OULUN AMMATTIKORKEAKOULU

WEB-BASED INTERFACE FOR REAL-TIME METEOROLOGICAL DATA VISUALIZATION: A CASE STUDY

Pedro Stradioto

Bachelor's thesis

Spring 2025

Degree Programme in Information Technology Engineering

Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology Engineering
Option of Web development

Author: Pedro Stradioto
Title of thesis: Web-based interface for Real-Time Meteorological Data Visualization: A case study
Thesis supervisor(s): Ville Majava
Term and year of completion: Spring 2025 Pages: for example, 24 + 5 appendices (or 1 appendix)

This work primarily focuses on the development of a web-based interface for visualizing real-time data transmitted from a weather balloon. For development purposes, the project was expanded to load and replay multiple files simultaneously, as well as to generate mock-up data according to specified criteria.
The practical aspect involved understanding, redesigning and expanding an existing Proof of Concept (PoC) created by the previous developer. The PoC was initially developed to test various web-development frameworks, leading to the conclusion that Preact.JS is the most suitable tool for this specific application.
The theoretical component will include a review of the technologies used, the coding techniques used, the data capture and processing methods, and the challenges encountered during the project.
The research methodology used a mixed approach that included both theoretical and empirical elements. Theoretical research involved studying books and articles, as well as consulting internal documentation on the hardware and software in use. Empirical testing and knowledge were gained through hands-on experiments and collaboration with experienced developers.
In addition, well-known and commercially available Artificial Intelligence (AI) technologies were occasionally consulted to enhance the research process.

Contents

GLOSSARY

AES Advanced Encryption Standard

AGL Above ground level

AI Artificial Intelligence

KML File format used to display geographic data

LED A semiconductor diode which glows when a voltage is applied

PBL Atmosphere part that is directly influenced by the earth's surface

PoC Proof of Concept

SoC System-of-Chip

UI User Interface

UX User Experience

VR Virtual Reality

1 Introduction

Data visualization refers to the graphical representation of information and data through charts, plots, infographics, animations, and even Virtual Reality (VR) simulations. In our data-driven world, where decision-making increasingly relies on collected data, the importance of data visualization has grown exponentially. After raw data is processed, many decisions will be made based on AI models within automated frameworks. However, these decisions will still require occasional validation from human peers. This human-data interaction underscores the importance of exploratory works like this. Data visualization allows new ways to interact with both raw and processed data, enabling humans to recognize trends and patterns that help in understanding the broader context and higher-level scenarios in which these data are involved.

The relevance of data visualization has not gone unnoticed by both large and small tech players and their stakeholders, leading to the creation of numerous new applications and libraries to meet the growing demand. Currently, there are options to suit all preferences and needs, from widely known programming languages like Python to veteran favorites like C++, and from well-established companies like Microsoft to new players like Grafana Labs.

The vast array of available frameworks and services is both a facilitator and an initial challenge for companies. Choosing the right software is the first step in any new project in modern companies. The numerous variables to consider, such as the project's purpose, data characteristics, system requirements, and licensing costs, among others, present a real challenge in finding the best fit.

Fortunately, this work builds on the diligent efforts of a previous main developer and expert developers at Sparv Embedded AB, who already narrowed the scope to a few tools considered the best fit. To limit the initial project footprint, a lightweight alternative to , known as , was used.

This addressed many of the initial variables, but it did not exempt the current developer from studying the subject to understand the decisions made and exploring the best approach among the various possible paths within the current framework. Questions such as how to divide components for better modularity, which libraries to

use for each component, evaluate which compiler to use for each component, and how to optimize the bundle size, particularly for those components that significantly impacted the performance.

Dealing with an ongoing project also brings the challenge of understanding and documenting changes. This involved not only the software project documentation but also documents related to data stream functionality and the hardware of the devices. Although further interactions with the devices that transmit the data and the data transfer protocols were not necessary within the project's scope, understanding these aspects was crucial.

The main interactions were through `.sounding` files, both recorded and artificially generated by a developer tool. The primary challenge was to carefully read and understand the parameters that determine each data point. Additionally, the frequency at which each data point was transferred was crucial to display updated information for the user. The data context component was already partially implemented, which facilitated further development, but it required a complete restructuring to handle new requirements, especially regarding the management of multiple `.sounding` files.

As for the frontend and the details of what data to present, the design aimed to leverage the main advantages of modern web frameworks, such as reactivity and portability. The primary concern was to keep the data display efficient and lightweight, where played a crucial role. The developer aimed to explore the best possible User Interface (UI), seeking a logical and lightweight UI. The displayed information and tabs to divide the application were mainly determined by trying to emulate the old desktop version of the application.

In summary, data visualization is critical in our data-driven world, it enables automated and human decision-making by providing intuitive ways to understand complex data. The growing demand for effective data visualization solutions has led to the development of infinitely tools and frameworks, each with its own strengths and challenges. By complementing the work of previous developers and carefully utilizing their selected tools, such as , this project aims to create a lightweight, efficient, and user-friendly interface for data interaction. The following sections will discuss the methodologies, challenges, and solutions implemented in this project, providing a comprehensive overview of the development process and its outcomes.

2 Background and Related Work

2.1 Overview of Previous Academic Works

Previous academic works targeting Windsond, `https://sparvembedded.com/products/windsond`, last accessed: 23 April 2025, aimed to describe and compare Windsond's capabilities in terms of data accuracy, recovery reliability, and deployment system with its competitors. For instance, (Quentin et al. 2019) focused on whether Windsond "has answered the call for less expensive but accurate reusable radiosonde...for boundary-layer observations collection."

Other studies explored the use of Windsond as a complementary tool to radars, enabling grounded observers to collect data and understand the factors leading to meteorological phenomena, rather than merely observing them (Markowski et al. 2018). These works are of marginal interest for the work developed here, as they help better understand the product, its use cases, and the in-field necessities that justify its development.

2.2 Overview of Previous Development Assessment

The work of the previous developer was documented in an internal report. It consisted of evaluating different modern web frameworks and developing PoC projects with them. The main idea was to assess the functionalities and data visualization libraries to decide on the best fit in terms of performance. A strong emphasis was placed on the reliability of performance as the work scaled up the number of functionalities.

Three widely used web development frameworks were tested: `Angular`, `Preact`, and `Vue.js`. From the start, `Vue` and `Preact` appeared as strong candidates due to their "very light core build-sizes" (Stameus 2024). The learning curve for users was also a strong factor in considering these, as Angular seems to have a longer one. After a promising start, the developer experienced problems with Vue handling the arrival of large amounts of real-time data, noting that `Preact` seemed to handle it better. The developer documentation noted the possible subjective nature of this decision, since his familiarity with `React.JS` may have facilitated his use of `Preact`. It is pru-

dent to make the same notation in this work.

Regarding the graph visualization libraries, `Chart.js`, `D3.js`, and `Plotly.js` were compared. It is noted that `Chart.js` and Plotly operate at a higher level than `D3.js`. Among the reasons for choosing Plotly was that it did not have the disadvantages of `Chart.js`, which uses canvas to render the charts, making them not natively reactive and unable to be formatted with CSS, among other faults. Plotly, on the other hand, can use SVG to render, which makes each element in the plot individual DOM nodes, allowing interaction and modification without giving coordinates. Plotly was also built on `D3.js`, requiring less manual settings than its ”predecessor.”

When analyzing the map libraries, the candidates were MapLibre, Pigeon Maps, and Leaflet. In this case, all three were very strong tools that could theoretically fulfill the project necessities.

Leaflet and Pigeon were both lightweight and possess sufficient tools for the project scope, a 2D map visualization tool that allowed the plotting of the flying path and relevant marks for the observers. The downside of Pigeon was the lack of documentation, with Leaflet being far superior in that regard at the time of the evaluation. MapLibre was a powerful library allowing for 3D visualization, camera animations, and more, being eliminated not for any flaw but for its inadequacy with the scope of the project itself.

2.3 About This Work

This work will not focus on atmospheric concepts or the reliability of data collection and hardware functionalities, as previous studies have already addressed these aspects. Instead, it will delve into building an understanding of the product and its use cases to expand the initial PoC while keeping a question in mind: how the user experience (UX) of the product could be improved.

Visualizing the collected data to better understand atmospheric phenomena is a common goal among researchers, clients, and software developers involved in the project. Fortunately, advancements in modern frameworks provide numerous options in this field. Therefore, data visualization is a strong focus in the practical work. Of course, UX goes beyond what is seen by the user in a well-designed frontend. Expanding the virtual environment of the tool will be essential for this improvement.

These improvements can be achieved by migrating the project to modern web frameworks, making it more portable and easily connected to cloud services (Wilson & Hawksworth 2019).
After considering these aspects, it seems obvious that the decisions regarding the use of `Preact` for the frontend, with Plotly (Inc. 2016) and Leaflet (Agafonkin & contributors 2011) libraries for data visualization and map functions respectively, made and documented by the previous developer, make sense. Future modifications were made with that in mind.

3 System's overview

The data visualization project was initiated to meet the needs of Sparv Embedded's new generation of products, particularly the Windsond S2 and its radio receiver, Sparv Receiver RR4. The RR4 receiver has been updated to enhance connectivity options, integrating all features of the previous model into one and receiving new functionalities for saving and interacting with data.
The Windsond S2 has been redesigned to leverage the experience in product development and to accommodate new use cases, such as operating near the ocean. Reusability is a key factor, so the development of the Windsond S2 and its receiver RR4 occurred following an idea of how the workflow of releasing and reacquiring the sond would work. That is described and illustrated below.
The workflow begins when the observer positions themselves at the launch point. The sonde, attached to a helium balloon, is powered on and searches for a GPS connection. The GPS status is displayed in the application (e.g., OK or Ready to Launch). Once the sonde is released, it enters "Ascent mode," during which data is recorded. The trajectory can be monitored in real-time via the application. At a predefined altitude or upon receiving a ground command, the sonde detaches from the helium balloon and enters "Descent mode," descending with its velocity controlled by a parachute. Upon reaching the ground, the sonde switches to "Recovery mode," conserving energy and emitting a beep to assist observers in locating it.

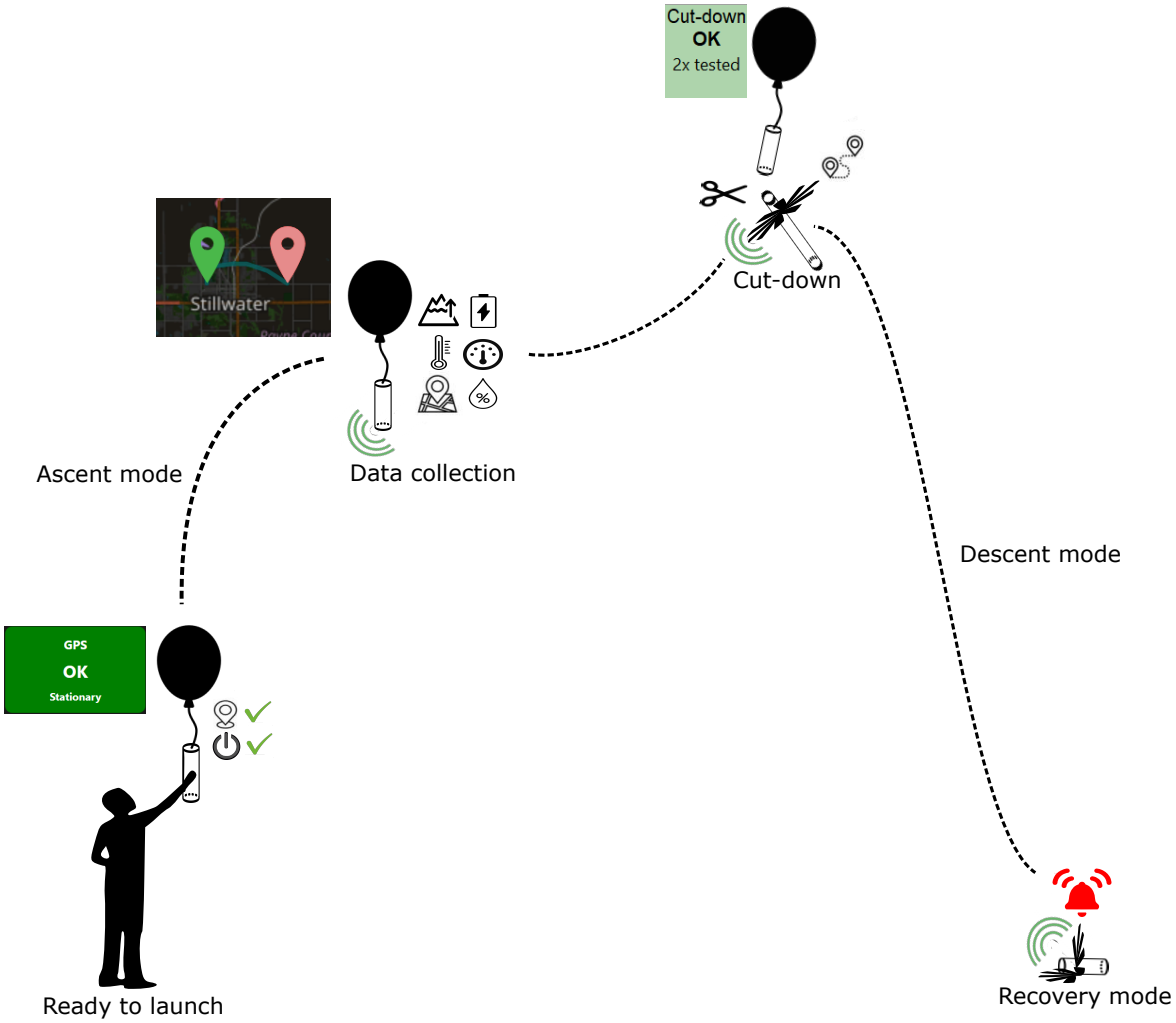

Figure 1: Expected deployment flow

3.1 Hardware components

3.1.1 Sparv Receiver RR4

The Sparv Receiver RR4 is designed for use with Windsond S2 sondes. It is a robust and weather-resistant device, featuring protective connector caps. With an IP65 rating, it is completely dustproof and can withstand exposure to low-pressure water. The RR4 connects to the Windsond and receives data via Long Range (LoRa) radio modulation (Augustin et al. 2016).

Sparv's radio receiver effectively utilizes the ESP32 chip, a System-on-Chip (SoC) product that supports the integration of various sensors, components, and wireless technologies (such as WiFi and Bluetooth) for embedded devices.

Previous receiver models, such as the RR1-250, RR2, and RR3 Ethernet, had spe-

cific connection methods between them and the observer's devices, making the choice of device dependent on the receiver model (see diagram in item 10.1.1 from the appendices). The RR4 introduces a versatile system that allows devices to connect to the receiver in multiple ways, including wireless, Bluetooth, and USB cable. A standout feature is its ability to serve a web application via the microcontroller, enabling multiple users to connect using laptops, tablets, or smartphones to visualize real-time data.

Additionally, the RR4 includes a micro SD slot, allowing captured information to be stored, thereby enhancing data storage capabilities and enabling backups.

A diagram illustrating the hardware involved in the data collection flow and their connections is attached below.

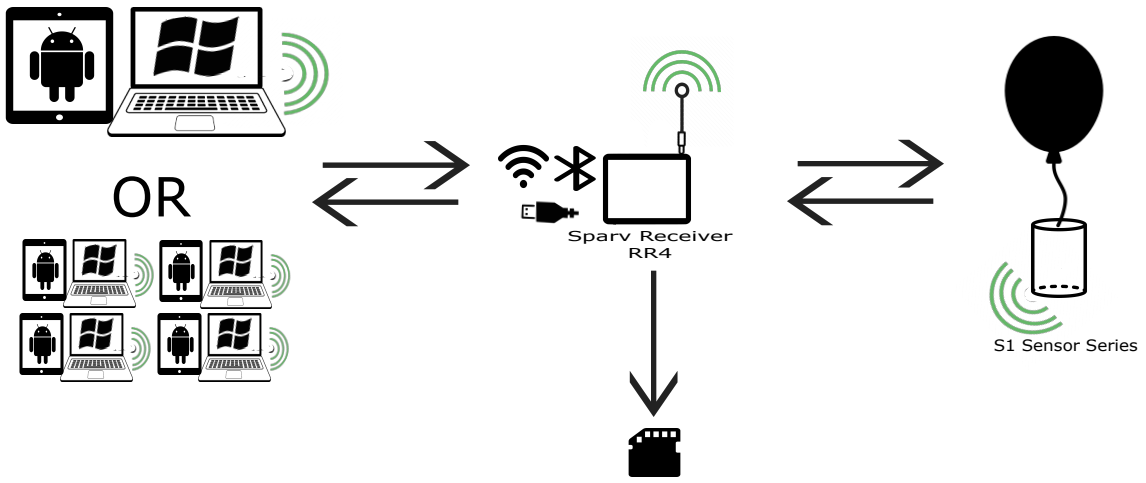

Figure 2: Workflow of Receiver RR4

The battery can keep the receiver active for up to 20 hours in untethered mode and can be recharged via USB. A short cable connects the RR4 to the Windsond S2, allowing it to charge the sonde, facilitate GPS acquisition using the receiver's built-in GPS, and synchronize radio settings. The RR4 also features a built-in barometer to verify the sonde's pre-launch pressure readings, an audible buzzer to indicate radio reception quality, and three status light-emitting diodes (LED) to provide feedback to observers. Additionally, it offers internal expansion slots for add-ons and the capability to install firmware updates on-site.

3.1.2 Windsond S2

The Windsond S2 is designed to be a compact, lightweight, and cost-effective reusable radiosounding system for acquiring real-time meteorological data of the planetary boundary layer (PBL) (e.g., wind velocity, temperature, humidity, etc.).

Figure 3: Windsond S2 after cut-down

Figure 4: Windsond S2 packaging

The reduced costs are not only reflected in the individual sonde price but also in the overall operational cost. Due to its smaller and lighter design compared to its predecessor, it requires only 30 liters of helium to be deployed and reach its desired altitude, up to 8 km above ground level (AGL). The ascent rate can be adjusted according to the quantity of helium but averages around 2 m/s.

Improvements over the Windsond S1 include a quicker GPS fix, a battery with better cycle life, and data collection capabilities in rainy conditions or near water (see picture in item 10.1.2 from the appendices).

The collected data in the S2 is stored internally, allowing for post-flight data analy-

sis and ensuring data redundancy. It possesses sensors for temperature, humidity, pressure, GPS, and voltage monitoring. Beyond embedding sensors, it utilizes algorithms to correct data with offsets and generate new types of data, which will be discussed later.

3.1.3 Hardware Suitability for Sensitive Applications

The hardware stack consists of a versatile radio receiver paired with a reliable, user-friendly sonde, enabling the system to be used in challenging weather conditions and in stressful situations.

The developed product extends beyond academic research or routine weather monitoring. Other use cases include firefighting during wildfires and military applications, specifically for artillery specialists. Such sensitive uses can be securely managed over LoRa communications, supported by encryption techniques, like AES-128 encryption (LoRa Alliance 2017) or physical layer security techniques (Porambage et al. 2021).

Experiments have been conducted with the product in this direction, and a variant of this product can be found in companies offering military sensor capabilities, `https://www.hensoldt.net/products/encrypted-weather-data-acquisition-system?utm_source`, last accessed: 29 April 2025.

3.2 Software stack

Due to the infinity of variables one observer can deal when collecting data in-field, selecting the correct software development tools to deliver an ecosystem that balances efficiency, performance and versatility in all its components can be challenging. To address this challenges it was suggested an previous assessment and PoC development, as we saw in the item 2.3, The software development frameworks and libraries assessed as more suitable for the project, `Preact`, and the libraries Plotly and Leaflet where selected.

3.2.1 Frontend

The framework `Preact`, a lightweight version of `React`, was chosen to address the challenges of creating a high-performance application capable of handling large amounts of real-time data, even on resource-constrained devices.

`Preact` focuses on maintaining a smaller size, with just 3KB when gzipped, and achieving better performance. It offers faster initial rendering compared to `React` (Wilson & Hawksworth 2019). To achieve this size and performance, `Preact` has fewer features. For instance, it supports JSX, Virtual DOM, and more, but there are trade-offs, such as limited event handling options compared to `React`. This makes `Preact` closer to how the browser operates, enhancing compatibility with ES Modules and no contextTypes are needed. It is possible to use `preact/compat` to achieve full compatibility with `React`. If performance becomes a concern in legacy projects, the `preact/compat` layer can serve as a drop-in replacement for `React`, minimizing migration efforts. Later, this text will also discuss Preact's compatibility with modern build tools and compiling workflows, such as Vite, the compiler used in this project.

`React`, on the other hand, is larger, with 30KB when gzipped, and includes more built-in features, making it slower and bulkier compared to `Preact`. It offers a wide array of tools, such as synthetic events, custom hooks, context, and specialized components.

`React` is a mature system with a vast ecosystem and is commonly used in many modern applications. The large community surrounding `React` is a significant advantage, providing extensive support and resources.

Both `React.js` and `Preact.js` use the concept of virtual DOM to efficiently update rendered content by tracking changes in the data set (Wilson & Hawksworth 2019). Understanding what a DOM node is, that is, individual elements, attributes, and text within the HTML structure, present in both the virtual and real DOM, is crucial.

These concepts explain how UI elements can `react` to events like clicks, hovers, or tooltips. This reactivity enables dynamic data display and allows users to interact with specific parts of the dataset. Most importantly, when these elements are accessed either by user input or programmatically, they can expose the data they refer to.

Aspect	Preact	React
Size	3KB (gzipped)	30KB (gzipped)
Performance	Faster loading and less memory	Good performance but slower than Preact
Events	Native browser events	Synthetic event system
API	Smaller, fewer features	More features (context, advanced hooks...)
Compatibility	Has ‘preact/compat‘ for React compatibility	Full React ecosystem support
Ecosystem	Smaller ecosystem	Larger ecosystem

Table 1: Comparison of Preact vs React

3.2.2 Visualization layer

The challenge was to create a software to visualize the data using graphics and maps, which would improve clarity. This introduces challenges in dynamically generating and rendering component elements as each node must be accessible, interactive, and capable of exposing its data efficiently.

The concept of Virtual DOM allows efficient updates for components state and layout logic, and the selected libraries make great use of this because they use embedded lifecycle methods to ensure smooth updates.

The Data visualization was enhanced with the integration of two libraries, Plotly for dynamically plotting in graphics and Leaflet for generating interactive maps, with allows the users to observe the data collection in real-time, or replays of it, though different layouts that would focus on specific factors. Users would be able to switch seamlessly between the layouts, with essential data always remaining visible.

Plotly and Leaflet have significant benefits when integrated into the `Preact` environment due to its component-based approach and reactive rendering. The Virtual DOM pattern allows UI components to update when data changes occur at runtime. This enables charts, map markers, and displayed data to be managed with a considerable reduction in re-renders, always when unnecessary full redraws are avoided, since these libraries handle their own rendering outside the Virtual DOM. In such cases, performance is improved, especially when dealing with real-time data visualization.

The modular design allows the construction of reusable components. For example, Plotly charts can be embedded in reusable components that `react` to state or prop changes, making it easier to display data dynamically for the user, including the pos-

sibility for developers to choose the best way to present it based on the dataset size. Similarly, Leaflet maps can be embedded in components so users can use inputs to zoom in, zoom out, search for their current position, or view hover information on markers.

3.2.3 Other software considerations

In the hardware layer we have a ESP32 microcontroller, programmed in C++, the choice of these technologies is due to its compatibility with wireless technologies (LoRa, WiFi, etc.) and efficient low power operation (Augustin et al. 2016). The embedded part, although very interesting, was not a direct part of the work developed in this work.

As a side note, it is mandatory to explain that a Windows desktop applications already exist and is commercially used with previous products versions.

The combination of these software development frameworks, programming languages and hardware allow for the delivery of a cohesive, resilient and energy wise system, ideal for operations in-field.

3.3 Data flow and Pattern

The data flow refers to the transmission of information between the devices within this system. It begins with the sonde capturing raw environmental data through its sensors. This data is then transmitted to a receiver connected to a mobile device, where the software processes it within the data processing component.

Each time the Sonde's environmental sensors acquire a new data set, it is transmitted in real time via a LoRa connection. LoRa enables the transmission of small amounts of data over long distances (Augustin et al. 2016), which is more than sufficient for sending sensor measurements, telemetry data (such as battery voltage and radio signal quality), and metadata, especially when the data is binary-encoded or compressed to minimize bandwidth usage.

The transmitted message is received and decoded by the receiver. The receiver then forwards the message to the mobile device using its available communication channel, which varies depending on the receiver model. For example, it could be a

standalone device tethered via USB or Ethernet, a USB adapter directly connected to the mobile device, or a standalone unit that creates a Wi-Fi network to host a web application like the one used in this project. The Sparv RR4 receiver can even store raw data on an SD card before forwarding it to a connected device. Once decoded, the data is passed to the software system through a socket interface when within range.

Within the web application, the DataContext component is responsible for parsing incoming payloads. This component handles real-time data calibration, computational functions, and the control of sonde states (Wilson & Hawksworth 2019). A diagram below illustrates the described data flow.

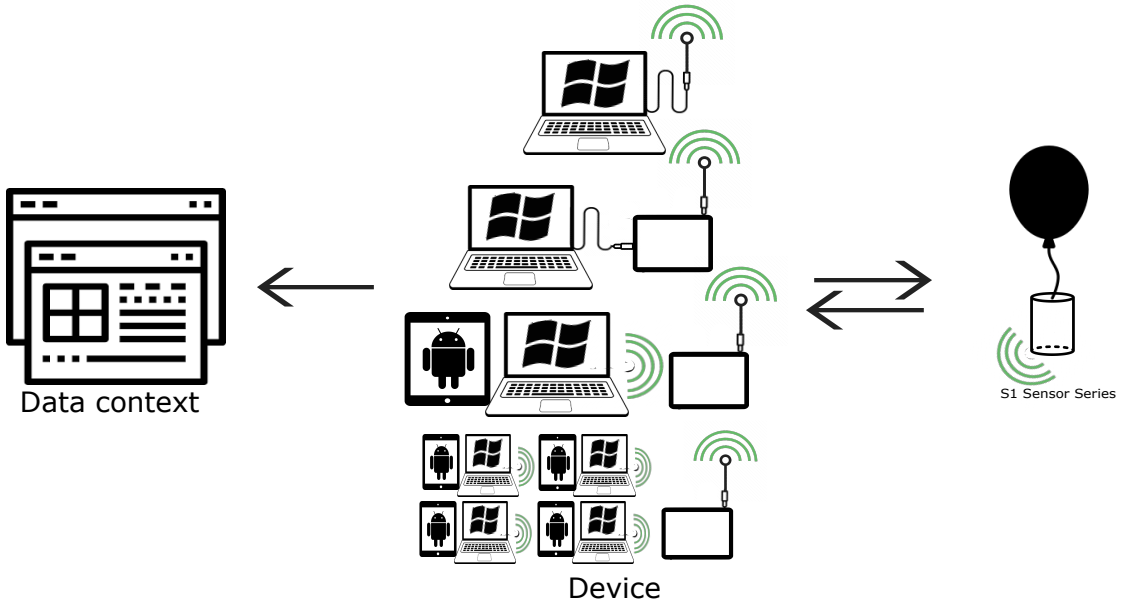

Figure 5: Data flow from windsond until data context

The data pattern refers to the format of the data in its raw, calibrated, or derived form, as well as how it is processed and displayed to the user. A consistent pattern ensures alignment between the data that is recorded, displayed, and exported from the application. Understanding this pattern is essential, not only for developing an application that can handle data in all its states, but also for extracting unique information without relying on specific sensors. For example, the S1H2 sonde estimates altitude using pressure sensor data, which is a common method in atmospheric sensing (Quentin et al. 2019).

To meet these objectives, the data system was built with a structured and well-

documented pipeline, covering everything from data collection and recording to processing and display. During each flight session, parameters such as temperature, humidity, pressure, GPS coordinates, and device-specific metadata are recorded (Markowski et al. 2018). The data is saved in `.sounding` files. A complete list of recorded parameters and their meanings is available in item 10.2.1 of the appendices.

Raw data is processed in the DataContext component, where algorithms apply mathematical methods to calibrate the data and generate new values.

The frontend was designed to support dynamic interaction with both real-time data and replays from saved `.sounding` files. This is achieved by integrating the incoming data into visualization components built using the described libraries. These libraries help display the data in dashboards with live charts, maps, and wind rose diagrams. Users can view the full dataset in spreadsheet format or through specific components, depending on their needs. Multiple input options are available to support data trend analysis and improve readability and comprehension.

The frontend also allows data export in several common formats, such as .CSV, spreadsheets, .KML for geographic paths, .TXT for plain text, and .PNG for saving graphs separately.

Although the DataContext component is covered in a separate chapter it has a strong interaction with the data types being covered here, the table below provides an overview of the types of data collected, the sensors used, and how each data type interacts with the software.

Measurement	Sensor Type	Software Involvement
Temperature	Temperature sensor	Calibration, error correction
Humidity	Humidity sensor	Calibration
Pressure	Pressure sensor	Offset correction, Altitude estimation
GPS (Position)	GNSS receiver	Used to derive wind speed/direction
Voltage	Battery monitor	Alerts for low power levels
Movement / Wind	GPS drift	Calculations, no direct anemometer
Radio Telemetry	RF transceiver	Signal quality monitoring, frequency tuning

Table 2: Sensors and software functionality

4 Project preact components

In this chapter, we review the project's codebase to understand the current state of its modules and how they interact. The modular architecture was designed to encapsulate each feature, enhancing maintainability and testability during development (Meyer & Webb 2005). This approach also allows for greater flexibility, as each part can be expanded independently, especially when multiple developers are working simultaneously. A complete diagram is provided in item 10.1.3 of the appendices.

The modular architecture is divided into six main modules: the Core application modules, which is responsible for wrapping the application in shared logic (e.g., `index.tsx`), the Data processing module, that handles incoming data (e.g., `SoundingDataContext.tsx`), the Input modules, that manage user inputs (e.g., `FileDropZone.tsx`), the Navigation modules, which enable switching between views (e.g., `StatusPage.tsx`), the Configuration module, that manages state changes across the application (e.g., `Configuration.tsx`), and the Visualization module, that handles display elements such as charts and map components.

This division enforces a clear separation of concerns and simplifies code maintenance. The core modules provide a central structure for initializing and wrapping the application. Data is processed separately through dedicated context providers.

Input and configuration modules are clearly defined, which helps manage state without interfering with visualization components. Navigation components are isolated, allowing for seamless transitions and easier route maintenance. Visualization components primarily rely on context and, occasionally, local variables to manage the state only when necessary.

The project uses variables in three main ways: through local `useState` variables, as props passed from parent to child components, and via context for global access (React Team 2024). Understanding this structure is key to analyzing individual components. A table with relevant information is included below.

Usage	**Scope**	**Updatable?**	**Shareable?**
`useState` in component	Local	Yes (internally)	No
Prop	Passed down	No (unless setter is passed)	Yes (limited)
Context	Global	Yes	Yes

Table 3: Comparison of React state handling methods

4.1 Core Application Modules

4.1.1 Component: `index.tsx`

After transitioning to the `Preact` framework, `index.tsx` was defined as the entry point of the application, where execution begins. It contains two main components. The first is <App/>, which defines the overall project structure. It wraps `<AppContent/>` inside `<SoundingDataProvider/>`, allowing any component within `<AppContent/>` to access shared data through context as a global variable (React Team 2024). The application is rendered using Preact's render function.

The second component is `<AppContent/>`, which contains the core logic of the application. This includes state management, routing paths, and rendering of UI components. It primarily uses Preact's `useState` hook to manage local state and `useEffect` to synchronize components with external systems (Bhavzlearn 2022). These features are used to manage a wide range of application states.

`<AppContent/>` also interacts with `SoundingDataContext.tsx` to access a filtered

data array (`filteredData`), a function to process the full dataset (`processFullData`), and to update the current file name in the application state (Byte 2022). It also handles toggling between dark and light modes. One essential state managed here determines whether the application is operating offline, uploading data from a file, or online, receiving data through a `WebSocket`, (Manfra 2021). The routing logic was also moved into `index.tsx`, so any changes to navigation paths should be made here (React Router Team 2024).

The project still includes the legacy `App.tsx` file, which previously defined the application structure and wrapped it in a global context provider. This file is no longer in use, as it was originally built with `React` and has been retained only for reference after the transition to `Preact`.

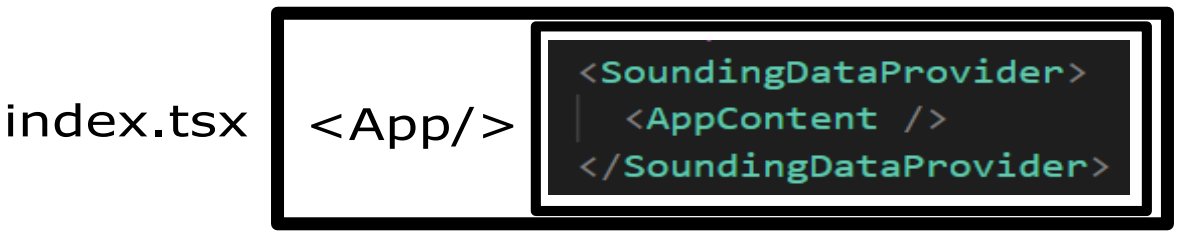

Figure 6: Index nesting

4.2 Data Processing Modules

4.2.1 Component: SoundingDataContext.tsx

This component acts as an intermediary between incoming data and the application. It is the most critical part of the project, supporting the majority of its functionalities. It creates a context using Preact's createContext method. This context is accessed globally by other components through `useContext`(`SoundingDataContext.tsx`), allowing shared variables to be updated and accessed without the need to pass them manually from parent to child at every level.

`SoundingDataContext.tsx` includes a provider component, `<SoundingDataProvider/>`, which manages the entire dataset. This data can be streamed from a sonde via `WebSocket` or uploaded from `.sounding` files containing records of previous flights. It also maintains information about which sonde

should be displayed on the front end, using the sonde's ID. Arrays are used to buffer multiple datasets, enabling the visualization of several datasets simultaneously.
The component also manages a `WebSocket` connection that automatically activates when an available server is detected, using the `useEffect` hook. The connectToWebSocket function, triggered by user input, initiates the connection and sets up event listeners. When data is received, it checks whether it is a full dataset or a single entry. If it is a dataset, the `processFullData` function is called to make the data available to the user. If it is a single entry, the data is parsed, buffered in real time, and periodically (Manfra 2021)..
Below is a visual representation to help understand the data flow from the sonde to the UI components.

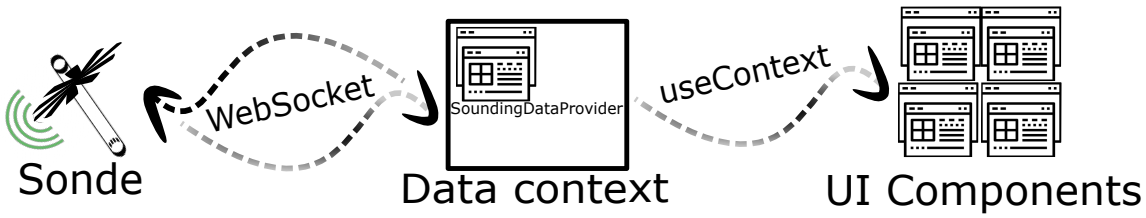

Figure 7: Data flow from sonde until components

Several interfaces are defined in the code, serving as blueprints for object types. The `SoundingData` interface represents a single line in the dataset, containing time-segmented weather data. The `Dataset` interface defines a collection of multiple `SoundingData` entries. The `SondeIdentifier` interface identifies the correct sonde by ID and associates it with the corresponding file, helping to avoid confusion when sondes share the same ID. The `SoundingDataContextType` interface defines the structure of the global context provided to the application.
A dataset can be set as primary using the `makeDataSetMain` function and removed using the `removeDataset` function. After removal, the `clearAllData` function ensures that no residual data remains in the application.
These changes aim to provide more controlled updates, better separation between UI and logic, and centralized state management.

4.3 Navigation Modules

4.3.1 Component: `navBar.tsx`

This component acts as the primary user interaction point. It provides navigation to various display layouts via Hypertext Reference (href) links and directly manages access to the `Configurations.tsx` component. Designed to be responsive across different screen sizes, it uses a handleResize function to adapt the navigation layout into a hamburger menu when needed. The component leverages Preact's `useState` to track which UI to render and `useEffect` to monitor screen width changes, using hooks to manage its internal state (React Team 2024). It does not contain any child components, as routed elements are already handled in `index.tsx`.

Figure 8: navBar screenshot

4.3.2 Component: `miniStatusBar.tsx`

This component combines elements of data visualization with user interaction inputs. As the bar remains constantly visible throughout the application, it uses `useState` to determine when certain data should not be displayed, such as status elements when in `/status` or the sonde selector when in `/overview`, corresponding to items 2 and 1 in Figure 9. This ensures that information is rendered only once on the screen.

Figure 9: miniStatus screenshot

The component has two child components. The first is `sondeSelector.tsx`, a dropdown menu with dynamic values. It updates the list of available sondes using `useEffect`, based on sonde IDs retrieved from `SoundingDataContext` via `useContext` (Byte 2022).. It also updates the selected sonde in the context. Several functions are dedicated to extracting and formatting strings for user-friendly display, minimizing confusion and handling errors, for example, by automatically selecting a sonde to avoid an empty state.

The second component is `fileUploadButton.tsx`, which manages file upload interactions. It includes a hover feature that displays uploaded files and allows for their deletion. It uses `useContext` to transfer files to `SoundingDataContext`, where the data becomes available. The first uploaded file is treated as the main dataset until deleted, at which point the next file in line is promoted. If no files remain, all data is cleared to ensure no residual information is left in the layout.

4.4 Visualization Modules

4.4.1 Component: `overview.tsx`

The overview was designed to serve as the main dashboard of the application. When in offline mode, its initial state renders a file dropzone that prompts the user to drag and upload one or more files. After receiving data, either from a `WebSocket` connection or an uploaded file, the component displays information for all sondes using a layout of Cards. These Cards, defined in `SoundInfo.tsx`, act as selectors for choosing a specific sonde and viewing more detailed information.

The application uses `useEffect` and `useState` hooks to enhance performance, similar to other components. Additionally, it employs `useMemo` to cache the result of hasData, preventing unnecessary re-renders when new data is not significantly different. `useContext` is also used, as in most components throughout the project.

`overview.tsx` has two child components that are conditionally rendered based on the application state. One is `FileDropZone.tsx`, a drag-and-drop area where users can upload files by dragging them from their system. This component checks whether the file is in a valid format (e.g., `.sounding` type) and, if so, passes it to a handler. It also displays error messages when needed, ensuring that issues are communicated clearly without disrupting the user experience.

The second child component is `SoundInfo.tsx`, which is responsible for displaying `.sounding` data. It determines which sonde and file are currently selected by accessing `currentDataIndex` via `useContext` (Byte 2022). Once a sonde is selected, the component displays detailed numerical information instead of just icons or visual elements.

4.4.2 Component: `StatusPage.tsx`

This component was designed to visualize one sonde at a time while allowing the display of multiple types of information simultaneously. For instance, it includes all weather graphs (combined into one), the map component, status components, and radio quality graphs. This is by far the most performance-intensive part of the application.

The code uses `useMemo` to cache important data, similar to `overview.tsx`, including `hasData`, `getTrend` (which calculates differences between current and previous data points to determine trend behavior), and `renderNewTable`, which renders these trends. To optimize performance under heavy usage, the component includes buttons that allow users to conditionally render or hide specific sections. The visibility of each section is managed using `useState` with booleans stored in `visibleSections`.

Its integrated components include `radioQualityPlot.tsx`, `Status.tsx`, `genereateKM Lcomp.tsx`, and `multiPlot.tsx`. At the top of the Status page, the status overview section, defined in `Status.tsx`, displays the most recent real-time or current step data, including temperature, humidity, GPS status, radio link quality, battery voltage, and barometric pressure. The color of each information box changes based on the data state, emulating the three-LED indicator found in the radio receiver hardware. The code structure is similar to other components, using `useState`, `useContext`, and `useEffect` to enhance performance by rendering only when necessary and accessing global variables. `useEffect` also monitors dataset length to detect when a new line of data is available for rendering.

A notable distinction is the use of `useRef` to store the last known voltage (`lastKnownVo ltage`) and speed (`lastKnownSpeed`) (Bhavzlearn 2022). This is necessary because not every data entry includes voltage or speed, which would be resource-intensive to transmit in an embedded system and is not the primary focus of data collection. If `useRef` does not contain valid data, the `findRecentData` function is called to search previous entries for the most recent valid values.

The ”Download KML” button interacts with `genereateKMLcomp.tsx`, allowing users to export sonde trajectory data in KML (Keyhole Markup Language) format. This file type enables 3D visualization of the sonde’s path in tools like Google Earth. The

code constructs a KML-formatted string using the available data.

`radioQualityPlot.tsx` and `multiPlot.tsx` are graphical components with different purposes but similar functionality. Both are built using the `plotly.js-dist-min` library, which supports interactive features such as zoom, pan, export, and tooltips. They follow the same rendering pattern as other graphical components in the application. Updates to props trigger re-renders, ensuring synchronization with the current application state.

`radioQualityPlot.tsx` specializes in representing signal quality as a percentage. It builds Plotly traces and incorporates performance optimizations using `useEffect`, `useContext`, `useRef`, and `useState`. The tick spacing for time and distance axes is dynamically adjusted to handle large datasets efficiently, and the data range can be manually configured. These adaptations significantly improve performance, especially when rendering within `StatusPage.tsx`, which already includes several other data visualization elements.

`multiPlot.tsx` was designed to combine multiple weather data graphs based on altitude. Each dataset is represented with distinct Plotly styles with variations in color, line type, etc., and aggregated into a single data array. Traces are rendered in separate layers, allowing users to toggle their visibility.

`leafletMap.tsx` and `dataLog.tsx` are not discussed here, as they are reused from more specialized components.

4.4.3 Component: `MapPage.tsx`

This component serves as a container for wrapping `LeafletMap.tsx` and passing values as props. These props include visualization customization options such as `isDarkMode`, and data-related controls like `isOnlineMode` and `currentDataIndex`.

`LeafletMap.tsx` is the child component. It utilizes the `preact-leaflet` library to render the map, plot markers, and trace the path of a navigating sonde (Agafonkin & contributors 2011). It also supports interactive features such as hovering over markers to display data, zooming, panning, and dynamic centering of the map. Helper functions are used to convert coordinate integers into standard decimal degrees, enabling compatibility with the mapping library. The icons were updated to SVG-based versions to resolve compiler incompatibilities.

All processing logic is handled within `LeafletMap.tsx`, while `MapPage.tsx` simply passes data not available in the context as props (React Team 2024).
`LeafletMap.tsx` is also reused in `StatusPage.tsx`, alongside other components. A visual representation is shown below.

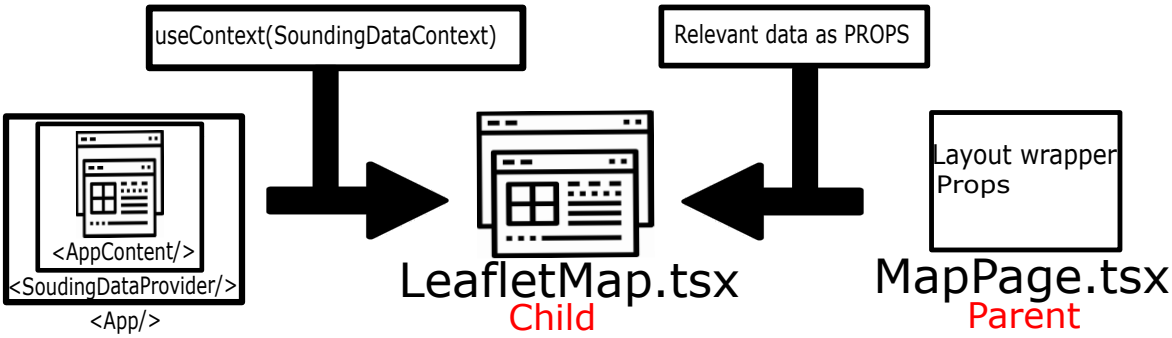

Figure 10: MapPage component data children flow

4.4.4 Component: `weather.tsx`

This component functions differently from `MapPage.tsx`, featuring a significantly larger number of child components and more complex conditional rendering rules. In this case, the parent component plays a more active role than simply wrapping its children, it determines what should be rendered based on the current state (React Team 2024).
It contains five child components: `WindSpeedPlot.tsx`, `TempHumPlot.tsx`, `WindRoseC hart.tsx`, `WindDirectionPlot.tsx`, and `WindDirectionPolarChart.tsx`. The user can control the layout by switching between dataset stages, defined by the mode (rising, falling, or both), or by toggling the visibility of individual child graphs. The parent component accesses context information, processes it, and passes the relevant data to its children (React Team 2024).
Below is a visual representation:

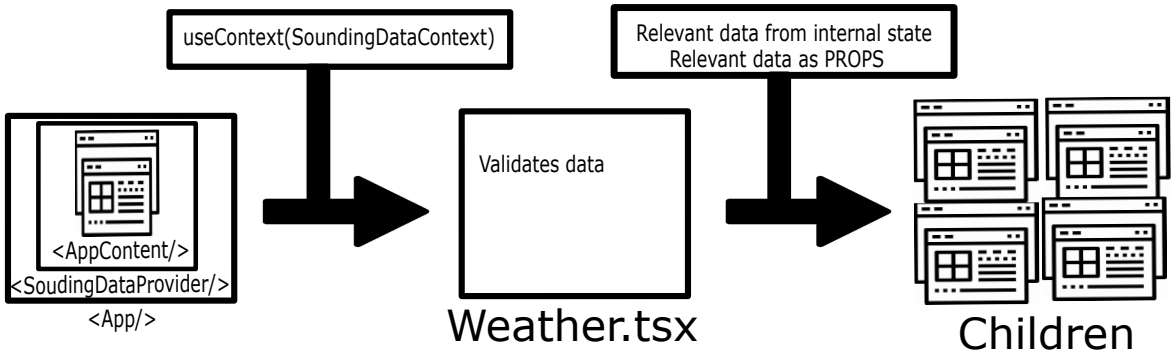

Figure 11: Weather component parent-children flow

The current child components of `weather.tsx` can be categorized by the type of graph they use. `WindSpeedPlot.tsx`, `WindDirectionPlot.tsx`, and `TempHumPlot.ts x` are Line/Cartesian plots that represent time-series data using line charts, bar charts, or other standard Cartesian coordinate systems. `WindRoseChart.tsx` and `WindDirec tionPolarChart.tsx` are Polar/Radial charts, commonly used in meteorology to display directional patterns using polar or radial plots.

4.4.5 Components: `textReport.tsx` and `dataLog.tsx`

A simple component designed to display formatted data as a text report. `useEffect` is used to track the report height and enable auto-scrolling, unless interrupted by user input and `useMemo` enhances performance by checking for changes in data length before triggering a re-render React Team (2024). The core functionality relies on context-aware functions that access the data to be displayed, along with `currentDataIndex` and `isOnlineMode`, to determine how the data should be streamed. The report can be viewed within the component and downloaded in two formats: plain text (.txt) and spreadsheet (.csv).

With a similar purpose, `dataLog.tsx` shares a comparable structure but focuses less on exportable formatting and more on displaying the maximum amount of information. It uses the same features (e.g., `useContext`, `useMemo`, `isOnlineMode`, etc.) and includes similar functionality to `textReport.tsx`, such as auto-scrolling. It also provides filtering functions based on user input and helps maintain a clean and organized layout.

As a reminder, this component is also reused in `StatusPage.tsx`, just like `LeafletMap.tsx`.

4.5 Configuration Module

4.5.1 Component: `Configuration.tsx`

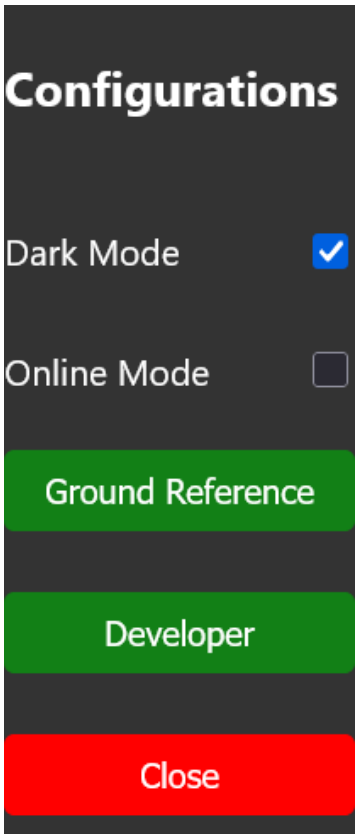

Figure 12: Configurations

This component is located inside `navBar.tsx` and opens as a modal view when the user clicks the configuration icon. Its purpose is to enhance the user experience by providing access to a range of customization and development tools, while also centralizing configuration inputs for easier maintainability.

This control panel uses toggle switches and buttons to allow users to switch between light and dark themes, open the developer toolkit, or set a new ground reference for measurements. It uses `useState` to manage the state of the modals. Notably, it also uses `useContext`, not just to consume data, but to modify the reference values for altitude and pressure. This demonstrates the utility of context for managing global variables that need to be updated dynamically.

Props are used to control the toggles, such as `isDarkMode` for theme switching and `isOnlineMode` to toggle between `WebSocket` and file upload modes.

4.6 Input Modules

4.6.1 Component: `sondeControl.tsx`

In its current state, this component serves as a shell for a UI element. It contains placeholder functions that need to be implemented to enable communication with the sonde. Once completed, it should be capable of sending commands to the sonde via `WebSocket` in response to user inputs. This would include functionalities such as triggering a beep for recovery, sending cut-down commands, or issuing commands

in CLI format.

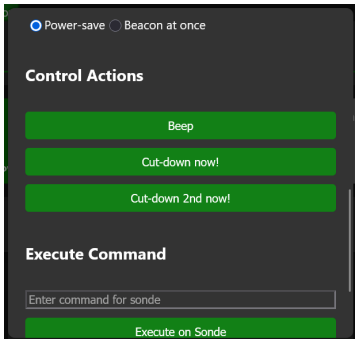

Figure 13: Sonde controller screenshot

4.6.2 Component: `StepCounter.tsx`

The desktop version of this application included a slider tool for navigating through data entry lines. Originally intended for development purposes, this tool was replicated in the web version. In addition to the slider for controlling the current step, the component features a play button for automatic progression, a velocity selector to adjust the playback speed, arrow buttons to move forward or backward by individual steps, and a text input box for jumping to a specific step number. The component is draggable, allowing it to be repositioned as needed.

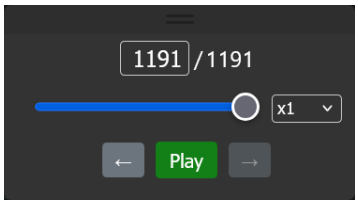

Figure 14: Counter screenshot

The component relies on props from its parent, `index.tsx`, to receive `filteredData`, as it does not access context directly. The main props are `setCurrentDataIndex` and `currentDataIndex`, which allow it to update the current index value based on user input via the slider, buttons, or text input box.

5 Simulators

5.1 Component: `DeveloperModeModal.tsx`

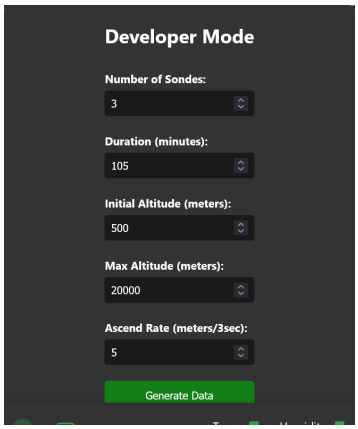

Figure 15: Developer modal

During development, it became evident that acquiring new `.sounding` files with specific characteristics for testing was challenging due to the in-field nature of data collection. To address this, a developer tool was created to quickly generate mock-up data that emulates real sonde flights.

The generated data aims to represent ideal conditions, which sometimes results in an excessive number of entries that would not typically be transmitted by a real sonde for optimization purposes. Additionally, some entries may appear unrealistic (e.g., abrupt and unnatural changes in flight path direction).

Despite lacking full realism, the generated data has proven useful for testing new features, such as visualizing multiple sondes simultaneously or reading `.sounding` files containing data from multiple sondes.

Currently, the generation is guided by a few parameters: the number of sondes in the file, the simulation duration, the starting and ending altitudes (cut-down), and the sonde's ascent rate. Based on these user-defined parameters, a series of functions emulate real-world values and generate entry lines, such as the example below:

00h00m00s400[#MET:rltime=1800.000,sid=1,rs=-109,snr=25,md=0,te=27.95,hu=36.53,pa=95406,alt=504,spd=5.84,ang=149.76,lat=36.078888,lon=-97.049574,spd1=5.84,ang1=149.76,te1=27.94,hu1=36.51,su=4.31]

Finally, the generated lines are organized by timestamp and exported as a `.sounding` file in the same project folder.

5.1.1 Several Simulators

One of the development challenges was reproducing the sonde behavior in-house. To address this, several simulator versions were developed. These simulators load data from `.sounding` files stored on the computer and transmit it to the web application via `WebSocket`.

They were created at different stages of development by various developers, so their functionalities can differ significantly, ranging from major differences to minor adjustments reflecting changes in the application.

The simplest simulator is `simulator3.cjs`, which sends all the data from a hard-coded file as soon as a connection with the client is detected. The data is in `JSON` format. This simulator is particularly useful in offline development mode, simulating the client-side reanalysis of collected data. UI elements are populated instantly, and the slider is set to the last data point.

The simulators `simulator.cjs` and `simulator.js` are structurally similar, with the former being a more advanced version of the latter. Both simulate real-time data transfer by introducing delays between transmissions to the UI. Each line marked with `MET` is sent individually. `simulator.js` parses timestamps in the format `00:45:300`, but does not support hours, which can cause layout issues for files longer than one hour. `simulator.cjs` was developed to resolve these issues and supports full timestamp parsing, including hours, minutes, seconds, and milliseconds. These simulators test the application's expected normal usage.

`simulator4.cjs` simulates the specific scenario where reconnection after a lost connection and resumption of data transfer from the last known data line. It internally tracks the number of lines sent and queries the client for the last received line before resuming. This simulator tests the application's behavior in the event of service disruptions.

6 Conclusion

The current project presents a substantial development from its previous state and offers a foundation for future work. The application is highly responsive and can be expanded or integrated into new platforms and devices, especially when compared

to its desktop version. It allows for the visualization of real-time or recorded data in a lightweight yet powerful way.

The objective of this study was to build a frontend application capable of receiving, parsing, and displaying tropospheric atmospheric data. The data should be displayed in a modular and clean layout, focusing on performance and interactivity. The web application should be able to run from a microcontroller, so resource constraints were considered. `Preact` was selected for this reason, as its reduced size allows for fast initial loading and low memory overhead, while retaining the core benefits of `React` with reduced complexity.

The real-time data and large datasets handled by the frontend in a project like this can easily overwhelm the rendering engine. Therefore, the use of the Virtual DOM to efficiently update the UI only when changes occur is essential for maintaining good performance. The integration of libraries like `Plotly` and `Leaflet` brought significant benefits for displaying dynamic charts and maps for real-time data visualization, especially since they can perform updates outside the Virtual DOM, updating only when necessary and avoiding unnecessary full re-renders.

In terms of architectural decisions, the application uses `Preact` hooks (e.g., `useState`, `useEffect`, `useMemo`, `useContext`, etc.) to manage state, optimize re-renders, and transfer data between components. The modular approach allows for component reusability and flexibility in the project's expansion.

There is room for improvement in all the features described above. Some mistakes were made and could be addressed to enhance the application. Due to the thesis proponent's limited experience, there was a lack of a clear architectural overview from the beginning. This affected components like `LeafletMap.tsx`, which followed a different standard from the rest of the visualization components, as seen in Figures 11 and 10 from Section 4.4. Drawing diagrams to illustrate how data flows could have improved clarity and positively influenced decision-making and development.

At some point during development, version control practices slowed down, becoming less frequent than at the start. Maintaining a better frequency and more descriptive commit messages would have helped in situations where reverting changes was necessary. This inconsistency affected task time estimations when problems were encountered. Perhaps this was one of the reasons for delays, and time constraints

ultimately prevented work on the radio receiver component, which would have enriched both the thesis and the proponent's experience.
In the end, while not all components were completed, a highly functional prototype of a web-based data visualization tool was developed. It can receive, process, and display data effectively. This shows that powerful modern tools can be optimized for use on resource-constrained hardware. Future work should focus on refining the application and fully integrating it into Sparv's ecosystem of tools and services.

7 Thoughts about future implementations

The application can be improved in several ways, depending on the decisions made regarding the objects defined in future discussions. These actions may focus on optimizing, scaling, or enhancing system maintainability, performance, or integration with other systems.
Currently, the application is designed to operate locally or be deployed in embedded environments. A major improvement would be the implementation of a cloud-based ecosystem. This would enable real-time access to recordings and analyses online across multiple user accounts, enhancing collaboration among teams in different locations. Such a system could be built using existing services like Amazon Web Services (AWS), Firebase, or custom solutions. Features like automated data evaluation, AI integration, and tools for generating reports or 3D maps could be included. This could even lead to the creation of a public climate data repository, establishing the project as a reference point for meteorological research.
A fully functional file generator could be developed to support testing and development. While the current implementation prints all necessary data, more realistic datasets would aid in debugging. Similarly, the simulator could be consolidated into a single codebase, with configurable switches to select modes, files, and connection types, which eliminates the need to launch each mode via the command line. To reduce the initial project size, tools could be made standalone or modular, with dependencies installed conditionally based on user needs.
Some decisions regarding component and variable names, as well as code patterns, were made hastily to accelerate development. The project would benefit from a thor-

ough refactoring, like renaming variables and functions to better reflect their roles and improving data flow clarity. Now that the project is more mature, the variable flow could be redesigned, with some variables added to context instead of being passed as props. Identifying and removing unused code could also be beneficial. Improved documentation would help future developers recognize and address these issues.

The cutdown component is an example of an unfinished feature, as is the interaction with the sonde. Completing these, along with finalizing the radio receiver, would move them beyond placeholders and give the project a more polished appearance more suitable for live demonstrations or early release.

Efforts have been made to reduce bundle size warnings during the build process with Vite. However, the compiler still issues warnings related to the map component. Further performance improvements could be achieved through lazy loading and code splitting.

These changes could pave the way for the project to evolve from a high-functioning prototype into a finished product.

8 References

Agafonkin, V. & contributors (2011), ‘Leaflet - javascript library for interactive maps’. Available at: https://leafletjs.com. Accessed: 21.5.2025.

Augustin, A., Yi, J., Clausen, T. & Townsley, D. (2016), ‘A study of lora: Long range & low power networks for the internet of things’, *Sensors* **16**(9), 1466.

Bhavzlearn (2022), ‘Demystifying useref and usememo in react’, https://dev.to/bhavzlearn/demystifying-useref-and-usememo-in-react-4jcl. Accessed: 2025-05-27.

Byte, F. (2022), ‘How to use react context api with usereducer, usememo hooks’, https://frontendbyte.com/how-to-use-react-context-api-usereducer-hooks. Accessed: 2025-05-27.

Inc., P. T. (2016), ‘plotly.js - the open source javascript graphing library’. Available at: https://github.com/plotly/plotly.js. Accessed: 21.5.2025.

LoRa Alliance (2017), ‘Lorawan security whitepaper’. Available at: https://lora-alliance.org/wp-content/uploads/2020/11/lorawan_security_whitepaper.pdf. Accessed: 26.5.2025.

Manfra, L. (2021), ‘Using websockets in react’, https://blog.logrocket.com/using-websockets-in-react/. Accessed: 2025-05-27.

Markowski, P. M., Richardson, Y. P., Richardson, S. J. & Petersson, A. (2018), ‘Aboveground thermodynamic observations in convective storms from balloonborne probes acting as pseudo-lagrangian drifters’, *Bulletin of the American Meteorological Society* **99**(4), 711–724.

Meyer, M. & Webb, P. (2005), Modular, layered architecture: the necessary foundation for effective software development, Technical report, MIT Sloan School

of Management.
URL:
https://web.mit.edu/deweck/Public/ESD39/Readings/MatlabCaseStudy.Meyer-Webb-2005.pdf

Porambage, P., Gür, G., Moya Osorio, D. P., Liyanage, M., Gurtov, A. & Ylianttila, M. (2021), 'The roadmap to 6g security and privacy', *IEEE Open Journal of the Communications Society* **2**, 1094–1122.

Quentin, G. E., Fosu-Amankwah, K., Petersson, A. & Brooks, B. J. (2019), 'Evaluation of windsond s1h2 performance in kumasi during the 2016 dacciwa field campaign', *Atmospheric Measurement Techniques* **12**(2), 1311–1324.

React Router Team (2024), 'React router: Declarative routing for react.js', `https://reactrouter.com/en/main/start/overview`. Accessed: 2025-05-27.

React Team (2024), 'React hooks documentation', `https://react.dev/reference/react/useState`. Accessed: 2025-05-27.

Stameus, M. (2024), 'Windsond web: Initial investigation'.

Wilson, M. B. & Hawksworth, P. (2019), *Modern Web Development on the JAMstack: Modern Techniques to Build Amazing Websites*, O'Reilly Media.

9 APPENDICES

9.1 Additional figures

9.1.1 Expected deployment flow

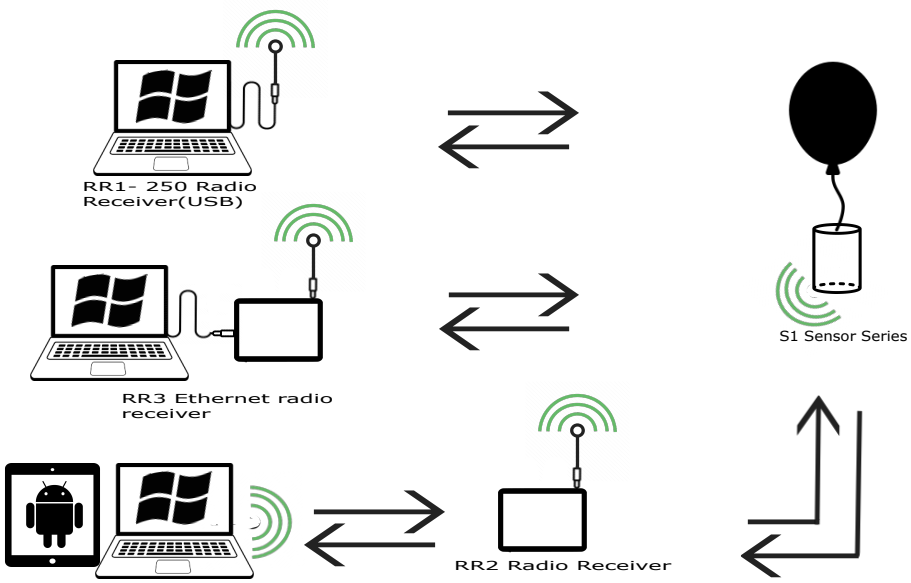

Figure 16: Other products use flow

9.1.2 Windsond S1

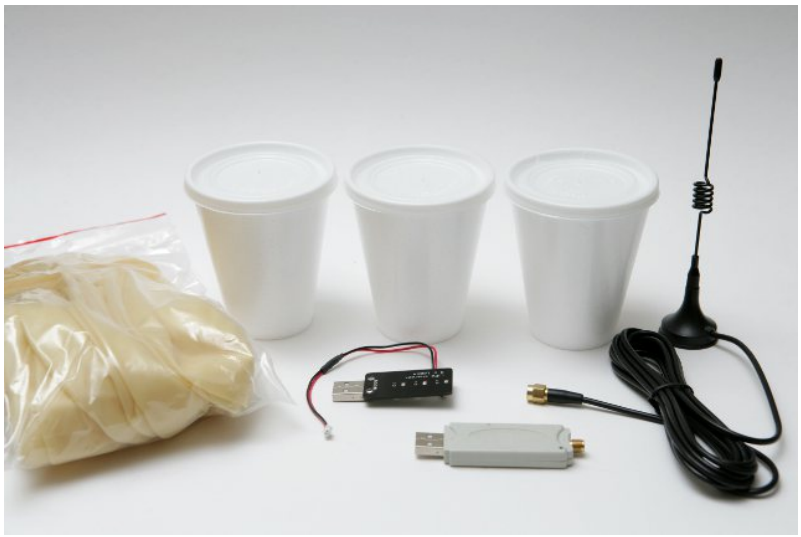

Figure 17: Windsond S1 and components

9.1.3 Full modular architecture

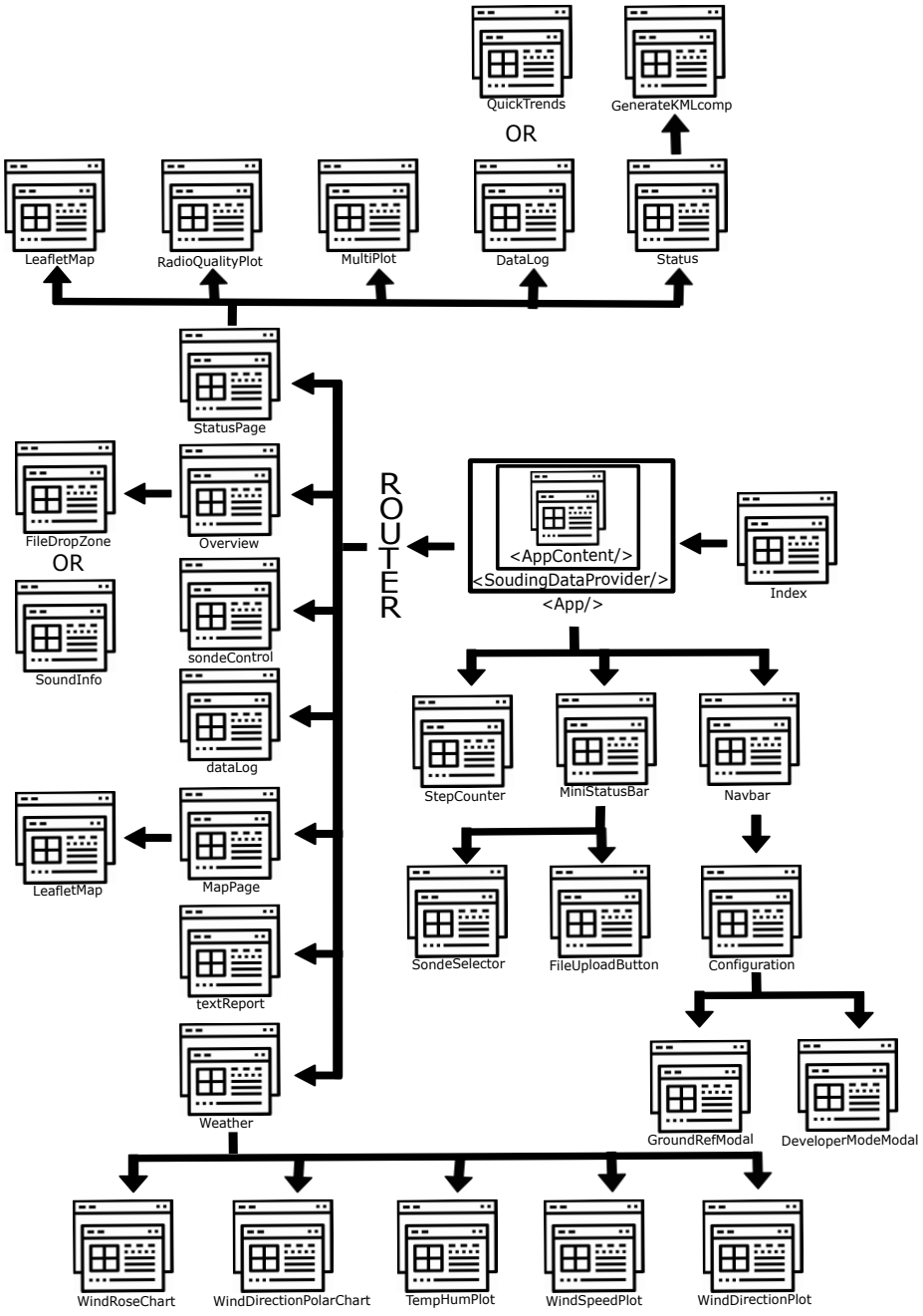

Figure 18: Modules architecture

9.2 Additional Tables

9.2.1 Data parameters from Windsond

Parameter	Meaning
syn	Time synchronization flag (currently unused).
r	Packet reception quality (0–255, where 255 is perfect).
afc	Automatic Frequency Correction (debug info on frequency drift).
id	Unique device ID of the sonde, receiver, or device.
sid	Session ID assigned by receiver.
md	Mode of the sonde: 0=INIT, 1=READY, 2=RISING, 3=FALLING, 4=POWERSAVE, 5=FINDME, 6=FINDME_SLOW, 7=CUTDOWN.
te	Temperature in °C.
hu	Humidity in % RH.
pa	Pressure in Pascals.
alt	GPS altitude (debug only, use pressure for calculations).
spd	Speed in m/s.
ang	Direction of motion (wind is opposite).
lat, lon	Full latitude and longitude: Format is XXXYYYYYY = XXX degrees and YY.YYYY minutes.
latm, lonm	Latitude and longitude (minute parts only).
latd, lond	Decimal part of the minutes (YYYY only).
spd1, spd2	Speed 1 or 2 seconds ago (in m/s).
ang1, ang2	Angle 1 or 2 seconds ago (in degrees).
te1, te2	Temperature 1 or 2 seconds ago (in °C).
hu1, hu2	Humidity 1 or 2 seconds ago (% RH).
su	Supply voltage (in V).
behlan	Behavior after landing (integer flag).
role	Device role: 0=Undefined, 1=Radiosonde, 2=One-time Radiosonde, 3=Dropsonde, 4=Dropsonde with cutdown, 5=UAV, 6=Receiver.

Continued on next page

Table 4 – continued from previous page

Parameter	Meaning
clk	Speed correction (clock correction factor).
pwr	Radio transmission power (7 = 100 mW).
fwver	Firmware version (e.g., 205 = v2.05).
rltime	Receiver local time since startup (seconds).
gpse	GPS enabled flag.
logst	Logging status.
mcnt	MET data packet counter.
cutalt	Cutdown altitude.
cutpr2	Secondary cutdown pressure.
supmul	Supply multiplier.
gpa	Ground pressure (altimeter reference).
galt	Ground altitude.
hw	Hardware version or status.
burn	Burn wire status.
ucnt	Usage counter (e.g., how many times the sonde has been used).
tim	System time.
rssi	RSSI from a scan (signal strength).
ctmr	Cutdown timer value.
net	Network settings (e.g., timeslot info).
twire	Test wire status.
blk	Blink (LED blink command/status).
beep	Beep status.
gpss	GPS status.

9.2.2 Variables and the Components that Use Them

Variable	Used In Components
currentDataIndex	Overview, MapPage, LeafletMap, StatusPage, GenerateKMLComp, QuickTrends, WeatherComponent, GroundRefModal, MiniStatusBar, StepCounter
SoundingDataContext	Overview, SoundInfo, StatusPage, RadioQualityPlot, MultiPlot, DataLog, Status, GenerateKMLComp, SondeSelector, FileUploadButton, WeatherComponent, GroundRefModal, DataLog (standalone)
isOnlineMode	FileDropZone, GenerateKMLComp, MiniStatusBar, StepCounter, GroundRefModal
currentFileName	FileDropZone (optional), FileUploadButton, MiniStatusBar
handleFileProcess	FileDropZone, FileUploadButton, MiniStatusBar
setCurrentFileName	FileDropZone, FileUploadButton, MiniStatusBar
config toggle	Navbar, DeveloperModeModal
config management	Configurations
isDarkMode	MiniStatusBar
showStatusElements	MiniStatusBar
currentUrl	MiniStatusBar
filteredData.length>0	StepCounter
data (Weather.tsx)	WindDirectionPlot, WindSpeedPlot, TempHumPlot, WindDirectionPolarChart, WindRoseChart