UNIVERSITY OF ZAGREB

FACULTY OF ORGANIZATION AND INFORMATICS

V A R A Ž D I N

Matej Desanić

WATER PARAMETER MEASUREMENT SYSTEM

FINAL THESIS

Varaždin, 2025.

UNIVERSITY OF ZAGREB

FACULTY OF ORGANIZATION AND INFORMATICS

V A R A Ž D I N

Matej Desanić

JMBAG: 0016155191

Study: Information and business systems

WATER PARAMETER MEASUREMENT SYSTEM

FINAL THESIS

Mentor:

prof. dr. sc. Ivan Magdalenić

Varaždin, July 2025.

Matej Desanić

Statement of authenticity

I declare that my final thesis is the original result of my work and that I have not used any sources other than those listed in it in its preparation. Ethically appropriate and acceptable methods and techniques were used in the preparation of the thesis.

The author confirmed by accepting the provisions in the FOI-papers system

---

Summary

This paper concerns itself with the development of the water parameter measurement system in the Internet of Things environment, by using LoRa wireless technology and Docker platform for running the servers. The project is based on theoretical ideas of the Moore's Law and the basic principles of IoT. The main objectives of the project are to create a measuring device that uses an ESP32 microcontroller with a LoRa module to collect and transmit water temperature data. The system consists of an integrated measuring device, a server that manages the data flow, and an Android application to display measurement results. Solutions have been developed that enable reliable data collection and transmission with minimal energy consumption. Docker containerization was chosen for its ease of use and management, enabling system scalability. The conclusions of the paper highlight the success of the LoRa technology implementation, and the potential for further development and application in various industries.

Key words: LoRa; Internet of Things (IoT); ESP32; Docker; water parameter measurement; wireless communication; Android application; temperature sensors;

Table of content

1. Introduction

In recent years, the rapid development of the Internet of Things (IoT) has revolutionized how we observe and interact with our physical environment. This shift has been especially impactful in fields where traditional infrastructure is either impractical or cost-prohibitive, such as remote environmental monitoring. Among the growing number of wireless technologies used in IoT systems, **LoRa (Long Range)** has emerged as one of the most suitable choices for low-power, long-distance communication. Its ability to transmit data over several kilometers with minimal energy consumption makes it particularly well-suited for sensor networks deployed in remote or inaccessible locations.

This thesis presents the design and implementation of a **water parameter measurement system** based on LoRa communication, built to be modular, power-efficient, and easy to deploy. The system combines embedded sensor devices based on the ESP32 microcontroller, a Node.js-based backend containerized with Docker, and an Android application for displaying collected data. Its architecture reflects the real-world needs of environmental data collection: reliable communication, long-term operation on limited power, and a clear separation between data acquisition, storage, and presentation.

According to Betancur et al. (2021), LoRa is increasingly being adopted in environmental and aquaculture applications due to its "*low deployment cost, long-range coverage, and suitability for device-level power constraints*." (Betancur, 2021) This work builds on that premise by demonstrating how LoRa can be effectively integrated into a complete, scalable system for monitoring water temperature. By containerizing the backend using Docker, the system remains portable and maintainable, while the use of REST APIs ensures interoperability with external systems or platforms.

Through the combination of embedded systems, wireless communication, and modern software development practices, this project provides a robust and extensible foundation for IoT-based environmental monitoring — with potential applications ranging from lake ecosystems and shellfish farming to urban infrastructure and climate research.

2. Working methods and techniques

For the development of a water parameter measurement system, the idea of a global system architecture (Figure 1) was first defined in order to decide on the technologies to be used in the development and design of the system. The main decision was related to the choice of the communication method between the measurement system, the Android application and the server(s) with the database, which created the need to choose between running a private virtual server (Virtual Private Server (VPS)) or using Docker containers to manage and develop the communication network between the system elements.

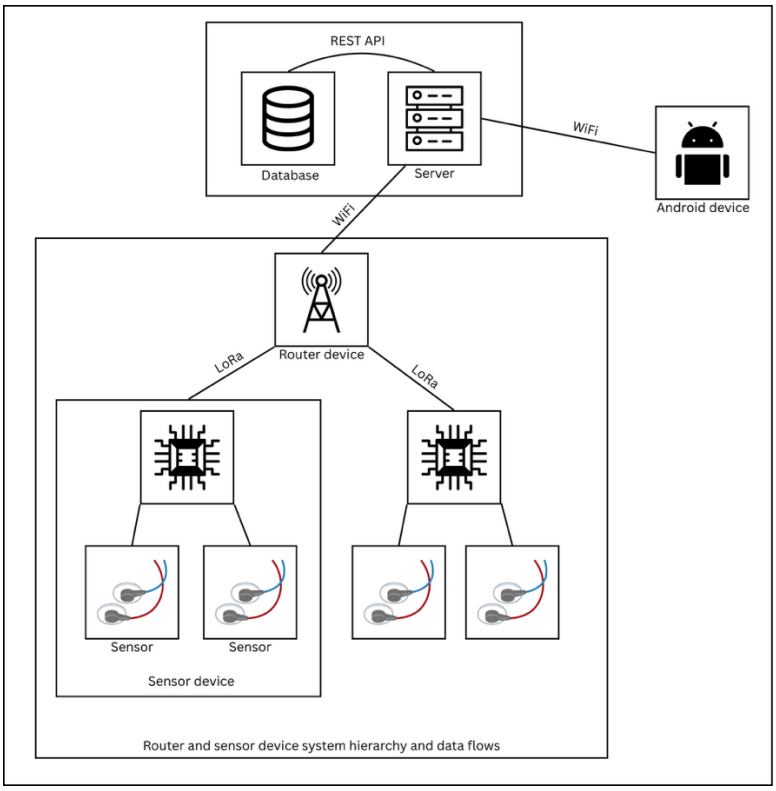

Figure 1: Global architecture of water parameter measurement systems (Canva, 2025)

Given that the metering system is a closed system that should not experience major changes in the number of connected devices, and taking into account that the main focus of the LoRa system is wireless modulation within the metering and transmitter system, it was decided to use a Docker container to run the server and development processes for this project. During research (Dima, 2022), it was found that the advantages provided by a VPS, namely the complete isolation of each virtual server, are not as important as the advantages offered by the Docker environment. The ability to quickly add containers to run separate parts

of the system and the relatively simple management and control procedures for each environment were decisive in choosing the Docker environment over the VPS option.

The microcontroller used on the measuring devices and the transmitter is an ESP32 with a built-in LoRa module. The devices are programmed in the C++ language within the Arduino environment because it provides a simple way to serially monitor the device's status and is compatible with the microcontroller used. The communication between the measuring device and the transmitter device is LoRa wireless modulation. LoRa communication was chosen because it is optimal for the type of measurement this system performs: periodic measurement of water parameters. Being suitable for LoRa modulation, the device turns on and consumes energy only when sending a message. It is not necessary for the LoRa module to be constantly on because it periodically sends data, making consumption much lower, and thus extending the device's battery life.

To start the server, a Node.js server is used, which runs inside a Docker container on a local port. The database has, also operates in its own separate Docker container, distinct from the server. The intention behind this architecture is for the server to manage all communication channels with the database, thereby maintaining data security and integrity by limiting the number of communication flows to the database to just one. Communication between the server and other system elements takes place over the Internet.

To display the data, an Android application was developed using the Kotlin programming language. The data is retrieved via the HTTP protocol by calling the REST service on the system server.

The android app was developed using IDEA by Jetbrains (Jetbrains, 2025), Docker containerization was run on Docker desktop app (Docker, 2025), and the codebase was developed using Visual Studio Code (VS Code) by Microsoft (Microsoft, 2025). Specifically, for embedded software development, we used the PlatformIO plugin for VS Code, which enables a cross-platform, multi-framework, and multi-architecture development environment for embedded systems, simplifying tasks like compiling, uploading firmware, and managing libraries. For creating tables we used Microsoft Office Excel (MS Excel) (Microsoft, 2025). For versioning, we opted for the classic option, GitHub (GitHub Inc., 2025), the link being https://github.com/mdesanic/zavrsni-rad-iot-lora.

[1]

[1] The schemes for the devices were designed and developed using Fritzing (Fritzing, 2025). For writing this thesis, I used LLM Gemini (Google, 2025)to lector my sentences and paragraphs, since English is not my first language.

3. System architecture

The system consists of three main elements: integrated meter and transmitter systems, a server that manages the data flow, and the Android application WaPamWatchdog. The integrated meter system is a single device consisting of an ESP32 microcontroller and two water temperature sensors. The transmitter device is simply an ESP32 microcontroller that serves to place meter data into a database via a server. Communication between these integrated systems is via a LoRa wireless module. The software solution for the meter and transmitter will be described in more detail below.

The server, which manages the data flow, is organized to run inside a Docker container and is accessed via a local IP address and port, specifically `127.0.0.1:3000` (IP_address:PORT). To access the server from the Android emulator, it was necessary to use the emulator's predefined "macro" address for connecting to the local computer (`10.0.2.2`). For communication from the transmitter device, the local computer's IP address is used. The server is run by the main application file, which provides RESTful services for data retrieval and recording. These REST services are organized into files dedicated to different data operations and are accessed via a specific path for server communication, in this case, `ip_server_address:port/path`. Database linking libraries have been created to handle server authentication before connecting to the database and to ensure secure data retrieval from it.

The database was launched via a MySQL Docker container, configured to handle its creation and management. Developing the database involved identifying the parameters to be monitored, normalizing data in each table for enhanced security and integrity, and creating an Entity-Relationship (ERA) model *(Figure 11: E-R Diagram*) along with an SQL script for its implementation within the MySQL environment.

The Android application is a simple tool designed solely for displaying data retrieved from the database. Developed in Kotlin, it uses the `okhttp3` library to make calls to the RESTful service on the system server. Each component will be described in more detail in the following text.

The core technology highlighted is the LoRa wireless module, and the majority of this work focuses on the system's integrated devices. The server and Android application primarily serve to visualize the data communicated between the meter and the transmitter via LoRa.

3.1. Docker

To understand Docker we need two definitions:

1. “*Docker is an open-source platform that enables developers to build, deploy, run, update and manage containers.”* (Susnjara & Smalley, IBM, 2024)
2. “*Containers are standardized, executable components that combine application source code with the operating system (OS) libraries and dependencies required to run that code in any environment*.” (Susnjara & Smalley, IBM, 2024)

Docker facilitates system development by running all necessary environments within containers. Its popularity stems from the increasing interest in "cloud programming," which emphasizes the development and deployment of numerous microservices to implement applications or systems. While software development engineers can create containers natively in Linux, Docker significantly accelerates this process by leveraging *Dockerfile* scripts for image creation and subsequent container instantiation.

3.1.1 Docker in comparison to virtual machines

The ability to containerize stems from the Linux kernel's capacity to isolate and virtualize processes, including the dynamic allocation of memory required for maintaining and managing a started process. Much like a hypervisor (Susnjara & Smalley, IBM, 2024) allows multiple virtual machines to share the resources of a single service processor and server memory, a container enables similar functionalities while consuming fewer resources. This also makes it easier to set up new Docker environments on different computers and run multiple distinct environments on the same machine, thereby making more effective use of available user resources.

3.1.1.1 Docker container system compared to other companies offering "cloud solutions"

The idea of containerizing parts of the system and running server calls in the “cloud” led to a decision on which system to use to make this possible. Among the various manufacturers and providers of “cloud solutions”, the shortlist for the system was:

- Microsoft Azure (Microsoft, 2025)
- Amazon Web Services (Amazon, 2025)
- Docker (Docker, 2025)

Microsoft Azure presents an attractive "hybrid cloud" option, facilitating easier integration of installed infrastructure, such as the meter and transmitter devices, with cloud-based infrastructure. However, its extensive integration with artificial intelligence and the dependency on Microsoft's services for product development led to the decision not to use Azure for this project's development process. Amazon Web Services (AWS) also appeared to be an interesting option, but the project does not require its simple product delivery options, nor will the system utilize the vast number of services, networked data centers, and global ecosystem features that AWS offers. In this context, Docker proved simpler, as the water metering system will not be so robust and complex as to necessitate greater management capabilities. Docker emerged as the superior choice primarily due to its ease of use and the team's prior knowledge of the technology. A significant additional advantage of Docker is its cost-free nature, unlike the other services mentioned.

3.1.2 Docker in the project

For this system to fully function, it was necessary to create two Docker containers:

- **iot-mysql** (database) – for starting and managing the database
- **iot-backend** (server) – server for data access through REST API

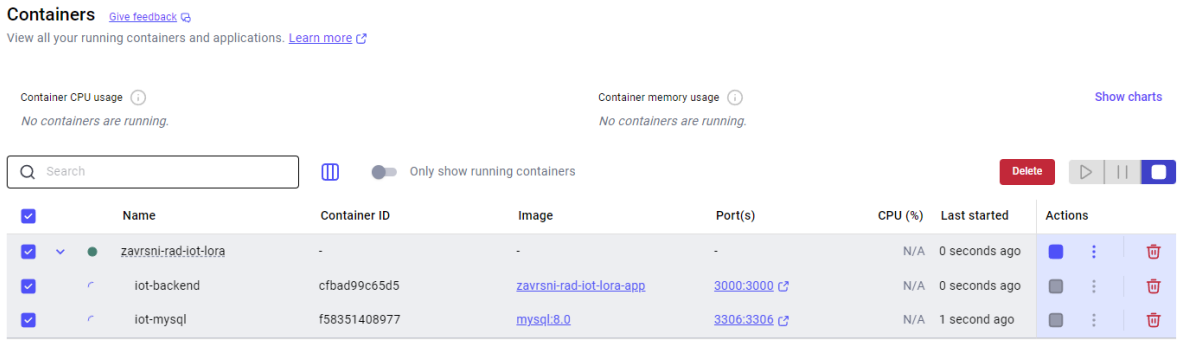

Figure 2: Docker containers

The *iot-mysql* container runs the database engine. It is built upon a foundational image downloaded from Docker Hub (Docker, 2025) and launched using a script within the *docker-compose.yml* file (Code 1). A service named *mysql* within this file defines the container for MySQL. The other settings determine how the image will run and on which port it can be accessed.

```yaml
version: '3.8'
services:
  mysql:
    image: mysql:8.0
    container_name: iot-mysql
    restart: unless-stopped
    env_file:
      - ./.env
    ports:
      - "${MYSQL_PORT}:3306"
    volumes:
      - ./mysql/init.sql:/docker-entrypoint-initdb.d/init.sql
    networks:
      - iot-network
  app:
    build:
      context: ./app
      dockerfile: Dockerfile
    container_name: iot-backend
    restart: unless-stopped
    ports:
      - "3000:3000"
    environment:
      - DB_HOST=iot-mysql
      - DB_USER=${MYSQL_USER}
      - DB_PASSWORD=${MYSQL_PASSWORD}
      - DB_DATABASE=${MYSQL_DATABASE}
    depends_on:
      - mysql
    networks:
      - iot-network

volumes:
  mysql-data:

networks:
  iot-network:
    driver: bridge
```

Code 1: docker-compose.yml

Similarly, the *iot-backend* container is started. After the initial successful installation of the Node.js image, a script is created, in this case the *Dockerfile* script, which is (Code 2). This is crucial because when the nodejsserver container image is started, the server also launches, requiring all necessary dependencies to be pre-installed. The Dockerfile uses *package.json* and *package-lock.jso*n files during the installation of these dependencies, ensuring proper startup and management of the Node.js server components.

```
1. FROM node:18-alpine
 2.
 3. WORKDIR /app
 4.
 5. COPY package*.json ./
 6.
 7. RUN npm install
 8.
 9. COPY . .
10.
11. EXPOSE 3000
12.
13. CMD ["node", "app.mjs"]
```

Code 2: Dockerfile for server container

Crucially, the volumes section is used to run a script. Specifically, *-./mysql/init.sql:/docker-entrypoint-initdb.d/init.sql* mounts the local init.sql file, located in the mysql directory, into the container's */docker-entrypoint-initdb.d/* directory. Docker automatically executes any *.sql* scripts placed in this directory when the MySQL container starts for the first time. This mechanism is used to initialize the database schema and populate it with any necessary default data. Finally, the service connects to the *iot-network*, ensuring it can communicate with other services within the same Docker network.

To run all Docker containers with minimal complication, the `function docker-compose up --build` is utilized. Executing this command in the console from within the directory containing the *docker-compose.yml* (Code 1) file initiates the construction of containers based on the images defined in the *Dockerfile* (Code 2) and the relative paths specified in `docker-compose.yml`.

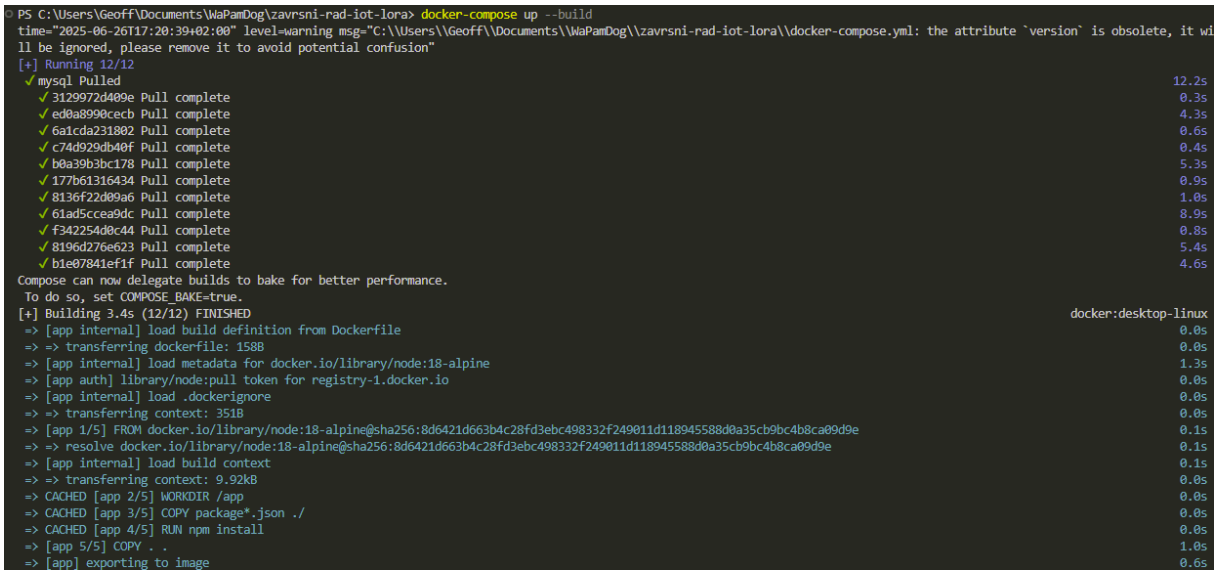

Figure 3: Logs after running docker-compose up --build

Following container construction from their respective images, all services necessary for normal functioning are activated. Furthermore, ports are configured for accessing the database and the server, enabling local devices on the same internet network to access endpoints via the internet, thus establishing communication between different parts of the system. The Android application and the transmitter will subsequently use the HTTP protocol to communicate with the server and access its RESTful services for database operations. Some parts of logs after running the script is shown in *(Figure 3: Logs after running docker-compose up --build).*

3.2. Transmitter device and sensor device embedded systems

The device system comprises two main roles: a sensor device and a data transmitter. The sensor device collects data and transmits it to a server on the Internet. This architecture was chosen to illustrate the following scenario:

Eg. *Sensor devices are positioned on the surface of water (e.g., a lake or sea), with their sensors submerged at specific depths, perhaps attached to a metal rod. Each sensor device can accommodate multiple sensors, though for this demonstration, a device with two sensors will be used. This device sends data via LoRa wireless communication to a separate transmitter device located on land. The land-based transmitter can be connected to a larger battery or a permanent power source, making it more suitable for sending data over the Internet, which consumes significantly more energy than the LoRa module. It receives data via its LoRa module and then uploads it to the database through RESTful access points on the server.*

The sensor device itself consists of an SX1276 ESP32 LoRa microcontroller on a development board, one 4.7K Ohm resistor, two DS18B20 Waterproof Temperature Sensors, and an antenna for establishing LoRa communication (Figure 4: Schematic of the project measuring device).

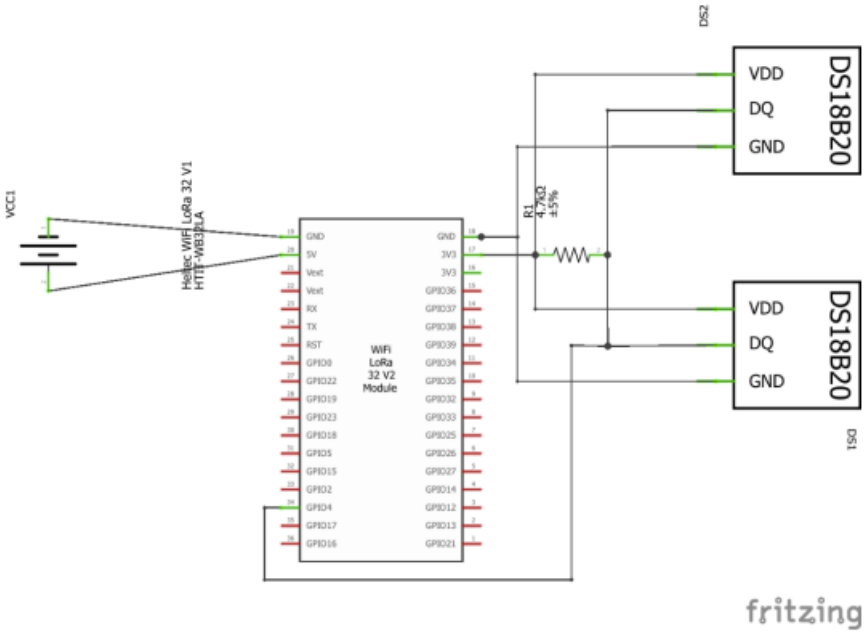

Figure 4: Schematic of the project measuring device (Fritzing, 2025)

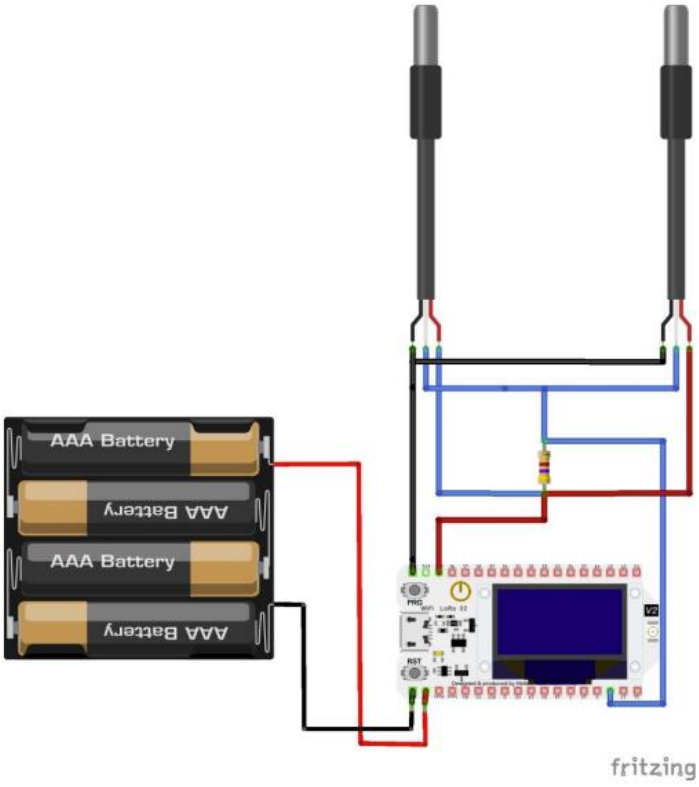

Figure 5: Schematic of a measuring device with real components (Fritzing, 2025)

The device that receives data via the LoRa module and sends it to the Internet has nothing but power supply connected to itself because the development board I use, LilyGO LoRa32 868/434MHz V1.3 - ESP32 - SX1276, has an antenna built in and a module already soldered on it.

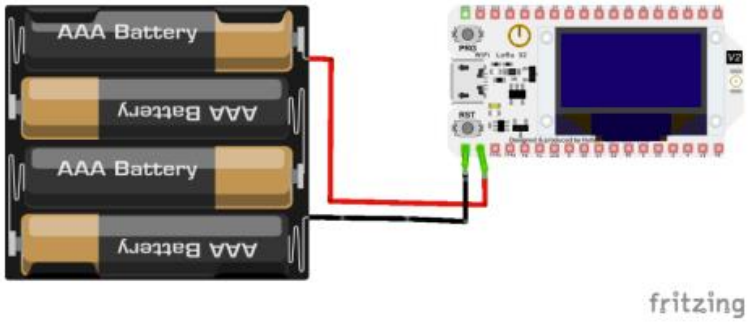

Figure 6: Transceiver device diagram with real components (Fritzing, 2025)

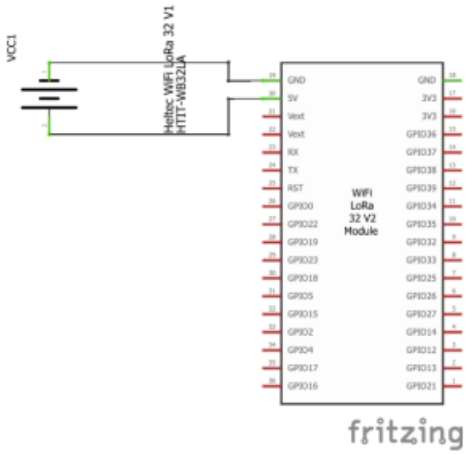

Figure 7: Transceiver device diagram (Fritzing, 2025)

3.2.1. LoRa wireless module

Long Range (LoRa) is a popular wireless communication standard primarily used in systems with low power requirements. It offers long-distance connectivity with very low power consumption, making it well-suited for Internet of Things (IoT) system development. Due to its open access and low deployment cost, LoRa is rapidly gaining traction in urban applications, smart metering, and aquaculture automation (for instance, a water measurement system can be utilized in shellfish farming). (Toro Betancur, Premsankar, Slabicki, & Di Francesco, 2021)

3.2.1.1. Modes of media space sharing that LoRa uses

Within an IoT system, information travels through the air. To successfully regulate this limited airspace and prevent signal and information overlap, LoRa employs various methods of sharing:

- **Frequency Division Multiplexing (FDM)**
    - The device uses different frequency channels to distinguish each one message
    - In Europe, LoRa uses the frequency 868MHz and 434MHz and divides it into more channel so that multiple messages can be sent simultaneously
- **Time Division Multiplexing (TDM)**
    - Each message is sent at precisely specified time intervals, limiting the chance of multiple messages being sent at the same time
    - LoRa does not allow any type of continuous communication

- **Spread Spectrum**
    - Using this method, a signal is sent at the same time from multiple end devices, but each signal has a specific structure, which allows the receiver to distinguish the real information from the noise

LoRa modulation utilizes Chirp Spread Spectrum, a technique distinct from classic coding methods. A "Chirp" (Compressed High Intensity Radar Pulse) refers to the type of signal generated by LoRa modulation, a concept originally employed in radar technology.

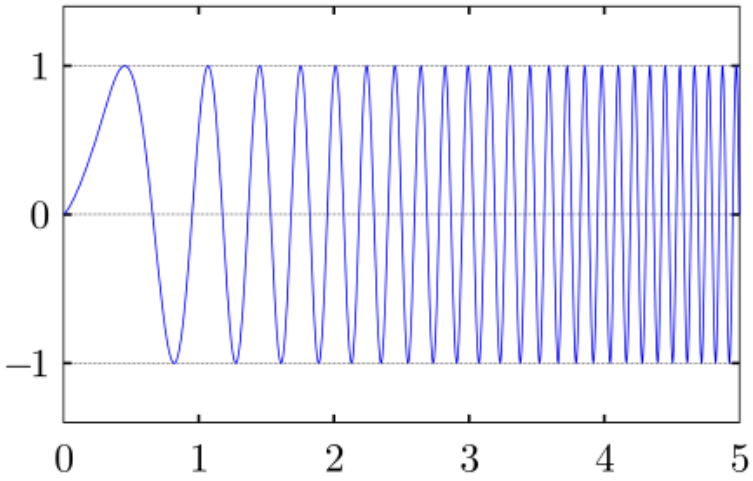

Figure 8: Linear chirp (Georg-Johann, 2010) (CC BY-SA 3.0)

3.2.1.2. Example of LoRa modulation (Montagny, 2022)

Take for an example this binary record:

- 0 1 0 1 1 1 0 0 0 1 1 0 0 1 0 0 0 1 1 0 1 0 0 1 1 0 0 1 0 0 0 1 1 0 1

We use SF10 LoRa modulation, separating the code into groups of 10:

- 0 1 0 1 1 1 0 0 0 1 | 1 0 0 1 0 0 0 1 1 0 | 1 0 0 1 1 0 0 1 0 0

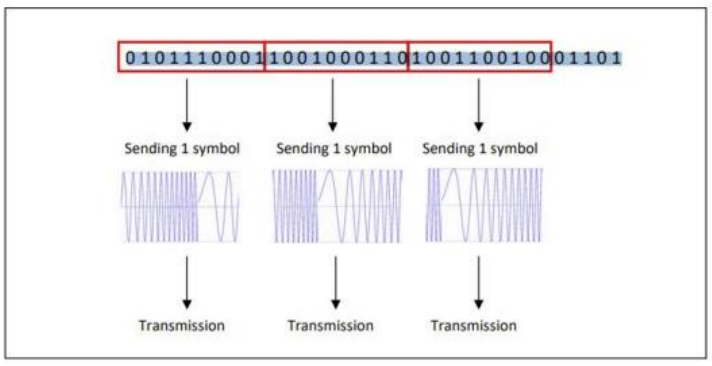

Figure 9: LoRa Chirp transmission example (Montagny, 2022)

3.2.1.3. Technical details

LoRa normally operates on the channels 434Mhz, 868 MHz and 915 MHz. Home and the final frequency of each signal is calculated by defining the central frequency with F_{ch}, and signal width (bandwidth (BW)), and the width is halved, subtracted from the center frequency for obtaining the initial frequency and add it to the central one to obtain the final frequency.

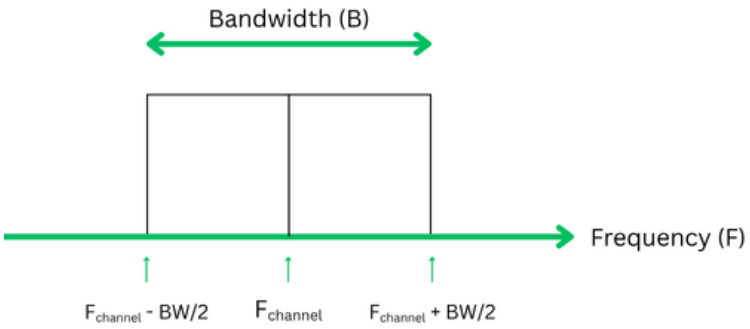

Figure 10: Visual representation of the starting and ending frequency of the signal (Canva, 2025)

Each symbol in the communication represents a set of Spreading Factor (SF) bits. Number of possible symbols is 2^{SF}. The signal transmission time is inversely proportional to the signal width and depends on SF.

$$T_{symbol} = \frac{2^{SF}}{BW}$$

- SF7: 1.024 ms
- SF8: 2.048 ms
- SF9: 4.096 ms
- SF10: 8.192 ms
- SF11: 16.384 ms

3.2.2. Bill of Materials

Part number	Manufacturer	Description	Quantity	Price (including VAT)	Total price	Documentation	Example
WiFi LoRa V3/V2	AITEX ROBOT	ESP32 Microcontroller with LoRa	1	11.08€	11.08€	AliExpress	AliExpress
Q310 V1.3 L	LilyGO TTGO	ESP32 Microcontroller with LoRa	1	19.50€	19.50€	TinyTronics	TinyTronics
4.7K Ohm	Gohjmy	4700 Ohm resistor	1	1.63€ (min. 100 pieces)	0.163€	AliExpress	/
DS18B20	HnxDIY	Waterproof sensors	2	4.88€ (min. 5 pieces)	1.95€	AliExpress	AliExpress
10cm 20cm 15cm	Dupont	Cables	4	1.87€ (min. 40 pieces)	0.187€	AliExpress	/
3 Slot AA Battery Holder	Blueendless	Battery holder	2	1.77€	2.56€	AliExpress	/

Table 1: Bill of Materials for the system

3.3. NodeJS

The technology chosen for running the server was based on personal familiarity, leading to the use of a Node.js server to host RESTful services. To initiate this system's server, it was necessary to include the express, fetch, mysql2, and node-fetch libraries, alongside the fundamental dependencies required for a standard Node.js server (Code 3: package.json). Within the same *package.json* file, a script ("npm start") is defined to launch the *app.mjs* file, which serves as the main server file. *This app.mjs* file contains the defined paths for all RESTful services and manages connections to the database.

```
{
  "name": "app",
  "version": "1.0.0",
  "type": "module",
  "main": "app.mjs",
  "scripts": {
    "start": "nodemon app.mjs",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "dependencies": {
    "express": "^4.18.2",
    "mysql2": "^3.6.5",
    "node-fetch": "^3.3.2"
  },
  "devDependencies": {
    "nodemon": "^2.0.22"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": ""
}
```

Code 3: package.json

3.3.1. app.mjs

I chose the *.mjs* extension for the server due to it being a more modern and current approach for Node.js servers compared to the traditional *.js* option. This choice was primarily driven by its standardized method for importing external files, which simplifies dependency

management for both external resources and internal software solutions like REST services and auxiliary modules.

```
1. import express from 'express';
2. import {
3. getAllLocations,
4. .
5. .
6. .
7. } from './api/RESTapi.mjs';
8.
9. const server = express();
10. const port = 3000;
11.
12. function startServer() {
13.   server.use(express.json());
14.
15.   // ✅ First define your actual routes
16.   server.get("/locations", getAllLocations);
17.   .
18.   .
19.   .
20.
21.   // ❗ Put 404 handler **after** all route definitions
22.   server.use((request, response) => {
23.     response.status(404).json({ desc: "no resources" });
24.   });
25.
26.   server.listen(port, () => {
27.     console.log(`Server started on port: ${port}`);
28.     console.log(`Available endpoints:`);
29.     console.log(`GET /locations - Fetch all locations`);
30.     .
31.     .
32.     .
33.   });
34. }
35.  startServer();
```

Code 4: app.mjs shortened version

The server launched by app.mjs (Code 4: app.mjs shortened version) is an **Express server** that uses **ES6 modules (.mjs)** to handle HTTP requests. The server is first initialized, and middleware is set up to parse JSON bodies of incoming HTTP requests.

In the startServer function, a **port** is defined on which the server will run. The server is accessible via the **local IP address (127.0.0.1)** and the designated **port (3000)**, making the server's access URL **127.0.0.1:3000**. This address serves as the **endpoint reference** for accessing all the features offered by the server.

Later in the text, it is explained how this address is used within an **Android application**, which slightly differs from standard interaction expectations.

The startServer function also defines the **routes** for accessing REST services, such as the path /database/locations, which triggers the getLocations function from the RESTapi.mjs module. Details about how the server interacts with the **database** are explained further in the document.

After defining the server startup function, app.mjs calls startServer(), which logs a message to the console indicating that the server has started on the defined port.

3.3.2. Other modules and services

The database.js module defines a **Database class** that manages the connection between the server and a **MySQL database** using the mysql2/promise library, which allows asynchronous operations with modern async/await syntax.

At the beginning of the file, the **database configuration** is defined using environment variables (process.env) to make the code adaptable for different deployment environments. If environment variables are not set, default values are used, such as localhost for the host and root for the user. The database name defaults to *water_temp_db*, and the standard MySQL port 3306 is used.

The Database class contains three main functionalities:

1. **connect()** – This asynchronous method establishes a connection to the MySQL database using the configuration provided. Upon a successful connection, a confirmation message is printed to the console. If the connection fails, the error is caught, logged, and rethrown.
2. **disconnect()** – This method safely closes the database connection. It checks if the connection exists, attempts to end it, and logs a success or error message accordingly. It also ensures the internal connection reference is cleared afterward.

3. **runQuery(sql, params)** – This asynchronous method executes SQL queries using the established connection. It accepts a SQL string and an optional array of parameters (for prepared statements). The method returns the resulting rows or throws an error if the query fails or the connection has not been established.

Finally, the Database class is exported using ES6 module syntax (export { Database }) so it can be imported and used by other parts of the application (e.g., REST API modules).

```
 1. import mysql from 'mysql2/promise';
 2.
10.
11. class Database {
12.   constructor() {
13.     this.connection = null;
14.   }
15.
16.   async connect() {
17.     this.connection = await mysql.createConnection(config);
18.     console.log('Connected to database!');
19.   }
20.
21.   async disconnect() {
22.     if (this.connection) {
23.       await this.connection.end();
24.       console.log('Disconnected from database!');
25.       this.connection = null;
26.     }
27.   }
28.
29.   async runQuery(sql, params = []) {
30.     if (!this.connection) {
31.       throw new Error('Database connection not established');
32.     }
33.     const [rows] = await this.connection.query(sql, params);
34.     return rows;
35.   }
36. }
37.
38. export { Database };
```

Code 5: database.js

databaseDAO.js functions as a Data Access Object (DAO) for the RESTful services, defining all necessary data operations. These operations include retrieving, deleting, updating, and recording data within the database. Each operation consistently follows a pattern: connecting to the database, executing a query, and then disconnecting from the database, utilizing the operations provided by *database.js (*Code 5: database.js*).*

```
async getAllLocations() {
        try {
            await this.database.connect();
            const sql = "SELECT * FROM Locations;";
            const data = await this.database.runQuery(sql);
            this.database.disconnect();
            return data;
        } catch (error) {
            console.error('Error in getAllLocations:', error);
            throw error;
        }
    }
```

Code 6: databaseDAO.js example

3.3.3. REST servisi

RESTService.js defines a set of REST services that enable communication between the Node.js server and a MySQL database. It uses the node-fetch module for HTTP requests and a custom DatabaseDAO.js module that handles all direct database operations.

Each REST service (e.g., getAllLocations (Code 7: REST API code for getAllLocations)) is written as an asynchronous function that accepts request and response objects. These services are non-blocking, meaning they use async/await to handle potentially long-running database operations without freezing the main application thread.

Inside each function:

- The response type is explicitly set to "application/json".
- A new instance of DatabaseDAO is created to interact with the database.
- A try-catch block is used to manage errors:
    - In the try block, data is fetched from the database using appropriate DAO methods.

- If successful, the response is returned with a 200 OK status and the data is serialized to JSON.
- If an error occurs, it's caught in the catch block, logged to the console, and a 500 Internal Server Error is sent back to the client along with an error message.

This REST-based architecture is chosen for its simplicity and ensures consistent and secure communication with the database. Only the backend has direct access to the database, reducing the risk of unintended access or modification. The overall scope of the REST API remains small, since the volume of data being stored and retrieved is relatively low.

To use a specific REST endpoint, the client simply calls the server's URL followed by the appropriate path (e.g., /database/getAllLocations), which maps to a specific service function.

```
1. export default async function getAllLocations(request, response) {
2.     response.type("application/json");
3.     let ddao = new DatabaseDAO();
4.     console.log(ddao);
5.     try {
6.         let locationsResponse = await ddao.getAllLocations();
7.         let locations = locationsResponse[0]; // Extract location data
from the first element
8.         response.status(200);
9.         response.send(JSON.stringify(locations));
10.     } catch (error) {
11.         console.error('Error while fetching users from the database:',
error);
12.         response.status(500).json({ error: 'Error while fetching users
from the database' });
13.     }
14. }
```

Code 7: REST API code for getAllLocations

3.4. Database

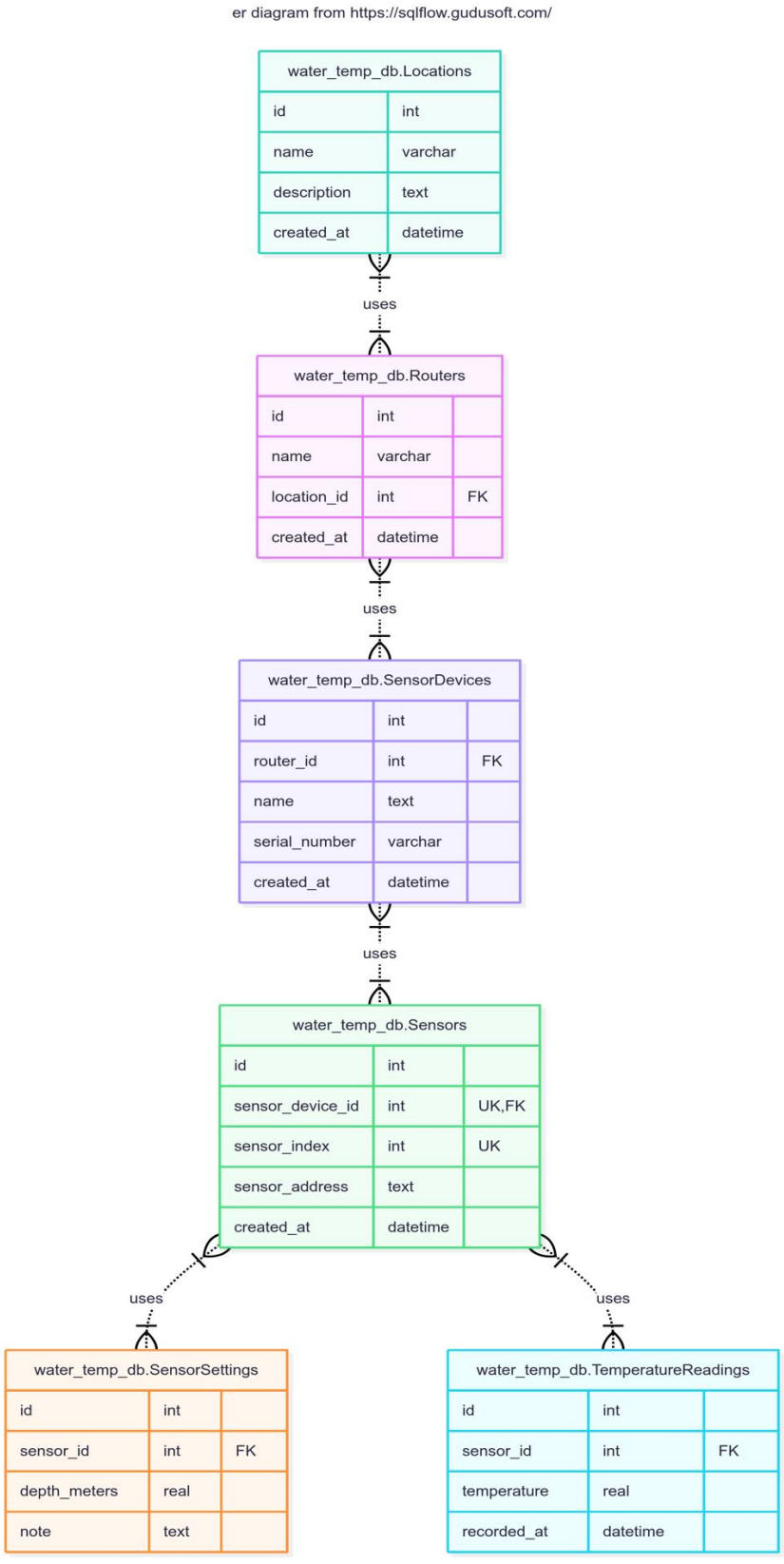

Figure 11: E-R Diagram (SQLFlow, 2025)

This database is part of a larger system for monitoring water temperature, designed for remote supervision of locations such as shellfish farms. Although this current iteration of the project includes only a limited number of physical measuring devices — due to hardware and budget constraints — the database has been fully prepared to support future expansion. The structure is modular and scalable, allowing seamless integration of additional devices, sensors, and monitoring sites.

At the core of the database is the goal of enabling centralized tracking of environmental conditions, specifically water temperature, across multiple physical sites. To achieve this, the system captures and organizes data from temperature sensors connected to sensor devices, which are in turn connected to router devices. These routers serve as intermediaries, uploading sensor readings to a central server over WiFi. Each component in the system is clearly represented in the database schema and logically linked to reflect real-world relationships between hardware elements.

The schema consists of several interconnected tables that form the foundation of the system's data layer. The `Locations` table represents distinct geographic or operational sites where equipment is deployed — for instance, a specific aquaculture zone. Each location is identified by name and optionally described, and each router is linked to one of these locations via a foreign key.

Routers, stored in the `Routers` table, are communication devices responsible for transmitting sensor data to the backend. Each router has a unique ID, name, and a reference to its assigned location. Routers are physically connected to one or more sensor devices, stored in the `SensorDevices` table. These devices are microcontroller-based units capable of managing multiple sensors. Every sensor device has its own name, serial number, and creation timestamp, and is linked back to the router it communicates through.

Each `SensorDevice` can host multiple sensors, which are stored in the `Sensors` table. A sensor is defined by its index (indicating its position on the device), its physical OneWire address (if applicable), and the device it belongs to. The `sensor_index` and `sensor_device_id` together form a unique constraint, ensuring that no two sensors are assigned the same index on the same device. This avoids addressing conflicts and allows fixed-position assignment — such as "Sensor 1 is always the shallowest sensor."

To capture sensor readings over time, the `TemperatureReadings` table records individual temperature measurements. Each reading references a specific sensor via a foreign key and logs the measured temperature along with the exact timestamp of the reading. This

structure enables time-series analysis of water temperature trends per sensor, per device, and ultimately per location.

In addition, the `SensorSettings` table is used to store configuration metadata for each sensor, such as the depth at which the sensor is submerged and optional notes. This allows the system not only to store raw temperature data, but also to attach meaningful context to each measurement — crucial for understanding environmental gradients in vertical water columns.

All of these tables work together to provide a normalized, logically structured representation of the physical system. One location may host multiple routers; each router can connect to multiple sensor devices; each sensor device supports multiple sensors — and each sensor may have hundreds or thousands of temperature readings over time. This relational model provides clarity, flexibility, and maintainability, making the system robust for both current needs and future scaling. Once the schema was finalized and the electronic designs of the routers and sensors were complete, it became much easier to determine which attributes needed to be tracked and how they relate to one another.

A wizard (SQLFlow, 2025) was used to auto-generate ER creation scripts based on the SQL script, which were then executed inside a Docker container configured via `docker-compose.yml`. The database is hosted on port 3306, and once launched, it was immediately ready for integration with the backend system

LOCATION (**location_id,** name, description, created_at)

ROUTER (**router_id**, name, location_id, created_at)

SENSOR_DEVICE (**sensor_device_id**, router_id, name, serial_number, created_at)

SENSOR (**sensor_id,** sensor_device_id, sensor_index, sensor_address, created_at)

SENSOR_SETTING (**sensor_setting_id**, sensor_id, depth_meters, note)

TEMPERATURE_READING (**reading_id**, sensor_id, temperature, recorded_at)

Figure 12: Relational schema

```
USE water_temp_db;

CREATE TABLE Locations (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE Transmitters (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    location_id INT,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (location_id) REFERENCES Locations(id) ON DELETE SET NULL
);

CREATE TABLE SensorDevices (
    id INTEGER PRIMARY KEY AUTO_INCREMENT,
    transmitter_id INTEGER NOT NULL,
    name TEXT NOT NULL,
    serial_number VARCHAR(255) UNIQUE NOT NULL,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (transmitter_id) REFERENCES Transmitters(id) ON DELETE CASCADE
);


CREATE TABLE TemperatureReadings (
    id INTEGER PRIMARY KEY AUTO_INCREMENT,
    sensor_id INTEGER NOT NULL,
    temperature REAL NOT NULL,
    recorded_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (sensor_id) REFERENCES Sensors(id) ON DELETE CASCADE
);

```

Code 8: SQL Script for creating the database 1/2

```
 1. CREATE TABLE SensorSettings (
 2.     id INTEGER PRIMARY KEY AUTO_INCREMENT,
 3.     sensor_id INTEGER NOT NULL UNIQUE,
 4.     depth_meters REAL,
 5.     note TEXT,
 6.     FOREIGN KEY (sensor_id) REFERENCES Sensors(id) ON DELETE CASCADE
 7. );
 8.
 9. CREATE TABLE Sensors (
10.     id INTEGER PRIMARY KEY AUTO_INCREMENT,
11.     sensor_device_id INTEGER NOT NULL,
12.     sensor_index INTEGER NOT NULL,
13.     sensor_address TEXT,
14.     created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
15.       FOREIGN KEY (sensor_device_id) REFERENCES SensorDevices(id) ON DELETE
CASCADE,
16.     UNIQUE(sensor_device_id, sensor_index)
17. );
18.
```

Code 9: SQL Script for creating the database 2/2

3.5. Android application

The Android application developed in this project functions primarily as a display interface for measurement results from the water parameter system. It represents a modern implementation, built using **Kotlin** and leveraging **Jetpack Compose** for its user interface. Unlike applications that might require complex architectural patterns for extensive user interaction, this application focuses on efficiently retrieving and presenting data, serving as a clear demonstration of the system's data flow from sensor to display. The development prioritizes reactive data handling and a modular structure.

3.5.1. Android manifest

The AndroidManifest.xml file serves as the foundational configuration for the Android application, instructing the Android system on its operational parameters. Within this project, the manifest (Code 10), (Code 12) defines essential properties such as the targeted Android API version, application icon, and theme. Crucially, it declares necessary permissions, most notably android.permission.INTERNET, which is vital for enabling network communication to fetch measurement data from the backend server. While external network security configurations like *network_security_config.xml* (Code 11) are common for handling secure connections, the core manifest ensures the application has the basic capabilities for launch and network access..

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3.     xmlns:tools="http://schemas.android.com/tools">
4.
5.     <application
6.         android:networkSecurityConfig="@xml/network_security_config"
7.         android:name=".MyApplication"
8.         android:allowBackup="true"
9.         android:dataExtractionRules="@xml/data_extraction_rules"
10.        android:fullBackupContent="@xml/backup_rules"
11.        android:icon="@mipmap/ic_launcher"
12.        android:label="@string/app_name"
13.        android:roundIcon="@mipmap/ic_launcher_round"
```

Code 10: androidmanifest.xml 1/2

```
 1.    android:supportsRtl="true"
 2.          android:theme="@style/Theme.WaPamWatchdog"
 3.          tools:targetApi="31">
 4.          <activity
 5.              android:name=".MainActivity"
 6.              android:exported="true"
 7.              android:label="@string/app_name"
 8.              android:theme="@style/Theme.WaPamWatchdog">
 9.              <intent-filter>
10.                  <action android:name="android.intent.action.MAIN" />
11.                                                              <category
android:name="android.intent.category.LAUNCHER" />
12.              </intent-filter>
13.          </activity>
14.      </application>
15.      <uses-permission android:name="android.permission.INTERNET" />
16.                                                      <uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE" />
17. </manifest>
```

Code 12: androidmanifest.xml 2/2

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <network-security-config>
3.     <domain-config cleartextTrafficPermitted="true">
4.         <domain includeSubdomains="true">10.0.2.2</domain>
5.     </domain-config>
6. </network-security-config>
7.
```

Code 11: network_security_config.xml

3.5.2. build.gradle.kts

For project build automation, the Gradle option was chosen. The key use of this file for build management with Gradle lies in including new dependencies, configuring the build, and setting other parameters.

The file defines several code blocks that further describe individual build items:

- **Plugins**: Indicate the application type (Android) and the programming language used (Kotlin).
- **Android**: After defining the application type, specific properties of the application are set.
- **Dependencies**: Define the dependencies required to run the application, such as the `okhttp3` library for making HTTP requests.

```
plugins {
    id("com.android.application")
    id("org.jetbrains.kotlin.android")
}
android {
    namespace = "com.example.wapamwatchdog"
    compileSdk = 34
    defaultConfig {
        applicationId = "com.example.wapamwatchdog"
        minSdk = 24
        targetSdk = 34
        versionCode = 1
        versionName = "1.0"
        testInstrumentationRunner =
            "androidx.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            isMinifyEnabled = false
                        proguardFiles(getDefaultProguardFile("proguard-
androidoptimize.txt"), "proguard-rules.pro")
        }
    }
    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_1_8
        targetCompatibility = JavaVersion.VERSION_1_8
    }
    kotlinOptions {
        jvmTarget = "1.8"
    }
```

Code 13: build.gradle.kts

3.5.3. Flow of application

The Android application serves as a display interface for measurement results, built with Kotlin and **Jetpack Compose** for the UI, following an **MVVM (Model-View-ViewModel) architecture** with a **Repository pattern**. This modern approach emphasizes modularity and reactive data handling.

```kotlin
// repository/LocationRepository.kt
package com.example.wapamwatchdog.data.repository

import ...

@Singleton
class LocationRepository @Inject constructor(
    private val apiService: ApiService
) {
    suspend fun getAllLocations(): List<Location> {
        return apiService.getLocations()
            .locations
            .map { it.toDomain() }
    }

    suspend fun getLocation(locationId: Long): Location {
        return apiService.getLocation(locationId).toDomain()
    }
}

class LocationRepositoryException(message: String, cause: Throwable?)
```

Code 14: Example of a repository in android app

The application's **Model Layer** distinguishes between clean **Domain Models** (e.g.,) and network-specific **DTOs** (e.g.,Code 17). Mapper functions (eg. Code 16) convert DTOs to domain models, abstracting network details.open window.

```kotlin
package com.example.wapamwatchdog.domain.model
data class Reading(
    val temperature: Double,
    val recordedAt: LocalDateTime?
)
```

Code 15: Example of a model in android app

```kotlin
@Serializable
data class ReadingDto(
    val temperature: Double,
    @SerialName("recorded_at") val recordedAt: String
)
```

Code 17: Example of DTO in android app

```kotlin
@RequiresApi(Build.VERSION_CODES.O)
fun ReadingDto.toDomain(): Reading {
    val parsedRecordedAt: LocalDateTime? = try {
        val offsetDateTime = OffsetDateTime.parse(this.recordedAt)
        offsetDateTime.atZoneSameInstant(ZoneOffset.UTC).toLocalDateTime()
    } catch (e: Exception) {
        println("ReadingDto.toDomain(): Error parsing recorded_at string '${this.recordedAt}', Error: ${e.message}")
        null
    }
    return Reading(
        temperature = this.temperature,
        recordedAt = parsedRecordedAt
    )
}
```

Code 16: Example of a mapper in android app

Data flow is unidirectional and reactive: **Data Source** → **Repository** → **ViewModel** → **UI**. User actions flow back: **UI** → **ViewModel** → **Repository** → **Data Source**. This ensures the UI updates dynamically

The View Layer, eg. ReportScreen (a Jetpack Compose Composable), observes data from the ViewModel (eg. Code 18) and handles user interactions. The ViewModel Layer, with ReportViewModel, manages UI-related data and state using Kotlin Flows (StateFlow), delegating data operations to the Repository. The Repository Layer, ReadingRepository, acts as the single source of truth, abstracting data sources and performing DTO-to-domain model mapping.

```
@HiltViewModel
class ReportViewModel @Inject constructor(
    private val readingRepository: ReadingRepository
) : ViewModel() {

   private val _reports = MutableStateFlow<List<Report>>(emptyList())
    val reports: StateFlow<List<Report>> = _reports.asStateFlow()

    private val _isLoading = MutableStateFlow(false)
    val isLoading: StateFlow<Boolean> = _isLoading.asStateFlow()

    private val _error = MutableStateFlow<String?>(null)
    val error: StateFlow<String?> = _error.asStateFlow()

    fun fetchReports(
        sensorDeviceId: Long,
        startDate: LocalDateTime,
        endDate: LocalDateTime
    ) {
        viewModelScope.launch {
            _isLoading.value = true
            _error.value = null
            try {
               _reports.value = readingRepository.getGroupedReadings(
                    sensorDeviceId,
                    startDate,
                    endDate
                )
            } catch (e: Exception) {
                _error.value = "Failed to load reports: ${e.message}"
                _reports.value = emptyList()
            } finally {
                _isLoading.value = false
            }
        }
    }
}
```

Code 18: Example of view model in android app

For Networking, the application uses Retrofit for declarative API calls and Kotlinx Serialization for efficient JSON parsing into DTOs. All network operations are asynchronous, handled by Kotlin Coroutines, preventing UI blocking.

The **UI Rendering** in **Jetpack Compose** uses composable functions like ReportScreen and efficient list rendering with **LazyColumn**. Data from the ViewModel is observed and used to dynamically build the interface, including local grouping of readings by date for display.

Dependency Management is handled by **Hilt**, which simplifies injecting dependencies like ViewModels and Repositories throughout the application. Basic **Error Handling** is implemented with try-catch blocks and error messages propagated to the UI via StateFlow.

4. Conclusion

The aim of this project was to develop an IoT system for measuring water parameters, designed to monitor conditions and transmit data to a remote location for server storage. Throughout this project, which focused on researching the Internet of Things, the primary challenge was identifying existing technologies and adapting them for a water parameter measurement system. Regarding hardware requirements, identifying the most optimal communication system for this type of IoT setup was crucial, and LoRa wireless signal modulation proved to be an excellent choice. The use of a powerful ESP32 microcontroller with a built-in LoRa module significantly streamlined the development of the hardware, largely due to its extensive documentation and the numerous open-source projects offering creative programming solutions.

Docker containerization of the server and database access points forms the backbone of this project's wireless internet communication. With a database designed in the MySQL environment, and dedicated Docker containers for running the database and a Node.js server for database access, the entire system deployment and internet accessibility process is encapsulated and simplified. An Android application, programmed in Kotlin, is used to display the collected data, while the integrated systems themselves were developed using C++.

The implementation of LoRa technology has proven highly successful in terms of communication efficiency, and Docker containerization has greatly facilitated system management and enabled scalability. The system has demonstrated practical applicability, particularly if enclosed in a waterproof housing. However, challenges remain. One significant challenge is optimizing energy consumption, especially for deployments in remote areas. Ensuring data accuracy requires the use of high-quality sensors and regular maintenance, which can increase the manufacturing cost of the device. Scaling the system to handle a growing volume of data presents another challenge. Despite these hurdles, the integration of modern technologies such as LoRa, Docker, and Android has resulted in a robust and reliable system that offers valuable information for decision-making. This system holds potential for wide application across various industries, contributing to environmental protection and the sustainable management of water resources.

Bibliography

Amazon. (2025). *Amazon*. Retrieved from Amazon Web Services: https://aws.amazon.com/

Betancur, V. T. (2021). Modeling communication reliability in LoRa networks with device-level accuracy. *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications.* IEEE. doi: https://doi.org/10.1109/INFOCOM42981.2021.9488807

Canva. (2025). *Canva*. Retrieved from Home - Canva: https://www.canva.com/

Dima, M. (2022). *Quora*. Retrieved from What's the difference between hosting websites directly on VPS and on Docker containers?: https://www.quora.com/Whats-the-difference-between-hosting-websites-directly-on-VPS-and-on-Docker-containers

Docker. (2025). *Docker*. Retrieved from The #1 containerization software for developers and teams: https://www.docker.com/products/docker-desktop/

Fritzing. (2025). *fritzing*. Retrieved from Welcome to Fritzing: https://fritzing.org/

Georg-Johann. (2010). *Linear-chirp.svg.* Retrieved from Wikimedia Commons: https://commons.wikimedia.org/wiki/File:Linear-chirp.svg

GitHub Inc. (2025). *Github*. Retrieved from GitHub: https://github.com/

Google. (2025). *Gemini*. Retrieved from Gemini: https://g.co/gemini/share/79cc840456dc

Jetbrains. (2025). *Jetbrains*. Retrieved from IntelliJ IDEA - The IDE for Professional Development in Java and Kotlin: https://www.jetbrains.com/idea/

Kotlin. (2025). *Kotlin*. Retrieved from Get started with Kotlin: https://kotlinlang.org/docs/getting-started.html

Microsoft. (2025). *Microsoft*. Retrieved from Azure: https://azure.microsoft.com/en-us

Microsoft. (2025). *Microsoft Office*. Retrieved from Get started with Microsoft products : https://setup.office.com/

Microsoft. (2025). *Visual Studio Code*. Retrieved from Your code editor. Redefined with AI: https://code.visualstudio.com/

Montagny, S. (2022). *LoRa - LoRaWAN and Internet of Things for Beginners.* Université Savoie Mont Blanc. Retrieved from https://www.univsmb.fr/lorawan/wp-content/uploads/2022/01/Book-LoRa-LoRaWAN-and-Internet-ofThings.pdf

Moore, G. E. (1965, April 19). Cramming more components onto integrated circuit. *Electronics*.

Oracle. (2022, May 9). *"What is IoT?"*. Retrieved from Oracle: https://www.oracle.com/internet-of-things/

PlatformIO Labs. (2025). *PlatformIO*. Retrieved from Your Gateway to Embedded Software Development Excellence: https://platformio.org/

SQLFlow. (2025). *SQLFlow*. Retrieved from SQLFlow E-R Diagram Generator from SQL: https://sqlflow.gudusoft.com/#/

Susnjara, S., & Smalley, I. (2024, June 6). *IBM*. Retrieved from What is Docker? | IBM: https://www.ibm.com/think/topics/docker

Susnjara, S., & Smalley, I. (2024, October 30). *IBM*. Retrieved from What are hypervisors? | IBM: https://www.ibm.com/think/topics/hypervisors

Toro Betancur, V., Premsankar, G., Slabicki, M., & Di Francesco, M. (2021). Modeling communication reliability in LoRa networks with device-level accuracy. *INFOCOM 2021 - IEEE Conference on Computer Communications.* IEEE.

Figure list

Table list

Code list

Attachments

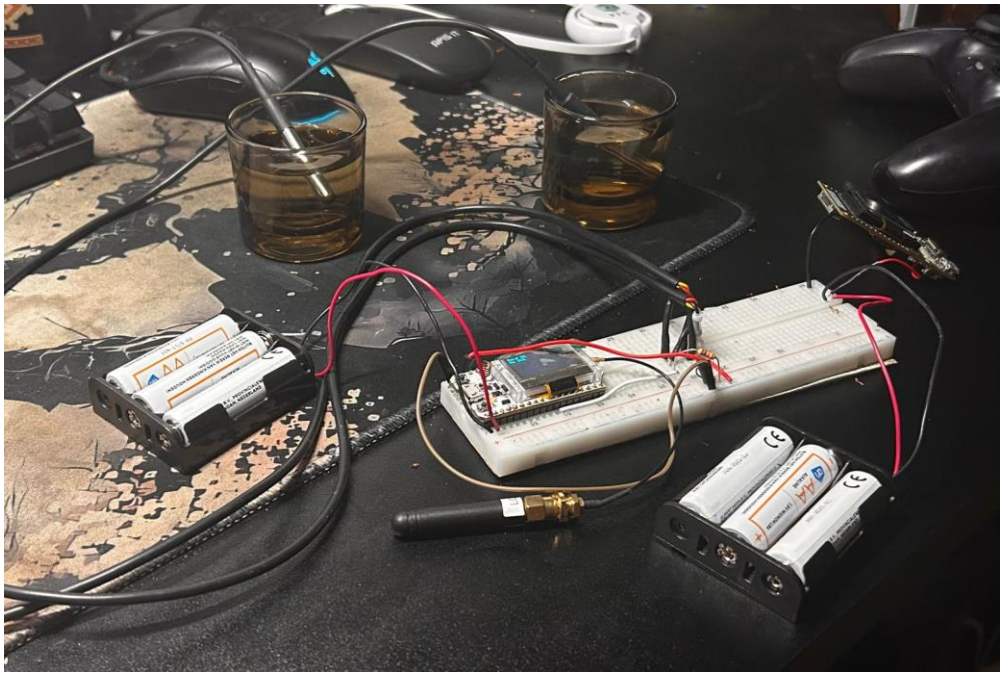

Figure 13: Embedded device in real life 1/4

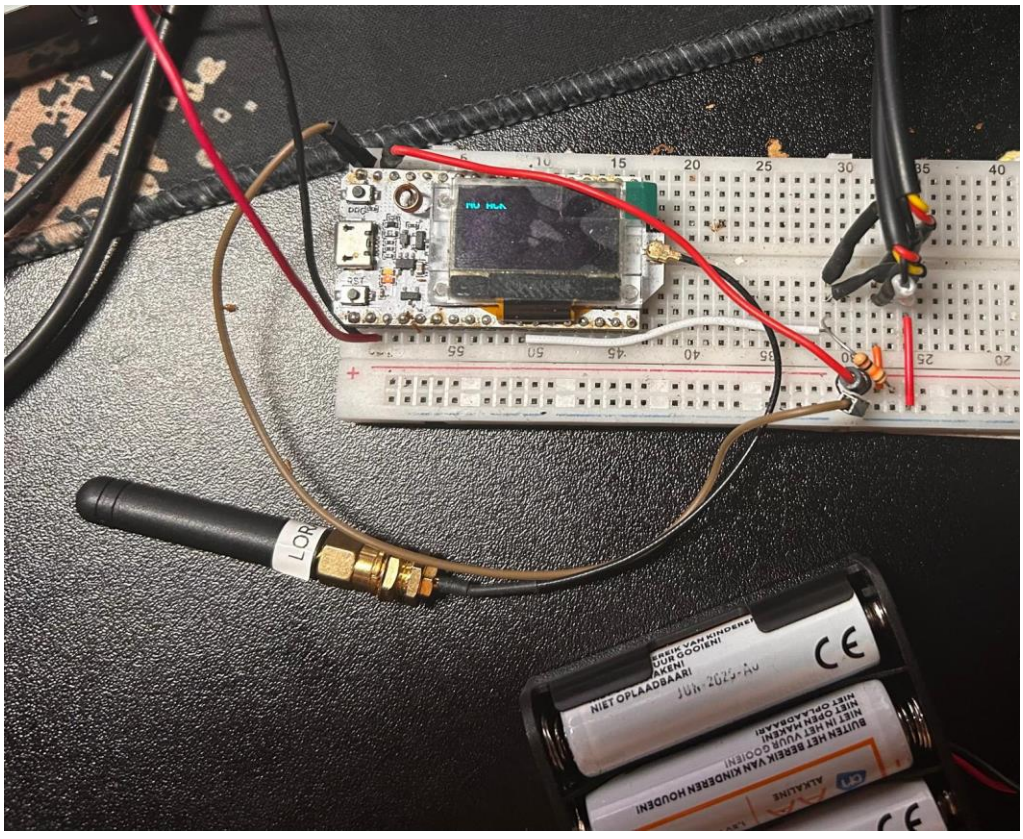

Figure 14: Embedded device in real life 2/4

Figure 15: Embedded device in real life 3/4

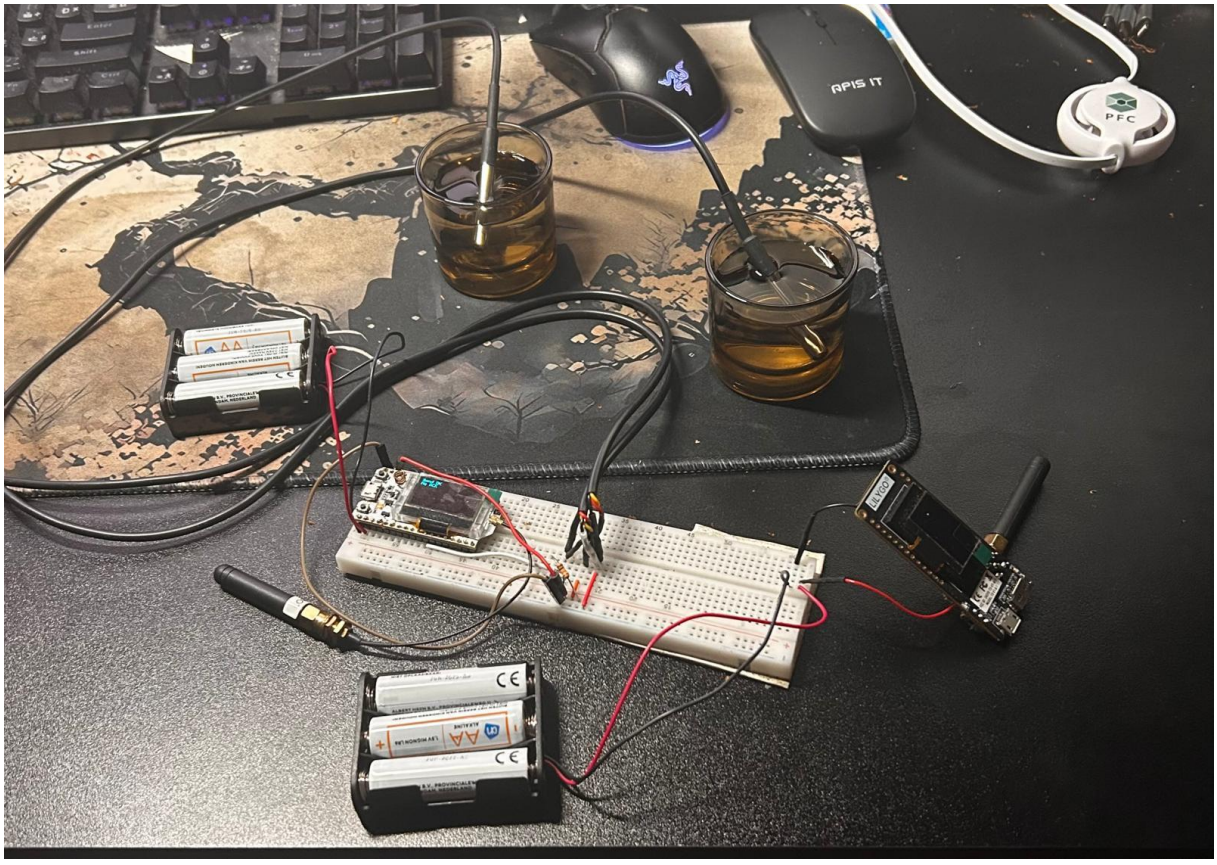

Figure 16: Embedded device in real life 4/4