled to provide a visual interface for managing backend resources, making development more efficient [21].

7.1 UI State Management and Providers Used

UI state management in Flutter follows a declarative approach, where the UI is rebuilt from scratch to reflect the current app state instead of modifying elements directly. Flutter distinguishes between *ephemeral state*, which is temporary and widget-specific, and *app state*, which persists across the app or multiple widgets. Ephemeral state is managed with Stateful Widget and setState(), while app state requires dedicated state management solutions like Provider, which enables efficient and scalable state handling. By using ChangeNotifier with Provider, Flutter apps ensure that UI updates dynamically when state changes, improving performance and maintainability [22].

In this app, *ChangeNotifierProvider* is used to supply state objects to the widget tree, ensuring efficient state updates. notifyListeners () is called whenever data changes, triggering UI updates. The LocationData provider manages real-time location updates, storing a list of LocationInfo objects that track latitude, longitude, accuracy, and other metadata. It also maintains a heartbeat value to monitor connectivity with AWS IoT Core. The *LocationMapProvider* handles multiple location maps and routes stored in the cloud, allowing dynamic retrieval and visualization of mapped paths. By leveraging these providers, the app efficiently updates UI components based on incoming real-time data.

7.2 Wi-Fi and BLE Connection Logic and UI

The home screen features two primary connection buttons: one for connecting to AWS IoT and another for connecting to the Raspberry Pi (RPi) via Bluetooth. The "Connect" button initiates the AWS IoT connection using fleet provisioning by claim certificates, and upon successful connection, the "App Connected" status updates to indicate a successful link. Additionally, the "Pi Online" status dynamically updates based on the heartbeat values received from the RPi through MQTT, signifying its connectivity. Similarly, the "Connect Bluetooth" button scans for BLE devices advertising a specific service, attempts to establish a connection, and subscribes to real-time data updates. Once connected, the "App Bluetooth Connected" status updates, confirming the app's connection, while the "Pi Bluetooth Connected" status reflects whether the RPi's Bluetooth service is active. These statuses provide real-time feedback, ensuring seamless monitoring of both cloud and Bluetooth connectivity.

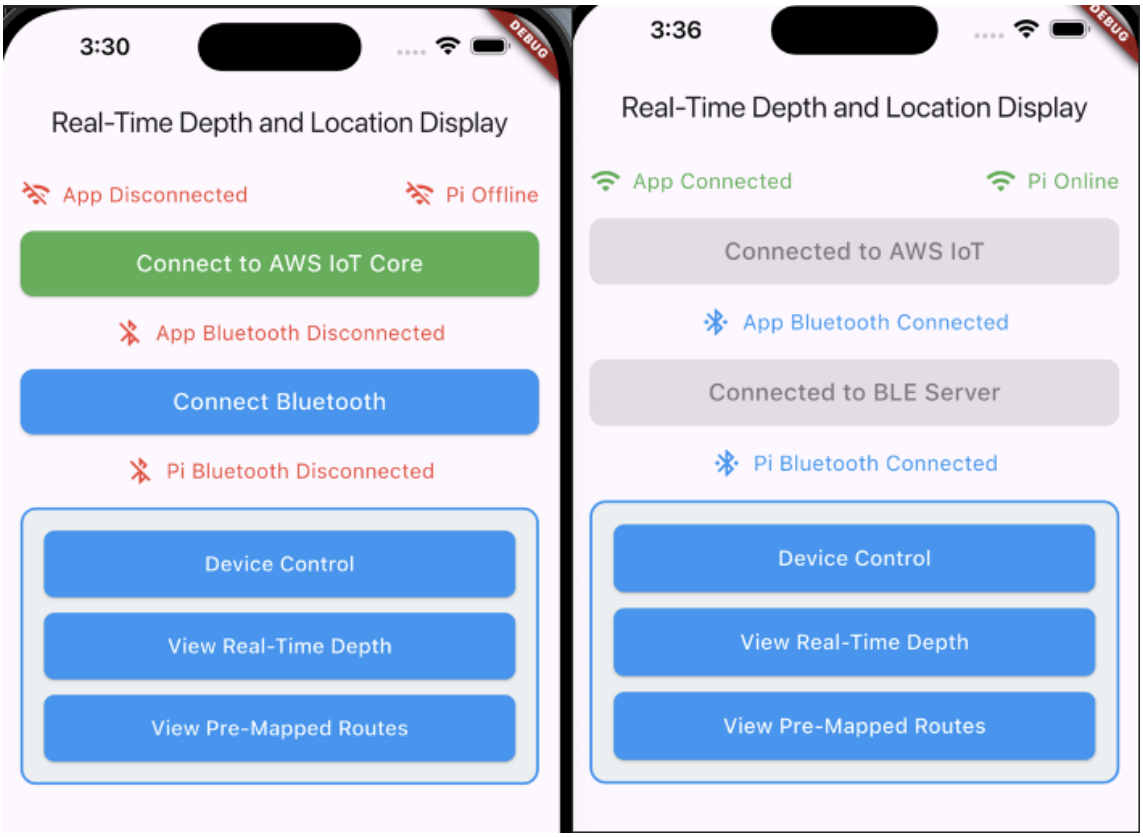


Figure 21 App Home Screen: Wi-Fi and BLE Connect Buttons and Buttons to other screens

The home screen also includes navigation buttons for accessing key functionalities: "Device Control", "View Real-Time Depth", and "View Pre-Mapped Routes". The "Device Control" button navigates to a screen for managing connected devices, while "View Real-Time Depth" opens a live data visualization of depth and location updates received via Bluetooth. The "View Pre-Mapped Routes" button allows users to access and display pre-mapped location data stored in the cloud, providing a visual representation of mapped routes.

7.3 Receive and Display Real Time Depth and Location Info

The real-time depth and location monitoring system receives data through two communication channels: *Wi-Fi (MQTT)* and *Bluetooth Low Energy (BLE).* These channels provide continuous updates on depth readings and GPS coordinates, which are processed and displayed dynamically on a Flutter Map with markers and polylines.

Receiving Real-Time Data

For BLE communication, the app subscribes to a specific characteristic of the connected device. When new data is received, the subscription callback decodes the incoming *UTF-8 payload* and passes it to the handleReadValuesResponse function, which extracts depth, confidence, latitude, and longitude information.

Similarly, for WiFi communication using MQTT, the app listens to subscribed topics such as "g9capstone/readValues", "g9capstone/piHeartbeat", and extracts relevant sensor readings. The parsed data is then processed to update the provider, ensuring that real-time updates are reflected in the UI.

Updating the Provider with New Data

The *LocationData* provider manages the list of received locations and triggers UI updates whenever new data is added. When real-time data is received via MQTT (WiFi) or BLE, it is processed and stored in _locationList, and the notifyListeners() method ensures that the UI updates dynamically.

Whenever a new location update is received, the provider:

1. Adds the new location to the _locationList
2. Notifies the UI to rebuild and display updated markers and polylines
3. Ensures smooth, real-time visualization of depth and location data

By leveraging providers for data management and Flutter Map for visualization, the system ensures an efficient and responsive real-time mapping interface, helping users track depth data and movement patterns with ease.

Additionally, the provider manages *heartbeat values* that monitor the connection status. The app continuously updates the _heartbeatValue, which can trigger alerts or UI changes if connection issues arise.

Updating and Displaying Real-Time Data on the Map

The Flutter Map component dynamically updates to visualize received depth and location data in real-time. The RealTimeDepthScreen can be accessed by used by clicking the View Real Time Depth Button on Home Screen. This UI utilizes a *Timer.periodic() function and providers*, which triggers UI rebuilds when providers are updated with new data on receiving messages from RPi4 via Wi-Fi or BLE ensuring that the latest data is always displayed.

Displaying Markers on Flutter Map

Each recorded location is represented by a marker, displaying:

- Depth readings (in mm)
- Confidence values (in %)
- Timestamps of when the data was received

The *MarkerLayer* dynamically iterates through the location provider's list, creating markers with a red pin icon for each recorded data point. When a user taps on a marker, a popup dialog appears, showing detailed information such as timestamp, depth, and confidence percentage. Additionally, a Tooltip provides a quick preview of the depth reading directly on the map without requiring a click.

Displaying Polylines for Path Tracking

To visualize the movement path over time, the app also connects consecutive location points using polylines. The *PolylineLayer* fetches all location points from the provider and renders a blue path, indicating the traveled route in real time.

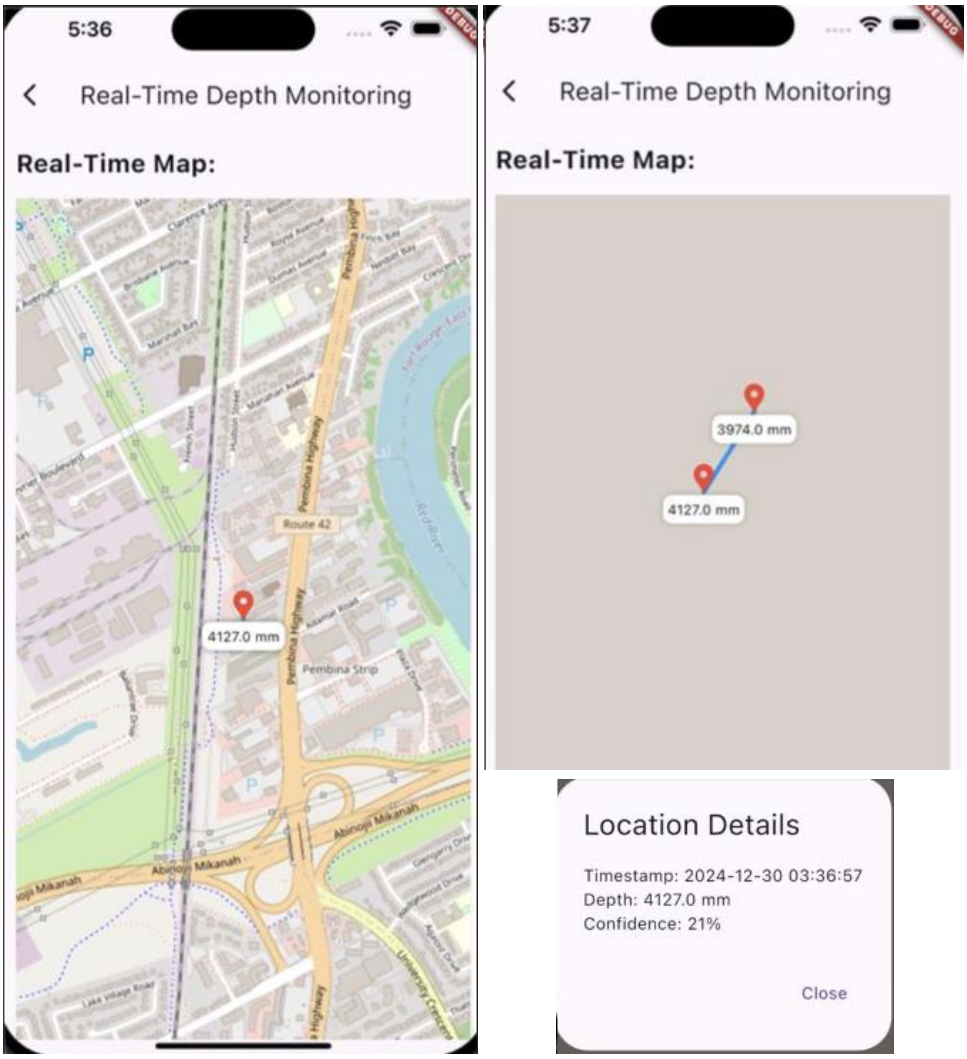


Figure 22 Real Time Depth and Location Monitoring Screen

7.4 Fetch Pre-Mapped Routes from Cloud and Display List

The pre-mapped list of data is retrieved from AWS S3 cloud storage whenever the user refreshes the screen. The system first clears any existing map data before fetching the latest files stored in the cloud. These files contain location-based data with key details such as timestamp, distance, confidence, latitude, longitude, accuracy, and prediction values.

Once the data is retrieved, it is carefully processed. If the dataset includes a safe route, it is extracted separately and stored for reference. If no safe route is found, a default placeholder is assigned. The processed location data, along with the corresponding map name, is then stored in a state management provider, ensuring that the application maintains an up-to-date record of all available maps.

After updating the provider, the UI is automatically refreshed to reflect the latest data. The updated list of maps is displayed dynamically, allowing users to select and explore different pre-mapped routes. Each entry provides access to a detailed view with a heatmap visualization, showing the recorded data points along with the safest suggested route if available. Additionally, users have the option to download maps for offline access, storing them locally for future use.

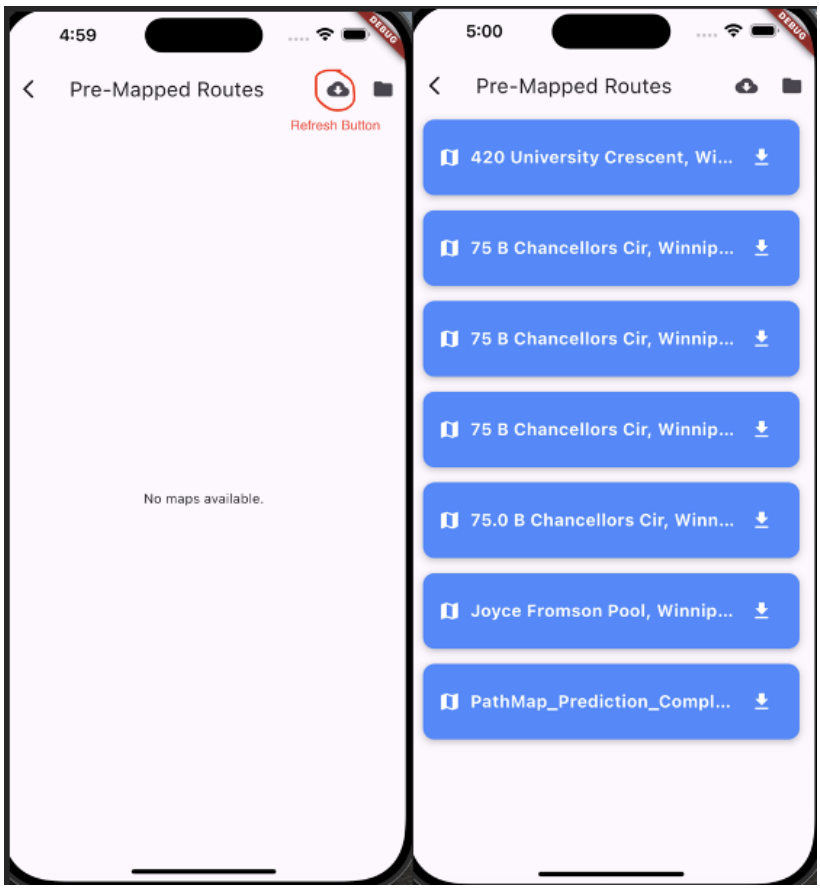


Figure 23 Pre-Mapped List fetched and display list

7.5 Display Pre-Mapped Route data

The heatmap screen visualizes pre-mapped routes using markers, polygons, and polylines to provide a clear representation of recorded location data and safety insights. Each location point from the dataset is displayed as a marker, placed using its latitude and longitude coordinates. These markers also include small labels

indicating distance values, ensuring users have contextual information about each recorded position. Additionally, small patches are added around the markers with a semi-transparent fill, enhancing visibility and differentiation between various data points. To define the overall mapped boundary, the system uses the computeConvexHull function, which generates a polygonal boundary encompassing all relevant route points. This ensures that users can see the extent of the tracked locations as the boundary. Each marker is assigned a color based on its prediction value, indicating different hazard levels as determined by the hazard differentiation algorithm. Green markers signify low-risk areas, yellow markers indicate moderate risk, orange markers show increasing danger, and red markers highlight high-risk locations. This dynamic color-coding allows users to quickly assess potential hazards along a given path. For added safety insights, a safe route is extracted from the dataset if available. This route is displayed as a polyline overlay, ensuring it stands out from the rest of the data points. The safe route is represented using cylindrical markers in blue, guiding users toward the most secure pathway within the mapped environment.

Figure 24 Pre-Mapped Route Visualization – Hazard and Safe Route Analysis

The visualization seamlessly integrates hazard analysis with route optimization for pre-mapped routes, helping users navigate efficiently while avoiding potential risks.

7.6 Notifications – Hazard Alerts

The flutter_local_notifications package [23] is used in the app to handle alert notifications effectively. First, it initializes the notification settings for both Android and iOS, ensuring platform-specific configurations such as categories and actions for iOS and high-priority settings for Android. The app requests necessary permissions for sending notifications, including alert, badge, and sound permissions for iOS/macOS and runtime notification permissions for Android. When the app is launched, it checks whether notifications are enabled and prompts the user to allow them if needed. Notifications are then handled in both foreground and background states, ensuring seamless interaction with users when alerts are received. Additionally, the app defines custom notification details, such as priority and importance levels, to ensure timely and prominent alerts for critical warnings.

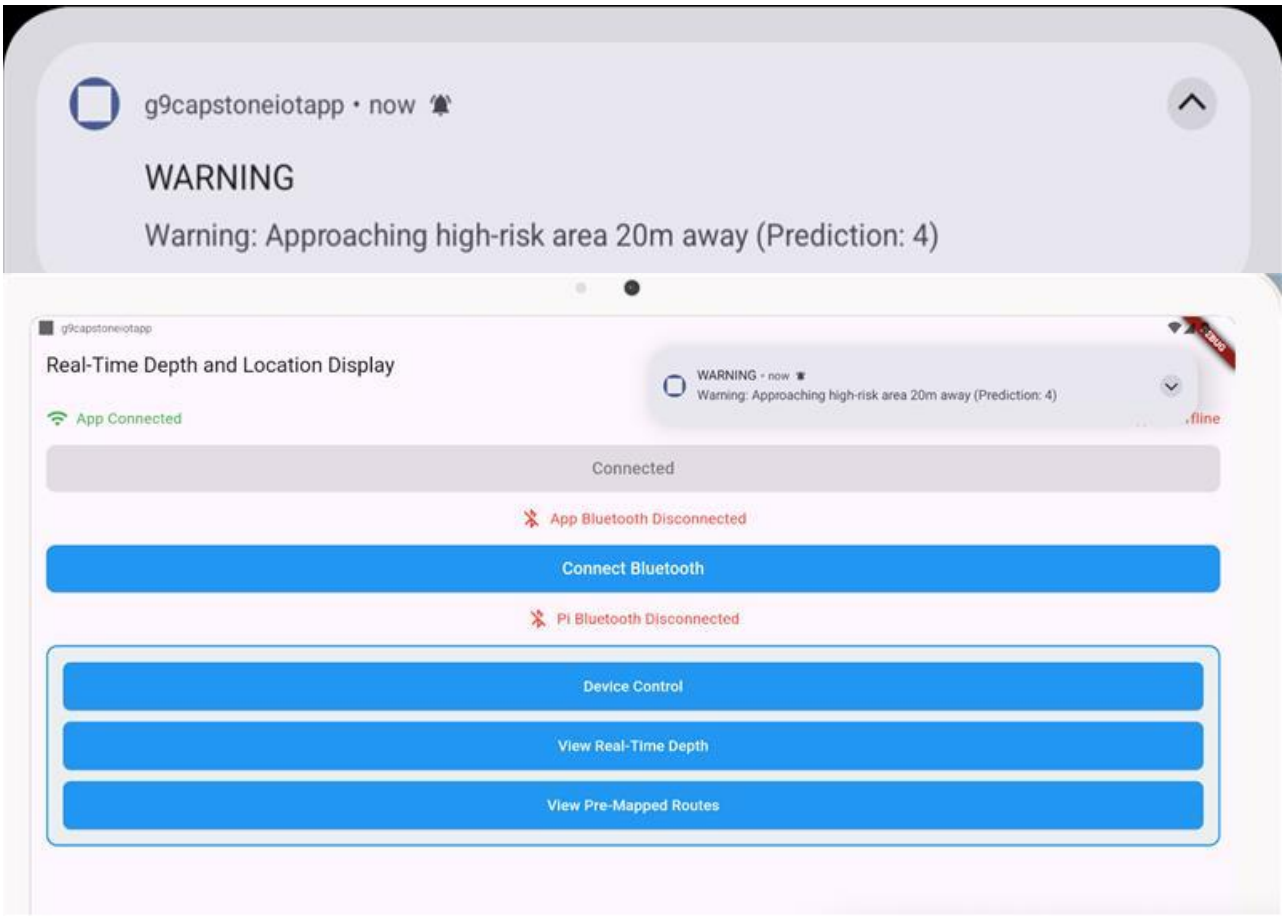


Figure 25 Hazard Alerts Display for pre-mapped routes

For hazard alerts related to user-selected routes, the app utilizes real-time location updates received from the Raspberry Pi 4 (RPi4). When a user selects a map and enables notifications by clicking the notification icon, the app saves that map data to compare with real time location data. The app compares the real-time coordinates received from RPi with pre-mapped predicted locations stored in the selected map, filtering points with a prediction value greater than 1. The distance between the real time position and these high-risk

points is continuously calculated. If the boat comes within 20 meters of any hazardous location, an alert notification is triggered, displaying a warning message with the corresponding prediction value. Users receive immediate alerts about potential dangers on their route, allowing them to take necessary precautions in time.

8 Software Integration on MCU, States and Data Flow

The combined script running on the Raspberry Pi 4 (RPi4) is designed as a multi-threaded, state-driven system that efficiently manages depth monitoring, data synchronization, hazard prediction, and routing algorithms. The system consists of multiple threads working in parallel to ensure real-time operation, uninterrupted communication, and seamless data processing.

8.1 Multi-threading Structure

1. **Wi-Fi Thread:** This thread is responsible for handling network connectivity, ensuring that the device connects to a Wi-Fi network whenever available. It checks RF-kill status, retries connections when necessary, and sets a Boolean flag indicating Wi-Fi status. If Wi-Fi is available, it facilitates MQTT communication with AWS IoT Core.
2. **State Machine Thread:** The core thread that manages different states, including reading depth data from the Ping2 sensor, retrieving GPS/geolocation data (via Wi-Fi-based geolocation indoors or direct GPS readings outdoors), and synchronizing this information into a structured JSON message. The state machine reacts to commands from the mobile app—*START*, *STOP*, and *UPLOAD*—which control the data collection process.
3. **Synchronizer Thread:** This thread is responsible for handling data transmission and local storage. It fetches JSON messages from a queue and either transmits them over MQTT (if Wi-Fi is available) or via Bluetooth (if Wi-Fi is unavailable). It also logs data locally in a list and periodically writes it to a file in the /home/g9pi/Downloads/readyToUpload directory.
4. **Heartbeat Thread:** This thread sends periodic (every 60 seconds) updates to AWS IoT Core, ensuring that the system remains connected to AWS and that mobile app users are aware of the device's status.

5. **Bluetooth Handling Thread (BLE Socket Server):** A Unix socket server that works with qtble_server to establish Bluetooth Low Energy (BLE) communication when Wi-Fi is unavailable. It listens for device control commands (START, STOP, UPLOAD) and transmits real-time depth monitoring data to the mobile app using a subscribed characteristic ID.

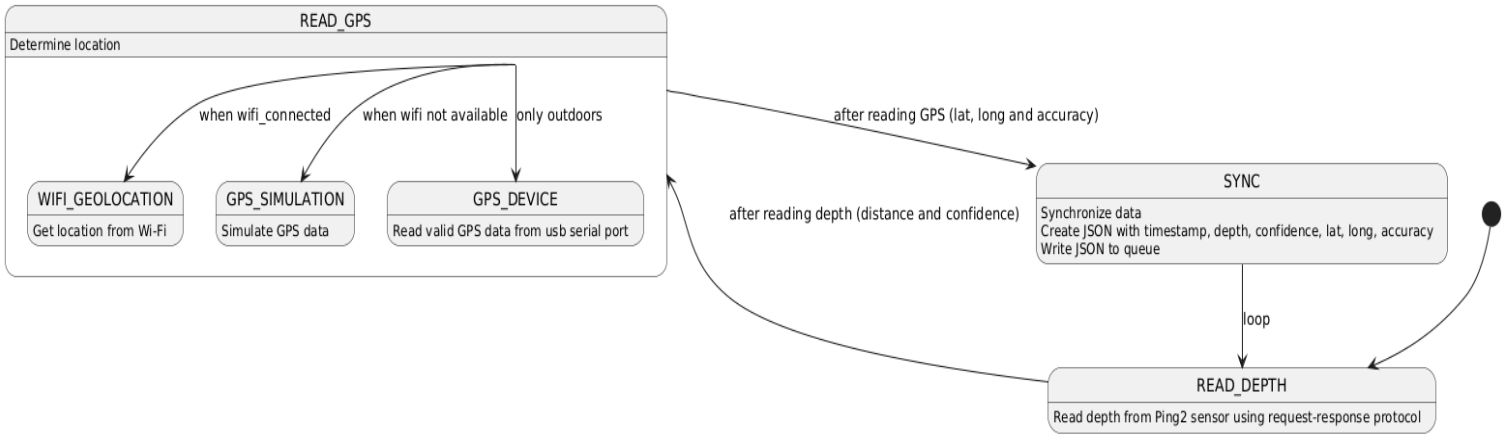


Figure 26 State Machine for collecting and syncing data

8.2 State Management and Data Flow

1. **IDLE State:**
    - The system remains in the IDLE state when it is not actively collecting data.
    - It waits for a *START* command from the mobile app via Bluetooth (BLE).
    - When the *START* command is received, the system moves to the *DATA COLLECTION state*.

2. **DATA COLLECTION State:**
    - The system starts collecting data by starting the state machine thread:
        - *READ DEPTH***:** Retrieves depth readings from the Ping2 sensor.
        - *READ GPS***:** Retrieves GPS coordinates using Wi-Fi-based geolocation (indoors) or direct GPS readings (outdoors).
        - *SYNC***:** Creates JSON messages containing timestamp, distance, confidence, latitude, longitude and accuracy.
    - The depth and GPS data are stored in a buffer and synchronizer thread checks the queue continuously and transmit data real time to the mobile app:
        - **Via Wi-Fi:** If connected, data is published using MQTT to AWS IoT Core.
        - **Via Bluetooth:** If Wi-Fi is unavailable, data is sent to the mobile app over BLE.

- The system remains in this state until a **STOP command** is received

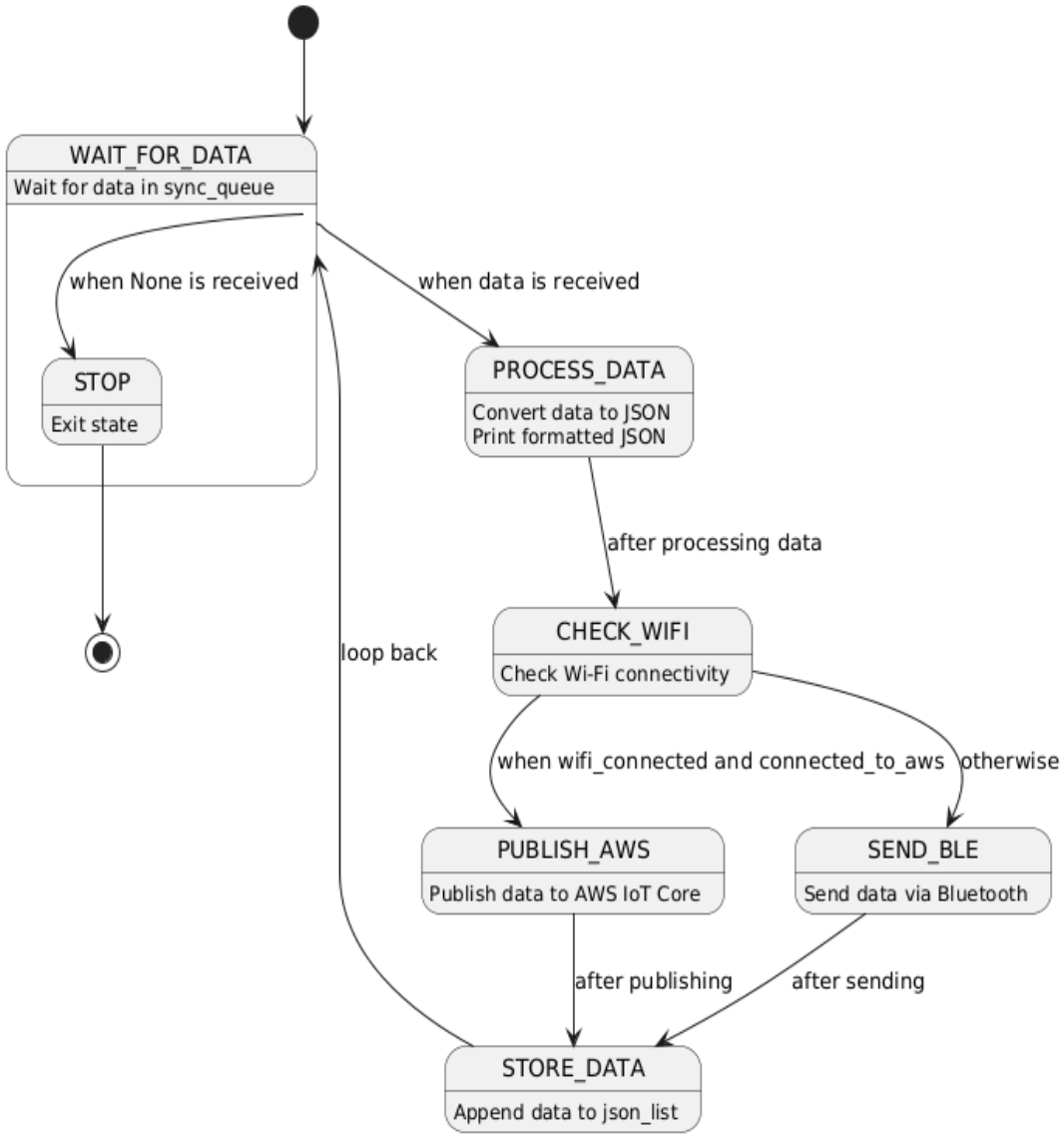


Figure 27 Flow of the Synchronizer thread to get data and communicate to App

3. **DATA PROCESSING State:**
   - When the *STOP command* is issued, the buffered data is saved locally in the /home/g9pi/Downloads/readyToUpload directory.
   - The pathMapping_HazardPrediction.py script runs in parallel to process this stored data.
   - The *Hazard Prediction Process* includes:
     - Downloading the latest hazard differentiation model from AWS S3.
     - Loading the model using joblib and applying it to predict hazard levels.

- The process_file() function validates and processes each JSON file, ensuring that unprocessed files (without PathMap_Prediction_Completed in their name) are analyzed.
- Hazard levels are predicted using predict_entry(), and the results are appended to the JSON file.

- The *Safe Route Calculation Process* includes:
  - Normalizing data, interpolating depth values, and generating a grid of coordinates.
  - Running the findRoute() algorithm to determine the safest path.
  - The computed route is added to the JSON data, and the updated file is saved with a PathMap_Prediction_Completed prefix.
  - The original unprocessed file is deleted to avoid redundant processing.

4. **UPLOAD State:**
   - When the *UPLOAD command* is triggered, all processed files in the /home/g9pi/Downloads/readyToUpload directory are uploaded to AWS S3.
   - The mobile app can then refresh to fetch the latest files from the cloud.
   - The app displays:
     - Mapped zones with color-coded markers.
     - Risk areas based on hazard predictions.
     - The computed safe route.
   - The system returns to the *IDLE state*, awaiting the next command.

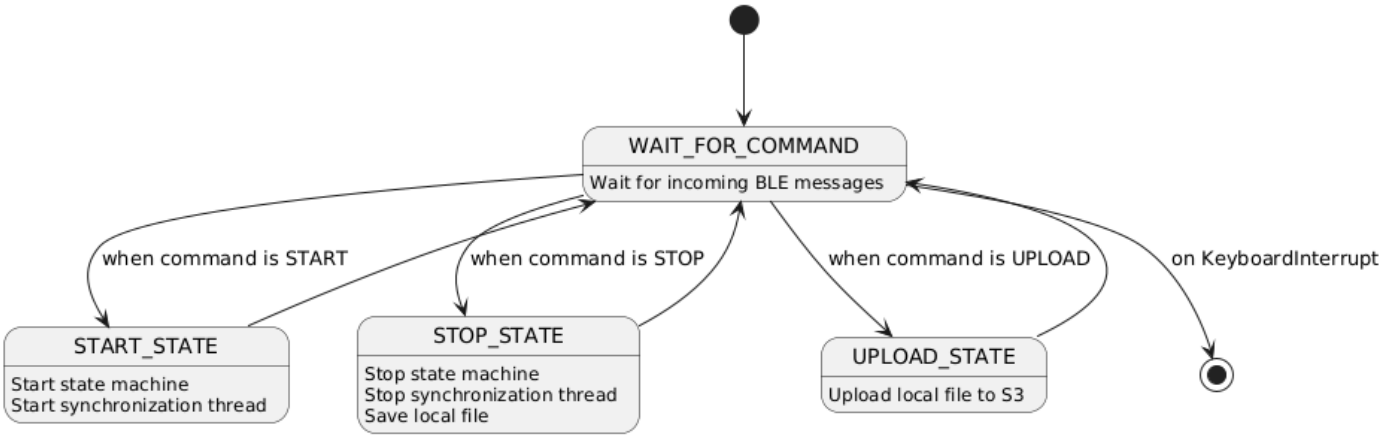


Figure 28 Receive incoming device control command and handle actions

8.3 Wi-Fi and Bluetooth Handling

The system seamlessly switches between Wi-Fi and Bluetooth based on network availability. When Wi-Fi is connected, data is transmitted via MQTT to AWS IoT Core. If Wi-Fi is unavailable, the BLE server activates, allowing the mobile app to receive real-time depth monitoring data over Bluetooth. The Unix socket facilitates communication between the main thread and qtble_server, enabling command exchange via BLE.

8.4 Local Storage and Cloud Uploads

All depth and GPS data collected during monitoring are saved locally in the /home/g9pi/Downloads/readyToUpload directory when STOP command is received. When the UPLOAD command is issued, these files are sent to AWS S3, where the mobile app can retrieve them for visualization. If Wi-Fi is unavailable, files remain on the RPi4 until connectivity is restored.

8.5 Running Algorithms for Hazard Prediction and Path Mapping

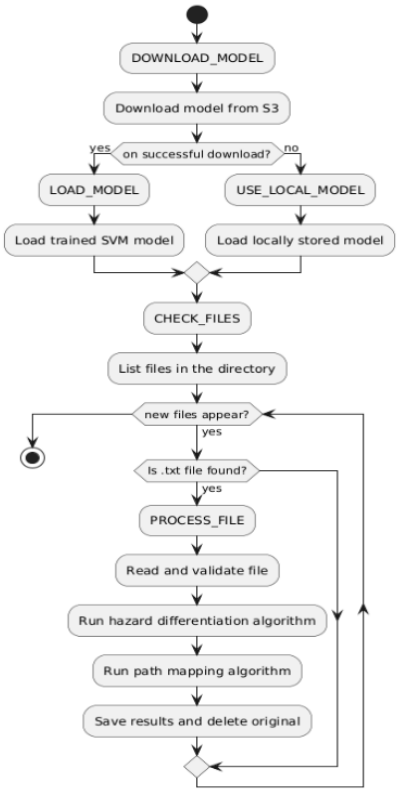


Figure 29 Processing Local File using Hazard Diff and Path Mapping Alg

- The *Path Mapping and Hazard Prediction script* continuously monitors the /home/g9pi/Downloads/readyToUpload directory for new JSON files containing depth and GPS data. When a new file is detected:
    - It loads the latest hazard differentiation model from AWS S3 using joblib and applies it to predict hazard levels.
    - The hazard differentiation predict function evaluates each data entry and classifies hazards based on sensor depth readings and assigns a prediction level for each entry.
    - The Interpolation and *Routing Algorithm* determines a safe route from start to endpoint while avoiding hazardous areas using the predictions resulted from hazard differentiation prediction.
    - The processed JSON file is saved with the *PathMap_Prediction_Completed* prefix, and the original file is deleted to prevent reprocessing.

8.6 Mobile App Integration

- Users can receive real-time data via Wi-Fi or BLE.
- The app allows sending START, STOP, and UPLOAD commands via BLE.
- Pre-mapped files can be downloaded from the cloud and displayed with hazard zones and safe routes.
- Notifications alert users when they are approaching a hazardous area within 20m.

8.7 Running the Scripts on Rpi4

Added the following files on *rc.local file* to run the scripts automatically as soon as RPi4 is turned on. This file can be accessed on RPi4 using command sudo nano /etc/rc.local:

- RPi4_Combined Script**:** handles wifi connection, bluetooth connection, listens for device control command from user by ble to START reading sensor, gps and sync, STOP reading and save the file locally, UPLOAD to upload the file to AWS S3 and handles real time depth monitoring via MQTT over Wi-Fi (when available) or BLE.
- BLE Server: starts the bluetooth server, advertises services, handles connections, listens for commands and handle reconnections.
- Algorithms Running Script: start the script to run the hazard differentiation algorithm to update with prediction values and path Mapping algorithm to suggest a safe route and write the results to file with update filename.

This integrated system ensures continuous depth monitoring, real-time hazard prediction, and seamless communication via multi-threading. It dynamically switches between Wi-Fi and Bluetooth, efficiently logs and synchronizes data, and processes hazard mapping to provide users with safe navigation routes. The combination of MQTT for remote monitoring, BLE for local communication, and cloud-based analytics makes this an advanced, reliable, and robust depth monitoring and hazard prediction system. This integrated system of data collection, hazard prediction, and cloud synchronization provides real-time, efficient hazard mapping and navigation support, guaranteeing seamless operation and continuous communication throughout the entire process. Users can receive the real time data both by Wi-Fi (for local and remote) and BLE (local) and view marker on App, control the system by sending Commands from App via BLE, get the pre-mapped files from cloud, view the map with color coded markers, safe route, download the map for later use, urn on notifications (hazard alerts while using the map) to get alert for approaching hazard 20m before.

9 Systems Integration, Testing and Validation

Finally, we integrated a USB-UART converter to enable seamless UART data transmission from the PING2 sensor. To enhance system functionality, we implemented Wi-Fi-based geolocation and GPS simulation, allowing for accurate indoor system testing. Additionally, data synchronization was established on our primary microcontroller, the Raspberry Pi 4 (RPi4), ensuring smooth data handling across various hardware components. As part of the final hardware integration, all devices are powered by a 5V power bank, which provides stable voltage to maintain consistent operation.

The Raspberry Pi 4 (RPi4) runs a multi-threaded, state-driven system for real-time depth monitoring, hazard prediction, and route calculation. It efficiently manages Wi-Fi connectivity, Bluetooth communication, data synchronization, and cloud integration through multiple parallel threads. The Wi-Fi thread ensures network connectivity and facilitates MQTT-based communication with AWS IoT Core, while the Bluetooth Handling Thread enables BLE communication when Wi-Fi is unavailable. The State Machine Thread controls core functions, including reading depth data from the Ping2 sensor, obtaining GPS coordinates, and synchronizing information into structured JSON messages. The Synchronizer Thread handles real-time data transmission over MQTT or BLE and logs data locally for further processing, while a Heartbeat Thread sends periodic updates to AWS IoT Core to maintain system visibility.

The system transitions through different states which are sent by the user on App to RPi4 using BLE: IDLE, where it waits for a START command from the mobile app; DATA COLLECTION, where it gathers depth and GPS data and transmits it in real time via MQTT or BLE; DATA PROCESSING on receiving STOP

command, where it saved the data mapped in a local file which triggers to run hazard prediction and safe route mapping using a machine learning model downloaded from AWS S3, applying hazard differentiation and routing algorithms before storing processed data with a "PathMap_Prediction_Completed" prefix; and UPLOAD, where processed files are transferred to AWS S3 for retrieval by the mobile app, which visualizes hazard zones and computed safe routes.

AWS Cloud Services ensure seamless data management and retrieval, with AWS IoT Core serving as the primary communication hub for real-time data exchange, AWS S3 securely storing mapped route data, and AWS Amplify optimizing data access and user interaction. Python scripts running on the RPi4 process depth and GPS data, synchronizing it with AWS S3 via API calls, while a Flutter-based mobile app fetches and visualizes the data through Amplify API calls, offering users an intuitive interface for monitoring depth readings, viewing hazard zones, and navigating safe routes in real time.

For real-time communication, the system leverages MQTT over Wi-Fi, with AWS IoT Core acting as the MQTT broker to ensure reliable, low-latency communication between the RPi4 and the mobile app. If Wi-Fi is unavailable, the system dynamically switches to *Bluetooth Low Energy (BLE)*, enabling continuous data transmission to the mobile app. This intelligent network switching ensures uninterrupted operation, allowing users to receive real-time hazard updates and depth monitoring data regardless of connectivity conditions. The integration of AWS Cloud Services enhances efficiency, enabling seamless communication, scalable data management, and real-time hazard mapping, making this a robust and reliable depth monitoring and hazard prediction system.

9.1 Testing Methodology

In this section, we provided a detailed explanation of the testing methodology employed for various purposes, including functionality, stability, and performance evaluations. We outline the specific objectives of each test, the step-by-step approach we followed, and the tools and techniques used to ensure accurate results. Additionally, we discuss the rationale behind our testing procedures and how they contributed to refining the overall design and performance of our system.

9.1.1 Swimming Pool Testing

The primary objective of testing in the university swimming pool was to evaluate the accuracy of the ultrasonic sensor in measuring depth when integrated with the RPi4. All the hardware components, including the RPi4, Ping2 sensor, Arduino Uno R3, and power bank—were fastened together using zip ties to ensure a

secure setup. These components were placed inside a box container mounted on top of the prototype boat. The results of this test are discussed in detail in the system validation section.

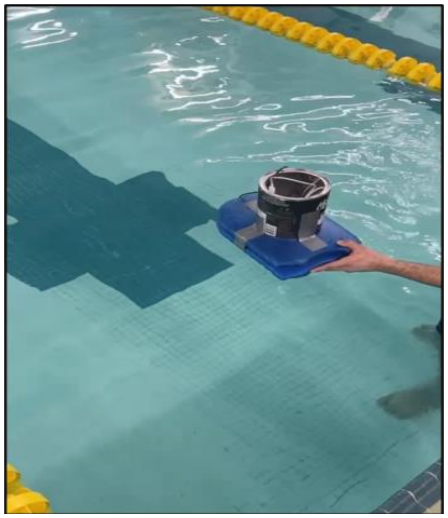


Figure 31 Initial Swimming Pool Testing

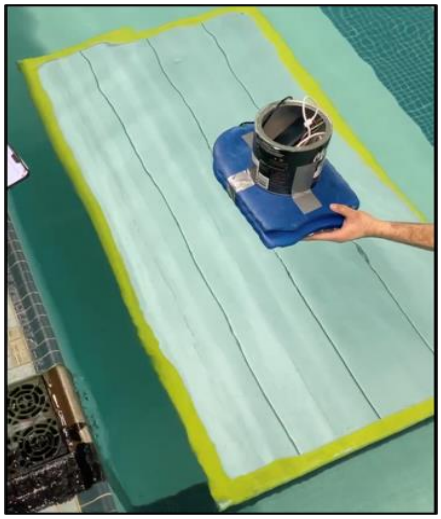


Figure 30 Testing sudden change in the depth

Once we confirmed that our system integration could reliably provide accurate depth measurements, we proceeded to test its response to sudden changes in depth. To simulate this scenario, we placed a calibrated table inside the pool and maneuvered our prototype boat over it. This setup allowed us to observe how the system reacted to abrupt depth variations. The results and analysis of this experiment are also presented in the system validation section.

Additionally, we intended to test the system in a real-world river or lake environment. However, due to unfavorable weather conditions outside the university campus, field testing was not feasible during the course timespan. As an alternative, we purchased a plastic testbed container, which allowed us to simulate natural conditions by adding sand, mud, and rocks. With assistance from our technical lead, Gordon, we conducted similar tests in this controlled environment. However, the results were suboptimal due to reflections from the container's walls, which interfered with sensor readings.

Despite this challenge, the experiment provided valuable insights that we later applied to improve our testing approach in subsequent trials.

Figure 32 Test bed testing

9.1.2 Civil Lab testing

After conducting tests in the plastic test bed, we realized it was not the most suitable environment for our experiment, as the confined space and reflective surfaces interfered with sensor accuracy. Given these challenges, we sought alternative testing options. Following a suggestion from Gordon, we reached out to Alex Wall, in-charge of the Civil Hydraulics Laboratory, for his assistance in testing our system integration. The controlled environment of the civil water flume closely resembled a natural riverbed, providing a more realistic setting for our tests. The flume featured a continuous water flow (manually adjustable) from one side

Figure 33 Placed an arbitrary block inside the water flume

to simulate different conditions. Additionally, with a maximum depth of 60 cm, it was well within the operational range of our sensor specifications. Moreover, we were permitted to place calibrated bricks inside the flume, allowing us to refine our system hazard differentiation feature by simulating underwater obstacles.

These improved testing conditions helped us overcome the limitations of the plastic test bed, enabling us to obtain more reliable and accurate results. The enhanced setup allowed for a more thorough evaluation of our system's performance in real-world scenarios, ultimately contributing to the refinement and optimization of our prototype.

9.2 System Validation

This section details our validation approach and experimental results for ensuring that the IoT Lake/River Real-time Depth Monitoring System performs reliably across different environments that were accessible to us during the design and testing process. Our validation process is divided into controlled tests in a swimming pool, custom test bed container and in a Civil Hydraulics Lab water flume, and rigorous specifications validation to confirm system performance metrics.

9.2.1 Swimming Pool Testing

In our initial validation, we used a swimming pool environment to evaluate sensor accuracy and communication performance. This test enabled us to verify that our system could accurately measure depth and detect obstacles, using a controlled setting where interference from sonar reflections off hard pool walls could be observed. Although the swimming pool provided a stable water surface, it had limitations in simulating the diverse conditions found in natural water bodies, prompting further testing in a more dynamic environment.

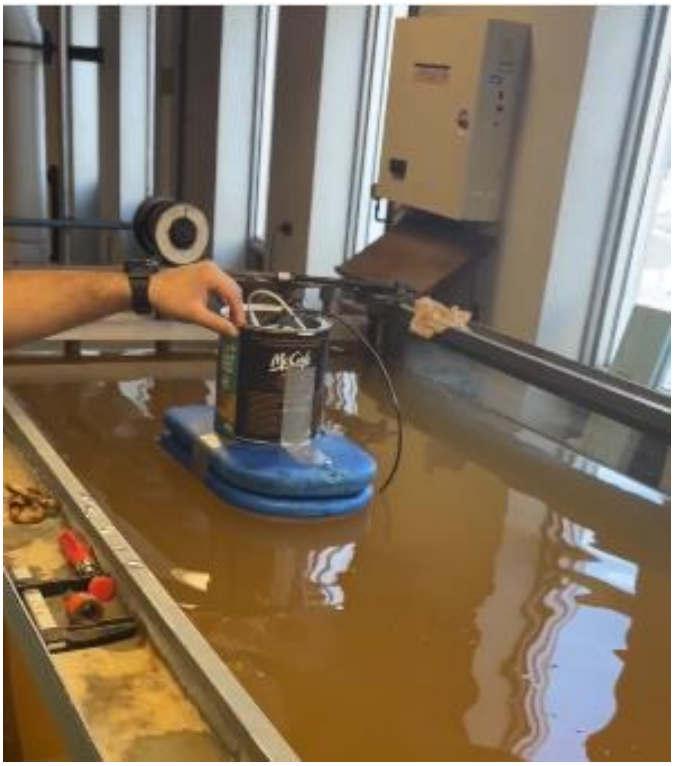


Figure 34 Civil Lab System testing

Observations from our tests revealed that when the Raspberry Pi is connected to Wi-Fi, location data via Wi-Fi geolocation is received at intervals of approximately 7–8 seconds. In contrast, when Wi-Fi is unavailable and GPS simulation is used via Bluetooth, the intervals shorten to 3–4 seconds. This suggests that the Wi-Fi geolocation process adds an average delay of about 4 seconds. However, in the real-word deployment, our device would be used and tested on rivers or lakes which would enable the GPS device to work with full functionality outdoor.

During shallow swimming pool tests, after adjusting for the sensor's vertical dimension (41 mm), the expected depth range was 1.17–1.41 m, and the measured depths ranged from 1.185 m to 1.462 m, resulting

in an average percentage error of roughly 1.28% to 3.69%. Tests in a deep swimming pool, where the expected depth range was 2.25–3.62 m (after subtracting the sensor's 0.04 m offset), yielded measured depths of 2.202–3.699 m with an average error of about 2.13–2.18%. Furthermore, tests that introduced an obstacle (an immersed table) in a shallow pool showed a drop in the average measured depth—from 1317.5 mm without the obstacle to 811.29 mm with it—indicating an approximate table height of 506.21 mm which was very close to the accurate measurement.

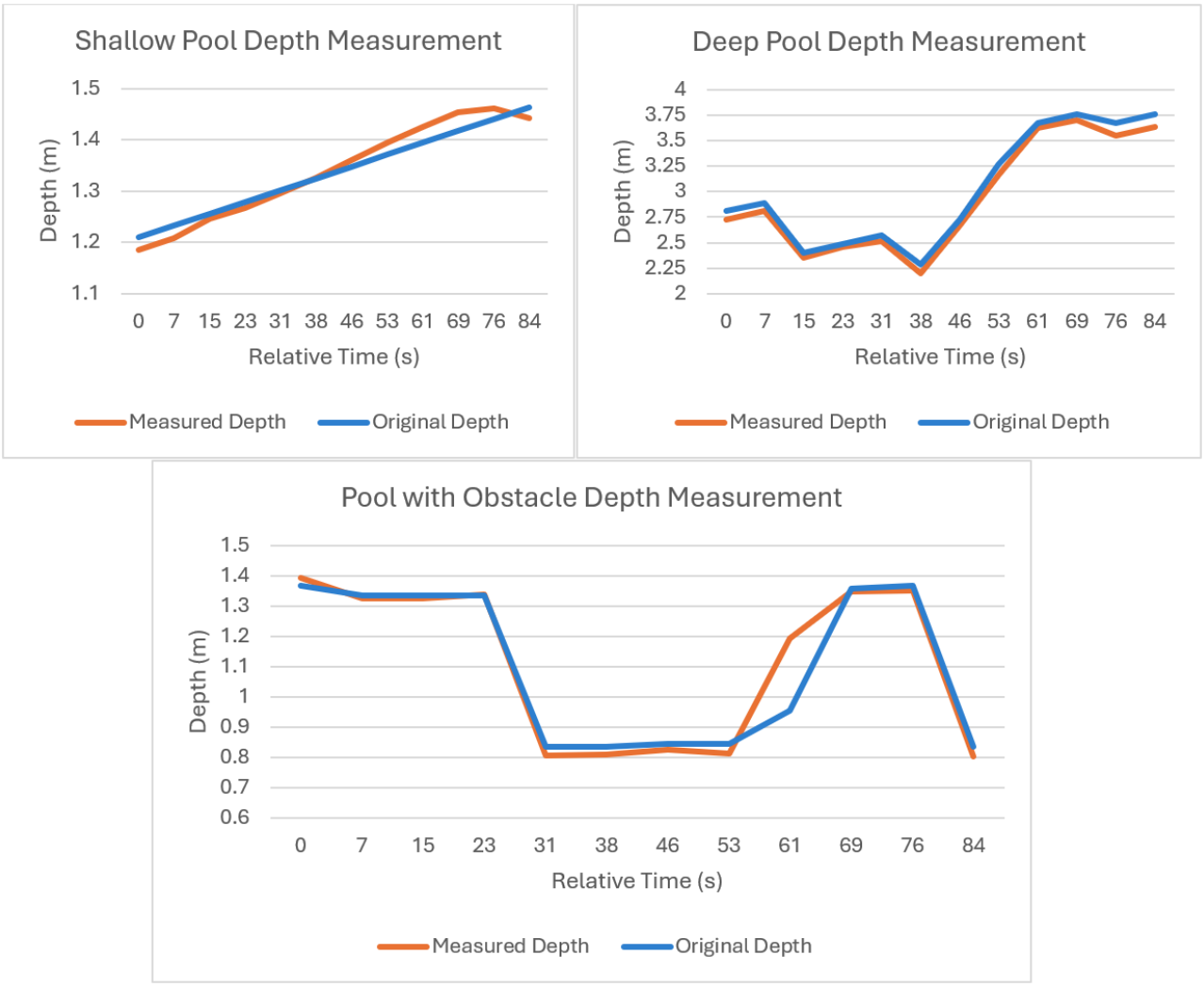


Figure 35 Validation on the Datasets obtained from Swimming Pool testing

9.2.2 Custom Test Bed Container

We have used a large container to prepare a custom testbed to introduce more diversity to our testing scenarios. The tests conducted on it exposed physical limitations in our testbed. Although the container initially satisfied the sensor's minimum depth requirement of 30 cm according to the manufacturer, the bending of the container walls under water pressure reduced the effective depth to about 21 cm, which

adversely affected the readings. Additionally, multi-reflection of sonar waves from the container walls introduced fluctuations in the depth values. We proposed reinforcing the container walls by placing heavy objects along the wider sides to prevent bending, which should allow the container to reach its full depth potential and reduce reflective interference. We have also narrowed down the angle of sonar wave beams to reduce interference from walls. Fortunately, we were able to conduct our tests in a river-model apparatus which was available in Civil Hydraulics Laboratory, which eliminated all the major limitations of our custom testbed container.

9.2.3 Civil Hydraulics Lab Water Flume Testing

The Hydraulics Lab's water flume provided a controlled yet dynamic environment to simulate riverine conditions. The flume's adjustable flow rate and modular brick obstacles enabled rigorous validation of hazard detection under turbulence. In our initial scenario, the tank was filled with water to a manually measured depth of 61 cm. Using our system, we collected real-time data via our mobile app, which was directly uploaded to the cloud. During this test, the sensor's depth readings were processed to include additional parameters such as the original measured depth and calculated range resolution; analysis of the plotted data revealed an average error of approximately 7 mm, confirming reliable sensor accuracy despite the challenging conditions.

Subsequently, we tested the system with obstacles placed in the water. We first arranged submerged bricks in fixed positions—with measured heights of 190.50 mm, 87.31 mm, and different shapes such as half cylinders—and then in random configurations to increase the complexity of the dataset. These tests provided valuable information regarding the response of the system when encountering physical objects or hazards, and the collected data was subsequently used for training our machine learning models. A further testing scenario involved initiating the depth measurement process (using the START command) while gradually filling the tank to observe the minimum depth at which the device could reliably register a 100% confidence reading; the device consistently recorded accurate values once the depth reached 0.379 m. On a later occasion, we conducted additional tests by placing different sized bricks to gather more varied data for model training.

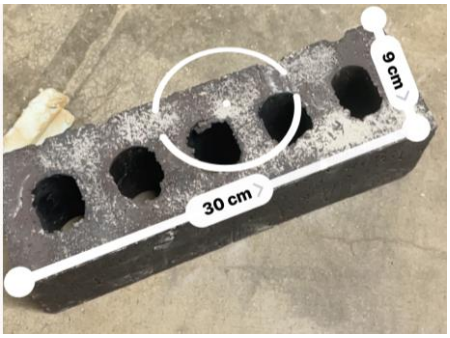


Figure 36 Measuring dimension of the used bricks

These comprehensive tests revealed several key insights. Firstly, the accuracy of the Ping2 transducer is confirmed with an average error of less than 1 cm, although our observations indicate that even a 1 cm difference in the effective water depth can be significant, particularly when ensuring that the prototype boat remains centrally located within the test area. The implementation of a gain setting of 2 in the sensor's code helped reduce noise by focusing on stronger signals, thereby improving measurement reliability compared to our earlier custom testbed experiments. Moreover, the diverse scenarios—from simple depth measurement to obstacle detection—underscore the necessity of incorporating a variety of objects beyond just bricks to further refine our machine learning model for hazard differentiation.

9.2.3 Specifications Validation

This section provides a detailed technical validation "Placing bricks at different known locations for testing and ML training" of key system performance metrics, assessed during controlled lab testing. In defining our performance specifications, several new metrics have been added, and some have been modified based on our testing results and practical constraints. Our system performance metrics can be found in *Project Specifications.*

Measuring and Validating Minimum and Maximum Depth: We validated the sensor's accuracy by comparing its readings with manual measurements in controlled settings. The sensor reliably recorded a minimum depth of 0.379 m, confirming accurate performance in shallow water. In deep swimming pool tests—after adjusting for sensor mounting offsets—the sensor measured a maximum depth of 3.69 m. It

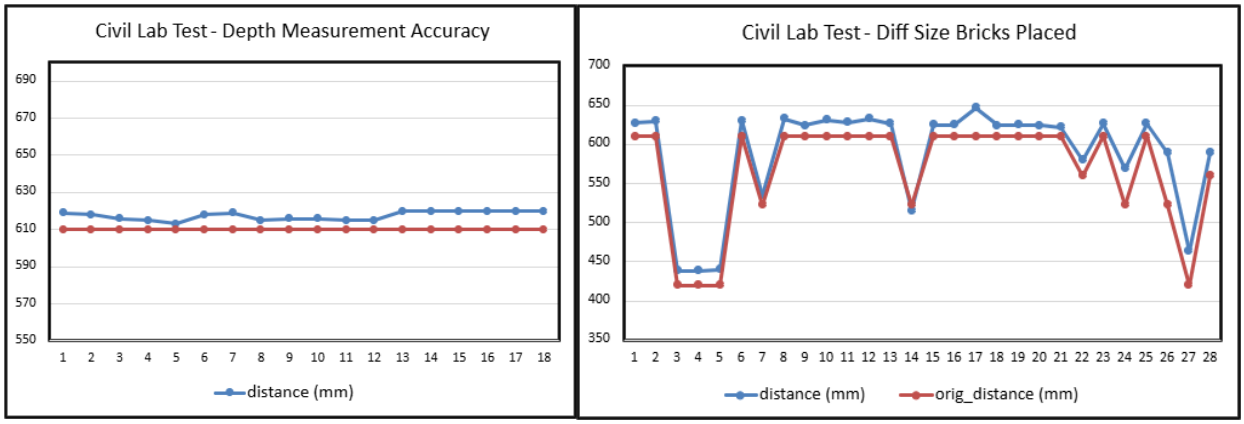


Figure 37 Validation on the Datasets obtained from the Civil Lab testing

should be noted that this maximum was limited by our testing facility, which could not simulate deeper water

conditions expected in natural lakes or rivers. Nonetheless, these results demonstrate consistent sensor performance within the tested range and serve as a robust proof-of-concept for our application.

Figure 38 Placing bricks at different known locations for testing and ML training

Real-Time Depth Monitoring Communication: In order to measure communication delay, we needed to make sure that the machine clock on RPi and Phone app are synced. By printing concurrent timestamps, we determined the calibration offset to consider in our calculation for communication delay, which is 154.37 milliseconds.

Table 6 Determining Clock Calibration between RPi and App

Current RPi Timestamp (ms)	Current App Timestamp (ms)	Calibration Offset (ms)
1738969174996.25	1738969174805.00	191.25
1738970271795.06	1738970271666.00	129.06
1738970367237.74	1738970367013.00	224.74
1738970458096.43	1738970457979.00	117.43
1738970542385.36	1738970542276.00	109.36
Average Calibration Offset (ms)		154.3680176

The system's real-time depth monitoring performance has been validated through two communication channels. When connected to Wi-Fi, the Raspberry Pi 4 delivers depth data to the application via MQTT at an average delay of approximately 0.469 seconds (see Table 8), while Bluetooth Low Energy (BLE)

communication—used in the absence of Wi-Fi—requires about 0.492 seconds (see Table 7). The target for real-time communication is a delay of less than 2 seconds, which has been comfortably met by both methods.

Table 7 Determining Real-Time Communication Delay using BLE

RPi Timestamp at Send (ms)	App Timestamp at Receive (ms)	Calibration Offset + Delay (ms)
1738891606592.37	1738891607154.00	716.00
1738891609751.96	1738891609957.00	359.41
1738891612881.67	1738891613164.00	436.70
1738891616027.26	1738891616366.00	493.11
1738891619137.13	1738891619437.00	454.24
Average Total Communication Delay (ms)		491.8920215

Table 8 Determining Real-Time Communication Delay using Wi-Fi and MQTT

RPi Timestamp at Send (ms)	App Timestamp at Receive (ms)	Calibration Offset + Delay (ms)
1738890386440.48	1738890386778.00	491.89
1738890393626.45	1738890393934.00	461.92
1738890400721.63	1738890401002.00	434.74
1738890408268.30	1738890408580.00	466.07
1738890415721.65	1738890416056.00	488.72
Average Total Communication Delay (ms)		468.6680469

Frequency of Readings: The frequency of sensor readings, defined as the time interval between successive measurements and subsequent logging, is a critical performance parameter. In testing, when simulating GPS values, the average interval was approximately 3.14 seconds (see Table 9). When Wi-Fi geolocation was employed to provide location data, the interval increased to about 7.32 seconds (see Table 10). These values reflect the processing delays inherent in the combined script, which manages three states—reading sensor data, reading GPS data, and synchronizing—resulting in a cumulative delay that meets the operational design requirements.

Table 9 Frequency of Readings for Simulation or Device

Timestamps of RPi Readings for simulation or device (ms)	Difference in Readings (ms)
1738891606592.37	-

1738891609751.96	3159.59
1738891612881.67	3129.71
1738891616027.26	3145.59
1738891619137.13	3109.87
Average Difference in Readings (ms)	3136.189941

Table 10 Frequency of Readings for Wi-Fi Geolocation

Timestamps of RPi Readings for Wi-Fi Geolocation (ms)	Difference in Readings (ms)
1738890386440.48	-
1738890393626.45	7185.97
1738890400721.63	7095.18
1738890408268.30	7546.67
1738890415721.65	7453.35
Average Difference in Readings (ms)	7320.29248

File Upload Speed: For data transmission, the system uploads log files—comprising JSON messages that include depth data and safe routing information—from the Raspberry Pi to AWS S3. Our validation tests measured an upload speed of approximately 27.13 kB/s, which exceeds the target upload speed of 20 kB/s. This metric confirms that file synchronization with the cloud is efficient, ensuring timely data availability for further processing and application display.

Table 11 Determining File Upload Speed to AWS Cloud Services from RPi

RPi Timestamp at Upload (ms)	Timestamp on AWS S3 server when uploaded (ms)	File Size (kB)	Delay (ms)	Speed (kB/s)
1738895297704.40	1738895300000.00	66.5	2295.600098	28.96846017
1738895300194.31	1738895301000.00	27.8	805.6899414	34.50458864
1738895301609.17	1738895303000.00	24.9	1390.830078	17.90297779
			Average Upload Speed (kB/s)	27.1253422

Application Response Time: The application's performance was evaluated by two key indicators. First, the real-time UI updates—based on state changes and provider notifications—are achieved within 1 second, well under the target of 2 seconds. Second, the time required to query AWS S3, retrieve pre-mapped routes, and

update the display in the app was measured at roughly 4.42 seconds, which is within our target of less than 5 seconds.

Table 12 Determining Application Response Delay

Timestamp of when user clicks and request to query (ms)	Timestamp when the list of maps is retrieved and showed on App (ms)	Delay (ms)
1738895941895.00	1738895946536.00	4641
1738896018161.00	1738896022426.00	4265
1738896043788.00	1738896048139.00	4351
	Average Response Delay	4419

Hazard Differentiation Accuracy: Finally, the performance of the hazard differentiation module was validated through machine learning experiments. Our model, trained on data from controlled lab tests, achieved an accuracy of 88.89% on the training dataset (see Figure 38) and 91.43% on the test dataset (see Figure 39). These results meet our performance targets of over 85% during training and over 80% during testing, confirming the viability of the SVM-based approach for reliably indicating hazard severity. However, after adding K-fold cross validation and more dataset into our model, the model accuracy became 81.61% (see Figure 40) which falls short by a few percentage from our target accuracy.

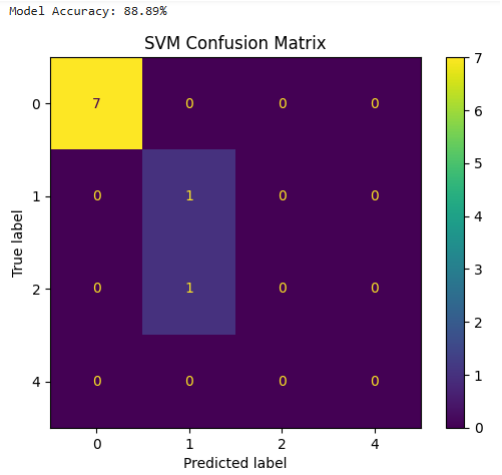


Figure 39 SVM Model Accuracy Test

```
Accuracy for each fold: [0.8286, 0.8286, 0.8, 0.8, 0.8235]
Average accuracy across 5 folds: 0.8161344537815125
```

Figure 40 Model Accuracy while Training with K-fold cross validation and more dataset

```
Predictions added. Output saved to civiltest2_predict_dataset.txt.
Accuracy during testing: 91.42857142857143% (based on 35 valid predictions)
```

Figure 41 Model Accuracy while Testing in Laboratory

10 Budget

In summary, the project exceeded the allocated budget by $69, surpassing the $600 provided by the department ($100 per team member for six members). The additional funds were covered through an external budget allocated to the team.

Table 13 Budget

	PROPOSED (including Shipping)			ACTUAL (including Shipping)		
PRODUCT PART	Unit Cost	Quantity	Total Cost	Unit Cost	Quantity	Total Cost
Ping2 Sonar Altimeter and Echosounder	$580	1	$580.00	$578.00	1	$578.00
Raspberry Pi4 Model B	$0	1	$0.00	$0.00	1	$0.00
Arduino UNO R3	$0	1	$0.00	$0.00	1	$0.00
SIM (2FF,3FF,4FF) VER 4G LTE	$8.98	1	$8.98	-	-	-
MICROSD CARD 32GB SANDISK	$21.30	1	$21.30	$0.00	1	$0.00
MICROSD CARD MODULE FOR ARDUINO	$8.33	1	$8.33	-	-	-
SIM808 GPS/GPRS/GSM Arduino Shield	$56	1	$56.00	-	-	-
RC Boat[1]	$68.31	1	$68.31	$0.00	1	$0.00
Battery	$43.66	1	$43.66	$0.00	1	$0.00
Custom PCB	$35	1	$35.00	-	-	-
GPS Device: YIC / GU-504GGB-USB	-	-	-	$59.11	1	$59.11
USB_UART Converter	-	-	-	$10.18	1	$10.18
Test Bed Container[1]	-	-	-	$0.00	1	$0.00
Subtotal			$822			$647.29
Contingency Costs		+25%	$205.40			
Total			$1,027			$647.29

Funds Available from ECE Department	$600	CR
Used ECE Department Fund	$578	DR
External Budget (Group Contribution)	$69.29	DR

[1] These parts were returned after use

11 Future Considerations

While significant progress has been made in developing a depth monitoring system for lakes and rivers, several improvements can further enhance its functionality, accuracy, and user experience. Below are key areas for future development:

- **Automatic Underwater Imaging and Cloud Integration:** Implementing an automated underwater imaging system will allow for real-time image capture and cloud uploads. This enhancement will:
    - Improve model accuracy by continuously feeding high-quality underwater images into the system.
    - Enable real-time feedback for users via the mobile app, providing insights into underwater conditions.
    - Facilitate automated anomaly detection, such as detecting debris, aquatic vegetation, or sudden depth changes.
- **Advanced Navigation Services:** Enhancing the app with smart navigation features will provide users with a more interactive and safer boating experience. Proposed improvements include:
    - Real-time route tracking with GPS guidance to assist users in navigating through lakes and rivers efficiently.
    - Hazard alerts based on time intervals (e.g., 5, 10, 20 minutes ahead) or proximity (e.g., 5, 10, 20 meters ahead).
    - Location-based recommendations for safer or optimal routes based on environmental conditions.
- **Speed Sensor Integration for High-Frequency Data Collection:** Adding a speed sensor to the system will improve the accuracy of boat movement tracking. Benefits include:
    - Collection of high-frequency speed data optimized for lake and river environments.
    - Enhanced data precision for better correlation between speed, depth variations, and navigation safety.
    - Potential for speed-based adaptive navigation recommendations.
- **Improved Data Visualization:** Currently, depth and location data are displayed as markers on a map. Future improvements could involve:
    - Using interpolation techniques to create a continuous raster representation of depth data.
    - Displaying this raster as a **3D surface plot** or **heatmap** for better visualization and analysis.
    - Improving user experience by making data more intuitive and visually engaging.
- **Alternative Routing Algorithms:** Optimizing the current routing algorithm will enhance navigation efficiency. Future developments can include:

    - Investigating alternative routing algorithms like A* or Dijkstra's algorithm which may provide better routes for an algorithm speed trade-off.
    - Customizable routing options based on user preferences (e.g., shortest route, safest route, scenic route).
    - Adaptive routing that dynamically updates based on real-time environmental changes.
- **Data Merging for Multiple Recording Sessions:** To improve the accuracy and usability of depth maps, future updates should allow for:
    - Merging data from multiple recording sessions to create more comprehensive depth information for the same lake or river.
    - Synchronizing past and current data to detect long-term depth variations and environmental changes.
    - Implementing an automated data-cleaning process to filter out inconsistencies or anomalies.
- **Map Search Functionality:** Enhancing the app with a search feature will improve usability and accessibility by allowing users to:
    - Search for specific locations, routes, or depth readings.
    - Bookmark frequently visited locations for easy access.
    - Integrate filters to refine search results based on parameters such as depth range, speed limits, or hazard zones.
- **Hazard Differentiation Dataset:** Expanding the dataset to include various hazard types will make the system more robust. Improvements include:
    - Differentiating hazards such as submerged obstacles, strong currents, and shallow areas.
    - Training the system to recognize and categorize hazards based on real-world testing data.
    - Providing detailed hazard descriptions and severity levels in the app for better decision-making.
- **Real-World Testing and Model Validation:** To ensure system accuracy and reliability, further real-world testing is essential. This includes:
    - Conducting field tests in diverse lake and river environments beyond controlled indoor simulations.
    - Validating model predictions with real-environment depth readings.
    - Gathering additional real-world data to improve machine learning model training and generalization.

By implementing these improvements, the system can evolve into a more advanced, reliable, and user-friendly tool for navigation, depth monitoring, and hazard detection in aquatic environments.

12 Conclusion

In conclusion, our "Design and Implementation of IoT Lake/River Real-time Depth Monitoring System" project has successfully integrated hardware, firmware, cloud services, and mobile application components into a cohesive system. Through extensive testing in varied environments—from controlled swimming pool experiments using custom test beds to dynamic water flume tests in the Civil Lab—we have validated critical performance metrics, including accurate depth measurement (with minimal error), robust real-time communication via both Wi-Fi (MQTT) and Bluetooth Low Energy, and effective data integration for path mapping and hazard differentiation. The experimental results and refined system validations confirm that our design meets the project specifications and provides a reliable platform for real-time bathymetric surveying and hazard detection.

The iterative testing phases—spanning controlled swimming pools, custom test beds, and the Hydraulics Lab's river-model flume—provided critical insights into environmental challenges such as acoustic reflections, turbidity, and GPS signal loss. Key innovations, including timestamp synchronization via fixed-ratio pairing, SVM-based hazard severity classification, and cloud-assisted model updates, addressed these challenges while maintaining compliance with IoT best practices. The system's adaptability was further proven through its ability to handle asynchronous sensor data, dynamic obstacle configurations, and real-time user interactions via the Flutter-based mobile app. These accomplishments underscore the project's alignment with academic research in embedded systems and industry standards for marine IoT applications.

Looking ahead, this work lays a foundation for expanding the system's capabilities through integration with autonomous vessels, multi-sensors and partnerships with environmental agencies for large-scale bathymetric mapping. Future iterations can focus on enhancing obstacle diversity in training datasets, refining edge-AI latency, and improving energy efficiency for prolonged deployments. By bridging the gap between theoretical machine learning and practical IoT deployment, this project not only enhances boater safety but also contributes to broader efforts in sustainable waterway management and smart marine infrastructure.

References

[1] Blue Robotics**,** "Ping Sonar Technical Guide," [Online]. Available: https://bluerobotics.com/learn/ping-sonar-technical-guide/#authors.

[2] GeeksforGeeks, "Difference between Arduino and Raspberry Pi," *GeeksforGeeks*, Sep. 4, 2024. [Online]. Available: https://www.geeksforgeeks.org/difference-between-arduino-and-raspberry-pi/.

[3] Raspberry Pi Foundation, "Getting started with your Raspberry Pi," *Raspberry Pi Documentation*, [Online]. Available: https://www.raspberrypi.com/documentation/computers/getting-started.html

[4] Raspberry Pi Forums, "[SOLVED] Wi-Fi is currently blocked by rfkill Issue," *Raspberry Pi Forums*, Feb. 2024. [Online]. Available: https://forums.raspberrypi.com/viewtopic.php?t=379975.

[5] Raspberry Pi Forums, "Getting Started with Network Manager," *Raspberry Pi Forums*, Aug. 2023. [Online]. Available: https://forums.raspberrypi.com/viewtopic.php?t=357739.

[6] Blue Robotics, "simplePingExample.py," *GitHub*, [Online]. Available: https://github.com/bluerobotics/ping-python/blob/master/examples/simplePingExample.py.

[7] YIC, "YIC-GU-504GGB Datasheet," *YIC Electronics*, Dec. 2024. [Online]. Available: https://www.yic.com.tw/wp-content/uploads/2024/12/YIC-GU-504GGB.pdf.

[8] Google Developers, "Get an API Key," *Google Maps Documentation*, [Online]. Available: https://developers.google.com/maps/documentation/geolocation/get-api-key#console_1

[9] "Geolocation Requests," *Google Maps Documentation*, [Online]. Available: https://developers.google.com/maps/documentation/geolocation/requests-geolocation.

[10] "Reverse Geocoding Requests," *Google Maps Documentation*, [Online]. Available: https://developers.google.com/maps/documentation/geocoding/requests-reverse-geocoding.

[11] Amazon Web Services, "What is MQTT?" *AWS Documentation*, [Online]. Available: https://aws.amazon.com/what-is/mqtt/.

[12] Amazon Web Services, "What is AWS IoT?" *AWS IoT Developer Guide*, [Online]. Available: https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html.

[13] Amazon Web Services, "Connecting to an Existing Device," *AWS IoT Developer Guide*, [Online]. Available: https://docs.aws.amazon.com/iot/latest/developerguide/connecting-to-existing-device.html

[14] Amazon Web Services, "Viewing MQTT Messages," *AWS IoT Developer Guide*, [Online]. Available: https://docs.aws.amazon.com/iot/latest/developerguide/view-mqtt-messages.html

[15] Nordic Developer Academy , “Lesson 1 - Bluetooth Le Introduction,” [Online]. Available: https://academy.nordicsemi.com/courses/bluetooth-low-energy-fundamentals/lessons/lesson-1-bluetooth-low-energy-introduction/.

[16] pub.dev, "flutter_reactive_ble," *Dart Packages*, [Online]. Available: https://pub.dev/packages/flutter_reactive_ble.

[17] Amazon Web Services, "upload_file - Boto3 S3 Client," *Boto3 Documentation*, [Online]. Available: https://boto3.amazonaws.com/v1/documentation/api/1.35.6/reference/services/s3/client/upload_file.html.

[18] IBM, "What Is Supervised Learning?" *IBM*. [Online]. Available: https://www.ibm.com/think/topics/supervised-learning.

[19] scikit-learn developers, "1.4. Support Vector Machines" *scikit-learn*. [Online]. Available: https://scikit-learn.org/stable/modules/svm.html.

[20] R. Wang, 2-D Interpolation, [Online]. Available: https://pages.hmc.edu/ruye/MachineLearning/lectures/ch7/node7.html.

[21] Amazon Web Services, "AWS Amplify," *AWS Documentation*, [Online]. Available: https://aws.amazon.com/amplify/.

[22] Amazon Web Services, "Amplify Flutter Documentation," *AWS Amplify Documentation*, [Online]. Available: https://docs.amplify.aws/flutter/.

[23] Flutter Dev, "State Management," *Flutter Documentation*, [Online]. Available: https://docs.flutter.dev/data-and-backend/state-mgmt.

[24] pub.dev, "flutter_local_notifications," *Dart Packages*, [Online]. Available: https://pub.dev/packages/flutter_local_notifications.

Appendix

Appendix A Technical Details of Ping2 Sensor

Technical Details

Specifications

Parameter	Value	
Electrical		
Supply Voltage	4.5–5.5 V	
Logic Level Voltage	3.3 V (5 V tolerant)	
Current Draw	100 mA (typical) 900 mA (peak)	
Communication		
Signal Protocol	TTL Serial (UART)	
Available Firmware Baud Rates	115200 bps (default) 9600 bps	
Message Protocol	Ping Protocol	
Message Subsets	common, ping1d	
Software Development Libraries	Arduino ArduPilot (limited) C++ Python Rust	
Cable		
Cable Used	BR-100452	
Cable Length	875 mm	34.5 in
Maximum Cable Length	8 m	26 ft
Conductor Gauge	24 AWG	
Wires	Black Red White Green	Ground Vin TX (out) RX (in)
Installed Penetrators		
Device Side	M06-4.5mm-LC	
Cable End	M10-4.5mm-LC	
Acoustics		
Frequency	115 kHz	
Source Level	198 dB re 1 uPa @ 1 m	
Beamwidth (-3 dB from peak)	25 degrees	

Typical Minimum Range	0.3 m	1 ft
Typical Usable Range[1]	100 m	328 ft
Range Resolution	0.5% of range	
Range Resolution at 100 m (waterfall range, not measured range)	50 cm	19.7 in
Range Resolution at 2 m (waterfall range, not measured range)	1 cm	0.4 in
Physical		
Depth Rating	300 m	984 ft
Temperature Range	0–30°C	32–86°F
Weight in Air (w/ cable)	187 g	0.41 lb
Weight in Water (w/ cable)	100 g	0.22 lb
Mounting Bracket Screw Size	M5x16 mm	
Internals		
Air Chamber O-Ring	AS568-030, Buna-N, 70A	

[1] Usable range depends on operating conditions (e.g. performance may suffer over variable density changes from tall vegetation, groups of fish, geysers, etc).

Appendix B Bluetooth Communication Protocol

There are two versions of Bluetooth: Bluetooth classic and Bluetooth Low Energy (BLE).

Bluetooth Classic

- Reliable communication with high data rate (1Mbps to 3Mbps)
- 2.402 - 2.480 GHz band
- 79 channels with a frequency width of 1 MHz
- like serial connection like UART
- Used for continuous communication like audio streaming and data transfer

BLE

- Designed for low power consumption at the expense of data rate by only transmitting data intermittently (125Kbps to 2Mbps)
- 2.402 - 2.480 GHz band
- 40 channels with a frequency width of 2 MHz
- like client/server connection
- Used for everything including audio streaming and data transfer
- Additional features like positioning and multiple communication topologies

For our project, we used BLE since our device is battery powered, therefore having low power consumption is beneficial. Also, we do not need a high data rate. Lastly, BLE is the most compatible version of Bluetooth. iOS devices exclusively use BLE.

There are two types of BLE devices: peripherals and centrals. A peripheral can handle many centrals and a central can connect to many peripherals.

Peripheral Device

Peripheral devices are usually battery powered devices like headphones, smart watches, or even smart bulbs. They act as severs in a client/server connection. Peripherals advertise their services to nearby devices.

Central Device

Central devices are usually more powerful devices like smart phones or laptops. They act as clients in a client/server connection. Central devices scan for nearby Bluetooth devices and the services they offer. They initiate a connection between the two devices.

Once a connection is established between these devices, a Generic Attribute Profile (GATT) is used to govern the data exchange. A GATT is a data structure made of a hierarchy of attributes listed

below. Each attribute can have a standardized 16-bit UUID defined by the Bluetooth SIG group or a custom 128-bit UUID.

Service Attribute

A service attribute is a service offered by a device. A device can have a standardized service that is registered with the Bluetooth SIG group, like the Battery Service. It can also have custom services. A device can offer multiple services.

Characteristic Attribute

A characteristic attribute is a characteristic offered by a service. The characteristic attribute contains the actual data value that is transferred between devices. A service can have a standardized characteristic that is registered with the Bluetooth SIG group, like Battery Level, Battery Power State, and/or Battery Level State. It can also have custom characteristics. A service can offer multiple characteristics.

Descriptor Attribute

A descriptor attribute is a descriptor that belongs to a characteristic. A characteristic can have a standardized descriptor that is registered with the Bluetooth SIG group, like Characteristic User Description and/or Client Characteristic Configuration. It can also have custom descriptors. A characteristic can have multiple descriptors.

These different types of attributes are combined to make an attribute table in a GATT server. The GATT server advertises its services. When a device connects, they can see the attribute table and exchange data.

Appendix C RBF Interpolant Calculation

Radial Basis Function (RBF) interpolation uses a kernel function to determine the weight of each sample based on distance. There are many kernel functions however the one we used is the gaussian kernel. The equation for the kernel is shown below. Gaussian kernels decay as distance between the two vectors increases.

$$K(\overrightarrow{x_1}, \overrightarrow{x_2}) = e^{-\frac{|\overrightarrow{x_1} - \overrightarrow{x_2}|^2}{\mu}}$$

The equation to interpolate a new point is shown below. It takes a list of samples points and calculates the interpolant by summing the contributions of each sample point on the new point. For each sample, it first determines the weight of the sample based on the distance between the sample vector and the new vector using the equation above. Then it is multiplied by another weight for the sample. The multiplication of the two weights results in the contribution of each sample on the interpolant.

x is the vector for the new point

$\boldsymbol{x_i}$ is the list of sample vectors

$\boldsymbol{w_i}$ is a weight vector for each sample point

f(x) is the interpolant

$$f(\vec{x}) = \sum_{i=1}^{N} w_i \, K(\overrightarrow{x_i}, \vec{x}) = \sum_{i=1}^{N} w_i \, e^{-\frac{|\overrightarrow{x_i} - \vec{x}|^2}{\mu}}$$

Now that we know how the equation works we can interpolate new points, however we do not know the vector $\boldsymbol{w_i}$. To find $\boldsymbol{w_i}$ we must use the depth value of the sample point as ***f(x)***. Then using the equation above, we solve for $\boldsymbol{w_i}$. Once all $\boldsymbol{w_i}$ are known, we can use the equation above to interpolate new points.

The calculation of $\boldsymbol{w_i}$ for all samples leads to a system of equations. Therefore, the problem can be solved with matrices as shown below.

d is the vector of depth values for each sample

w is the vector of weight values for each sample

G is the matrix of gaussian kernel values for each sample

$$\vec{d} = \vec{w}G$$

$$\vec{w} = \vec{d}G^{-1}$$

$$G = \begin{bmatrix} K(\vec{x_1}, \vec{x_1}) & K(\vec{x_1}, \vec{x_2}) & \cdots & K(\vec{x_1}, \vec{x_N}) \\ K(\vec{x_2}, \vec{x_1}) & \ddots & & \\ \vdots & & & \\ K(\vec{x_N}, \vec{x_1}) & & & K(\vec{x_N}, \vec{x_N}) \end{bmatrix}$$

After the sample weight are determined new points can be interpolated. The matrix equations are shown below.

x is the vector for the new point

w is the vector of weight values for each sample

k is the vector of gaussian kernel values for each sample

f(x) is the interpolant

$$f(\vec{x}) = \vec{w}\vec{k}^{-1}$$

$$\vec{k} = \left[K(\vec{x_1}, \vec{x})\ K(\vec{x_2}, \vec{x})\ \cdots\ K(\vec{x_N}, \vec{x})\right]$$

```
# Calculate Weights from x, y, and z samples points
# mu: constant 0.03
# sample_coors        (n,2): list of coors from readings
# z                   (n,1): list of depths from readings
def calcWeights(sample_coors, z, mu):
        # sample_mat (n,2)*(2,n) = (n,n)
        sample_mat = np.matmul(sample_coors, sample_coors.transpose())
        # sample_diag (n,1)
        sample_diag = np.diagonal(sample_mat)
        # X_sqr, Y_sqr (n,n)
        X_sqr, Y_sqr = np.meshgrid(sample_diag, sample_diag)
        # gramMatrix (n,n)
        gramMatrix = np.exp(-1/mu * (X_sqr - 2*sample_mat + Y_sqr))
        # weights (n,1)
        weights = np.matmul(np.linalg.pinv(gramMatrix), z)
```

```
        return weights
```

Code snippet for weight calculation

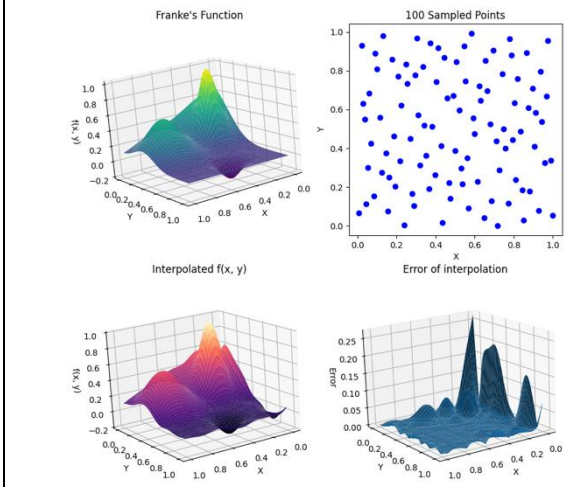
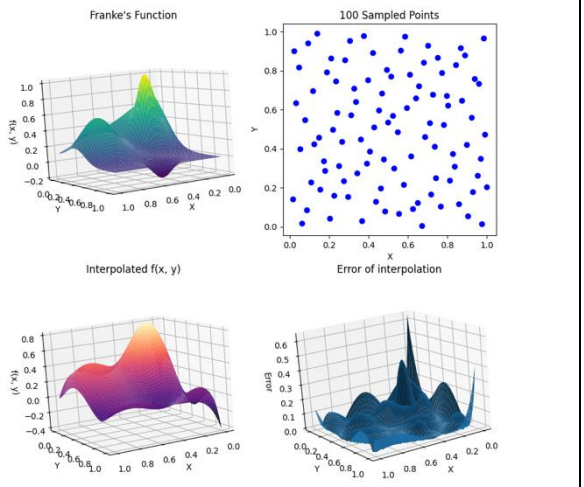
```
# Interpolate z values from grid of x and y points
# mu: constant 0.03
# X                 (m,p): longitude matrix
# Y                 (m,p): latitude matrix
# w                 (n,1): weights
# sample_coors      (n,2): list of coors from readings
def interpolate(X, Y, w, sample_coors, mu):
        # interp_coors (m x p, 2)
        interp_coors = np.array([X.flatten()[:], Y.flatten()[:]]).transpose()
        # sample_mat (n,2)*(2,n) = (n,n)
        sample_mat = np.matmul(sample_coors, sample_coors.transpose())
        # sample_diag (n,1)
        sample_diag = np.diagonal(sample_mat)
        # interp_mat (m x p, 2)*(m x p, n) = (m x p, m x p)
        interp_mat = np.matmul(interp_coors, interp_coors.transpose())
        # interp_diag (m x p, 1)
        interp_diag = np.diagonal(interp_mat)
        # Y_sqr, X_sqr (n, m x p)
        Y_sqr, X_sqr = np.meshgrid(interp_diag, sample_diag)
        w = w.reshape((1,w.size))
        # gramMatrix (n, m x p)
        gramMatrix = np.exp(-1/mu * (X_sqr - 2*np.matmul(sample_coors,
interp_coors.transpose()) + Y_sqr))
        # z_vals (1, n)*(n, m x p) = (1, m x p) which is reshaped to (m,p)
        z_vals = np.matmul(w, gramMatrix).reshape(X.shape)

        return z_vals
```

Code snippet for interpolation

Appendix D RBF Parameter Optimization

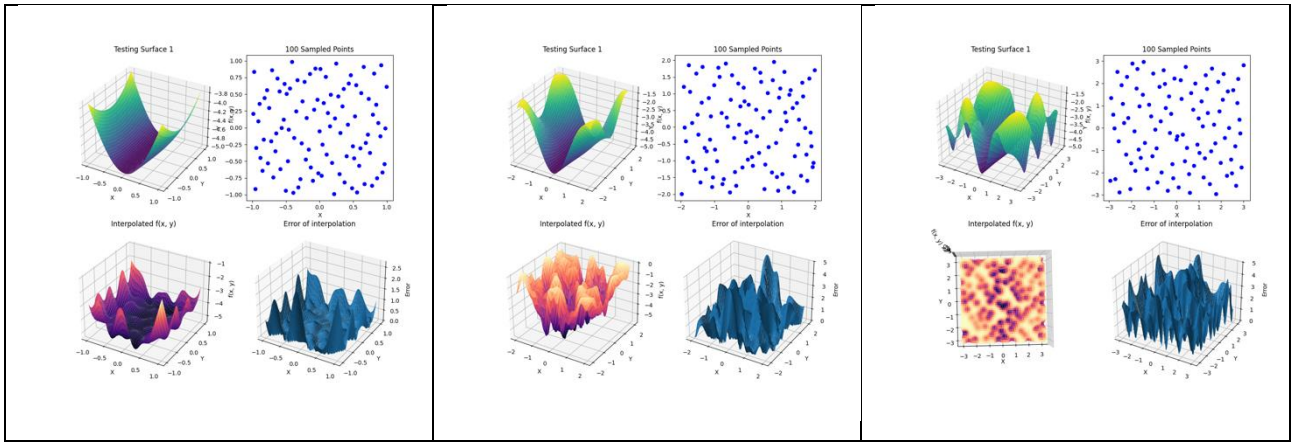
$$K(\overrightarrow{x_1}, \overrightarrow{x_2}) = e^{-\frac{|\overrightarrow{x_1} - \overrightarrow{x_2}|^2}{\mu}}$$

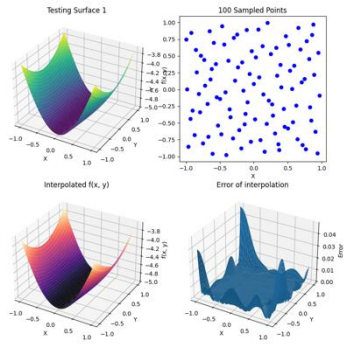
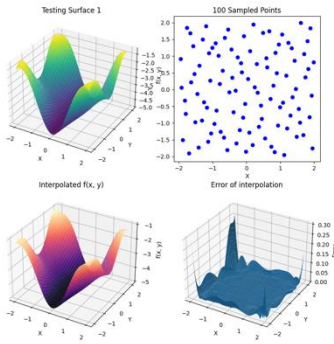
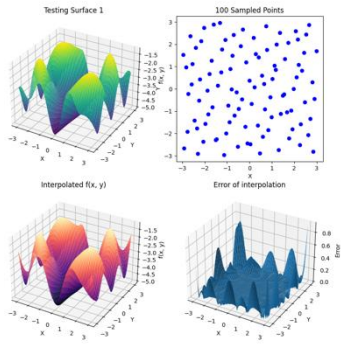
The RBF interpolation algorithm uses a Gaussian kernel. This kernel has one tuneable parameter, μ, which is shown in the denominator of the exponential in the equation above. Adjusting this parameter affects the quality of the interpolation.

	
μ = 0.05	μ = 20

When the parameter is set to a low value, the Gaussian kernel decays faster. When μ is set low, the new interpolated values pass closely through the sample points. When μ is set high, the new interpolated values get flattened. The data is smoothened since the kernel decays slower. This is an undesirable trait for our use, so the parameter should be set low.

Another case to consider is when the domain of the sample space varies. So far, all the plots shown of Franke's Function have been in the same domain. The domain of x and y is [0,1].



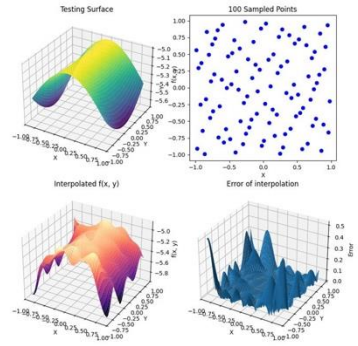
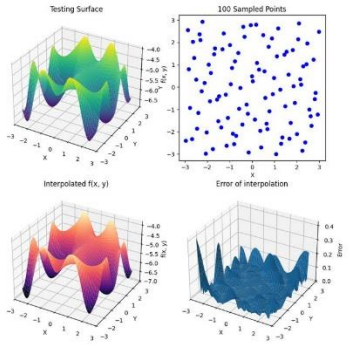
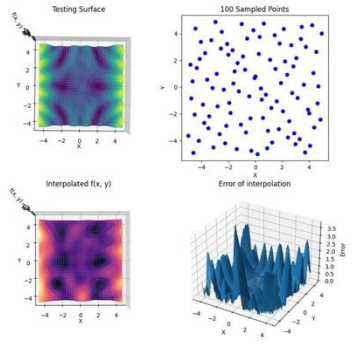
μ = 0.05, x&y domain = [-1,1]	μ = 0.05, x&y domain = [-2,2]	μ = 0.05, x&y domain = [-3,3]
		
μ = 0.35, x&y domain = [-1,1]	μ = 0.8, x&y domain = [-2,2]	μ = 1.5, x&y domain = [-3,3]

The plots above show a new testing surface with various domains. In the top row, μ is kept constant at 0.05, while the domain increases. We can see that as domain increases, the interpolation becomes worse. In the bottom row, μ is adjusted to a value that minimizes error. We can see that as domain increases, mu increases, and the interpolation is closer to the original surface. This means that μ must be scaled with the domain.

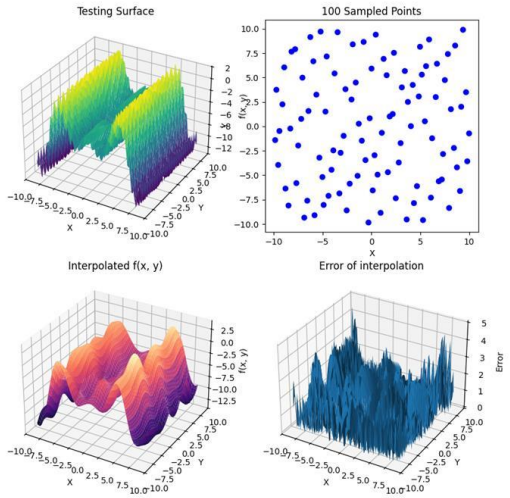
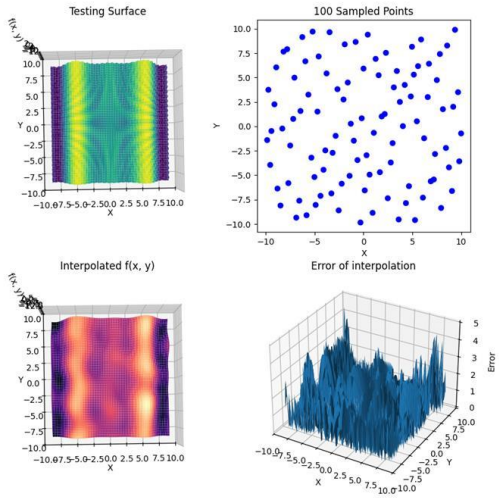
After considering the cases shown above, the optimal value of μ was found. First, we used gradient descent to find the optimal value of μ where the domain of x and y is [0,1]. The initial value of μ was set to 0.1 since low values of μ are desirable. The mean square loss was calculated by taking the difference between the actual surface and interpolated surface. Then the difference was summed up and the mean of the entire matrix was taken. This was repeated for 100 iterations. The optimal value was found to be around μ = 0.03.

Next, to address the issue in the second case, the domain of the sample space was scaled so that the domain of x and y is [0,1]. This was done by finding the minimum and maximum values of the samples. Using those values, the samples were translated to coordinates (0,0). Then, the samples were normalized to a value between 0 and 1. Once the interpolation is complete, the samples were converted back to the original sample space using those values.

The results of the adjustments are shown in the table below. μ was kept constant at 0.03, even though the domain was increasing.

		
x&y domain = [-1,1]	x&y domain = [-3,3]	x&y domain = [-5,5]

The interpolation method works as needed. It does struggle when the surface rapidly changes, however a surface like that is not likely to be recorded on lakes and rivers. In the table below, the surface is very jagged. This emulates some randomness in depth. The interpolation handles this randomness well and smoothens the surface.

	
x&y domain = [-10,10]	x&y domain = [-10,10] TOP VIEW

Appendix E RPi4_Combined Script

- The code is available on GitHub: https://github.com/sukhmeet468/G09-CapstoneProject/blob/g9capstone_app/RPi4_Scripts/V1_RPi4_CombinedScript.py

Appendix F BLE Server Script

- The code is available on GitHub: https://github.com/sukhmeet468/G09-CapstoneProject/blob/g9capstone_app/RPi4_Scripts/qtble_server.py

Appendix G Algorithms Running Script

- The code is available on GitHub: https://github.com/sukhmeet468/G09-CapstoneProject/blob/g9capstone_app/RPi4_Scripts/pathMapping_HazardPrediction.py

Appendix H Application (Flutter)

- The app is on the GitHub repository: https://github.com/sukhmeet468/G09-CapstoneProject.git

Appendix I Testing Videos and Images

- A file with Testing video and images was made and publicly available at: https://github.com/sukhmeet468/G09-CapstoneProject/blob/g9capstone_app/Testing%20Videos.md